

Programmer's Guide

SunTM ONE Identity Server

Version 6.0

December 2002
816-6687-10

Copyright © 2002 Sun Microsystems, Inc. All rights reserved.

Sun, Sun Microsystems, and the Sun logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Federal Acquisitions: Commercial Software -- Government Users Subject to Standard License Terms and Conditions. The product described in this document is distributed under licenses restricting its use, copying, distribution, and decompilation. No part of the product or this document may be reproduced in any form by any means without prior written authorization of the Sun Microsystems, Inc. and its licensors, if any.

THIS DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Some preexisting portions Copyright (c) 1999 The Apache Software Foundation. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment:

"This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>)."

Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear.

4. The names "" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org.

5. Products derived from this software may not be called "Apache", nor may "Apache" appear in their name, without prior written permission of the Apache Software Foundation.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright © 2002 Sun Microsystems, Inc. Tous droits réservés.

Sun, Sun Microsystems, le Sun logo, et iPlanet sont des marques dposes ou des marques dposes registre de Sun Microsystems, Inc. aux Etats-Unis et d'autres pays.

Le produit dé crit dans ce document est distribué selon des conditions de licence qui en restreignent l'utilisation, la copie, la distribution et la décompilation.

Aucune partie de ce produit ni de ce document ne peut être reproduite sous quelque forme ou par quelque moyen que ce soit sans l'autorisation écrite préalable de Sun Microsystems, Inc., le cas échéant, de ses bailleurs de licence.

CETTE DOCUMENTATION EST FOURNIE "EN L'ÉTAT", ET TOUTES CONDITIONS EXPRESSES OU IMPLICITES, TOUTES REPRÉSENTATIONS ET TOUTES GARANTIES, Y COMPRIS TOUTE GARANTIE IMPLICITE D'APTITUDE À LA VENTE, OU À UN BUT PARTICULIER OU DE NON CONTREFAÇON SONT EXCLUES, EXCEPTÉ DANS LA MESURE OÙ DE TELLES EXCLUSIONS SERAIENT CONTRAIRES À LA LOI.

Contents

About This Guide	11
About Identity Server 6.0	11
What You Are Expected to Know	11
Identity Server Documentation Set	12
Documentation Conventions Used in This Guide	13
Typographic Conventions	13
Terminology	13
Related Information	14
Documentation Comments	15
Chapter 1 Introduction	17
Identity Server Overview	17
Data Management Components	17
Application Management Services	19
Managing Access	20
Extending Identity Server	21
Service Definition With XML	21
Identity Server Console Customization	22
Java Packages	22
Identity Server File System	24
Chapter 2 The Identity Server Console	27
Overview	27
Console Interface	28
Architecture	28
Customizing The Console	29
Default Console Directory	30
Creating Custom Organization Files	30
Precompiling JSP Files	32
Customizing The User Profile View	32

Miscellaneous Customizations	33
Changing Default Attribute Display	33
Localizing The Console	34
Customizing Background Colors	34
Labelling The Module Tab	35
Displaying Container Objects	35
Console Sample	36
Chapter 3 Authentication Service	37
Overview	37
Accessing The Authentication Service	38
Authenticating The Request	39
Miscellaneous Features	41
The Authentication User Interface	42
Customizing The Authentication Interface	43
JSP Templates	45
Authentication Module Configuration Files	47
Default Authentication Modules	47
Core Authentication Service	47
Proprietary Authentication Modules	47
Assigning The Authentication Method	48
Custom Authentication Modules	54
Creating A New Authentication Module	55
Configuring Localization Properties	56
Configuring Module Credential Requirements	57
Modifying amAuth.xml	61
Application Authentication	61
Authentication API For Java Applications	62
Authenticating Non-Java Applications	63
The remote-auth.dtd Structure	63
Authentication SPI	71
URL Parameters	71
C Programs and Authentication	73
Authentication Request / Response Flow	74
Authentication Samples	77
Remote Client API	77
Login Module	77
Chapter 4 Single Sign-On	79
Overview	79
Contacting A Policy Agent	80
Creating A Session Token	80

Providing User Credentials	80
Cookies and Session Tokens	81
Cross-Domain Support For SSO	81
Enabling Cross-Domain Single Sign-On	82
Configuring For Cross-Domain SSO	83
SSO API	85
Non-Web-Based Applications	85
API Overview	86
Sample API Code	90
Sample SSO Java Files	94
SSO Servlet Sample	94
Remote SSO Sample	95
Command Line SSO Sample	95
Chapter 5 Identity Management	97
Overview	97
Abstract Objects	98
Object Templates	99
Structure of ums.xml	100
Modifying ums.xml	101
Identity Server SDK	102
SDK Interfaces	103
The SDK And Cache	106
Installing the SDK Remotely	107
amEntrySpecific.xml	107
amEntrySpecific.xml Schema	107
Management Sample Functions	109
Create, Delete Or Modify Users	109
Create Organization	109
Retrieve Templates	110
Create Users With Modified LDAP Schema	111
Chapter 6 Service Management	113
Overview	113
XML Service Files	114
Document Type Definition Structure Files	114
Service Management SDK	115
Service Definition	115
Defining A Service	115
Creating A Service File	117
Extending The Directory Server Schema	120
Importing the XML Service File	122

Configuring Localization Properties	123
Updating Files For Abstract Objects	124
Registering The Service	124
DTD Files	125
The sms.dtd Structure	126
The amAdmin.dtd Structure	135
XML Files	150
Default XML Service Files	150
Batch Processing XML Files	153
Customizing User Pages	156
Service Management SDK	156
Chapter 7 Policy Service	159
What Is Policy?	159
Policy Service	160
Architecture	161
Policy Types	162
Subjects	163
Policy Definition Type Document	164
Policy Element	164
Rule Element	164
ServiceName Element	165
ResourceName Element	165
AttributeValuePair Element	165
Subjects Element	166
Subject Element	166
Referrals Element	166
Referral Element	166
Conditions Element	167
Condition Element	167
Java SDK For Policy	167
Policy Evaluation Java APIs	168
Policy Management Java APIs	169
Policy Plugin Java APIs	170
C Library For Policy	171
C APIs for Policy Evaluation	172
am_properties_t	185
Information And Utility APIs	192
am	194
am_policy	196
Specialization Methods	201
Initialization Variables	203
Specialization Methods For Web Agents	205

Initialization Variables	218
Chapter 8 Using The SAML Service	219
Overview	219
Assertion Types	221
Profile Types	222
SAML SOAP Receiver	224
Accessing The SAML Service	226
amSAML.xml	226
SAML SDK	227
com.sun.identity.saml	227
com.sun.identity.saml.assertion	228
com.sun.identity.saml.common	228
com.sun.identity.saml.plugins	229
com.sun.identity.saml.protocol	229
com.sun.identity.saml.xmlsig	231
SAML Service Samples	231
Chapter 9 Federation Management	233
Overview	233
The Liberty Alliance Project	234
Liberty Specification Concepts	235
Federation Management Process	236
Federation Management Protocols	238
Federation Management API	240
Customizing The Module	241
Federation Management Samples	242
Chapter 10 Logging Service	245
Overview	245
Logging Architecture	246
Logging Service XML File	247
Log Security	247
Log Message Formats	247
Flat File Format	247
Relational Database Format	248
Logging API	249
Logger Class	249
LogRecord Class	249
Logging Exceptions	250
Sample Logging Code	251
Logging SPI	251

Plugin Log Verifier	251
Plugin Authorization Mechanism	252
Log Files	252
SSO-related Logs	252
Console-related Logs	253
Authentication-related Logs	253
Federation-related Logs	253
Debug Files	253
Secure Logging	254
Chapter 11 Client Detection	255
Overview	255
Client Data	256
Client Detection API	257
Client Detection Module Interface	258
Chapter 12 Identity Server Utilities	263
Backup And Restore	263
Backup Script	264
Restore Script	267
Utility API	268
API Summary	268
Appendix A AMConfig.properties File	271
Overview	271
Deployment Directives	272
Identity Server Directives	272
Directory Server	273
Configuration Directives	274
Debug Service	274
Stats Service	275
SAML	276
Miscellaneous Services	276
Read-Only Directives	279
Base Directory	280
Shared Secret	280
Deployment Descriptors	280
Session Properties	280
Cross Domain Single Sign-On Support	281
SecureRandom Properties	282
SocketFactory properties	282
Encryption	282

Remote Installation	283
IP Address Checking	283
Remote Policy API Directives	283
FQDN Map	285
Appendix B Directory Server Concepts	287
Overview	287
Roles	288
Managed Roles	288
How Identity Server Uses Roles	290
Access Control Instructions (ACIs)	292
Defining ACIs	293
Format of Predefined ACIs	293
Class Of Service	296
CoS Definition Entry	297
CoS Template Entry	297
Conflicts and CoS	298
Application Schema	298
Index	299

About This Guide

The Sun™ ONE *Identity Server Programmer's Guide* offers information on how to deploy and customize the Sun ONE Identity Server. This preface contains the following sections:

- About Identity Server 6.0
- What You Are Expected to Know
- Identity Server Documentation Set
- Documentation Conventions Used in This Guide
- Related Information

About Identity Server 6.0

Sun ONE Identity Server, prior to the 6.0 release, was known as iPlanet Directory Server Access Management Edition (DSAME). The product was renamed shortly before the launch of version 5.1. Identity Server is designed to help organizations manage identities and enforce secure access to their network services and web-based resources. It contains a number of services towards this end as well as the Sun ONE Directory Server as a data store. For the latest information about new features and enhancements in this release of Identity Server, please see the online release notes at <http://www.sun.com/software/> or the Sun ONE Identity Server Product Brief.

What You Are Expected to Know

This Programmer's Guide is intended for use by IT administrators and custom software developers who manage identities and access to their web resources using Sun ONE servers and software. It is recommended that administrators understand directory server technologies, including Lightweight Directory Access Protocol

(LDAP), and have some experience with Java™, Java Server Pages, Hyper Text Markup Language (HTML) and eXtensible Markup Language (XML). Particularly, they should also be familiar with Sun ONE Directory Server and the documentation provided with that product.

Identity Server Documentation Set

The Identity Server documentation set contains the following titles:

- *Product Brief* provides an overview of the Identity Server application and its features and functions.
- *Installation and Configuration Guide* provides details on how to install and deploy the Identity Server on Solaris™, Linux and Windows® 2000 systems.
- *Administration Guide* describes how to use the Identity Server console as well as manage user and service data via the command line.
- *Programmer's Guide* (this guide) documents how to customize an Identity Server system specific to your organization. It also includes instructions on how to augment the application with new services using the public APIs.
- *Policy Agent Guide* documents how to install and configure an Identity Server policy agent on a remote server. It also includes troubleshooting and information specific to each agent.
- *Getting Started Guide* documents how to use various features of the Identity Server product to set up a simple organization with identities, policies and roles.
- The *Release Notes* file gathers an assortment of last-minute information, including a description of what is new in this release, known problems and limitations, installation notes, and how to report problems.

NOTE Be sure to check the Identity Server documentation web site at <http://docs.sun.com/db/prod/slidsrv#hic> for updates to the release notes and for revisions to the documentation. Updated books will be marked with a revision date.

Documentation Conventions Used in This Guide

In the Identity Server documentation, certain typographic conventions and terminology are used. These conventions are described in the following sections.

Typographic Conventions

This book uses the following typographic conventions:

- *Italic type* is used within text for book titles, new terminology, and emphasis.
- `Monospace font` is used for sample code and code listings, APIs and programming language elements (such as function names and class names). It is also used for filenames, pathnames, directory names, HTML tags, URLs, and any text that must be typed on the screen.
- `<sample text>` is used to represent a variable placeholder. When this convention is used in a directory path or URL, the text and surrounding carats should be replaced with deployment-specific information. For example, the following command uses `<filename>` as a variable placeholder for an argument to the `gunzip` command:

```
gunzip -d <filename>.tar.gz
```

NOTE Notes, Cautions and Tips highlight important conditions or limitations. Be sure to read this information.

Terminology

Below is a list of general terms used in the Identity Server documentation set:

- `<identity_server_root>` is a variable placeholder for the path to the home directory where Sun ONE Identity Server is installed.
- `<directory_server_root>` is a variable placeholder for the path to the home directory where Sun ONE Directory Server is installed.

Related Information

In addition to the documentation provided with Identity Server, there are several other sets of documentation that might be helpful. This section lists these and additional sources of information.

Sun ONE Directory Server Documentation

iPlanet Directory Server 5.1 documentation can be found at
http://docs.sun.com/db/coll/S1_ipDirectoryServer_51.

Sun ONE Web Server Documentation

iPlanet/Sun ONE Web Server documentation can be found at
http://docs.sun.com/db/coll/S1_ipwebservree60_en.

Sun ONE Certificate Server Documentation

iPlanet Certificate Server documentation can be found at
http://docs.sun.com/db/coll/S1_s1CertificateServer_47.

iPlanet Proxy Server Documentation

iPlanet Proxy Server documentation can be found at
http://docs.sun.com/db/coll/S1_ipwebproxysrvr36.

Other iPlanet Product Documentation

Documentation for all other iPlanet and Netscape servers and technologies can be found at <http://docs.sun.com/db/prod/sunone>.

Download Center

Links to download any of Sun's Sun ONE/iPlanet software are at
<http://www.sun.com/software/download/>.

Sun ONE Technical Support

Technical Support can be contacted through
<http://www.sun.com/service/support/software/iplanet/index.html>.

Professional Services Information

Professional Service can be contacted through
<http://www.sun.com/service/sunps/iplanet/>.

Sun Enterprise Services for Solaris Patches And Support

Solaris patches and support can be obtained through
<http://www.sun.com/service/>

Developer Information

Information on Identity Server, LDAP, the Sun ONE Directory Server, and associated technologies can also be found at

<http://developer.iplanet.com/tech/directory/>

Documentation Comments

Sun Microsystems and the documentation writers of Identity Server are interested in improving the documentation and welcome any comments and suggestions.

Please email these comments to docfeedback@sun.com.

Introduction

The Sun™ One Identity Server *Programmer's Guide* describes the programmatic and back-end aspects of Identity Server. It includes instructions on how to augment the application with new services using the eXtensible Markup Language (XML) files for configuration, the public Java™ APIs for integration and the Java Server Pages (JSP) for customization. This guide also includes instructions on how to customize an Identity Server application for use by a specific organization. This introductory chapter contains the following sections:

- Identity Server Overview
- Extending Identity Server
- Identity Server File System

Identity Server Overview

An *identity* is a Lightweight Directory Access Protocol (LDAP) representation of a user or an object. The Sun ONE Identity Server integrates identity management with the capability to create and enforce authentication processes and access policies. These capabilities enable organizations to deploy a comprehensive system that helps to secure and protect their enterprise assets as well as deliver their web-based applications. Towards this end, Identity Server contains components and application management utilities or *services*.

Data Management Components

Identity Server provides the following components to simplify the administration of identities and the management of data:

- **Service Management**—provides a solution for customizing and registering application management parameters. Configuration parameters or *attributes* are grouped into *services* which can then be managed using the Identity Server. The solution includes an Document Type Definition (DTD) that defines the structure for creating an XML service file as well as Java interfaces that are used to integrate and manage the service.
- **Identity Management**—provides a solution for managing identities. It includes Java interfaces for creating, modifying and removing identity-related objects (users, roles, groups, people containers, organizations, organizational units and sub-organizations) as well as an XML template that defines an object's LDAP attributes.
- **Policy Management**—provides a solution for defining and retrieving access privileges to an enterprise's secure resources. It includes Java interfaces that applications can use to obtain an identity's policy settings. The applications then use these settings to evaluate policy decisions when a user requests action on a secure resource.
- **Federation Management**—provides a solution for defining authentication domains, service providers and identity providers in order to give users the functionality of federation. This module integrates the Liberty Alliance Project's Version 1.0 specifications.
- **Session Management**—provides a solution for viewing user session information and managing same. It keeps track of various session times as well as allowing the administrator to terminate a session.
- **Sun ONE Directory Server**—provides the storage facility in an Identity Server deployment. It holds all configured identity data as well as access policies. The majority of the data is stored in the Directory Server using LDAP; certain of it is stored as XML.
- **Sun ONE Web Server**—provides the container in which the Identity Server is run. Because Identity Server uses Java and JSP technologies, the Web Server is needed to implement the Servlet API.

Application Management Services

When Identity Server is installed, a number of application management utilities or *services* are installed. A *service* is a grouping of an application's configuration parameters (also called *attributes*). The attributes can be randomly grouped together for easy management or specifically grouped together for one purpose. Additional information on services can be found in Chapter 6, "Service Management" and the *Sun ONE Identity Server Administration Guide*. The current installed services are:

- **Administration**—provides properties for the configuration of the Identity Server application and attributes for the customization of the application specific to each organization.
- **Authentication**—provides an interface for gathering user credentials and issuing single sign-on (SSO) session tokens that integrates HTML, XML and `http/https`. It contains an SDK for writing plug-ins for different authentication servers. It also contains a SSO SDK for integrating token validation and authentication credential storage into the plug-in.
- **Client Detection**—provides an interface and configurable properties for detecting the client type of the browser attempting to access Identity Server.
- **Logging**—provides Java interfaces for audit trail and logging ability. Both file-based logs and logs stored in a relational database are supported. On Solaris, Identity Server uses `var/opt/SUNWam` as the default directory for logs and debug files. On Windows® 2000, `<identity_server_root>` is the default.
- **Naming**—provides configurable attributes that allow client browsers to find the correct URL for all services in a deployment that is running more than one Identity Server. This ensures that the URL returned for the service is the one for the host that the user session was created on.
- **Platform**—provides configurable attributes for the entire Identity Server deployment.
- **Policy Configuration**—provides properties for the configuration of the Identity Server application and attributes to configure the Policy Service specific to each organization.
- **Security Assertion Markup Language (SAML)**—provides an interface integrating SAML, Simple Object Access Protocol (SOAP) and `https` for sending and receiving security information. This service encrypts data passed between different security entities. APIs are provided to this end.
- **Session**—provides attributes to configure the session properties inherited by all identities for each organization.

- User—provides attributes to configure the user properties inherited by all identities for each organization.
- Security Service—provides a certificate authority service for users and components. For users, it issues and revokes certificates. For components, it issues user certificates for agents or server certificates for Sun ONE servers.

In addition to its configured services, Identity Server provides a graphical user interface that allows the application user to manage identity objects, services and policy information via a web browser. This Identity Server console is built using the Sun ONE Application Framework and can be called by all users, from top level administrator to end users. (A *policy* defines the specific access privileges for each user.) The console can be customized for each configured organization by modifying and integrating a set of JSP and related files. For data backup and restoration, schema management and metadata integration, Identity Server offers command-line executables. Information on both of these topics can be found in Chapter 2, “The Identity Server Console.”

Managing Access

Identity Server can manage access to its protected resources in either of two ways: an administrator can authenticate and access Identity Server via a web browser or, a Java application can access Identity Server directly, requesting user authentication information through the use of Identity Server APIs.

Web Access

When a user requests access to a secure application or page using a web browser, they must first be authenticated. The request is directed to the Authentication Service which determines the type of authentication process to initiate based on the method associated with the requestor's profile. For instance, if the user's profile is associated with LDAP authentication, the Authentication Service would send an HTML form to their web browser asking for an LDAP user name and password. (More complex types of authentication might include requesting information for multiple authentication types.) Having obtained the user's credentials, the Authentication Service calls the respective provider to perform the authentication. (The provider in the LDAP example would be the Directory Server.) Once verified, the service calls the SSO API to generate a Single Sign-On (SSO) token which holds the user's identity and then generate a token ID, a random string associated with the SSO token. This complete token is sent back to the requesting browser in the form of a cookie. The authentication component then directs the user to the requested secure application or page. Additional information on the Authentication Service can be found in Chapter 3, “Authentication Service.”

NOTE Web access might also include an additional security measure to evaluate a user's access privileges; this includes web agents. For more information, see the Sun ONE Identity Server Policy Agent documentation.

Application Access

Java applications can access Identity Server directly, requesting user configuration information using the Identity Server APIs. For example, a mail service might store its users' mailbox size information in Identity Server and the Identity Server SDK can be used to retrieve this information. To process this request, the system running the application must have the Identity Server SDK installed. There must also be at least one instance of the Sun ONE Web Server running the Identity Server. Additional information on the Identity Server SDK can be found in Chapter 5, "Identity Management."

NOTE Some services can also be accessed by C applications. Please see Chapter 3, "Authentication Service" and Chapter 7, "Policy Service" for further information on this functionality.

Extending Identity Server

One of the architectural goals of Identity Server is to provide an extensible interface. This extensible interface is defined by the following functions:

1. Custom services can be defined for the deployment using XML.
2. Console templates can be modified and/or customized for each organization using Java Server Pages (JSP).
3. Default services can be implemented using a set of Java APIs.

Service Definition With XML

As mentioned in the "Identity Server Overview," on page 17, Identity Server contains a number of application management utilities or *services*. A service is a grouping of configuration parameters defined under one name. These *attributes* can be randomly grouped together for easy management or specifically grouped together for one purpose. (Identity Server ships with a number of internal services of the latter type. More information on these internal services can be found in the *Sun ONE Identity Server Administration Guide*.) All Identity Server services are

written using the XML. Administrators or service developers can modify the internal XML service files or configure the custom XML service files based on their need. More information on services and how they are integrated into the Identity Server deployment can be found in Chapter 6, “Service Management.”

NOTE Identity Server services manage attribute values that are stored in Sun ONE Directory Server. They do not implement the behavior of the attributes or dynamically generate code to interpret them. It is up to an external application to interpret or utilize these values.

Identity Server Console Customization

The Identity Server console is used for managing and monitoring identities, services and protected resources throughout the Identity Server deployment. It’s framework uses XML files, JSP templates and Cascading Style Sheets (CSS) to control the look and feel of the screens that a user accesses. These files can be duplicated and modified to make changes to the design for each registered organization; for instance, an organization’s logo can be added in place of the Sun logo. The entire template can also be replaced with an organization’s custom HTML page. Additional information on customizing the Identity Server console can be found in Chapter 2, “The Identity Server Console.”

Java Packages

The Identity Server packages provide public interfaces to implement the behavior of Identity Server’s default or customized services. The packages are:

Identity Server SDK

Identity Server provides the framework to create and manage users, roles, groups, people containers, organizations, organization units, and sub-organizations. It also includes the functionality to create and modify service templates. This API is the core of the identity, service and policy management modules and provides Java classes that can be used to customize them. The API package name is `com.iplanet.am.sdk`.

Service Management SDK

The Identity Server provides Java APIs for service management. These interfaces can be used by developers to register services and applications, and manage their configuration data. The API package name is `com.sun.identity.sm`.

Utility API

This API provides a number of Java classes that can be used to manage system resources. This includes, among others, thread management and debug data formatting. The API package name is `com.iplanet.am.util`.

Logging API

The Logging service records, among other things, access approvals, access denials and user activity. The Logging API can be used to enable other Java applications to call it. The API package names begin with `com.sun.identity.log`.

Client Detection API

Identity Server can detect the type of client that is attempting to access its resources and respond with the appropriately formatted pages based on its type. The API package used for this purpose is `com.iplanet.services.cdm`.

SSO API

Identity Server provides Java interfaces for validating and managing the single sign-on (SSO) tokens, and for maintaining the user's authentication credentials. All applications wishing to participate in the SSO solution can use this API. The API package name is `com.iplanet.sso`.

Java SDK For Policy

The Policy API can be used to evaluate and manage Identity Server policies as well as provide additional functionality for the Policy Service. The API package names begin with `com.sun.identity.policy`.

SAML SDK

Identity Server uses the SAML API to exchange acts of authentication, authorization decisions and attribute information. The API package names begin with `com.sun.identity.saml`.

Federation Management API

Identity Server uses the Federation Management API to add functionality based on the Liberty Alliance Project specifications. The API package name is `com.sun.liberty`.

NOTE The complete set of Javadocs can be accessed from any web browser by copying the `<identity_server_root>/SUNWam/docs/` directory into `<identity_server_root>/SUNWam/public_html/` and using `http://<Identity Server_host.domain>:<port>/docs/index.html`

Identity Server File System

Identity Server installs its packages and files in a directory named `SUNWam`. The file system layout for a Solaris installation is as follows:

`<identity_server_root>/SUNWam/`

- `bin/` contains Identity Server executables such as `amserver` and `amadmin` in addition to LDAP command line applications.
- `capi/` contains the C API for integrating C applications with the Identity Server.
- `config/` contains Identity Server configuration files as well as the XML files which define Identity Server services.
- `docs/` contains Identity Server documentation.
- `dtd/` contains the defining XML DTDs used by Identity Server applications and services.
- `java/` contains the Java Development Kit.
- `ldaplib/` contains files needed to run the `ldapmodify` application.
- `ldif` contains the Identity Server LDAP schema.
- `lib/` contains Identity Server jar files as well as platform specific C libraries.
- `locale/` contains the internationalization resource files.
- `migration/` contains tools for Sun ONE Directory Server data migration from earlier versions to version 5.1.
- `public_html/` contains pre-authentication HTML files used by Identity Server. This directory is also configured as the root of the Sun ONE Web Server therefore, copying the `docs` directory into it will allow accessibility to the product documentation through a web browser on a non-Solaris machine.

- `samples/` contains sample Java programs on how to use the Identity Server APIs.
- `servers/` contains the files and documentation for the deployed Sun ONE Web Server.
- `web-apps/` contains the WAR-based deployments and their associated files: *Services* (authentication, policy management, identity management, SSO, service management, etc.) and *Applications* (Identity Server console).

The Identity Server Console

The Identity Server console is a web interface for managing and monitoring identities, services and resources throughout the Identity Server deployment. It is built with Sun™ One Application Framework, a Java™ 2 Enterprise Edition (J2EE) web application framework used to help developers build functional web applications. XML files, Java Server Pages (JSP) and Cascading Style Sheets (CSS) are used to define the look of the web pages. This chapter explains the console, its pluggable architecture and how to customize it. It contains the following sections:

- Overview
- Customizing The Console
- Customizing The User Profile View
- Miscellaneous Customizations
- Console Sample

Overview

The Identity Server console is a browser-based interface for creating, managing and monitoring identities, web services and enforcement policies throughout an Identity Server deployment. It allows administrators with different levels of access restrictions to, among other things, create organizations, add (or remove) users to (or from) those organizations, and establish enforcement policies that protect and limit access to the organization's resources. Towards this end, the console ships with four modules: Identity Management (including user and policy management), Service Configuration, Current Sessions and Federation Management. Customization of these modules and the Identity Server console can be achieved, in varying degrees, by modifying the JSP and XML files of the graphical user interface (GUI) as well as extending the JATO ViewBeans.

NOTE The client web browser must support JavaScript, v. 1.2 and Cascading Style Sheets.

Console Interface

The console is divided into three frames as pictured in Figure 2-1: Header, Navigation and Data. The Header frame displays branding information as well as the Full Name of the currently logged-in user. (The Full Name refers to the value of the `cn` attribute in the user's LDAP profile.) The Common Name, which may or may not be the same as the user ID, also links to the user's profile. The Header frame also contains a set of tabs to allow the user to switch between the management modules, hyperlinks to the Identity Server Help, a Search function for searching the directory information tree (DIT) and a Logout link. Actions performed in the Header frame affect the other two frames. The Navigation frame displays the object hierarchy of the module chosen. The Data frame displays the attributes of the object selected from the hierarchy in the Navigation frame.

Plug-In Modules

An external application may also be plugged-in to the Identity Server console as a module, gaining complete control of the Navigation and Data frames for its specific functionality. In this case, a tab with the name of the custom application needs to be added to the Header frame. An XML definition of the module name, class, and `isbn` filename is used to track registered views, and route request traffic to them. The application developer would create the JSPs for both left and right frames, and all view beans, and models associated with them.

Architecture

When the Identity Server console receives an http(s) request from a web browser, it first determines whether the requestor has been authenticated. If there is no valid single sign-on (SSO) token, the request is redirected to the Authentication Service. When the user has successfully authenticated to the Identity Server, the Authentication Service redirects the original request back to the console. The console will be dynamically built for the authenticated user based on the access assigned to them.

Figure 2-1 The Identity Server Console

Customizing The Console

The Identity Server console uses JSP, CSS and XML files to define the look and feel of the HTML pages used to generate its frames. An administrator can customize the console by changing the tags in these files accordingly. All of these files can be found in the `<identity_server_root>/SUNWam/web-apps/applications/console` directory. The files in this directory provide a default interface. To customize the console for a specific organization, this console directory could be copied and renamed to reflect the name of the organization (or any value). It would be placed at the same directory level as the default and the files within it would then be modified as needed. For example, the customized console files for the organization `dc=example, dc=com` could be found in the `<identity_server_root>/SUNWam/web-apps/applications/example` directory. (The console can also be modified by simply replacing the default images in `<identity_server_root>/SUNWam/web-apps/applications/console/images`, with new, similarly named images.)

Default Console Directory

The look and feel of the console is defined by both CSS and JSP. These files are contained in the default console directory, located in `<identity_server_root>/SUNWam/web-apps/applications/`. When copied and renamed for a specific organization, the files can be modified to reflect the organization's standards. Following is the default structure of the directory:

- `auth` contains JSPs for the Authentication Service.
- `base` contains JSPs that are not service-specific.
- `css` contains `adminstyle.css` which defines styles for the console.
- `dss` contains JSPs related to the Security Service.
- `federation` contains JSPs related to the Federation Management module.
- `html` contains miscellaneous HTML files.
- `images` contains images referenced by the JSP.
- `js` contains JavaScript files.
- `policy` contains JSPs related to the Policy Service.
- `service` contains JSPs related to the Service Management module.
- `session` contains JSPs related to the Session Management module.
- `user` contains JSPs related to identity management. This includes views for creating and displaying objects.

CAUTION JSP are HTML files that include references to tag library descriptor files (`.tld`) and Java classes which, when generated, form a web page. New tags can not be introduced into the JSP although tags can be removed.

Creating Custom Organization Files

To customize the Identity Server console for a specific organization, the default console directory should first be copied and renamed (ideally to reflect the name of the organization). The copy is placed on the same level as the default directory and the files modified as needed.

1. Change to the directory where the default templates are stored.

```
cd <identity_server_root>/SUNWam/web-apps/applications
```

2. Create a new directory at that level.

The directory name could be the name of the organization.

3. Copy all the properties and JSP files from the `console` directory into the new directory.

In the `<identity_server_root>/SUNWam/web-apps/applications` directory there is already a `console` folder that contains the properties and JSP files that should be copied into the organization's new directory. Ensure that any image files are also copied into this directory.

4. Customize the files in the new directory.

Modify any of the files in the new directory to reflect the organization.

5. Modify the `AMBase.jsp` file.

This file can be found in `<identity_server_root>/SUNWam/web-apps/applications/console/base`. The line `String console = "../console"`; needs to be changed to `String console = "../<new_directory_name>"`; The `String consoleImages` tag also needs to be changed to reflect a new image directory, if applicable. The contents of the file are copied in Code Example 2-1.

Code Example 2-1 The AMBase.jsp File

```
<!--
  Copyright © 2002 Sun Microsystems, Inc. All rights reserved.
  Use is subject to license terms.
-->

<% String console = "../console";
    String consoleUrl = console + "/";
    String consoleImages = consoleUrl + "images";
%>
```

6. Modify the JSP Directory Name attribute in the particular organization's Administration Service.

This attribute will point the Authentication Service to the directory which contains the organization's customized console interface.

Precompiling JSP Files

The JSP files used for the console interface need to be compiled. By default, the files are compiled automatically when the first user accesses the console. Because of this, the first user must wait before they are directed to the interface. The system administrator can precompile the JSPs by running the following command:

```
<identity_server_root>/SUNWam/servers/bin/https/bin/jspc -webapp
<identity_server_root>/SUNWam/web-apps/applications
```

Customizing The User Profile View

The Identity Server console creates a default User profile view based on information defined in `amUser.xml`. (Attributes defined as User attributes in specific XML service files can also be displayed.) A customized User profile view with functionality more appropriate to the organization's environment can be defined by creating a new `ViewBean` and/or a new JSP.

NOTE A `ViewBean` is a `JavaBean` written specifically for rendering display. In Identity Server, each identity has its own `Profile ViewBean`. For example, the user profile has the `UMUserProfileViewBean`.

To illustrate, an organization might want User profile attributes to be formatted differently than the default vertical listing provided. Another customization option would be to break up complex attributes into smaller ones. (Currently, the server names are listed as `<protocol>://<Identity Server_host.domain>:<port>`. Instead, the display can be customized with three fields:

```
<protocol_chooser_field>://<server_text_field>:<port_text_field>.
```

For a third option, JavaScript can be added to the `ViewBean` to dynamically update attribute values based on other defined input. The custom JSP should be placed in the `<identity_server_root>/SUNWam/web-apps/applications/console/user` directory and the `ViewBean` placed in the classpath

```
com.iplanet.am.console.user.
```

The value of the attribute `User Profile Display Class in the Administration Service` (`iplanet-am-admin-console-user-profile-class` in the `amAdminConsole.xml` service file) would then be changed to the name of the newly created `ViewBean`. The default value of this attribute is `com.iplanet.am.console.user.UMUserProfileViewBean`.

Miscellaneous Customizations

Included in this section are instructions for several customizations that can be configured for the Identity Server console.

Changing Default Attribute Display

The console auto-generates pages based on the definition of a service's attributes in an XML service definition file. As documented in "The sms.dtd Structure," on page 126, each service attribute is defined with XML attributes `type` and `syntax`. Type specifies the kind of value the attribute will take; syntax defines the format of the value. These syntax can be changed to alter the console display. Table 2-1 is a listing of the different values that can be used with these XML attributes.

Table 2-1 Attribute Display Elements

Type	Syntax	UI Element
Single	boolean	checkbox
		radio button
	string	text field
		link
		button
		text field
list	password	scrolling text field
		paragraph
	string	value list choices
single_choice	string	value list choices
		pull-down menu choices
		radio button choices
multiple_choice	string	choice list

For example, an attribute of the `single_choice` type displays its values as a drop down list which allows only one value to be selected. This list can also be presented as a set of radio buttons which allows only one value to be selected. Code Example 2-2 specifies the `uitype` for the attribute named `test-attribute` as radio button choices. Deleting `uitype` from the attribute schema and the default torpedoing menu is displayed.

Code Example 2-2 `uitype` XML Attribute Sample

```
<AttributeSchema name="test-attribute"
  type="single_choice"
  syntax="string"
  any="display"
  uitype="radio"
  i18nKey="d105">
  <ChoiceValues>
<ChoiceValue i18nKey="u200">Daily</ChoiceValue>
<ChoiceValue i18nKey="u201">Weekly</ChoiceValue>
<ChoiceValue i18nKey="u202">Monthly</ChoiceValue>
  </ChoiceValues>
  <DefaultValues>
    <Value>Active</Value>
  </DefaultValues>
</AttributeSchema>
```

Localizing The Console

All textual resource strings used in the console can be found in the `<identity_server_root>/SUNWam/locale/amAdminModuleMsgs.properties` file. The default language is English (`en_US`). Modifying this file with messages in a foreign language will localize the console.

Customizing Background Colors

All background colors are configurable using the Identity Server style sheet `adminstyle.css` located in the `<identity_server_root>/SUNWam/web-apps/applications/console/css` directory. For instance, to change the background color for the navigation frame, modify the `BODY.navFrame` tag. It takes either a text value for standard colors (blue, green, red, yellow, etc.) or a hexadecimal value (`#ff0000`, `#aadd22`, etc.). Replacing the default with another value will change the background color of the navigation frame after the console is closed and reopened. Code Example 2-3 illustrates this concept.

Code Example 2-3 BODY.navFrame Portion of adminstyle.css

```
BODY.navFrame {
    color: black;
    background: #ffffff;
}
```

Labelling The Module Tab

The attribute View Menu Entries in the Administration Service (iplanet-am-admin-console-view-menu in the amAdminConsole.xml service file) points to the ViewBeans that carry the label information used on the four module tabs plugged into the Identity Server console. The label information itself is found in the console properties file amAdminModuleMsgs.properties, located in <identity_server_root>/SUNWam/locale/. To modify the label for each tab, find the key and value pair in amAdminModuleMsgs.properties, change the value and restart the Identity Server. The labels are identified in this file as:

Code Example 2-4 Module Tab Key And Value Pairs

```
module101_identity=Identity Management
module102_service=Service Configuration
module103_session=Current Sessions
module104_federation=Federation Management
```

Displaying Container Objects

To get container objects to display in the Identity Server console, the following attributes need to be enabled in the Administration Service.

- Show People Containers
- Display Containers In Menu
- Show Group Containers

Display Containers In Menu must be enabled in order for the console to show either people containers or group containers.

Console Sample

Sample files have been included to help understand how the Identity Server console can be customized. They help to explain the Java™ 2 Enterprise Edition (J2EE) web application framework used. In addition, Java classes are extended from the console APIs and new JSP files are created. Existing XML and properties files are also used. These files are located in `<identity_server_root>/SUNWam/samples/console`. Open the `README` file in this directory for instructions on how to run the sample.

Authentication Service

The Authentication Service is the point of entry for Sun™ One Identity Server. A user must pass an authentication process before being allowed access to the console or any resource that is secured using the Identity Server. As well, an application or service must pass the authentication process before it can be considered a trusted source by Identity Server. This chapter explains the process, its pluggable architecture, and the authentication APIs. It contains the following sections:

- Overview
- The Authentication User Interface
- Default Authentication Modules
- Custom Authentication Modules
- Application Authentication
- Authentication SPI
- URL Parameters
- C Programs and Authentication
- Authentication Samples

Overview

Identity Server provides secure access to web-based (and non-web-based) applications and the data that they store. Gaining access to either of these resources requires that the user or application be given permission by the Authentication Service. When the requestor tries to access a protected resource, it is directed to submit credentials to one (or more) of several authentication modules; for instance,

the LDAP module generally requires authentication with the user's Directory Server ID and password while the SafeWord™ module requires authentication to the ACE/Server. The Authentication Service then acts as an *authority*, granting or denying access upon completion of the required process. If access is granted, a session token is created and assigned to the requestor who is then, by default, directed to the Identity Server console. When the console finds and validates the token, the appropriate page is displayed based on the identity's request and their parent organization. If there is no valid session token, the console denies access.

NOTE Upon successful authentication, the requestor is directed to the Identity Server console unless they have initially requested an external protected resource, have a `goto` parameter in their URL or have an authentication method which contains values for redirect URLs based on success or failure.

Accessing The Authentication Service

There are a number of ways to authenticate to the Identity Server in order to allow access to protected resources. Java™ applications can use the authentication API while C applications can open a connection using a web browser. End users also access the Authentication Service using a web browser. In addition, there is a `remote-auth.dtd` that defines the XML structure used to format the XML request messages.

Authentication For Java Applications

External Java applications can authenticate to the Identity Server or any of its protected resources (including the Sun ONE Directory Server data store) using the Authentication API for Java. This API provides interfaces to initiate the authentication process and communicate authentication credentials to the Authentication Service. The API is defined in a Java package called `com.sun.identity.authentication`. Developers incorporate the classes and methods from this package into their Java applications to allow communication with the Authentication Service. The application's Java request is first converted to an XML message format and passed to the Identity Server over `http` or `https`. (XML messages are structured according to the `remote-auth.dtd` which is discussed further in "The remote-auth.dtd Structure," on page 63.) The XML message is then converted back into Java which is able to be interpreted by the Authentication Service. These API are discussed further in "Authentication API For Java Applications," on page 62.

Authentication For C Applications

Identity Server includes the resources for C applications to authenticate to the Identity Server. They must first open a connection to the Identity Server using the URL, `http://<Identity_Server_host.domain>/<service_deploy_uri>/authservice`. Then the application sends a request in XML message format and passes it to the Identity Server over `http` or `https`. The XML message is then processed by the Authentication Service. After passing the authentication process, a validated session token will be sent to the C application. XML messages are structured according to the `remote-auth.dtd` which is discussed further in “The remote-auth.dtd Structure,” on page 63.

Authentication For Users Using A Web Browser

A user with a web browser can authenticate to Identity Server using the authentication user interface. The URI for this web-based interface is `http://<Identity_Server_host.domain>/<service_deploy_uri>/UI/Login`. Once authenticated, the user gains access to the Identity Server console and based on their privileges:

- Administrators can access the identity administration portion of the console in order to manage authentication data.
- End users can modify authentication data in their own user profiles.

Authenticating The Request

The authentication framework has the job of validating the authentication request. The framework integrates the standard Java Authentication and Authorization Service (JAAS) API with proprietary APIs that support Identity Server-specific features. Identity Server adds features on top of the JAAS framework including account locking, web-based UI, and an XML over HTTP interface that allows authentication APIs to work on a remote machine. Because of this architecture, any custom JAAS authentication modules will also work within the Authentication Service.

NOTE

This guide is not intended to document the JAAS framework. For more information on these APIs, see the *Java Authentication And Authorization Service Developer's Guide* at <http://java.sun.com/security/jaas/doc/api.html>. Additional information can be found at <http://java.sun.com/products/jaas/>.

Requesting Authentication

The authentication framework starts the authentication process by reading a Java request. When a request is received, the framework creates a session for the requestor and begins the authentication process by calling the Authentication Service Provider Interface (SPI) to allow interaction between the requestor and the login module for credential gathering. The SPI references the authentication module configuration file to determine the login requirements for the specific authentication module. (More information on the authentication module configuration file can be found in “The Auth_Module_Properties.dtd Structure,” on page 57.)

Using The Authentication SPI

Identity Server provides the Authentication SPI to invoke a specific authentication module. The SPI is defined in the `com.sun.identity.authentication.spi` package. All configured modules, custom and default, can extend the `AMLoginModule` and implement the `process`, `init` and `getPrincipal` methods. These methods access Identity Server objects for post-processing of user profiles, service templates, and attributes. More information on the Authentication SPI can be found in “Authentication SPI,” on page 71.

NOTE The Authentication API are also able to invoke authentication modules written using the pure JAAS API.

Client Detection

Within the framework, the first step in authenticating a user is to identify the client type making the HTTP request. The Authentication Service uses the URL information to retrieve the browser type’s characteristics. Based on these characteristics, the correct authentication pages are sent back to the client browser; for example, HTML pages. Once the user is validated, the client type is added to the session token where it can be retrieved by other services. More information on client detection can be found in Chapter 11, “Client Detection.”

NOTE Although Identity Server has the capability to support multiple clients (including wireless), currently it only defines client data for HTML clients.

Miscellaneous Features

The Authentication Service includes a number of new features. Account locking and Fully Qualified Domain Name (FQDN) Mapping are explained below.

Account Locking

Account locking prohibits the user from authenticating after a fixed number of consecutive unsuccessful login attempts. Identity Server allows two additional login attempts after the user is warned about an impending lockout. The locking can be initiated by changing the status of an attribute in LDAP or by memory locking which occurs when the time-out duration is greater than 0. Email notifications are sent to administrators regarding any account lockouts. (Account locking activities are also logged.) For more information on the account locking attributes, see the *Sun ONE Identity Server Administration Guide*.

NOTE Only modules that throw an Invalid Password Exception can leverage the Account Locking feature.

FQDN Mapping

FQDN Mapping enables the Authentication Service to take corrective action in the case where a user may have typed in an incorrect URL (such as specifying a partial host name or using an IP address to access protected resources). FQDN Mapping is enabled by modifying the `com.sun.identity.server.fqdnMap` attribute in the `AMConfig.properties` file. The format for specifying this property is:

```
com.sun.identity.server.fqdnMap[<invalid-name>]=<valid-name>
```

The value `<invalid-name>` would be a possible invalid FQDN host name that may be typed by the user, and `<valid-name>` would be the actual host name to which the filter will redirect the user. Any number of mappings can be specified in `AMConfig.properties` (as illustrated in Code Example 3-1) as long as they conform to the stated requirements.

:
Code Example 3-1 FQDN Mapping Attribute In `AMConfig.properties`

```
com.sun.identity.server.fqdnMap[isserver]=isserver.mydomain.com
com.sun.identity.server.fqdnMap[isserver.mydomain]=isserver.mydo
main.com
com.sun.identity.server.fqdnMap[<IP
address>]=isserver.mydomain.com
```

Possible Uses For FQDN Mapping

This property can be used for creating a mapping for more than one host name which may be the case if applications hosted on this server are accessible by more than one host name. This property can also be used to configure Identity Server to not take corrective action for certain URLs. For example, if no redirect is required for users who access applications by using an IP address, this feature can be implemented by specifying a map entry such as

```
com.sun.identity.server.fqdnMap[<IP>]=<IP>.
```

NOTE If more than one mapping is defined, ensure that there are no overlapping values in the invalid FQDN name. Failing to do so may result in the application becoming inaccessible.

The Authentication User Interface

The Authentication Service has a separate user interface from the Identity Server console. It provides the web-based interface for all authentication modules installed in the Identity Server deployment. The interface provides a dynamic and customizable means for gathering user credentials for authentication by presenting the web-based login requirement pages to a user requesting access. Figure 3-1 is a screenshot of the default user interface for LDAP authentication.

Figure 3-1 Default LDAP Authentication User Interface



Like the Identity Server console, the Authentication Service interface is built with J2EE Assisted Take-Off (JATO), a Java 2 Enterprise Edition (J2EE) application framework used to help developers build functional web applications. It uses the Authentication API to authenticate users to their specified authentication module. (This component can be deployed on non-Identity Server machines.)

The Authentication Service interface uses Java Server Pages (JSP) and XML files to convey graphical-based user information such as login, logout and time-out information as well as error messages. The JSP templates define the layout of the pages and are located in `<identity_server_root>/SUNWam/web-apps/services/config/auth/default`. The XML templates are the authentication module configuration files, discussed in “Configuring Module Credential Requirements,” on page 57. They are also located in `<identity_server_root>/SUNWam/web-apps/services/config/auth/default`. Both of these types of files can be modified to customize the user experience at the following levels:

- Organization
- Locale
- Sub-organization
- Service or Application
- Client type

Customizing The Authentication Interface

The JSP templates and module configuration properties files can be modified to reflect an organization’s branding or to add organization-specific functionality. For example, if there are three organizations in the Identity Server deployment—`org1`, `org2`, `org3`—and `org1` has customized templates, these templates will be located in `<identity_server_root>/SUNWam/web-apps/services/config/auth/org1`. Any organization that does not have its own directory of modified templates will continue to use the default set of files located in `<identity_server_root>/SUNWam/web-apps/services/config/auth/default`. In the example, both directories contain a full set of templates. `Org1` would use the set in the `org1` directory while `org2` and `org3` would use the set in the default directory.

Creating A Directory

1. Go to the directory where the JSP templates are stored.

```
cd <identity_server_root>/SUNWam/web-apps/services/config/auth/
```

2. Create a directory using the appropriate path based on the level of customization.

The directory name should match the name that appears in the Identity Server console. Table 3-1 lists the different path names for each level based on its customization of either a default or configured organization.

Table 3-1 Directory Locations Based On Customization Level

Level	Default Organization Location	Configured Organization Location
Organization	<identity_server_root>/SUNWam/web-apps/services/config/auth/default	<identity_server_root>/SUNWam/web-apps/services/config/auth/org1
Sub-organization	<identity_server_root>/SUNWam/web-apps/services/config/auth/default/sub-org1	<identity_server_root>/SUNWam/web-apps/services/config/auth/org1/sub-org1
Locale	<identity_server_root>/SUNWam/web-apps/services/config/auth/default_locale	<identity_server_root>/SUNWam/web-apps/services/config/auth/org1_locale
Service or Application	<identity_server_root>/SUNWam/web-apps/services/config/auth/default/service1	<identity_server_root>/SUNWam/web-apps/services/config/auth/org1/service1
Client Type (HTML, WML, etc.)	<identity_server_root>/SUNWam/web-apps/services/config/auth/default/clienttype1	<identity_server_root>/SUNWam/web-apps/services/config/auth/org1/clienttype1

3. Copy all the module configuration properties files and JSP templates into the new directory.

All of the files in the <identity_server_root>/SUNWam/web-apps/services/config/auth/ directory need to be copied into the new directory.

4. Customize the files in the new organization directory.

Make sure that any image files (for example, JPEG or GIF files for logos) are copied into <identity_server_root>/SUNWam/web-apps/services/login_images. Any HTML specific tags can be modified to other desired HTML tags or any other client type specific tags.

NOTE All JSP background colors, page layouts, and fonts are configurable using the style sheets located in `<identity_server_root>/web-apps/services/css`.

JSP Templates

The following JSP templates can be found in `<identity_server_root>/SUNWam/web-apps/services/config/auth/default`. Strong HTML skills as well as an understanding of web servers can help in the modification of these files.

Table 3-2 List of Customizable JSP Templates

GUI Template	Purpose
<code>auth_error_template.jsp</code>	Informs the user when an internal authentication error has occurred.
<code>authException.jsp</code>	Informs the user that an error has occurred during authentication.
<code>account_expired.jsp</code>	Informs the user that their account has expired and should contact the system administrator.
<code>configuration.jsp</code>	Informs the user that there has been a configuration error.
<code>disclaimer.jsp</code>	This is a sample, customizable disclaimer page used in the Self-registration authentication module.
<code>Exception.jsp</code>	Informs the user that an error has occurred.
<code>invalid_domain.jsp</code>	Informs the user that there is no such domain.
<code>invalidPCookieUserid.jsp</code>	Informs the user that a persistent cookie user name does not exist in the persistent cookie domain.
<code>invalidPassword.jsp</code>	Informs the user that the password entered does not contain enough characters.
<code>Login.jsp</code>	This is a Login/Password template.
<code>login_denied.jsp</code>	Informs the user that no profile has been found in this domain.
<code>login_failed_template.jsp</code>	Informs the user that authentication has failed.
<code>Logout.jsp</code>	Informs the user that they have logged out.
<code>maxSessions.jsp</code>	Informs the user that the maximum sessions have been reached.

Table 3-2 List of Customizable JSP Templates (*Continued*)

GUI Template	Purpose
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 chosen authentication module has been denied.
module_template.jsp	A customizable module page.
noConfig.jsp	Informs the user that no configuration has been defined/found for them.
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	This page is 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	This page carries a link to a page that has been moved.
register.jsp	A 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	This page is called if a new user is registering with a user name that already exists.
userPasswordSame.jsp	Called if a new user is registering with a user name field and password field have the same value.

Table 3-2 List of Customizable JSP Templates (*Continued*)

GUI Template	Purpose
user_inactive.jsp	Informs the user that they are not active.
wrongPassword.jsp	Informs the user that the password entered is invalid.

Authentication Module Configuration Files

The authentication module configuration XML files are based on the `Auth_Module_Properties.dtd` and the syntax of this DTD should be followed when customizing these files. Modifying elements in these XML files will automatically and dynamically customize the authentication interface. More information on modifying this type of file can be found in “Configuring Module Credential Requirements,” on page 57.”

Default Authentication Modules

Identity Server is installed with a set of default authentication modules that can be used to communicate with proprietary technologies. These modules are explained below.

Core Authentication Service

The Core authentication service is the configuration base for all other proprietary authentication modules. It must be registered to an organization before any user can log in using one of the default authentication modules. It allows the Identity Server administrator to define default values for the Core authentication parameters. These values can then be picked up if no overriding value is set in the specific authentication module chosen. The default values for the Core service are defined in the `amAuth.xml` file and stored in the Directory Server after installation.

Proprietary Authentication Modules

Identity Server provides authentication modules able to communicate with proprietary technologies. The authentication modules currently included are:

- **Anonymous**—This module allows a user to log in without specifying a user name and/or password. Additionally, an Anonymous user can be created. Log in as Anonymous is then possible without a password. Anonymous connections are generally customized by the Identity Server administrator to have limited access to the server.
- **Certificate**—This module allows a user to log in through a personal digital certificate (PDC) that could optionally use the Online Certificate Status Protocol (OCSP) to determine the state of a certificate. Sun ONE Certificate Server (CS) can be installed as a validation authority. For more information on CS, see the documentation set located at http://docs.sun.com/db?p=coll/S1_slCertificateServer_47.
- **LDAP**—This module allows for authentication using LDAP bind, an operation which associates a user ID password with a particular LDAP entry.
- **Membership**—This module allows a new user to register themselves for authentication with a login and password as well as other fields such as first name, last name, etc.
- **NT**—This module allows for authentication using a Windows NT server.
- **RADIUS**—This module allows for authentication using an external Remote Authentication Dial-In User Service (RADIUS) server.
- **SafeWord**—This module allows for authentication using Secure Computing's servers and tokens.
- **Unix**—This Solaris only module allows for authentication using a user's UNIX identification and password.

The default authentication module used after installation is LDAP.

Assigning The Authentication Method

An authentication module can be assigned as the method to authenticate identities that belong to any of the following objects:

- organization
- role
- service
- user
- module

Once a module is defined as the authentication method for one of these objects, it can then be configured with URLs to redirect the identity on either a successful authentication or a failed authentication. When more than one URL is set, Identity Server has a defined hierarchy to pick the proper redirection URL.

NOTE For more information on how to define the authentication module using the Identity Server console, see the *Sun ONE Identity Server Administration Guide*.

Authentication By Organization

The authentication method for an organization is set by registering the Core Authentication service to the organization and configuring the Organization Authentication Configuration attribute. The authentication service(s) defined must also be registered to the organization.

NOTE Authentication by organization is the default definition. It can also be accessed by providing the “org” parameter in the authentication URL. More information on this function can be found in “URL Parameters,” on page 71.

Successful Organization-based Authentication Redirection URLs

Identity Server bases a successful organization-based authentication redirection determination by checking for a redirection URL in the following places:

1. A URL set by the module.
2. A URL set by a `goto` URL parameter.
3. A URL set for `clientType` in `iplanet-am-user-success-url` in user's `amUser.xml` service file.
4. A URL set for `clientType` in `iplanet-am-auth-login-success-url` in the user's role.
5. A URL set for `clientType` in `iplanet-am-auth-login-success-url` in the organization.
6. A URL set for `clientType` in `iplanet-am-auth-login-success-url` as the global default.
7. `iplanet-am-user-success-url` set at user entry
8. `iplanet-am-auth-login-success-url` set at user's role entry
9. `iplanet-am-auth-login-success-url` set at the org entry

10. `iplanet-am-auth-login-success-url` as the global default if not set in `org`

Failed Organization-based Authentication Redirection URLs

Identity Server bases a failed organization-based authentication redirection determination by checking for a URL in the following order:

1. A URL set by the module.
2. A URL set by a `gotoOnFail` parameter.
3. A URL set for `clientType` in `iplanet-am-user-failure-url` set at the user entry.
4. A URL set for `clientType` in `iplanet-am-auth-login-failure-url` set at the user's role entry.
5. A URL set for `clientType` in `iplanet-am-auth-login-failure-url` set at the organization entry.
6. `iplanet-am-auth-login-failure-url` as the global default if it is not set in the organization.
7. `iplanet-am-user-failure-url` set at the user entry.
8. `iplanet-am-auth-login-failure-url` set at the user's role entry.
9. `iplanet-am-auth-login-failure-url` set at the organization entry.
10. `iplanet-am-auth-login-failure-url` as the global default if it is not set in the organization.

Authentication By Role

The authentication method for a role is set by registering the Core Authentication service to the role and configuring the Organization Authentication Configuration attribute. The authentication service(s) defined must also be registered to the organization in which the role exists.

NOTE Authentication by role is accessed by providing the "role" parameter in the authentication URL. More information on this function can be found in "URL Parameters," on page 71.

Successful Role-based Authentication Redirection URLs

Identity Server bases a successful role-based authentication redirection determination by checking for a URL in the following order:

1. A URL set by the module.
2. A URL set by the `goto` parameter.
3. A URL matching `clientType` in `iplanet-am-user-success-url` set in the user entry.
4. A URL matching `clientType` in `iplanet-am-auth-login-success-url` set at the role to which the user authenticates.
5. A URL matching `clientType` in `iplanet-am-auth-login-success-url` set at the user's role.
6. A URL matching `clientType` in `iplanet-am-auth-login-success-url` set at the organization.
7. A URL matching `clientType` in `iplanet-am-auth-login-success-url` as the global default.
8. `iplanet-am-user-success-url` set in the user entry.
9. `iplanet-am-auth-login-success-url` set at the role to which the user authenticates.
10. `iplanet-am-auth-login-success-url` set at the user's role.
11. `iplanet-am-auth-login-success-url` set at the organization.
12. `iplanet-am-auth-login-success-url` from the global default.

Failed Role-based Authentication Redirection URLs

Identity Server bases a failed role-based authentication redirection determination by checking for a URL in the following order:

1. A URL set by the module.
2. A URL set by the `gotoOnFail` parameter.
3. A URL matching `clientType` in `iplanet-am-user-failure-url` set in the user entry.
4. A URL matching `clientType` in `iplanet-am-auth-login-failure-url` set at the role to which the user authenticates.
5. A URL matching `clientType` in `iplanet-am-auth-login-failure-url` set at the user's role.
6. A URL matching `clientType` in `iplanet-am-auth-login-failure-url` set at the organization.

7. A URL matching `clientType` in `iplanet-am-auth-login-failure-url` as the global default.
8. `iplanet-am-user-failure-url` set in the user entry.
9. `iplanet-am-auth-login-failure-url` set at the role to which the user authenticates.
10. `iplanet-am-auth-login-failure-url` set at the user's role.
11. `iplanet-am-auth-login-failure-url` set at the organization.
12. `iplanet-am-auth-login-failure-url` from the global default.

Authentication By Service

The authentication method for a service is set by registering the Core Authentication service and configuring the Organization Authentication Configuration attribute. The authentication service(s) defined must also be registered to the organization in which the role exists.

NOTE Authentication by service is accessed by providing the “service” parameter in the authentication URL. More information on this function can be found in “URL Parameters,” on page 71.

Successful Service-based Authentication Redirection URLs

Identity Server bases a successful service-based authentication redirection determination by checking for a URL in the following order:

1. A URL set by the module.
2. A URL set by the `goto` parameter.
3. A URL matching `clientType` in `iplanet-am-user-success-url` set in the user entry.
4. A URL matching `clientType` in `iplanet-am-auth-login-success-url` set at the service entry.
5. A URL matching `clientType` in `iplanet-am-auth-login-success-url` set at the user's role.
6. A URL matching `clientType` in `iplanet-am-auth-login-success-url` set at the organization.
7. A URL matching `clientType` in `iplanet-am-auth-login-success-url` set at the global default.

8. `iplanet-am-user-success-url` set at the user entry.
9. `iplanet-am-auth-login-success-url` set at the service entry.
10. `iplanet-am-auth-login-success-url` set at user's role entry.
11. `iplanet-am-auth-login-success-url` set at the organization entry.
12. `iplanet-am-auth-login-success-url` as the global default if not set in the organization.

Failed Service-based Authentication Redirection URLs

Identity Server bases a failed role-based authentication redirection

1. A URL set by the module.
2. A URL set by the `gotoOnFail` parameter.
3. A URL matching `clientType` in `iplanet-am-user-failure-url` set at the user entry.
4. A URL matching `clientType` in `iplanet-am-auth-login-failure-url` set at the service entry.
5. A URL matching `clientType` in `iplanet-am-auth-login-failure-url` set at the user's role entry.
6. A URL matching `clientType` in `iplanet-am-auth-login-failure-url` set at the organization entry.
7. A URL matching `clientType` in `iplanet-am-auth-login-failure-url` as the global default if not set in the organization.
8. `iplanet-am-user-failure-url` set at the user entry.
9. `iplanet-am-auth-login-failure-url` set at the service entry.
10. `iplanet-am-auth-login-failure-url` set at user's role entry.
11. `iplanet-am-auth-login-failure-url` set at the organization entry.
12. `iplanet-am-auth-login-failure-url` as the global default if not set in the organization.

Authentication By User

The authentication method for a user is set by registering the Core Authentication service to the user and configuring the Organization Authentication Configuration attribute. The authentication service(s) defined must also be registered to the organization in which the role exists. More information on how this is done can be found in the *Sun ONE Identity Server Administration Guide*.

NOTE Authentication by user is accessed by providing the “user” parameter in the authentication URL. More information on this function can be found in “URL Parameters,” on page 71.

Authentication By Authentication Level

The authentication method for a particular authentication level is set by the administrator. More information on how this is done can be found in the *Sun ONE Identity Server Administration Guide*.

NOTE Authentication by authentication level is accessed by providing the “authLevel” parameter in the authentication URL. More information on this function can be found in “URL Parameters,” on page 71.

Authentication By Module

The authentication method for a module is set by registering the Core Authentication service to the module and configuring the Organization Authentication Configuration attribute. More information on how this is done can be found in the *Sun ONE Identity Server Administration Guide*.

Custom Authentication Modules

The Authentication Service framework allows an organization to plug-in custom authentication modules. The following section discusses the steps necessary to create a custom authentication module.

NOTE To write a custom authentication module, knowledge of the JAAS API is necessary, especially for defining the module’s configuration properties.

Creating A New Authentication Module

1. Create an XML service file for the new authentication module.

The XML service file is written, and imported, into Identity Server in order to manage the authentication module's parameters using the Identity Server console. The name of the XML service file follows the format `amAuth<modulename>.xml` (for example, `amAuthSafeWord.xml` or `amAuthLDAP.xml`) and it is located in `<identity_server_root>/SUNWam/config/xml`. Information on writing this XML service file, based on the `sms.dtd`, can be found in Chapter 6, "Service Management."

2. Create a localization properties file for the new module.

The localization properties file defines language-specific screen text for the attribute names of the module. It is located in the directory `<identity_server_root>/SUNWam/locale/`. More information on this file and how to configure it can be found in "Configuring Localization Properties," on page 56.

3. Create an authentication module configuration file.

An authentication module configuration file specifies the credentials required from an identity (either user, service, or application) in order to authenticate to a specific authentication module. It is located in `<identity_server_root>/SUNWam/web-apps/services/config/auth/default`. The required credentials might include, but are certainly not limited to, user name and password. Information on how to create the file, based on the syntax of the `Auth_Module_Properties.dtd`, can be found in "Configuring Module Credential Requirements," on page 57.

4. Modify the `amAuth.xml` file.

The `amAuth.xml` defines the "parent" Core Authentication service. It is located in `<identity_server_root>/SUNWam/config/xml`. This file must be modified in order for the Authentication Service to recognize any custom authentication module. Information on `amAuth.xml` modifications can be found in "Modifying `amAuth.xml`," on page 61. Information on modifying XML service files in general can be found in Chapter 6, "Service Management."

5. Import the custom module's XML service file into the Authentication Service using the `amadmin` command line tool.

The syntax of the `amadmin` command line tool and instructions on how to use it can be found in the *Sun ONE Identity Server Administration Guide*.

NOTE Identity Server contains a sample exercise for creating a custom authentication module. For more information, see the “Login Module,” on page 77.

Configuring Localization Properties

A localization properties file specifies the localized screen text and messages that an administrator or user will see when directed to an Authentication Service’s attribute configuration page. Each authentication module has a corresponding localization properties file. The name of the file follows the format `amAuth<modulename>.properties`; for example, `amAuthLDAP.properties`. The default character set is ISO-8859-1 (English). Each authentication module has its own localization properties file, located in `<identity_server_root>/SUNWam/locale/`. This directory contains a sub-directory for each locale. The default English directory is `en_US`. For reference, Code Example 3-2 is a portion of the file `amAuthLDAP.properties`. (The file is in the `<identity_server_root>/SUNWam/locale/en_US` directory.) Following are the concepts behind the configuration of this file.

- The data following the equal (=) sign in each key/value pair (displayed in English here) would be translated to a specific language as necessary and copied into the corresponding locale directory. In Code Example 3-2, the alphanumeric keys (`a1`, `a2`, etc.) map to fields defined by the `i18nKey` attribute in the `amAuthLDAP.xml` service configuration file.
- The alphanumeric keys determine the order in which the fields are displayed in the Identity Server console. The keys are taken in the order of their ASCII characters (`a1` is followed by `a10`, followed by `a2`, followed by `b1`). For example, if an attribute needs to be displayed at the top of the service attribute page, the alphanumeric key should have a value of `a1`. The second attribute could then have a value of either `a10`, `a2` or `b1`, and so forth.

Code Example 3-2 Portion of `amAuthLDAP.properties`

```

...
PInvalid=Current Password Entered Is Invalid
PasswdSame=Password should not be same
PasswdMinChars=Password should be at least 8 characters
a1=Primary LDAP Server and Port
a2=Secondary LDAP Server and Port
a3=DN to Start User Search
a4=DN for Root User bind
a5=Password for Root User Bind

```

Code Example 3-2 Portion of `amAuthLDAP.properties` (Continued)

```

...
PInvalid=Current Password Entered Is Invalid
a6=User Naming Attribute
a7=User Entry Search Attribute
...

```

Configuring Module Credential Requirements

The authentication module configuration file specifies each authentication module's credential requirements by defining the screens that a user might see when directed to authenticate. Modifying elements in this XML file will automatically and dynamically customize the authentication interface. The name of this file follows the format `<modulename>.xml`; for example, `SafeWord.xml` or `LDAP.xml`. Each authentication module has its own configuration file, located in `<identity_server_root>/SUNWam/web-apps/services/config/auth/default`.

If there is more than one organization in the Identity Server deployment, each organization has its own authentication directory named `<identity_server_root>/SUNWam/web-apps/services/config/auth/org_name`. If an organization has more than one locale, the files are stored separately, in directories appended with a locale, as in `<identity_server_root>/SUNWam/web-apps/services/config/auth/org_name_locale`. Additionally, with service authentication, there might be an authentication directory corresponding to the service under the LDAP organization tree.

NOTE Customization of the authentication screens are only supported at the organization, sub-organization and service levels. In a search for the correct module configuration properties files, Identity Server first searches the `org_name_locale` directory, followed by the `org_name`, the `default_locale` and the `default` directories.

The `Auth_Module_Properties.dtd` Structure

The `Auth_Module_Properties.dtd` defines the structure for the XML-based authentication module configuration files. It provides definitions to initiate, construct and send the required authentication interface to the authentication framework. The DTD is located in `<identity_server_root>/SUNWam/dtd`. An explanation of the elements defined by the `Auth_Module_Properties.dtd` follows. Each element includes required and/or optional XML attributes.

ModuleProperties Element

ModuleProperties is the root element of the authentication module configuration file. It must contain at least one *Callbacks* sub-element. The required XML attributes of *ModuleProperties* are `moduleName` which takes a value equal to the name of the module and `version` which takes a value equal to the version number of the authentication module configuration file itself. Code Example 3-3 below is the `LDAP.xml` file that defines the screens for the LDAP authentication module. Note the *ModuleProperties* element on the first line of code.

Code Example 3-3 LDAP.xml

```

...
<ModuleProperties moduleName="LDAP" version="1.0" >
  <Callbacks length="2" order="1" timeout="120"
    header="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" >
    <PasswordCallback echoPassword="false" >
      <Prompt>#REPLACE#&lt;BR&gt; Old Password </Prompt>
    </PasswordCallback>
    <PasswordCallback echoPassword="false" >
      <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."
    error="true" >
  </Callbacks>
</ModuleProperties>

```

Callbacks Element

The *Callbacks* element is used to request 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. It can contain one or more of four sub-elements: *NameCallback*, *PasswordCallback*, *ChoiceCallback* or *ConfirmationCallback*. The required XML attributes of *Callbacks* are `length` which takes a value equal to the number of callback requests for the defined element and `order` which takes a value equal to the number this particular callback is in the sequence of callbacks. The `order` attribute value starts with the number '1'. The optional XML attributes are `timeout`, `template`, `image`, `header` and `error`.

- `timeout`—takes a value equal to the amount of time in seconds before the request for information times out. It ensures that the user responds in a timely manner. If greater than the timeout value, a timeout page will be sent.
- `template`—defines the file used as a display template for this screen.
- `image`—defines a custom background image to be displayed on this screen.
- `header`—defines text information that can be displayed in the browser window for this screen.
- `error`—takes a true or false value which defines whether the error message generated by the authentication module will be used.

Code Example 3-3 on page 58 defines three screen's callback elements that can be called by the LDAP Authentication module. The first asks the requestor for a name and password. The second screen allows the requestor to change their password. The final screen sends a message to reset the password.

NameCallback Element

The *NameCallback* element is used to request data that is entered by the user, for example, a user identification. It can contain one sub-element: *Prompt*. The optional XML attributes are `isRequired` and `attribute`. `isRequired` takes a value of true or false and defines whether the element is required information. (A value of true displays an asterisk next to the attribute's name in the GUI.) `attribute` takes a character value of the corresponding LDAP attribute of this value.

PasswordCallback Element

The *PasswordCallback* element is used to request password data that is entered by the user. It can contain one sub-element: *Prompt*. The XML attributes are `echoPassword`, `isRequired` and `attribute`. `echoPassword` is required and takes a value of true or false and defines whether the password should be displayed on the

screen or not. `isRequired` is optional and takes a value of `true` or `false` and defines whether the element is required information. (A value of `true` displays an asterisk next to the attribute's name in the GUI.) `attribute` is also optional and takes a character value of the corresponding LDAP attribute of this value.

ChoiceCallback Element

The *ChoiceCallback* element is used when the application user must choose from multiple values. It can contain two sub-elements: *Prompt* or *ChoiceValues*. The XML attributes are `multipleSelectionsAllowed`, `isRequired` and `attribute`. `multipleSelectionsAllowed` is a required attribute and takes a value of `true` or `false`. It defines whether the user can choose a number of values or only one from the available choices. `isRequired` is optional and takes a value of `true` or `false`. (A value of `true` displays an asterisk next to the attribute's name in the GUI.) `attribute` is also optional and takes a character value of the corresponding LDAP attribute of this value.

ConfirmationCallback Element

The *ConfirmationCallback* element is used to send 'button' information, such as button text which needs to be rendered on the module's screen, as well as receive the button information, such as which button is clicked by the user. It can contain one sub-element: *OptionValues*. There are no XML attributes.

Prompt Element

The *Prompt* element is used to set the prompt that will display on the browser screen to request the information. It has no sub-elements or XML attributes.

ChoiceValues and ChoiceValue Element

The *ChoiceValues* element provides a list of values from which the user can select. It must contain at least one sub-element of the type *ChoiceValue* which defines one choice. *ChoiceValue* must contain the sub-element *Value*. *ChoiceValues* has no XML attributes but *ChoiceValue* can contain the XML attribute `isDefault`. `isDefault` specifies if the defined value has to be selected by default when displayed; it takes a value of `true` or `false`.

OptionValues and OptionValue Element

The *OptionValues* element provides a list of text information for buttons that need to be rendered on the login screen. It must contain at least one sub-element of the type *OptionValue* which defines one button text value. *OptionValue* must contain the sub-element *Value*. *OptionValues* has no XML attributes but *OptionValue* can contain the XML attribute `isDefault`. `isDefault` specifies if the defined value has to be selected by default when displayed; it takes a value of `true` or `false`.

Value Element

The *Value* element is used by the client to return a value provided by the requestor back to the Identity Server. It has no sub-elements or XML attributes.

Modifying amAuth.xml

The `amAuth.xml` defines the “parent” authentication service named Core. Following are the attributes in this file that need to be extended in order for the Authentication Service to recognize a new authentication module. `amAuth.xml` is located in `<identity_server_root>/SUNWam/config/xml`.

- `iplanet-am-auth-authenticators`—specifies the Java classes of the authentication services available to an organization within the Identity Server deployment. By default, this includes the Anonymous, Certification, LDAP, Membership, RADIUS, SafeWord, and Unix modules. To define a new authentication module, this field takes a value equal to a text string that specifies the full class name (including package) of the new module.
- `iplanet-am-auth-allowed-modules`—lists the authentication modules available to the specific organization. An administrator can choose the authentication method for their organization. The default authentication method is LDAP.

After modifying `amAuth.xml`, the command line tool `amadmin` is used to remove the old Core service file and load the modified one.

1. `amadmin --runasdn <admin_dn> --password <password> --deleteService iPlanetAMAuthService`
2. `amadmin --runasdn <admin_dn> --password <password> --schema amAuth.xml`

More information on the command line tool can be found in the *Sun ONE Identity Server Administration Guide*.

Application Authentication

Java™ applications use the authentication API to access, and authenticate to, the Authentication Service while C applications use a web browser. Both types of applications use the `remote-auth.dtd` to format the structure of the XML request messages used to transfer the authentication information.

Authentication API For Java Applications

External Java applications use the Authentication Java API to initiate the authentication process and communicate with the required authentication module. These Authentication Java API are organized in a package called `com.sun.identity.authentication` and can be executed locally or remotely to communicate locally with the Authentication Service. Communication between the API and the framework occurs by sending XML messages over HTTP(s).

NOTE The Identity Server Javadocs can be accessed from any browser by copying the complete `<identity_server_root>/SUNWam/docs/` directory into the `<identity_server_root>/SUNWam/public_html` directory and pointing the browser to `http://<server_name.domain_name>:<port>/docs/index.html`.

The `AuthContext` class is defined for each request desiring to authenticate to the Identity Server. Since Identity Server can handle multiple organizations, the `AuthContext` class must be initialized, at least, with the name of the organization to which the requestor is authenticating. Typical code would instantiate this class to begin the login process. The caller would then use the `getRequirements` method to ask for the requestor's credentials. The credentials are then submitted to the class using `submitRequirements`. If more information is required, the above process continues until all the required information has been supplied. The `getStatus` method is then called to check if the user has been successfully authenticated. If successful, the caller can then get the `Subject` and `SSOToken` for the user; if not, the caller obtains a `LoginException`. Identity Server is shipped with a sample that uses this class; for more information, see the section "Remote Client API," on page 77.

NOTE The Authentication API are also able to invoke authentication modules written using the pure JAAS API.

Authenticating Non-Java Applications

Non-Java applications can also authenticate to the Identity Server. Using the URL `http://<host.domain:port>/<service_deploy_uri>/authservice`, the application opens a connection and then exchanges XML messages with the Identity Server. The XML messages are structured according to the `remote-auth.dtd`. Information on this document can be found in “The `remote-auth.dtd` Structure,” on page 63. An example of the messages used by C applications can be found in “C Programs and Authentication,” on page 73.

The `remote-auth.dtd` Structure

Authentication requests and responses are sent to and received by the Authentication Java API or non-Java applications using an XML structure. The structure of these messages is defined in the `remote-auth.dtd`. The `remote-auth.dtd` defines the structure for the XML-based messages sent to, and received by, the Identity Server console. It provides definitions to initiate the collection of credentials and perform authentication. It is located in the `<identity_server_root>/SUNWam/dtd` directory. An explanation of the elements defined by the `remote-auth.dtd` follows. Each element has required and/or optional XML attributes.

AuthContext Element

AuthContext is the root element of the XML-based message. It must contain a *Request* or *Response* sub-element. The required XML attributes of *AuthContext* are `version` which takes a value equal to the version number.

Request Element

The *Request* element is used by the client to initialize and pass user credentials to the Authentication Service. It may contain one or more of the following sub-elements: *NewAuthContext*, *QueryInformation*, *Login*, *SubmitRequirements*, *Logout* or *Abort*. The required XML attribute of *Request* is `authIdentifier` which takes a value equal to a unique random number set by the Authentication Service and used to keep track of the authentication session.

NewAuthContext Element

The *NewAuthContext* element initiates the authentication process by initializing the Authentication Service and creating a session token for each request. It contains no sub-elements. The required XML attribute of *NewAuthContext* is `orgName` which takes a value equal to the name of the organization or sub-organization for which the process is defined.

QueryInformation Element

The *QueryInformation* element is used by the remote client to get information about the authentication modules supported by the Identity Server or the organization. It contains no sub-elements. The required XML attribute of *QueryInformation* is `requestedInformation` which takes a value equal to the authentication module plug-ins configured for an organization or sub-organization.

Login Element

The *Login* element is used to initialize the authentication session. It will have an *Empty* sub-element, or can have an *IndexTypeNamePair*. The *IndexTypeNamePair* element can be used to specify the defined authentication type and value. It has no required XML attributes.

SubmitRequirements Element

The *SubmitRequirements* element is used by the remote client to submit the identity's authentication credentials to the Identity Server. It has a *Callbacks* sub-element and no required XML attributes.

Logout Element

The *Logout* element is used by the remote client to indicate that user wants to logout. It has an *Empty* sub-element and no required XML attributes.

Abort Element

The *Abort* element is used by the remote client to indicate that the user wants to end the login process. It has an *Empty* sub-element and no XML attributes.

Response Element

The *Response* element is used by the Authentication Service to ask the remote client to gather user credentials or to inform the remote client on the success or failure of the login as well as any errors that might have occurred. It may contain one or more of the following sub-elements: *QueryResult*, *GetRequirements*, *LoginStatus* or *Exception*. Table 3-3 shows the Request sub-elements and the possible Responses for each.

Table 3-3 Request Sub-Elements And Possible Responses

Request	Possible Responses
NewAuthContext	LoginStatus or Exception
QueryInformation	QueryResult or Exception
Login	GetRequirements, LoginStatus or Exception
SubmitRequirements	GetRequirements, LoginStatus or Exception
Logout	LoginStatus or Exception
Abort	LoginStatus or Exception

The required XML attribute of *Response* is *authIdentifier* which takes a value equal to a unique random number set by the Authentication Service and used to keep track of the authentication session.

QueryResult Element

The *QueryResult* element is used by Identity Server to send query information requested by the remote client. It must contain a *Value* sub-element. The required XML attribute of *QueryResult* is *requestedInformation* which takes a value equal to the authentication module plug-ins configured for an organization or sub-organization.

GetRequirements Element

The *GetRequirements* element is used by the Identity Server to request authentication credentials from the client. It has a *Callbacks* sub-element and no required XML attributes.

LoginStatus Element

The *LoginStatus* element is used by the Identity Server to indicate the status of the authentication process. It will have an *Empty* sub-element if a *Subject* or *Exception* sub-element is not defined. The XML attributes are `status`, `ssoToken`, `successURL` or `failureURL`; the latter three are optional. If the *LoginStatus* is successful, the sub-element *Subject* will be returned with the authenticated user names. The attribute `ssoToken` will have the session token status set to `inprogress` when a new *AuthContext* is created, to `success` when a login has been successful, to `failed` when a login has not been successful and `completed` when the user logs out. The `successURL` attribute represents the URL that the identity will be redirected to upon successful authentication and `failureURL` represents the URL that the identity will be redirected to upon failed authentication.

Exception Element

The *Exception* element is used by the Identity Server to inform the client about an exception that occurred during the login process. It has an *Empty* sub-element and four optional XML attributes: `message` which takes a value equal to that of the exception message, `tokenId` which takes a value equal to that of the user ID of the failed authentication, `errorCode` which takes a value equal to that of the error message code and `templateName` which takes a value equal to the name of the JSP template which will be used for this particular exception.

IndexTypeNamePair Element

The *IndexTypeNamePair* element identifies the defined authentication method that will be used to validate the client. It has the *IndexName* sub-element. The required XML attribute is `IndexType` which takes a value equal to that of the generic level at which the authentication method has been defined: `authLevel`, `role`, `user`, `moduleInstance` and `service`.

IndexName Element

The *IndexName* element identifies the specific name of the value specified by the `IndexType` attribute in the *IndexTypeNamePair* element. The authentication method can be defined at the organization level, the role level, the user level, the authentication level or the service/application level. The `IndexType` attribute defines this level; the *IndexName* element takes a value equal to that of the specific name of the level at which the authentication method has been defined. It has no sub-elements and no XML attributes.

Subject Element

The *Subject* element identifies a collection of one or more identities. It has no sub-elements and no XML attributes.

Callbacks Element

The *Callbacks* element is used to request and transfer user credentials between the remote client and Identity Server. Identity Server constructs callback objects for information gathering. The client program collects the credentials by prompting the user and returns the callback objects with the required data. The *Callbacks* element may contain one or more of the following sub-elements: *NameCallback*, *PasswordCallback*, *ChoiceCallback*, *ConfirmationCallback*, *TextInputCallback*, *TextOutputCallback*, *LanguageCallback*, *PagePropertiesCallback* and *CustomCallback*. The required XML attribute is `length` which takes a value equal to that of a token.

NameCallback Element

The *NameCallback* element is used to obtain the name of the user (or service) that is requesting authentication. It may contain one or more of the following sub-elements: *Prompt* or *Value*. It has no required XML attributes.

PasswordCallback Element

The *PasswordCallback* element is used to obtain the password of the user (or service) that is requesting authentication. It may contain one or more of the following sub-elements: *Prompt* or *Value*. The required XML attribute is `echoPassword` which takes a value of true or false. The default value of false indicates that there will be no password confirmation.

ChoiceCallback Element

The *ChoiceCallback* element is used when the user must choose from a selection of values. It may contain one or more of the following sub-elements: *Prompt*, *ChoiceValue* or *SelectedValues*. The required XML attribute is `multipleSelectionsAllowed` which takes a value of true or false. The default value of false indicates that the user can not choose more than one from the selection.

ConfirmationCallback Element

The *ConfirmationCallback* element is used by the Identity Server to request a confirmation from the user. It may contain one or more of the following sub-elements: *Prompt*, *OptionValues*, *SelectedValue*, and *DefaultOptionValue*. The required XML attributes are `messageType` (which defines the type of message, either information, warning or the default, error), and `optionType` which specifies the type of confirmation (`ok_cancel`, `yes_no_cancel`, `unspecified` or the default, `yes_no`).

TextInputCallback Element

The *TextInputCallback* element is used to get text information from the user. It may contain one or more of the following sub-elements: *Prompt* or *Value*. There are no required XML attributes.

TextOutputCallback Element

The *TextOutputCallback* element is used when the user must choose from a selection of values. It may contain the sub-element *Value*. The required XML attribute is `messageType` which defines the type of message, either information, warning or the default, error.

LanguageCallback Element

The *LanguageCallback* element is used by the Identity Server to obtain the user's locale information. It must contain the *Locale* sub-element. There are no required XML attributes.

PagePropertiesCallback Element

The *PagePropertiesCallback* element contains all GUI-related information. It may contain any of the following sub-elements: *ModuleName*, *HeaderValue*, *ImageName*, *PageTimeoutValue*, or *TemplateName*. The required XML attribute is `isErrorState` which takes a value of true or false. The default value is false which indicates that this page is not an error page.

ModuleName Element

The *ModuleName* element is takes a value equal to the name of the authentication module. It contains no sub-elements and no XML attributes.

HeaderValue Element

The *HeaderValue* element is takes a value equal to the header that will be displayed. It contains no sub-elements and no XML attributes.

ImageName Element

The *ImageName* element takes a value equal to the name of the image to be displayed. It contains no sub-elements and no XML attributes.

PageTimeOutValue Element

The *PageTimeOutValue* element is the page time-out value in seconds. It contains no sub-elements and no XML attributes.

TemplateName Element

The *TemplateName* element takes a value equal to the name of the template to be rendered. It contains no sub-elements and no XML attributes.

CustomCallback Element

The *CustomCallback* element is used to define user-defined Callbacks. It may contain the *AttributeValuePair* sub-element. The required XML attribute is the `className` which takes a value equal to that of the *Callback* name.

AttributeValuePair Element

The *AttributeValuePair* element contains the attribute and values for a *Callback*. It must contain the *Attribute* sub-element and it can contain the *Value* sub-element. There are no required XML attributes.

Attribute Element

The *Attribute* element defines the *Callback* parameter. It contains no sub-elements. The required XML attribute is `name` which takes a value equal to the name of the *Callback* parameter.

Prompt Element

The *Prompt* element is used by Identity Server to request the remote client to display the prompt. It contains no sub-elements and there are no required XML attributes.

Locale Element

The *Locale* element contains the value of the locale that will be used for authentication. It contains no sub-elements. The optional XML attributes are `language` (which represents the language code), `country` (which represents the country code) and `variant` (which represents the variant code).

ChoiceValues Element

The *ChoiceValues* element provides a list of choices. It must contain at least one the *ChoiceValue* sub-element. There are no required XML attributes.

ChoiceValue Element

The *ChoiceValues* element provides a single choice. It must contain at least one *Value* sub-element. The required XML attribute is `isDefault` which takes a value of yes or no. The default value of no specifies if the value has to be selected by default when displayed.

SelectedValues Element

The *SelectedValues* element provides a list of values selected by the user. It must contain at least one *Value* sub-element. There are no required XML attributes.

SelectedValue Element

The *SelectedValue* element provides a value selected by the user. It must contain at least one *Value* sub-element. There are no required XML attributes.

OptionValue Element

The *SelectedValues* element provides a single user-defined option value. It must contain at least one *Value* sub-element. There are no required XML attributes.

DefaultOptionValue Element

The *DefaultOptionValue* element is the default option value. The default value depends on whether user-defined values or predefined values are used in the callback. If user-defined values are used, the default value will be an index in the *OptionValues* element; if predefined, it will be one of the predefined option values. It must contain at least one *Value* sub-element. There are no required XML attributes.

Value Element

The *Value* element is used by the remote client to return a value, provided by the user (or service), back to the Identity Server. It must contain at least one *Value* sub-element. There are no required XML attributes.

Authentication SPI

The Authentication SPI implements the JAAS LoginModule API, and provides methods to access the Authentication Service and module configuration properties files. The SPI are organized in the `com.iplanet.authentication.spi` package and contain the abstract class, `AMLoginModule`, which must be sub-classed with the name of an authentication module. The class must also implement the `init()`, `process()` and `getPrincipal()` methods in order to communicate with the module configuration properties files. The callbacks are then dynamically generated based on this file. Other methods that can be defined include the `setLoginFailureURL` and `setLoginSuccessURL` which set the URLs to send the user to based on a failed or successful authentication, respectively. More information on the SPI can be found in the Javadocs located at `<identity_server_root>/SUNWam/docs`.

URL Parameters

A URL parameter is a name/value pair appended to the end of a URL. The parameter starts with a question mark (?) and takes the form `name=value`. If more than one URL parameter exists, each parameter is separated by an ampersand (&). URL parameters pass information from the browser to the Identity Server. The following parameters, when appended to an authentication URL and typed in a web browser's Location bar, will redirect the user to the appropriate resource after authentication.

- goto**—Adding a `goto=<auth_success_URL>` query parameter tells the Authentication Service to send the user to the noted URL when successful authentication has been completed. An example `goto` URL might be `http://<host.domain:port>/<service_deploy_uri>/UI/Login?goto=http://www.sun.com`. A `goto=<logout_URL>` query parameter can also be used to tell the Authentication Service to send the user to the noted URL when they have logged out. An example `goto` URL might be `http://<host.domain:port>/<service_deploy_uri>/UI/Login?goto=http://www.sun.com/logout.html`.
- gotoOnFail**—Adding a `gotoOnFail=<auth_fail_URL>` query parameter tells the Authentication Service to send the user to the URL noted if user authentication has failed. An example `gotoOnFail` URL might be `http://<host.domain:port>/<service_deploy_uri>/UI/Login?gotoOnFail=http://www.sun.com/auth_fail.html`.

- **org**—The Authentication Service needs to know the requesting user's organization when the user first accesses the Identity Server. From this information the correct login page, based on the organization and the locale setting in the particular organization will be displayed. Adding an `org=<org_name>` query parameter tells the Authentication Service the user's organization. An example `org` URL might be `http://<host.domain:port>/<service_deploy_uri>/UI/Login?org=iplanet.`
- **user**—Authentication can be defined at the user level. For example, one user's profile can be configured to authenticate using the Certification module while another might be configured to authenticate using the LDAP module. Adding a `user=<user_name>` query parameter tells the Authentication Service to send the user to their configured authentication process. An example `user` might be `http://<host.domain:port>/<service_deploy_uri>/UI/Login?user=test.`
- **role**—Authentication can be configured on a per-role basis. Adding a `role=<role_name>` query parameter tells the Authentication Service to send the user to the authentication process configured for that role. An example `role` URL might be `http://<host.domain:port>/<service_deploy_uri>/UI/Login?role=manager.`
- **module**—A specific authentication module can be requested by adding a `module=<auth_module_name>` query parameter. An example `module` URL might be `http://<host.domain:port>/<service_deploy_uri>/UI/Login?module=unix.`
- **service**—Different authentication schemes can be configured for different services using the Authentication Configuration Service. For example, an online paycheck application might require authentication using a more secure Certification module while an organization's employee directory application might require LDAP authentication only. An authentication scheme can be configured, and named, for each of these services. The `service=<auth_scheme_name>` query parameter tells the Authentication Service that this user desires to use the `<auth_scheme_name>` authentication configuration. An example `service` URL might be `http://<host.domain:port>/<service_deploy_uri>/UI/Login?service=sv1.`

NOTE The Authentication Configuration Service is used to define a scheme for service-based authentication. More information on this service can be found in the *Identity Server Administration Guide*.

- **arg=newsession**—The Authentication Service will destroy an existing session token and perform a new login in one request with the use of the `arg=newsession` query parameter. This option is typically used in the Anonymous Authentication module. The user first authenticates with an anonymous session, and then hits the register or login link. An example `arg=newsession` URL might be `http://<host.domain:port>/<service_deploy_uri>/UI/Login?arg=newsession`.
- **authlevel**—Each authentication module is defined with a fixed integer authentication level. Adding an `authlevel=<value>` query parameter tells the Authentication Service to call a module with, at the least, the configured authentication level. An example `authlevel` URL might be `http://<host.domain:port>/<service_deploy_uri>/UI/Login?authlevel=1`.
- **domain**—This parameter allows a user to login to the specified domain. An example `domain` URL might be `http://<host.domain:port>/<service_deploy_uri>/UI/Login?domain=sun.com`.
- **iPSPCookie**—This parameter allows a user to login with a persistent cookie. A persistent cookie is one that continues to exist after the browser window is closed. In order to use this parameter, the organization to which the user is logging in must have Persistent Cookies enabled in their Core service. An example `iPSPCookie` URL might be `http://<host.domain:port>/<service_deploy_uri>/UI/Login?org=example&iPSPCookie=yes`. Once the user authenticates and the browser is closed, the user can login with a new browser session and will be directed to console without having to reauthenticate. This will work until the Persistent Cookie Max Time specified in the Core Service elapses.
- **Login.Token1 (<username>)** , **Login.Token2 (<password>)**—This parameter allows a user to login without having to access the authentication interface.
- **Login.Token0**—This parameter allows a user to login anonymously.

C Programs and Authentication

Following is an example of how customers create XML messages to call the Authentication Service from a C program.

Authentication Request / Response Flow

1. Create an AuthContext for the organization name to login to. The successful response for this is the authIdentifier which is the sessionID for this request. Code Example 3-4 illustrates what the Request XML message would look like.

Code Example 3-4 Request XML Message For C Applications

```

"POST /amserver/authservice HTTP/1.0
Accept text/xml
Content-Length: <len>
Content-Type: text/xml; charset=UTF-8

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<RequestSet vers="1.0" svcid="auth" reqid="1">
<Request><![CDATA[<?xml version="1.0" encoding="UTF-8"?>
<AuthContext version="1.0"><Request authIdentifier="0">
<NewAuthContext orgName="/"></NewAuthContext>
</Request></AuthContext>]]></Request>
</RequestSet>

```

Code Example 3-5 illustrates what the Response XML message (which returns the unique session ID for the request) would look like.

Code Example 3-5 Response XML Message From C Applications

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ResponseSet vers="1.0" svcid="auth" reqid="1">
  <Response><![CDATA[<?xml version="1.0"
encoding="ISO-8859-1"?>
  <AuthContext version="1.0">
    <Response authIdentifier="<sessionid>"
      <LoginStatus
status="in_progress"></LoginStatus>
    </Response>
  </AuthContext>]]>
  </Response>
</ResponseSet>

```

2. Start the authentication process using the Login Element. Code Example 3-6 illustrates this message.

Code Example 3-6 XML Message To Start Authentication Process

```

"POST /amservice/authservice HTTP/1.0
Accept text/xml
Content-Length: <len>
Content-Type: text/xml; charset=UTF-8

<xml version="1.0" encoding="UTF-8" standalone="yes"
<RequestSet vers="1.0" svcid="auth" reqid="1">
<Request><![CDATA[<?xml version="1.0" encoding="UTF-8" ?>
  <AuthContext version="1.0"><Request
authIdentifier="<sessionid">"><Login/></Request>
</AuthContext>]]></Request>
  </RequestSet>

```

The Response to this Request will be the credentials required for the organization's configured authentication method. (Passing the credentials is done using JAAS Callbacks.) Assuming that the default authentication module is LDAP, Code Example 3-7 illustrates the Response.

Code Example 3-7 XML Response To Authentication Process Request

```

<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<ResponseSet vers="1.0" svcid="auth" reqid="1">
  <Response><![CDATA[<?xml version="1.0"
encoding="ISO-8859-1" ?>
    <AuthContext version="1.0">
      <Response authIdentifier="<sessionId">">
        <Callbacks length="3">
          <PagePropertiesCallback isErrorState="false">
            <ModuleName>LDAP</ModuleName>
            <HeaderValue>This server uses LDAP
Authentication
            </HeaderValue>
            <ImageName>null</ImageName>
            <PageTimeout>120</PageValue>
            <TemplateName>null</TemplateName>
          </PagePropertiesCallback>
          <NameCallback><Prompt> User Name:
Authentication
          </Prompt></NameCallback>
          <PasswordCallback echoPassword="false">
            <Prompt> Password:</Prompt>
Authentication
          </PasswordCallback>
        </Callbacks>
      </Response>
    </AuthContext>]]>

```

Code Example 3-7 XML Response To Authentication Process Request

```

    </Response>
  </ResponseSet>

```

3. Send the required credentials back to the Identity Server. Code Example 3-8 illustrates this XML message. Code Example 3-8 illustrates the XML message that contains the authentication credentials.

Code Example 3-8 XML Request With Authentication Credentials

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
  <RequestSet vers="1.0" svcid="auth" reqid="1">
    <Request><![CDATA[<?xml version="1.0"
encoding="ISO-8859-1"?>
  <AuthContext version="1.0">
    <Request authIdentifier="<sessionId">
      <SubmitRequirements>
        <Callbacks length="2">
          <NameCallback><Prompt> User Name: </Prompt>
            <Value> <value> </Value>
          </NameCallback>
          <PasswordCallback echoPassword="false">
            <Prompt> Password:</Prompt>
            <Value> <password> </Value>
          </PasswordCallback>
        </Callbacks>
      </SubmitRequirements>
    </Request>
  </AuthContext>]]></Request>
</RequestSet>

```

The Response for the above might be callbacks if the module requires more information or the LoginStatus. Assuming that there are no more callbacks, the XML Response is illustrated by Code Example 3-9.

Code Example 3-9 XML Response To Authentication Credentials

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
  <ResponseSet vers="1.0" svcid="auth" reqid="1">
    <Response><![CDATA[<?xml version="1.0"
encoding="ISO-8859-1"?>
  <AuthContext version="1.0">
    <Response authIdentifier="<sessionId">

```

Code Example 3-9 XML Response To Authentication Credentials

```

                <LoginStatus status="success"
ssotoken="sessionid" successurl="http://www.yahoo.com">
                </LoginStatus>
            </Response>
        </AuthContext>]]></Response>
    </ResponseSet>

```

Authentication Samples

Authentication Service samples have been provided and can be found in the directory `<identity_server_root>/SUNWam/samples/authentication`. They include:

- Remote Client API
- Login Module

Remote Client API

This sample program demonstrates how to integrate the Remote Client API for authenticating users with the Identity Server. It uses LDAP authentication although it can be modified to use other existing or customized authentication modules. The instruction file is the `readme.html` file found in the `<identity_server_root>/SUNWam/samples/authentication/LDAP` directory.

Login Module

This sample demonstrates the steps needed to integrate a custom login module into the Identity Server. All the files needed to compile, deploy and run the sample authentication module that is shipped with Identity Server can be found in the `<identity_server_root>/SUNWam/samples/authentication/providers` directory. The instruction file is the `Readme.html` file in the same directory.

Single Sign-On

The Sun™ One Identity Server provides a single sign-on (SSO) solution that enables a user to authenticate once to access multiple resources. In other words, successive attempts by a user to access protected resources will not require them to provide authentication credentials for each attempt. This chapter explains the solution, how it works and the SSO APIs. It contains the following sections:

- Overview
- Cookies and Session Tokens
- Cross-Domain Support For SSO
- SSO API
- Sample SSO Java Files

Overview

Identity Server uses access control instructions (ACIs) to define administrative privileges that will protect an organization's proprietary data and web resources from unauthorized persons. A user wanting to access these protected resources must first pass validating credentials through the Authentication Service. A successful authentication gives the user authorization to access the protected resources, based on their assigned policies, roles or other such instructions. If a user wants to access *several* resources protected by Identity Server, the Session (or SSO) Service provides proof of authorization so there is no need to re-authenticate. As different domains generally have common users who need to generate access to their services in a single user session, Identity Server has also added a cross-domain functionality to the Session Service.

Contacting A Policy Agent

When a user attempts to access a protected resource via a web browser, a policy agent installed on the server that hosts the resource intercepts the request. The policy agent then inspects the request to see if a user session identifier, or *token*, exists. If none exists, the request is passed to the Identity Server where it first contacts the Session Service to create a session token and then the Authentication Service which pushes a login page to verify the user.

NOTE Policy agents police the web server or application server on which the protected resource lives and enforce user policy. They are available for installation separately from the Identity Server. Additional information can be found in the *Sun ONE Identity Server Policy Agent Guide*.

Creating A Session Token

Before a user's credentials can be authenticated, a session token is generated by the Session Service. Each token contains a randomly-generated Identity Server session identifier which ultimately represents the authenticated user. Once created, the Authentication Service inserts the token into a cookie and assigns it to the client browser. At the same time the token is assigned, a login page is returned to the user based upon their organization's method of authentication (LDAP, RADIUS, Unix, etc.).

NOTE The session token, at this point, is in an *invalid* state and will remain in one until the user has completed authentication.

Providing User Credentials

The user, having received the correct login page as well as a session token, fills in the appropriate authorization information based on the login page returned. After the user enters their credentials, the data is sent to the authentication provider (LDAP server, RADIUS server, etc.) for verification. Once the provider has successfully verified the credentials, the user is authenticated. The user's specific session information is retrieved from the token and the session state is set to *valid*. The user can now be redirected to the resource they were attempting to access.

Cookies and Session Tokens

A *cookie* is an information packet generated by a web server and passed to a web browser. It maintains information about the user's habits with regards to the web server it is generated by. It does not imply that the user is authenticated. Cookies are domain-specific; for example, a cookie generated by `domainone.com` cannot be used in `domaintwo.com`. In an Identity Server implementation, the cookie is generated by the Session Service and set by the Authentication Service. In addition, Identity Server cookies are session cookies that are stored in memory only.

A *session token* is generated by the Session Service and inserted into a cookie. It is generated using a secure random number generator and contains Identity Server-specific session information. Before a protected resource is accessed, the user is validated by the Authentication Service and a SSO token is created.

Cross-Domain Support For SSO

Identity Server supports cross-domain SSO. A user authenticated to Identity Server in one domain can access resources protected by a web agent in another domain. For example, in one scenario, the Identity Server instance for DomainA is the authentication provider. A user authenticates to Identity Server in DomainA and, after authentication, the token is set for DomainA. ServerB is protected by a web agent talking to an Identity Server in DomainB.

NOTE It is not obligatory to have an installed instance of Identity Server in both domains to use the cross-domain feature.

The Identity Server in DomainB recognizes the DomainA server as its authentication provider. If UserA accesses a resource on ServerB after authenticating to DomainA, the policy agent at DomainB checks for a SSO token and finds that there is no token authorizing access to DomainB. In a cross-domain SSO scenario, the agent will redirect the user to the URL of the cross-domain component running with the Identity Server instance in DomainB. This component redirects the request to the cross-domain component in DomainA since the Identity Server in DomainA is the authentication provider. This request contains the SSO token set by Identity Server in DomainA in the cookie header. The cross-domain component at DomainA will send a response back to the component in DomainB with access authorization if their configured policy permits it. The DomainB component validates the SSO token from DomainA and creates an SSO token for the user in DomainB. This process sets a cookie for the user in DomainB.

If a user accesses a resource directly at DomainB without authenticating at DomainA, the user is redirected to authentication at DomainA. If the authentication is successful, the SSO token is sent to DomainB from DomainA. The ServerB validates the SSO token with DomainA, creates it for DomainB and redirects the user to the original requested resource.

NOTE Identity Server uses a combination of URL parameters and cookies to implement cross-domain SSO. If a cookie is set in DomainA, the cookie value is carried over to DomainB using parameters, and a new cookie will be set with the same cookie name and value, but a different cookie domain.

Enabling Cross-Domain Single Sign-On

To enable cross-domain SSO, the administrator needs to install two different components: the Cross Domain Controller and the CDSSO Component. The Cross Domain Controller component comes bundled, and is installed, with Identity Server. The CDSSO Component needs to be installed separately onto all participating DNS domain servers.

NOTE The administrator can choose not to enable the cross-domain feature; in this case, the CDSSO component would function within a single domain.

Cross Domain Controller

The Cross Domain Controller (CDC) is associated with the Identity Server that is protecting a specific domain. It redirects a request to either the Authentication Service or to the SSO Component. When a HTTP request comes into the CDC and no SSO token information is found, the request is redirected to the Authentication Service. If a SSO token is found for another domain, the request is redirected to the SSO Component with the appropriate session information appended to the query string.

CDSSO Component

The CDSSO Component is deployed in each Identity Server-protected domain. When a user attempts to access a resource, the request is intercepted by the policy agent as discussed in “Contacting A Policy Agent,” on page 80. If no SSO token is found, the request is redirected to the CDSSO Component in the domain where the

resource exists. The CDSSO Component searches the query string again for the SSO token. As no token is found, the request is redirected to the Cross Domain Controller associated with the Identity Server that protects the resource. From this point, the authentication process will be followed.

NOTE If a SSO token is found by the policy agent when the request is made, the CDSSO Component would not receive the request as the agent would validate the token as described in Chapter 3, “Authentication Service.”

Configuring For Cross-Domain SSO

The SSO components need to be enabled in order to allow the cross-domain SSO function to work. Assuming a single Identity Server instance:

1. Install Identity Server in a primary DNS domain.

This will install the complete Identity Server application as well as the CDC component. The default CDC service URL, after installation, is `http(s)://Identity_Server_host:port/amserver/cdcservlet`.

2. Run the installer again on a machine in all participating DNS domains and choose the Cross-Domain Support option.

All machines in participating DNS domains need to have an instance of the CDSSO component installed. After running this option, a CDSSO directory is created in `<identity_server_root>/SUNWam/web-apps`. The default CDSSO Component service URL is `http(s)://<CDSSO_domain_host>:<port>/uri/cdssso`.

NOTE Install the CDSSO Component on any web server with host services (in all participating DNS domains) that need to be protected.

3. Edit the `com.iplanet.services.cdssso.cookieDomain` property in the `cdssso.properties` file found in the `<identity_server_root>/SUNWam/web-apps/cdssso/WEB-INF/classes` directory.

The `com.iplanet.services.cdssso.cookieDomain` property must be set to the domain name which hosts the CDSSO component installed in Step 2. Code Example 4-1 is copied from the file itself.

Code Example 4-1 Portion of `CDSSO.properties` file

```

...
/*
 * The following keys will be used for Cross Domain SSO support.
 * The user if needs cross domain sso support should change
 * "com.iplanet.services.cdssso.CDCURL" property to point to the
 * cdcervlet running with the Identity Server instance
 * "com.iplanet.services.cdssso.cookieDomain" property should
 * specify a comma separated list of domains for which the cdssso
 * servlet will set a SSO token.
 * Ex:com.iplanet.services.cdssso.cookieDomain=.sales.com,
 .eng.com,.marketing.com
 */

com.iplanet.services.cdssso.CDCURL=http://example.domain_name.com
:8080/amserver/cdcervlet
com.iplanet.services.cdssso.cookieDomain=.sales.com
/*
...

```

4. The following three properties, specific to the policy agent, need to be edited in each policy agent's `AMAgent.properties` file if the agent was not originally installed with CDSSO enabled.
 - o Change the value of `com.iplanet.am.policy.agents.ccdssso-enabled` to enable cross-domain SSO. Code Example 4-2 illustrates this property.

Code Example 4-2 Portion of `AMAgent.properties` file

```

...
#Cross-Domain Single Sign On URL
#Is CDSSO enabled
com.sun.am.policy.agents.ccdssso-enabled=true
...

```

- o Modify the SSO redirect URL. Code Example 4-3 illustrates this property.

Code Example 4-3 Second portion of `AMAgent.properties` file

```

...
/*This is the URL the user will be redirected to after successful
 *login in a CDSSO Scenario.
com.sun.am.policy.agents.cdssso-component.url=http://<cdssso_host>
:<cdssso_port>/<uri>/cdssso
...

```

- o Add the SSO service URL to the *not enforced* list. Code Example 4-3 illustrates this property.

Code Example 4-4 Third portion of `AMAgent.properties` file

```
...
/*If cross domain sso support is enabled notenforcelist should be
*edited to add cdsso servlet URL in it
com.sun.am.policy.agents.notenforcedList=*/amcdsso/*
...
```

This instance of Identity Server and all its participating DNS domains are now cross-domain SSO enabled.

NOTE The cross-domain SSO solution assumes a single Identity Server instance; therefore all user and policy information needs to be centralized in that instance. Multiple Identity Server instances are allowed only if they are all in the same domain.

SSO API

The SSO solution provides Java API to allow external applications to participate in the SSO functionality. All Identity Server services (except for Authentication) need a valid SSO token to process a HTTP request. External applications wishing to use the SSO functionality must use the SSO token to validate the user's identity. With the SSO API, an external application can get the token and, in turn, the identity of a user and related authentication information. Once a user is authenticated, this information is used to determine whether or not to provide access to the requested resource based on the validated user's policy. The SSO API can also be used to create or destroy a SSO token, to check the token's validity or to listen for token *events*. (An event might be a token timing out because the user has reached the token's maximum time limit.)

Non-Web-Based Applications

Identity Server provides the SSO component primarily for web-based applications, although it can be extended to any non-web-based applications with limitations. With non-web-based applications, there are two possible ways to use the API.

1. The application has to obtain the Identity Server cookie value and pass it into the SSO client methods to get to the SSO token. The method used for this process is application-specific.
2. Command line applications, such as `amadmin`, can be used. In this case, SSO tokens can be created to access the Directory Server directly. There is no session created, making the Identity Server access valid only within that process or VM.

API Overview

The primary purpose of the SSO API is to allow any service or application to make use of the SSO functionality. They are provided for the implementation of a SSO solution in external applications. Using these APIs, the identity of the user and related authentication information can be called. The application then uses this information to determine whether to provide user access to a protected resource. The SSO client applications get the information from the SSO token. For example, assume a user authenticates to `http://www.DomainA.com/Store` successfully and later tries to access `http://www.DomainB.com/UpdateInfo`. Rather than having the application authenticate the user again, it can use the API to determine if the user is already authenticated. If the methods indicate that the user is valid and has already been authenticated, access to this page can be given without the user authenticating again. Otherwise, the user is prompted to authenticate again.

Each time a user attempts to access a protected application, the application needs to verify their validity. Generally, the SSO component generates a SSO token for a user once the user is authenticated. After generation, the token is carried with the user as the user moves around the web. When the user attempts to access an application or service that is *SSO-enabled*, this token is used for user validation. Specifically, an instance of the `SSOTokenManager` class is created to allow access to the `createSSOToken`, `destroyToken` and `isValidToken` methods. An instance of the `SSOToken` class is then called; it contains the session information. Between the two, an application can determine if the user is authenticated. Another way to use the API is to invoke the `SSOTokenListener` interface which notifies the application when a token has become *invalid* in order for the application to terminate its access.

NOTE The Identity Server Javadocs can be accessed from any browser by copying the complete `<identity_server_root>/SUNWam/docs/` directory into the `<identity_server_root>/SUNWam/public_html` directory and pointing the browser to `http://<server_name.domain_name>:<port>/docs/index.html`.

SSOTokenManager Class

The `SSOTokenManager` class must be implemented to create one instance per token. It contains the three methods needed to create, get, validate and destroy SSO tokens. The `createSSOToken()` method is called to create a session token. It contains methods for doing this using the command line or through the internet. The `destroyToken()` method is called to delete a token when its session has ended. The `isValidToken()` and `validateToken()` methods can be called to verify the authenticity of a token. `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.

NOTE `SSOTokenManager` is a final class and a singleton. `SSOToken` and `SSOTokenID` are Java interfaces. Additionally, `SSOTokenListener` and `SSOTokenEvent` are provided to support notification when SSO tokens are invalidated.

Sample SSOTokenManager Code

The `SSOTokenManager` class can be used in the following way to determine if a user is authenticated:

Code Example 4-5 Sample SSOTokenManager Code

```
try {
    /* create the sso token from http request */
    SSOTokenManager manager = SSOTokenManager.getInstance();

    /* The request here is the HttpServletRequest. */
    SSOToken token = manager.createSSOToken(request);

    /* use isValid to method to check if the token is valid or not
     * this method returns true for valid token, false otherwise*/
    if (token.isValid()) {

        /* user 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,
         * by some means. If access can be given based on this
         * further check on user information is not necessary.
         */

        /* let us get some user information */
        String host = token.getHost();
        java.security.Principal principal = token.getPrincipal();
        String authType = token.getAuthType();
        int level = token.getAuthLevel();

        .....
    }
}
```

Code Example 4-5 Sample SSOTokenManager Code

```
try {  
    } else {  
        /* token is not valid, redirect the user login page */  
    }  
    ...  
}
```

SSO Implementations

The `SSOTokenManager` maintains a configuration database of valid implementations for `SSOProvider`, `SSOToken` and `SSOTokenID`. A request to `SSOTokenManager` gets delegated to the `SSOProvider`. Hence, the `SSOProvider` performs the bulk of the function of `SSOTokenManager`. The `SSOToken` is the SSO token that contains the crucial information about the token, and `SSOTokenID` is a string representation of SSO token. Although `SSOTokenManager` could support multiple and disparate providers, the only valid SSO provider is `SSOProvider`.

Additional Classes

The following classes can be used to implement customized SSO functionality in an application that does not use the default `SSOProvider` provided.

SSOToken

The `SSOToken` class represents a “single sign-on” token and contains information like the user validation, the authentication method, the host name of the client browser that sent the request, and session information (maximum session time, maximum session idle time, session idle time, etc.). Code Example 4-5 on page 87 also makes use of the `SSOToken` interface.

SSOTokenEvent

The `SSOTokenEvent` class represents a token event. An event is, for instance, when a token becomes invalid due to idle time-out or hitting a time limit maximum. A token is granted when a change in the state of the token, like those mentioned, occurs. An application must come to know of events in order to terminate access to the application for a user whose token has become invalid. The `SSOTokenListener` class would need to be implemented by applications to receive SSO token events.

Sample SSOTokenEvent Code. The `SSOTokenEvent` class can be used in the following way to get SSO Token events:

Code Example 4-6 Sample SSOTokenEvent Code

```

public class AppTokenListener implements SSOTokenListener {
    public void ssoTokenChanged(SSOTokenEvent event) {
        try {
            SSOToken token = event.getToken();
            int type = event.getType();
            long time = event.getTime();
            SSOTokenID id = token.getTokenID();
            System.out.println("Token id: " + id.toString() + "is
not valid anymore");
            /* redirect user to login */
            .....
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
...
SSOTokenListener myListener = new AppTokenListener();
token.addSSOTokenListener(myListener);

```

SSOTokenID

The `SSOTokenID` class is used to identify the `SSOToken` object. Additionally, the `SSOTokenID` string contains a random number, the SSO server host, and server port. The random string in the `SSOTokenID` is unique on a given server. In the case of services written using a servlet container, the `SSOTokenID` can be communicated from one servlet to another either:

- as a cookie in a HTTP header; or
- as an implementation of the `SSOTokenListener` interface by the applications to receive the SSO token events.

SSOTokenListener

The `SSOTokenListener` interface provides a mechanism for applications that need notification when an SSO token expires. (It could expire if it reached its maximum session time, or idle time, or an administrator might have terminated the session.) Applications wishing to be notified must invoke the `addSSOTokenListener` method using the `SSOToken` interface; this method implements the `SSOTokenListener` interface. A callback object will be invoked when the SSO token expires. Using the `SSOTokenEvent` (provided through the callback), applications can determine the time, and the cause of the SSO token expiration.

NOTE Once an application registers for SSO token events using `addSSOTokenListener`, any SSO token event will invoke the `ssoTokenChanged` method. The application can take suitable action in this method.

Sample API Code

Following are examples of code that illustrate various operations that can be performed by the SSO API.

User Authentication Sample Code

This code can be used to determine if a user is authenticated. (Additionally, the API can be used to perform a query on a token for information such as host name, IP address, or idle time).

Code Example 4-7 Code Sample To Determine If User Is Authenticated

```
try {
    ServletOutputStream out = response.getOutputStream();

    /* create the sso token from http request */
    SSOTokenManager manager =
SSOTokenManager.getInstance();
    SSOToken token = manager.createSSOToken(request);

    /* use isValid method to check if the token is valid
    * this method returns true for valid token, false non
    */
    if (manager.isValidToken(token)) {
        /* let us get all the values from the token */

        String host = token.getHostName();
        java.security.Principal principal =
token.getPrincipal();
        String authType = token.getAuthType();
        int level = token.getAuthLevel();
        InetAddress ipAddress = token.getIPAddress();
        long maxTime = token.getMaxSessionTime();
        long idleTime = token.getIdleTime();
        long maxIdleTime = token.getMaxIdleTime();
        out.println("SSOToken host name: " + host);
        out.println("SSOToken Principal name: " +
principal.getName());
        out.println("Authentication type used: " +
authType);
        out.println("IPAddress of the host: " +
ipAddress.getHostAddress());
    }
}
```

Code Example 4-7 Code Sample To Determine If User Is Authenticated (*Continued*)

```

    }
    /* try to validate the token again, with another method
     * if token is invalid, this method throws exception
     */
    manager.validateToken(token);

    /* get the SSOTokenID associated with the token */
    SSOTokenID tokenId = token.getTokenID();

    String id = tokenId.toString();

    /* print the string representation of the token */
    out.println("The token id is " + id);

    /* set properties in the token. We can get the values
     * of set properties later
     */
    token.setProperty("Company", "Sun Microsystems");
    token.setProperty("Country", "USA");
    String name = token.getProperty("Company");
    String country = token.getProperty("Country");

    out.println("Property: Company is - " + name);
    out.println("Property: Country is - " + country);

    out.println("SSO Token Validation test Succeeded");
    /* add a listener to the SSOToken. Whenever a token
     * event arrives, ssoTokenChanged method of the
     * listener will get called.
     */
    SSOTokenListener myListener = new
SampleTokenListener();

    token.addSSOTokenListener(myListener);
    out.flush();
    } catch (Exception e) {
        System.out.println("Exception Message: " +
e.getMessage());
        e.printStackTrace();
    }
}
}

```

In some cases, it might be more efficient and convenient to use `SSOTokenManager.validateToken(token)` than `SSOTokenManager.isValidToken(token)`. `SSOTokenManager.validateToken(token)` throws an exception when the token is invalid, thus terminating the method execution right away.

Get Token Sample Code

This sample code can be used to get the SSO token if the `SSOtokenID` string is passed to the application.

Code Example 4-8 Code Sample To Get Token from Token ID

```
try {
    /* create the sso token from SSO Token Id string */
    SSOTokenManager manager=SSOTokenManager.getInstance();
    SSOToken token = manager.createSSOToken(tokenString);
    /* let us get the SSOTokenID associated with the token
    */
    SSOTokenID id = token.getTokenID();

    String tokenId = id.toString();

    /* print the string representation of the token */
    System.out.println("The token ID is " + tokenId);

    /* set properties in the token. We can get the values
    * of set properties later */
    token.setProperty("Company", "Sun Microsystems");
    token.setProperty("Country", "USA");
    String name = token.getProperty("Company");
    String country = token.getProperty("Country");

    System.out.println("Property: Company is - " + name);
    System.out.println("Property: Country is - " +
country);

    System.out.println("SSO Token Validation test
Succeeded");
    /* add a listener to the SSOToken. Whenever a token
    * event arrives, ssoTokenChanged method of the
    * listener will get called.
    */
    SSOTokenListener myListener = new
SampleTokenListener();

    token.addSSOTokenListener(myListener);
} catch (Exception e) {
    System.out.println(e.getMessage());
    e.printStackTrace();
    SSOTokenManager manager=SSOTokenManager.getInstance();
    SSOToken token = manager.createSSOToken(tokenString);
}
}
```

Listen For Event Code Sample

Applications can listen for SSO token events. It is possible that while a user is using an application, an SSO token may become invalid because, for example:

- the user's access times out because of the maximum time limit; or,
- the user fails to log out of an application and the idle time-out expires.

The application must be informed of these *events* to follow-up on the invalid token by terminating the user's access. The following two sample codes can be used to get token events.

Code Example 4-9 Code Sample To Register For SSOToken Events

```
SSOTokenListener myListener = new SampleTokenListener();
token.addSSOTokenListener(myListener);
```

where `SampleTokenListener` is a class defined as:

Code Example 4-10 Code Sample Defining SampleTokenListener Class

```
public class SampleTokenListener implements SSOTokenListener {
    public void ssoTokenChanged(SSOTokenEvent event) {
        try {
            SSOToken token = event.getToken();
            int type = event.getType();
            long time = event.getTime();

            SSOTokenID id = token.getTokenID();

            System.out.println("Token id is: " + id.toString());

            if (SSOTokenManager.getInstance().isValidToken(token))
            {
                System.out.println("Token is Valid");
            } else {
                System.out.println("Token is Invalid");
            }

            switch(type) {
            case SSOTokenEvent.SSO_TOKEN_IDLE_TIMEOUT:
                System.out.println("Token Idle Timeout event");
                break;
            case SSOTokenEvent.SSO_TOKEN_MAX_TIMEOUT:
                System.out.println("Token Max Timeout event");
                break;
            }
        }
    }
}
```

Code Example 4-10 Code Sample Defining SampleTokenListener Class (*Continued*)

```

        case SSOTokenEvent.SSO_TOKEN_DESTROY:
            System.out.println("Token Destroyed event");
            break;
        default:
            System.out.println("Unknown Token event");
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
}
}

```

After the application registers for SSO token events using `addSSOTokenListener`, any SSO token events will invoke the `ssoTokenChanged()` method. The application can take a suitable action in this method.

Sample SSO Java Files

Identity Server provides three groups of sample Java files. With these samples, a developer can create an SSO token in several ways:

1. An SSO token can be created for an application that runs on the Identity Server server.
2. An SSO token can be created for an application that runs on a server other than the Identity Server server.
3. An SSO token can be created by a session ID string can be passed through the command line.

The files are in the `<identity_server_root>/SUNWam/samples/sso` directory.

SSO Servlet Sample

This sample can be used to create a token for an application that resides on the same server as the Identity Server application. The files used for this sample are:

- `Readme.html`
- `SampleTokenListener.java`
- `SSOTokenSampleServlet.java`

The instructions in `Readme.html` can be followed to run this code.

Remote SSO Sample

This sample can be used to create a token for an application that resides on a different server from the one on which the Identity Server application lives. The files used for this sample are:

- `remote.html`
- `SSOTokenFromRemoteServlet.java`
- `SSOTokenSampleServlet.java`

The instructions in `remote.html` can be followed to run this code.

Command Line SSO Sample

This sample illustrates how to validate a user from the command line using a session ID string. The files used for this sample are:

- `ssocli.txt`
- `CommandLineSSO.java`
- `SSOTokenSample.java`

The instructions in `ssocli.txt` can be followed to run this code.

Identity Management

The Identity Management module of Sun™ ONE Identity Server contains an XML template file and application programming interfaces (APIs) that provide functionality to, among other operations, create, delete and manage identity entries in the Sun ONE Directory Server used for data storage. This chapter offers information on the public APIs. It contains the following sections:

- Overview
- Object Templates
- Identity Server SDK
- amEntrySpecific.xml
- Management Sample Functions

Overview

The Identity Management module of Identity Server allows for the management of identity-related objects stored in the Directory Server. Towards this end, it provides interfaces for creating and managing identity-related objects in the Directory Server. The management functions that can be performed include the creation and deletion of specific objects as well as the ability to get, add, modify, or remove the attributes of these objects. The interfaces include a set of templates, defined in the `ums.xml` file, that contain LDAP configuration information for identity-related objects and a Java Software Development Kit (SDK) to embed the management functions into applications or services.

Abstract Objects

Identity Server represents the objects it manages abstractly; in other words, an organization in Identity Server does not necessarily map to an LDAP organization in the Directory Server. The default *abstract objects* are:

- organization
- organizational unit
- people container
- static group
- filtered group
- assignable dynamic group
- group container

NOTE For more information, see “amEntrySpecific.xml Schema,” on page 107.

Marker Object Classes

Abstract objects are identified in the Directory Server by object classes that are referred to as *marker* and defined in an Identity Server schema. The marker object classes are then used in LDAP object entries. For example, the Directory Server may use organizational units for their first level structure; by adding the Identity Server organization marker object class, `iplanet-am-managed-org`, to the LDAP entries of these organizational units, Identity Server can manage them as organizations. It is the use of marker object classes that allows Identity Server to manage most directory structures, regardless of the LDAP object classes and naming attributes deployed. The marker object classes are:

- `iplanet-am-managed-filtered-group`
- `iplanet-am-managed-assignable-group`
- `iplanet-am-managed-static-group`
- `iplanet-am-managed-org`
- `iplanet-am-managed-org-unit`
- `iplanet-am-managed-people-container`
- `iplanet-am-managed-group-container`

NOTE The marker object classes are defined in the Identity Server-specific LDAP schema `ds_remote_schema.ldif` which can be found in `<identity_server_root>/SUNWam/ldif`. It is loaded into the Directory Server when Identity Server is installed.

Object Templates

The `ums.xml` provides a set of parameters, known as Templates, that contain LDAP configuration information for all identity-related objects. Identity Server uses these templates to define the configuration of the Directory Server entries that store the Identity Server entry information created by the Identity Server SDK. The file can be found in the `<identity_server_root>/SUNWam/config/ums` directory and is based on the `sms.dtd`. The templates provide LDAP structure for:

- Users
- Groups
- Organizations
- Roles
- Organization Units
- Group Containers
- People Containers

The templates are used by the Identity Server SDK for the creation of identity-related objects in the Directory Server, as well as the dynamic generation of the object's roles and the construction of object searches. (These templates can be modified by administrators to alter the behavior of the Java interfaces.) Using these templates and the LDIF schema, parameters are configured for all identity-related objects.

When Identity Server is installed, the `ums.xml` file is stored in the Directory Server as the *DAI* service. (DAI is a service in the Identity Server whose configuration is not made available through the console.) The Identity Server SDK gets the configuration information from this node when it is being asked to create an identity-related object, generate a role or perform a search. Any attribute specified in the `ums.xml` can be set for a created object.

CAUTION Because `ums.xml` defines templates for directory entries created by the SDK, if it is modified and reloaded, there will be inconsistencies between the entries created prior and the newer ones to be created. Therefore, modifications to this file are not recommended unless Identity Server is being installed as a brand new entity.

Structure of `ums.xml`

The `ums.xml` defines three types of templates: Structure, Creation and Search. Structure templates define the Directory Server DIT attributes for the object. Creation templates define an LDAP template for the object being created. Search templates define guidelines for performing searches using LDAP.

Structure Templates

Structure templates define the form an Identity Server object will take in the Directory Server DIT. This conforms to where the object is located within the DIT; the objects are strictly LDAP entries. There are six attributes that need to be defined for each subschema.

- `class`—This attribute represents the name of the Java class that will implement the object. This attribute is fixed and should never be modified.
- `name`—This attribute defines the entry type of the object. For example, an organization object has `o=org` as its name.
- `childNode`—This attribute specifies the child nodes that will be created in tandem with the object.
- `template`—This attribute specifies the Creation template used to create this object.
- `filter`—This attribute specifies a filter that will be used to identify the object.
- `priority`—This attribute is defined as 0.

Creation Templates

Every identity object that Identity Server creates has a corresponding creation template which defines the LDAP schema for the object. It specifies which object classes and attributes are mandatory or optional and which default values, if any, should be set. This conforms to the actual LDAP entry in the Directory Server. There are six attributes that need to be defined for each subschema.

- `name`—This attribute defines the name of the object the template will create. It is also the name of the template itself.
- `javaclass`—This attribute defines the name of the Java class used to instantiate the object.
- `required`—This attribute defines the required LDAP attributes for the object.
- `optional`—This attribute defines the optional LDAP attributes for the object.
- `validated`—This attribute is reserved for future use.
- `namingattribute`—This attribute specifies the LDAP entry type.

Search Templates

Search templates are used to define how searches for Identity Server objects are performed in the Directory Server. This template defines a default search filter and the attributes returned in a search. For example, a search filter is constructed which defines and specifies which attributes and values are to be retrieved from the Directory Server.

- `name`—This attribute defines the name of the search template.
- `searchfilter`—This attribute defines the LDAP search filter.
- `attrs`—This attribute specifies the LDAP attributes that need to be returned.

Modifying ums.xml

Any new LDAP attributes or object classes must be added to the `ums.xml` file in order for them to be recognized by the Identity Server. In most cases, the attributes that service developers might want to add may already exist in the `inetorgperson` and the `inetuser` object classes. If, for example, a custom mail service is being added with, specifically, an `employee_id` attribute, the `ums.xml` file does not need to be modified because this attribute already exists in the `inetorgperson` object class. Generally, as in the example, the `ums.xml` file does not need to be modified. The only circumstances where this file would need to be modified are:

- if DSAME is being installed against a legacy DIT.
- if new object classes are being added to users or organizations.
- if service developers want to change the default organizations or roles.
- if service developers need to change an entry's naming attribute.

Additional information on when and how to modify the `ums.xml` file is covered in the section on installing against a legacy DIT in the *Sun ONE Identity Server Installation and Configuration Guide*.

CAUTION It is highly recommended that the `ums.xml` configuration file is duplicated before any modifications are made.

Adding Custom Object Classes

If a service developer wanted to add new or customized object classes to the Directory Server for Identity Server's use, they would need to modify the templates in the `ums.xml` file. Then, to manage them from the Identity Server console, these new object classes and attributes have to be modelled as an XML service file and imported into Identity Server using the procedures described in Chapter 6, "Service Management."

NOTE `umsExisting.xml` contains objectclasses and user object class tags which will be replaced after installation and is used when installing Identity Server with an existing directory server information tree.

Identity Server SDK

The Identity Server SDK contains APIs for identity management. These interfaces can be used by developers to integrate management functions into external applications or services that will be managed by the Identity Server. The APIs function to create or delete identity-related objects as well as get, modify, add or delete the object's attributes. The `com.iplanet.am.sdk` package contains all the interfaces and classes necessary to perform these operations in the Directory Server.

NOTE The Identity Server Javadocs can be accessed from any browser by copying the complete `<identity_server_root>/SUNWam/docs/` directory into the `<identity_server_root>/SUNWam/public_html` directory and pointing the browser to `http://<server_name.domain_name>:<port>/docs/index.html`.

SDK Interfaces

Below are brief explanations of the Identity Server SDK interfaces.

AMConstants

`AMConstants` is the base interface for all identity-related objects. It is used to define the scope of a search of the Directory Server. It can search for a specific object, a particular level of the DIT or an attribute.

AMObject

`AMObject` provides basic methods to manage identity-related objects. Since this is a generic class, it does not have any Templates (defined in “Object Templates,” on page 99) associated with it.

AMOrganization

The `AMOrganization` interface provides the methods used to manage organizations. Associated with this interface are the following `ums.xml` Templates that define its behavior at runtime. The name of the structural template used by this class is *Organization*; the name of the creation template used is *BasicOrganization*, and the name of the search template is *BasicOrganizationSearch*.

AMOrganizationalUnit

The `AMOrganizationalUnit` interface provides the methods used to manage organizational units. Associated with this object are the following `ums.xml` Templates that define its behavior at runtime. The name of the structural template used by this class is *OrganizationalUnit*; the name of the creation template used is *BasicOrganizationalUnit*, and the name of the search template is *BasicOrganizationalUnitSearch*.

AMPeopleContainer

The `AMPeopleContainer` interface provides the methods used to manage people containers. Associated with this object are the following `ums.xml` Templates that define its behavior at runtime. The name of the structural template used by this class is *PeopleContainer*; the name of the creation template used is *BasicPeopleContainer*, and the search template is *BasicPeopleContainerSearch*.

AMGroupContainer

The `AMGroupContainer` interface provides the methods used to manage group containers. Associated with this object are the following `ums.xml` Templates that define its behavior at runtime. The name of the structural template used by this class is *GroupContainer*; the name of the creation template used is *BasicGroupContainer*, and the search template is *BasicGroupContainerSearch*.

AMGroup

The `AMGroup` interface provides the methods used to manage groups. This is the basic class for all derived groups, such as static groups, dynamic groups and assignable dynamic groups. No default templates are defined for this class.

AMStaticGroup

The `AMStaticGroup` interface provides the methods used to manage static groups. This class extends the base `AMGroup` interface. The name of the creation template used with this class is *BasicGroup*; and the search template used is *BasicGroupSearch*. It does not have a pre-defined structural template.

AMDynamicGroup

The `AMDynamicGroup` interface provides the methods used to manage dynamic groups. This class extends the base `AMGroup` interface. Associated with this object are the following `ums.xml` Templates that define its behavior at runtime. The creation template used is named *BasicDynamicGroup*; and the search template used is named as *BasicDynamicGroupSearch*. It does not have a pre-defined structural template.

AMAssignableDynamicGroup

The `AMAssignableDynamicGroup` interface provides the methods used to manage assignable dynamic groups. This class extends the base `AMGroup` interface. Associated with this object are the following `ums.xml` Templates that define its behavior at runtime. The creation template used is named *BasicAssignableDynamicGroup*; and the search template used is named *BasicAssignableDynamicGroupSearch*. It does not have a pre-defined structural template.

AMRole

The `AMRole` interface provides the methods used to manage roles. Associated with this object are the following `ums.xml` Templates that define its behavior at runtime. The creation template used is named *BasicManagedRole*; and the search template used is named *BasicManagedRoleSearch*. It does not have a pre-defined structural template.

AMUser

The `AMUser` interface provides the methods used to manage users. Associated with this object are the following `ums.xml` Templates that define its behavior at runtime. The creation template used is named *BasicUser*; and the search template used is named *BasicUserSearch*. It does not have a pre-defined structural template.

AMTemplate

The `AMTemplate` interface represents a service template associated with a `AMObject`. Identity Server distinguishes between virtual and entry attributes. Per Sun ONE Directory Server terminology, a *virtual attribute* is an attribute not physically stored in an LDAP entry but still returned with it as a result of a LDAP search. Virtual attributes are analogous to *inherited* attributes. Entry attributes are non-inherited attributes.

NOTE More information on virtual attributes can be found in “Virtual Attribute,” on page 289 of Appendix B, “Directory Server Concepts.”

For `AMOrganization`, `AMOrganizationalUnit` and `AMRole`, virtual attributes can be grouped in a Template on a per-service basis; there may be one service Template for each service for any given `AMObject`. Such templates determine the service attributes inherited by the users within the scope of this object. There are three types of templates: `POLICY_TEMPLATE`, `DYNAMIC_TEMPLATE` and `ORGANIZATION_TEMPLATE`. `POLICY_TEMPLATE` and `DYNAMIC_TEMPLATE` are implemented using CoS Templates; `ORGANIZATION_TEMPLATE` does not have virtual attributes.

Template Priority

When any object inherits more than one template for the same service (by virtue of being in the scope of two or more objects with service templates), the conflict is resolved through template priorities. In this priority scheme, zero is the highest possible priority with the lower priorities extending towards infinity. Templates with higher priorities will be favored over and to the exclusion of templates with lower priorities. Templates which do not have an explicitly assigned priority are

considered to have the lowest priority possible, or no priority. In the case where two or more templates are being considered for inheritance of an attribute value, and they have the same (or no) priority, the result is undefined, but does not exclude the possibility that an arbitrarily chosen value will be returned.

AMStoreConnection

The `AMStoreConnection` class represents a connection to the Identity Server data store; the Identity Server data store is the Directory Server. This class controls and manages access to the Directory Server by providing methods to create, remove and get different types of identity-related objects. A `SSOToken` is required in order to instantiate a `AMStoreConnection` object.

The SDK And Cache

Caching in the Identity Server SDK is used for storing all `AMObject` attributes (i.e., attributes of identity-related objects) that are retrieved from the Directory Server. The cache does not hold `AMObject` directly, only its attributes. All attributes retrieved from the Directory Server using the interface methods `AMObject.getAttributes()`, `AMObject.getAttribute(String name)` or `AMObject.getAttributes(setAttributeNames)` will be cached.

Cache Properties

The following cache properties can be configured by accessing the `AMConfig.properties` file. They are:

- `com.ipplanet.services.stats.state`—Depending on whether this property is set to `file` or `console`, the cache statistics will be printed to either a `amSDKStats` file or the Identity Server console.
- `com.ipplanet.services.stats.directory`—The value of this property is the directory in which the `amSDKStats` file is created.
- `com.ipplanet.am.statsInterval`—The interval at which cache statistics are printed can be specified as the value of this property. It indicates the number of seconds after which the stats will be printed. For example, a value of 3600 would cause the cache statistics to be printed after 3600 seconds. This will be used only if `com.ipplanet.services.stats.state` is set to `file` or `console`.

Table 5-1 is explains the information that is recorded in the statistics files.

Table 5-1 Recorded Cache Properties

Information Name	What is recorded
Interval	Number of get requests during the specified interval
Hits during interval	Number of hits during the specified interval
Hit ratio for this interval	Hit ratio for the specified interval
Total number of requests	Overall number of get requests since a server re-start
Total number of Hits	Overall number of hits since a server re-start
Overall Hit ratio	Overall hit ratio since a server re-start

Installing the SDK Remotely

It is possible for an external application to perform management functions on the Identity Server data store (Directory Server) without installing the full Identity Server application at the external location. By installing the `SUNWamsdk` package using the `pkgadd` utility and answering NO to the first question, “Install the remote client only”, the full SDK will be installed. Answering YES, the Identity Server SDK can be installed on a non-Identity Server machine.

amEntrySpecific.xml

The purpose of the `amEntrySpecific.xml` service file is to define the attributes that will display on the Create, Properties and Search pages specific to each of the Identity Server *abstract objects*. Each Identity Server abstract object can have its own schema definition in the `amEntrySpecific.xml` file which is based on the `sms.dtd` as described in Chapter 6, “Service Management.”

amEntrySpecific.xml Schema

Each abstract object can have a schema defined in the `amEntrySpecific.xml` file. The schema defines what attributes will be displayed on the function pages used to manage abstract type objects:

- Create—The Create page is displayed when the administrator clicks `New`.

- **Properties**—The Properties Page is displayed when the Properties icon (an arrow in a box) next to an abstract type object is clicked.
- **Search**—The Search link is in the top left frame of the Identity Server console.

If a service developer wants to customize these Identity Server function pages for any of the abstract objects, they would need to modify the `amEntrySpecific.xml`. For example, to display an attribute on the group page, the new attribute needs to be added to the `amEntrySpecific.xml` file. Any abstract object with customized attributes in the Directory Server would need to have those attributes reflected in the `amEntrySpecific.xml` file also. (Most often, a service developer would only be customizing the organization pages.) Code Example 5-1 is the organization attribute subschema that defines the display of an organization's Organization Status and its choice values.

Code Example 5-1 Organization Subschema of `amEntrySpecific.xml`

```

...
<SubSchema name="Organization">
    <AttributeSchema name="inetdomainstatus"
        type="single_choice"
        syntax="string"
        any="optional|filter"
        i18nKey="o2">
        <ChoiceValues>
            <ChoiceValue>Active</ChoiceValue>
            <ChoiceValue>Inactive</ChoiceValue>
        </ChoiceValues>
    </AttributeSchema>
</SubSchema>
...

```

If the `type` attribute is not specified in `amEntrySpecific.xml`, the defaults will be used. A default setting means that only the name of the entry will display on the object function pages in the Identity Server console.

All the attributes listed in the schema definitions in the `amEntrySpecific.xml` file are displayed when the abstract type object pages are displayed. If the attribute is not listed in a schema definition in the `amEntrySpecific.xml` file, the Identity Server console will not display the attribute. For additional information on the Identity Server abstract objects and marker object classes, see the *Sun ONE Identity Server Installation and Configuration Guide*.

NOTE The User service is not configured in the `amEntrySpecific.xml` file but in its own `amUser.xml` file.

Management Sample Functions

Following are several samples that illustrate identity management functions using the Identity Server.

Create, Delete Or Modify Users

Users can be created, deleted or modified can be accomplished using the SDK. There is an interface that can be called for any SDK user creation, deletion, or modification. The property `com.ipplanet.am.sdk.userEntryProcessingImpl` should be set to the implementation in `AMConfig.properties`.

Code Example 5-2

```
public interface AMUserEntryProcessed {
    /**
     * Method which gets invoked whenever a user is created
     * @param token the SSOToken
     * @param userDN the DN of the user being added
     * @param attributes a map consisting of attribute names and
     * a set of values for each of them
     */
    public void processUserAdd(SSOToken token, String userDN,
        Map attributes);
}
```

Create Organization

The following code sample creates a new organization with one user by opening a connection to the Directory Server with `AMStoreConnection`. A new top organization (`newtoporg.com`) is then created with its own attributes. User John Smith is also created as a member of the new organization.

Code Example 5-3 Create New Organization And One User

```
...
    // instantiate a store connector from SSO Token
    AMStoreConnection amsc = new AMStoreConnection(ssoToken);
    // create a new top level organization without non-default
    attributes
    AMOrganization org =
    amsc.createTopOrganization("newtoporg.com", new HashMap());
```

Code Example 5-3 Create New Organization And One User *(Continued)*

```

        // set attribute for the newly created organization
        org.setStringAttribute("description", "organization
description");
        // save new attribute to the organization object
        org.store();

        // create new user "john" with "cn", "sn" attribute
        // Map to hold all users to be created, key is the string
value for user naming attribute,
        // value is a Map which contains all the initial values for
the user
        Map usersMap = new HashMap();
        // Map to hold attributes for the user
        Map attrsMap = new HashMap();
        // set cn = John Smith
        Set values = new HashSet();
        values.add("John Smith");
        attrsMap.put("cn", values);
        // set sn = Smith
        values = new HashSet();
        values.add("Smith");
        attrsMap.put("sn", values);
        // set put user john in the usersMap with "cn" & "sn"
specified above
        usersMap.put("john", attrsMap);
        // create user john in the organization
        Set users = org.createUsers(usersMap);
        ...

```

Retrieve Templates

The following code sample retrieves a service's dynamic templates by opening a connection to the Directory Server with `AMStoreConnection`. It retrieves a service's dynamic template by defining the DN of the top organization (`toporg.com`) as well as the string attribute of the specific service to be retrieved.

Code Example 5-4 Retrieve Service's Dynamic Template

```

...
        // instantiate a store connector from SSO Token
        AMStoreConnection amsc = new AMStoreConnection(ssoToken);
        // retrieve top level organization by DN
        AMOrganization org =
        amsc.getOrganization("o=toporg.com,o=isp");
        // retrieve Dynamic type AMTemplate for
iPlanetAMSessionService

```

Code Example 5-4 Retrieve Service's Dynamic Template *(Continued)*

```

    AMTemplate template =
    org.getTemplate("iPlanetAMSessionService",
    AMTemplate.DYNAMIC_TEMPLATE);
    // retrieve attributes
    String maxSessionTime =
    template.getStringAttribute("iplanet-am-session-max-session-time
    ");
    ...

```

Create Users With Modified LDAP Schema

There might be a need to modify the Directory Server schema in order to create users with non-default object classes. Here are the steps to create users with extended object classes:

1. Modify Directory Schema with the new set of attributes and object classes.
For more information on this function, see the Sun ONE Directory Server documentation.
2. Write a new XML service file which contains the definitions for the new object classes and attributes.
When writing this file, the object classes should be defined under the Global element and the attributes should be defined under the User element. More information can be found in Chapter 6, "Service Management."
3. Write a new module configuration properties file.
This file contains the key-value pairs for the internationalization keys used in the file created in Step 2. More information can be found in "Configuring Module Credential Requirements," on page 57 of Chapter 3, "Authentication Service."
4. Load the two files using the amadmin command line interface.
More information on this tool can be found in the Sun ONE Identity Server Administration Guide.
5. Restart the Directory Server and Identity Server.
6. Register the new service to the desired organization using the Console.

For getting more details about registering a new service, refer to the Sun ONE Identity Server Administration Guide.

7. Select the new service to create a user with the additional object classes.

When creating new user there is an option to select the newly configured service.

NOTE Instead of creating a new XML service file, `amUser.xml` can be modified. In this case, un-register the old `amUser` service, modify the file and re-register the modified service. Key-value pairs still need to be included in the `amUser.properties` file for newly defined internationalization keys. `ums.xml` does not need to be modified for this option.

Service Management

Sun™ One Identity Server uses eXtensible Markup Language (XML) files and Java™ interfaces for the integration and management of services into the Identity Server configuration. This chapter provides information on the structure of the XML files and the service management application programming interfaces (API). It contains the following sections:

- Overview
- Service Definition
- DTD Files
- XML Files
- Service Management SDK

Overview

A *service* is a group of attributes, defined in an XML file, that are managed together by the Identity Server console. The attributes can be the *configuration parameters* of a software module or they might just be related information with no connection to a software configuration. As an example of the first scenario, after creating a payroll module, a developer defines an XML service file that might include attributes to define an employee name, an hourly pay rate and a tax percentage. This file is imported into the Sun ONE Directory Server so the attributes and their values can be stored. When the service is registered to an organization, the attributes can be managed using the Identity Server console.

Identity Server provides the mechanisms for administrators to define, integrate and manage groups of attributes as an Identity Server service. Preparing a service for management involves creating an XML service file, configuring an LDAP Data Interchange Format (LDIF) file with any new object classes and importing both, the XML service file and the new LDIF schema, into the Directory Server.

Administrators can then register the service to identity objects using the Identity Server console. Once registered, the attributes can be managed and customized.

NOTE Throughout this chapter, the term *attribute* is used for two concepts. An Identity Server or service attribute refers to the configuration parameters of a defined service. An XML attribute refers to the parameters that qualify an XML element in the XML files.

XML Service Files

XML service files enable Identity Server to manage attributes that are stored in Directory Server. Identity Server does not implement any behavior or dynamically generate any code to interpret the attributes; it can only set or get the attribute values. Out-of-the-box, Identity Server loads a number of services to manage the attributes of its own features. For example, the Logging attributes are displayed and managed in the Identity Server console, while code implementations within the Identity Server use these configured attributes to record the operations of the application. All XML service files are located in `<identity_server_root>/SUNWam/config/xml`. For more specific information on XML service files, see “XML Files,” on page 150.

NOTE Any application with LDAP attributes can have this data managed using the Identity Server console by configuring a custom XML service file and loading it into the Directory Server. For more information, see “Service Definition,” on page 115.

Document Type Definition Structure Files

The format of an XML file in Identity Server is based on a structure defined in a DTD file. In general, a DTD file defines the elements and qualifying attributes needed to write a well-formed and valid XML document. Identity Server exposes the DTD files that are used to define the structure for different types of XML files. The DTDs are located in `<identity_server_root>/SUNWam/dtd`. Additional information on them can be found in “DTD Files,” on page 125.

NOTE Knowledge of XML is necessary to understand DTD elements and how they are integrated into Identity Server. When creating an XML file, it might be helpful to print out the relevant DTD and a corresponding sample XML file.

Service Management SDK

Identity Server also provides a service management SDK that provides application developers with interfaces to register and un-register services as well as manage their schema and configuration information. These interfaces are bundled in a package called `com.sun.identity.sm`. More information on the SDK can be found in “Service Management SDK,” on page 156.

Service Definition

To define a service for registration and management with the Identity Server the service developer must create an XML service file as well as configure an LDIF file with any object classes. Both, the XML service file and the new LDIF schema, must them be imported into the Directory Server. Once imported, the service can be registered by an administrator and its attributes managed and customized. The following sections describe the procedures to define and register a service.

Defining A Service

The following procedures must be completed in order to define a service and use the Identity Server to integrate and manage it.

1. Create an XML service file for the component.

This XML file must conform to the `sms.dtd`. A simple way to create a new XML service file would be to copy and modify an existing one. More information on creating an XML service file can be found in “Creating A Service File,” on page 117. The DTD syntax can be found in “The `sms.dtd` Structure,” on page 126.

2. Extend the LDAP schema in the Directory Server using `ldapmodify`, if necessary.

Loading an LDIF file into the Directory Server will add any new or modified object classes and attributes to the DIT. This step is only necessary when defining dynamic, policy and user attributes. (Global and organization attributes are stored in the Directory Server as XML, not LDAP.) Instructions on extending the LDAP schema can be found in “Extending The Directory Server Schema,” on page 120. See the Sun ONE Directory Server documentation for additional information.

3. Import the XML service file into Directory Server using `amadmin` using the `--schema` or `-S` option.

Information on importing an XML service file can be found in “Importing the XML Service File,” on page 122.

4. Configure a localization properties file and copy it into the `<identity_server_root>/SUNWam/locale` directory.

The localization properties file must be created with accurate `i18nKey` fields that map to names defined in the XML service file. If no localization properties file exists, Identity Server will display the actual attribute names. More information on the localization properties file can be found in “Configuring Localization Properties,” on page 123.

5. Update the `amEntrySpecific.xml` or `amUser.xml` files, if necessary.

The `amEntrySpecific.xml` file defines the attributes that will display on the Create, Properties and Search pages specific to each of the Identity Server *abstract objects*. The `amUser.xml` file can be modified to add User attributes to the User Service. (Alternately, User attributes can be defined in the actual XML service file in which case, `amUser.xml` would not need to be modified.) Information on updating `amEntrySpecific.xml` can be found in Chapter 5, “Identity Management.” Information on modifying `amUser.xml` can be found in “Modifying A Default XML Service File,” on page 151.

6. Register the service.

After importing the service into Directory Server, it can be registered and the attributes managed through the Identity Server console. Information on how this can be done is in the *Sun ONE Identity Server Administration Guide*. Information on how to register using the command line can be found in “Registering The Service,” on page 124.

Creating A Service File

The information in this section corresponds to Step 1, creating an XML service file. The XML service file defines the attributes of an Identity Server service. It must follow the structure defined in the `sms.dtd` which enforces the service developer to combine attributes into one of five groups, allowing the developer to differentiate between those attributes applicable to, for example, a service instance or a user.

Service File Naming Conventions

When creating a new XML service file, there are some naming conventions that must be followed.

- The name of a service (other than an authentication module service) as defined in the XML service file can be any string as long as it is unique.
- The name of an authentication module service as defined in the XML service file must be in the form `iPlanetAMAuthmodule_nameService`.)
- Any defined authentication level attribute must be configured as `iplanet-am-auth-module_name-auth-level`.

Service Attributes

The `sms.dtd` requires the service developer to define attributes into one of five groups. These groups differentiate between those attributes applicable to, for example, the Identity Server deployment, a service or a user.

Global Attributes

Global attributes are defined for the entire Identity Server installation and are common to all data trees, service instances and integrated applications within the configuration. Global attributes can not be applied to users, roles or organizations as their purpose is to configure the Identity Server itself. Server names, port numbers, service plug-ins, cache size, and maximum number of threads are examples of global attributes that are configured with one value. For example, when Identity Server performs logging functions, the log files are written into a directory. The location of this directory is defined as a global attribute in the Logging Service and all Identity Server logs, independent of their purpose, are written to it. Identity Server administrators can modify these default values using the Identity Server console. Global attributes are stored in the Directory Server as an XML *blob* within an attribute of an LDAP object. Therefore, the LDAP schema does not need to be extended to add a new global attribute.

NOTE If a service has only global attributes, it can not be registered to an organization nor can a service template be created.

Organization Attributes

Organization attributes are defined and assigned at the organization level. Attributes for an Authentication Service are a good example. When the Authentication Service is registered, attributes are configured depending on the organization to which it is registered. The `LDAP Server` and the `DN To Start User Search` would be defined at the organization level as this information would be different depending on the address of an organization's LDAP server and the structure of their DIT, respectively. Organization attributes are stored as an XML *blob* within an attribute of an LDAP object. Therefore, the LDAP schema does not need to be extended to add a new global attribute.

NOTE Organization attributes are not inherited by sub-organizations. Only dynamic and policy attributes can be inherited. For additional information, see "Attribute Inheritance," on page 120.

Dynamic Attributes

Dynamic attributes are *inheritable* attributes that work at the role and organization levels as well as the sub-organization and organizational unit levels. Services are assigned to organizations; roles have access to any service assigned to its parent organization. The dynamic attributes are then inherited by users that possess the role or belong to the organization. Because the attributes are assigned to roles or organizations instead of set in a user entry, they are *virtual* attributes inherited by users using the concept of *Class of Service* (CoS). When these attributes change, the administrator only has to change them once, in the role or organization, instead of a multitude of times in each user entry.

NOTE Dynamic attributes are modeled using *class of service* (CoS) and *roles*, both features of the Sun ONE Directory Server. For information on these features, see Appendix B, "Directory Server Concepts" or refer to the Sun ONE Directory Server documentation.

An example of a dynamic attribute might be the address of a common mail server. Typically, an entire building might have one mail server so each user would have a mail server attribute in their entry. If the mail server changed, every mail server attribute would have to be updated. If the attribute was in a role that each user in

the building possessed, only the attribute in the role would need to be updated. Another example might be the organization's address. Dynamic attributes are stored within the Directory Server as LDAP objects, making it feasible to use traditional LDAP tools to manage them. A Directory Server LDAP schema needs to be defined for these attributes.

Policy Attributes

Policy attributes are a special type of dynamic attribute. The main difference is that policy attributes provide a way to control resource access by defining a user's permissions. These defined permission attributes are then used to create *named policy*. For example, *allowURLList* is a named policy that defines a list of URLs a user is allowed to access; **.red.ipplanet.com*, **.eng.sun.com* are the permitted URLs defined as policy attributes. Named policies are assigned to roles or organizations; once assigned, the policy attribute is available in the user entry as an LDAP attribute, making it feasible to use traditional LDAP tools to manage them. (Named policies are not stored within the Directory Server as LDAP objects.) A Directory Server LDAP schema needs to be defined for these attributes.

NOTE Currently, Identity Server has only two services that use policy attributes: URL Policy Agent and URL Domain Access.

User Attributes

User attributes belong specifically to a single user. User attributes are not inherited from the role, organization, or sub-organization levels. They are typically different for each user, and any changes to them would affect only the particular user. Examples of user attributes could be an office telephone number, a password or an employee ID. The values of these attributes would be set in the user entry and not in a role or organization. User attributes can be a part of any service but, for convenience, Identity Server has grouped a number of the most widely-used attributes into a service defined by the `amUser.xml` service file. User attributes are stored within the Directory Server as LDAP objects, making it feasible to use traditional LDAP tools to manage them. A Directory Server LDAP schema needs to be defined for these attributes.

NOTE When defining user attributes in an XML service file other than `amUser.xml`, the service must be explicitly assigned to the user in order to display them on the User's Profile page. In addition, the User Profile Display Option in the Administration Service) must be set to `Combined`. For more information, see the *Sun ONE Identity Server Administration Guide*.

Attribute Inheritance

After creating and loading an XML service file, an administrator can assign the service's organization, dynamic and policy attributes by registering it to an identity object and creating a service template. (Any number of services can be assigned to these objects.) Then when a user possesses a role or belongs to an organization which possesses a service, the user inherits the dynamic and policy attributes or the organization, dynamic and policy attributes, respectively. Inheritance only occurs, though, if the service possessed is also explicitly assigned to the user. A user can inherit attributes from multiple roles or parent organizations.

NOTE Attributes defined as *User* have no inheritance; they are set and modified in each User entry. For example, if 70 attributes are defined as *User* and an organization has two million users, each attribute is stored two million times.

ContainerDefaultTemplateRole Attribute

Dynamic and policy attributes are used in an XML service file if an administrator wants to define a service in which all identity objects, with the specified service assigned to them, would inherit those attributes. After uploading the XML service file and assigning the service to an organization or role, all users in the sub-trees, with the specified service assigned to them, will inherit the dynamic and policy attributes. To accomplish this, Identity Server uses classic CoS (as described in Appendix B, "Directory Server Concepts") and role templates.

`ContainerDefaultTemplateRole` is a default *filtered* role configured for each organization. The filter is `objectClass=iplanet-am-managed-person`. Since every identity object in Identity Server carries this attribute, every identity in the organization possesses this role. Identity Server then creates a separate CoS template for each registered service which points to the service's default attributes. Any identity who has the role will then get all the dynamic and policy attributes.

Extending The Directory Server Schema

The information in this section corresponds to Step 2, extending the LDAP schema in the Directory Server. When configuring an XML service file for Identity Server, it might also be necessary to modify the Directory Server schema. First, any customized dynamic, policy or user attributes defined in an Identity Server service that are not already defined in the Directory Server schema need to be associated with an LDAP object class. Then the attribute(s) and object class(es) need to be added to the LDAP schema using `ldapmodify` and an LDIF file as input; thus, the Directory Server can store the data.

NOTE The order in which the LDAP schema is extended or the XML service file is loaded into Directory Server is not important.

1. Create an LDIF file to define any new or modified LDAP object classes and attributes.
2. Change to the Identity Server `bin` directory.

```
cd <identity_server_root>/SUNWam/bin
```

3. Run `ldapmodify` using the LDIF file as input. The syntax is `ldapmodify -D <userid_of_DSmanager> -w <password> -f <path_to_LDIF_file>`

By default, `<userid_of_DSmanager>` is `cn=Directory Manager`. If the LDIF was created correctly, the result of this command would be to modify the entry `cn=schema`.

NOTE After extending the schema using `ldapmodify`, it is not necessary to restart the Directory Server but, as `ldapmodify` is server-specific, the schema needs to be extended on all configured servers. Information on how this is done can be found in the Sun ONE Directory Server documentation.

4. Run `ldapsearch` to ensure that the schema has been created. The syntax is `ldapsearch -b "cn=schema" -s base -D <userid_of_DSmanager> -w <password> "(objectclass=*)" | grep -i "servicename"`

If the LDIF was created correctly, the result of this command would be a listing of the object classes as illustrated in Code Example 6-1 below.

Code Example 6-1 Sample LDIF Listing For Mail Service

```
objectClasses: ( 1.2.NEW
NAME 'iplanet-am-sample-mail-service'
DESC 'iPlanet SampleMail Service' SUP top AUXILIARY
MAY ( iplanet-am-sample-mail-service-status $
iplanet-am-sample-mail-root-folder $
iplanet-am-sample-mail-sentmessages-folder $
iplanet-am-sample-mail-indent-prefix $
iplanet-am-sample-mail-initial-headers $
iplanet-am-sample-mail-inactivity-interval $
iplanet-am-sample-mail-auto-load $
iplanet-am-sample-mail-headers-perpage $
iplanet-am-sample-mail-quota $
iplanet-am-sample-mail-max-attach-len $
iplanet-am-sample-mail-can-save-address-book-on-server )
X-ORIGIN 'user defined' )
```

Code Example 6-1 Sample LDIF Listing For Mail Service

```
attributeTypes: ( 11.24.1.996.1
  NAME 'iplanet-am-sample-mail-service-status'
  DESC 'iPlanet SampleMailService Attribute'
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
  X-ORIGIN 'user defined' )
```

Adding Object Classes To Existing Users

If a new service is created and the service's users already exist, the object classes need to be added to the user entries. In order to do this, Identity Server provides migration scripts for performing batch updates to user entries in the DIT. No LDIF file need be created when using them. These scripts are described in the *Sun ONE Identity Server Installation and Configuration Guide*. Alternatively, registered services can be added to the user by selecting the service from their Properties page.

NOTE To modify user entries using `ldapmodify`, an LDIF file needs to be created. For information, see the Sun ONE Directory Server documentation. (It is not recommended to use `ldapmodify` to create entries for Identity Server.)

Verifying The Directory Server Modification

To verify that the Directory Server has been populated correctly, an administrator can use `ldapsearch` or the following:

1. Change to the Directory Server install directory:

```
cd /<directory_server_root>/slapd-<directory_server_hostname>
```

2. Export the Directory Server contents into an LDIF file using

```
db2ldif -s <orgnamingattribute=top_level_org_name>
```

This command results in the name of a LDIF schema file stored under `<directory_server_root>/slapd-slapd-<directory_server_hostname>/ldif` which can be viewed to ensure that the new object classes have been created.

Importing the XML Service File

The information in this section corresponds to Step 3, importing an XML service file into the Identity Server.

1. Change to the Identity Server install directory:

```
cd <identity_server_root>/SUNWam/bin
```

2. Run following command line application: `./amadmin --runasdn <DN_of_directory_server_administrator> --password <password_directory_server_administrator> --verbose --schema <xml_service_file_path.>`

More information on the `amadmin` command line tool can be found in the *Sun ONE Identity Server Administration Guide*.

NOTE If changing an existing service, the original XML service file needs to be deleted before importing the modified XML service file.

Configuring Localization Properties

The information in this section corresponds to Step 4, configuring a localization properties file. A localization properties file specifies the locale-specific screen text and messages that an administrator or user will see when directed to a service's attribute configuration page. The files are located in the

`<identity_server_root>/SUNWam/locale/` directory. Code Example 6-2 is the localization properties file for Identity Server's sample mail service.

Code Example 6-2 Sample Mail Service Localization Properties File

```
...
iplanet-am-sample-mail-service-description=Sample Mail Service
Profile
a1=Mail Status
a2=Root Folder
a3=Sent Messages Folder
a4=Reply Prefix
a5=Initial Headers to Load
a6=Check New Mail Interval (minutes)
a7=Automatic Message Load at Disconnect
a8=Headers Per Page
p1=Mail Quota
p2=Auto-download Maximum Attachment Length
p3=Save Address Book on Server
```

The localization properties files consist of a series of key/value pairs. The value of each pair will be displayed on the service's Properties page in the Identity Server console. The keys (a1, a2, etc.) map to the `i18nKey` attribute fields defined for a service in the XML service file. The keys also determine the order in which the fields are displayed on screen, taken in alphabetical and then numerical order (a1, a2 is followed by b1, b2 and so forth). Note that the keys are strings, so a10 comes before a2.

NOTE If modifying a localization properties file, Identity Server needs to be restarted. If importing a localization properties file, Identity Server does not need to be restarted.

Identifying The Localization Properties File

Identity Server needs to be able to locate the localization properties file so it is located in the default `<identity_server_root>/SUNWam/locale` directory. If the file is kept in another directory, the `jvm.classpath=` entry in the `jvm12.conf` file needs to be modified to include the new directory path name.

NOTE If the `jvm12.conf` file is modified, the Identity Server server needs to be restarted.

Updating Files For Abstract Objects

For information corresponding to Step 5, updating the `amEntrySpecific.xml`, see Chapter 5, "Identity Management." For information corresponding to Step 5, updating the `amUser.xml`, see "XML Files," on page 150.

Registering The Service

The information in this section corresponds to Step 6, registering a new service to an identity object. The preferred way to register a service is to use the Identity Server console. Information on how this is done can be found in the *Sun ONE Identity Server Administration Guide*. Alternately, services can be registered using the `amadmin` command line executable.

1. Change to the Identity Server install directory:

```
cd <identity_server_root>/SUNWam/bin
```

2. Run following command line application `./amadmin --runasdn <DN_of_identity_server_administrator> --password <password_identity_server_administrator> --schema <xml_service_file_path.>`

More information on the `amadmin` command line tool can be found in the *Sun ONE Identity Server Administration Guide*.

DTD Files

Identity Server contains DTD files which are used to define the structure for XML files used within the Identity Server configuration. The DTDs are located in `<identity_server_root>/SUNWam/dtd` and include:

- `sms.dtd`—which defines the structure for XML service files. Information on this document can be found in “The `sms.dtd` Structure,” on page 126.
- `amAdmin.dtd`—which defines the structure for XML files used to perform batch LDAP operations on the directory tree using the command line tool `amAdmin`. Information on this document can be found in “The `amAdmin.dtd` Structure,” on page 135.
- `policy.dtd`—defines the structure for XML files used to define policies for servers and services. Information on this document can be found in Chapter 7, “Policy Service.”
- `Auth_Module_Properties.dtd`—defines the structure for XML files used by each authentication module to specify the properties for the Authentication Service interface. Information on this document can be found in Chapter 3, “Authentication Service.”
- `server-config.dtd`—defines the structure for `serverconfig.xml` which details ID, host and port information for all server and user types. Information on this document can be found in “The `amAdmin.dtd` Structure,” on page 135.

CAUTION None of these DTD files should be modified in any way. They contain rules and definitions that control how certain operations are performed and any alterations might hinder these operations.

The sms.dtd Structure

The `sms.dtd` defines the data structure for all XML service files. It is located in the `<identity_server_root>/SUNWam/dtd` directory. The `sms.dtd` enforces the developer to define each attributes as one of five schema types which are then stored and managed differently. For instance, some of the attributes are applicable to an entire Identity Server installation (such as a port number or server name), while others are applicable only to individual users (such as a password). The attribute types are:

- Global
- Organization
- Dynamic
- User
- Policy

An explanation of the elements defined by the `sms.dtd` follows. Each element includes a number of XML attributes which are also explained. Identity Server currently supports only about 20% of the elements contained in `sms.dtd`; this section discusses only those elements.

NOTE Customized attribute names in XML service files should be written in lower case as Identity Server converts all attribute names to lower case when reading from the Directory Server.

ServicesConfiguration Element

ServicesConfiguration is the root element of the XML service file. It's immediate sub-element is *Service*. Code Example 6-3 on page 126 illustrates the *ServicesConfiguration* element as defined in a file named `sampleMailService.xml`.

Code Example 6-3 ServicesConfiguration and Service Element

```

...
<ServicesConfiguration>
  <Service name="sampleMailService" version="1.0">
  <Schema...>
...

```

Service Element

The *Service* element defines the schema for a given service. Multiple services can be defined in a single XML file with this element, but it is recommended that only one be defined per XML service file. Currently, Identity Server supports the sub-element *Schema* which, in turn, defines Identity Server attributes as either *Global*, *Organization*, *Dynamic*, *User* or *Policy*. The required XML service attributes for the *Service* element are the name of the service, such as *iPlanetAMLoggingService*, and the version number of the XML service file itself. Code Example 6-3 on page 126 also illustrates the *Service* element and its attributes.

Schema Element

The *Schema* element is the parent of the elements that define the service's specific Identity Server attributes (global, organization, dynamic, user or policy) and their default values. The sub-elements can be *Global*, *Organization*, *Dynamic*, *User* or *Policy*. The required XML attributes of the *Schema* element include `serviceHierarchy` which defines where the service will be displayed in the Identity Server console, `i18nFileName` which defines the name of the localization properties file, and `i18nKey` which defines the attribute in the localization properties file from which this particular defined value will be taken.

NOTE The *Schema* element is required in all XML service files.

serviceHierarchy Attribute

When adding a service, this attribute must be defined in order to display the service in the Identity Server console. When a new service is registered, it is dynamically displayed based on this value. The value is a "/" separated string. Code Example 6-4 on page 127 illustrates the `serviceHierarchy` attribute as defined in the file named `sampleMailService.xml`. The name *sampleMailService* is used to find the localization properties file which defines what will be displayed below the Other Configuration header in the Identity Server console.

Code Example 6-4 `i18nFileName`, `i18nKey` and `serviceHierarchy` Attributes

```

...
<Schema
  serviceHierarchy="/other.configuration/sampleMailService"
    i18nFileName="sampleMailService"
    i18nKey="iplanet-am-sample-mail-service-description">
...

```

i18nFileName And i18nKey Attributes

These two XML attributes both refer to the localization properties files. The `i18nFileName` attribute takes a value equal to the name of the localization properties file for the defined service (minus the `.properties` file extension). The `i18nKey` is a text string that maps to a property value defined in the localization properties file (specified, as discussed, in the `i18nFileName` attribute.) For example, Code Example 6-4 on page 127 defines the name of the properties file as `sampleMailService` and the text-based value of the `i18nKey` maps to its final value as defined in `sampleMailService.properties`. The final value is the name of the service as it will be displayed in the Identity Server console; in this case, *Sample Mail Service Profile* is the name defined in `sampleMailService.properties`. More information on the localization properties file can be found in Chapter 3, “Authentication Service.”

Schema Sub-Elements

The next five elements are sub-elements of *Schema*; they are the declarations of the service’s Identity Server attributes. When defining a service, each attribute must be defined as one of these types: Global, Organization, Dynamic, Policy and User. Any configuration (all or none) of these elements can be used depending on the service. Each Identity Server attribute defined within these elements is itself defined by the sub-element `AttributeSchema`.

Global Element

The Global element defines Identity Server attributes that are modifiable on a platform-wide basis and applicable to all instances of the service in which they are defined. They can define information such as port number, cache size, or number of threads, but Global elements also define a service’s LDAP object classes. For additional information, see “Service File Naming Conventions,” on page 117.

serviceObjectClasses Attribute. The `serviceObjectClasses` attribute is a global attribute in each XML service file that contains dynamic or policy attributes. This optional attribute is used by the SDK to set the object class for the service in the user entries. When an organization registers a service with the `serviceObjectClasses` attribute defined, the service’s dynamic or policy attributes, if any exist, are automatically assigned to any user object which has been assigned the service.

CAUTION If the `serviceObjectClasses` attribute is not specified and the service has defined dynamic or policy attributes, an object class violation is called when an administrator tries to create a user under that organization.

Multiple values can be defined for the *serviceObjectClasses* attribute. For example, if a service is created with two attributes each from three different object classes, the *serviceObjectClasses* attribute would need to list all three object classes as *DefaultValues*. Code Example 6-5 illustrates a *serviceObjectClasses* attribute with two defined object classes.

Code Example 6-5 *serviceObjectClass* Defined As Global Element

```

...
<Global>
    <AttributeSchema name="serviceObjectClasses"
        type="list"
        syntax="string"
        i18nKey="">
        <DefaultValues>
            <Value>iplanet-am-sample-mail-service</Value>
            <Value>iplanet-am-other-sample-service</Value>
        </DefaultValues>
    </AttributeSchema>
</Global>
...

```

Organization Element

The Organization element defines Identity Server attributes that are modifiable per organization or sub-organization. For example, a web hosting environment using Identity Server would have different configuration data defined for each organization it hosts. A service developer would define different values for each organization attribute *per* organization. These attributes are only accessible using the Identity Server SDK. For additional information, see “Organization Attributes,” on page 118.

Dynamic Element

The Dynamic element defines Identity Server attributes that can be inherited by all user objects. Examples of Dynamic elements would be user-specific session attributes, a building number, or a company mailing address. Dynamic attributes always use the Directory Server features, CoS (Class Of Service) and Roles. For additional information, see “Dynamic Attributes,” on page 118.

User Element

The User element defines Identity Server attributes that exist physically in the user entry. User attributes are not inherited by roles or organizations. Examples include password and employee identification number. They are applied to a specific user only. For additional information, see “User Attributes,” on page 119.

Policy Element

The `Policy` element defines Identity Server attributes intended to provide privileges. This is the only attribute element that uses the `ActionSchema` element to define its parameters as opposed to the `AttributeSchema` element. Generally, privileges are `get`, `post`, and `put`; examples include `canChangeSalaryInformation` and `canForwardEmailAddress`. See Code Example 6-7 on page 133 for an example of a `Policy` schema definition from the `sampleMailService.xml` file. For additional information, see “Policy Attributes,” on page 119.

SubSchema Element

The `SubSchema` element can specify multiple sub-schemas of global information for different defined applications. For example, logging for a calendar application could be separated from logging for a mail service application. The required XML attributes of the `SubSchema` element include `name` which defines the name of the sub-schema, `inheritance` which defines whether this schema can be inherited by one or more nodes on the DIT and `maintainPriority` which defines whether priority is to be honored among its peer elements.

NOTE The `SubSchema` element is used only in the `amEntrySpecific.xml` file. It should not be used in any external XML service files.

AttributeSchema Element

The `AttributeSchema` element is a sub-element of the five schema elements discussed in “Schema Sub-Elements,” on page 128 as well as the `SubSchema` element described in “SubSchema Element,” on page 130. It defines the structure of each attribute. The sub-elements that qualify the `AttributeSchema` can include `IsOptional?`, `IsServiceIdentifier?`, `IsResourceNameAllowed?`, `IsStatusAttribute?`, `ChoiceValues?`, `BooleanValues?`, `DefaultValues?`, or `Condition`. The XML attributes that define each portion of the attribute value are `name`, `type`, `uitype`, `syntax`, `cosQualifier`, `rangeStart`, `rangeEnd`, `minValue`, `maxValue`, `validator`, `any`, and `%i18nIndex`. Code Example 6-6 on page 130 illustrates the `AttributeSchema` element, its attributes and their corresponding values. Note that this example attribute is a Dynamic attribute.

Code Example 6-6 AttributeSchema Element With Attributes

```

...
<Dynamic>
  <AttributeSchema name="iplanet-am-sample-mail-service-status"
    type="single_choice"
  >

```

Code Example 6-6 AttributeSchema Element With Attributes (*Continued*)

```

        syntax="string"
        i18nKey="a1">
        <ChoiceValues>
            <ChoiceValue>Active</ChoiceValue>
            <ChoiceValue>Inactive</ChoiceValue>
            <ChoiceValue>Deleted</ChoiceValue>
        </ChoiceValues>
        <DefaultValues>
            <Value>Active</Value>
        </DefaultValues>
    </AttributeSchema>
    ...

```

name Attribute

This required XML attribute defines the LDAP name for the attribute. Any string format can be used but attribute names must be in lower-case. Code Example 6-6 on page 130 defines it with a value of

`iplanet-am-sample-mail-service-status`.

type Attribute

This attribute specifies the kind of value the attribute will take. The default value for type is `list` but it can be defined as one of the following:

- `single` specifies that the user can define one value.
- `list` specifies that the user can define a list of values.
- `single_choice` specifies that the user can chose a single value from a list of options.
- `multiple_choice` specifies that the user can chose multiple values from a list of options.

ChoiceValues Sub-Element. If the `type` attribute is specified as either `single_choice` or `multiple_choice`, the `ChoiceValues` sub-element must also be defined in the `AttributeSchema` element. Depending on the type specified, the administrator or user would choose either one or more values from the choices defined. The possible choices are defined in the `ChoiceValue` element. Code Example 6-6 on page 130 defines the attribute type as `single_choice` so the `ChoiceValues` attribute defines the list of options as `Active`, `Inactive` and `Deleted`.

syntax Attribute

The `syntax` attribute defines the format of the value. The default value for `syntax` is `string` but, it can be defined as one of the following:

- `boolean` specifies that the value is either true or false.
- `string` specifies that the value can be any string.
- `password` specifies that user must enter a password, which will be encrypted.
- `dn` specifies that the value is a LDAP Distinguish Name.
- `email` specifies that the value is an email address.
- `url` specifies that the value is a URL address.
- `numeric` specifies that the value is a number.
- `percent` specifies that the value is a .
- `number` specifies that the value is a number.
- `decimal_number` specifies that the value is a number with a decimal point.
- `number_range` specifies that the value is a range of numbers.
- `decimal_range` specifies that the value is a range of numbers that might include a decimal figure.

DefaultValues Sub-Element. Defining any of these `syntax` values also necessitates defining a value for the `DefaultValue` sub-element. A default value will then be displayed in the Identity Server console; this default value can be changed for each organization when creating a new template for the service. For example, all instances of the LDAP Authentication Service use the `port` attribute so a default value of `389` could be defined in the XML service file. Once registered, this value can be modified for each organization using the Identity Server console. (The default value is also used by integrated applications when a service template has not been registered to an organization.) In the Code Example 6-7 on page 133, for example, the *Save Address Book On Server* field will display a default value of `false`. The user has the option to change the value to `true`, if desired. (The default value for `password` would be an encrypted password, generally the same as the one used for Identity Server.)

Code Example 6-7 sms.dtd:ActionSchema Element With Boolean Syntax

```

...
<AttributeSchema
name="iplanet-am-sample-mail-can-save-address-book-on-server"
    type="single"
    syntax="boolean"
    i18nKey="p3">
    <DefaultValues>
        <Value>>false</Value>
    </DefaultValues>
</ActionSchema>
...

```

cosQualifier Attribute

This attribute defines how Identity Server will resolve conflicting `cosQualifier` attributes assigned to the same user object. This value will appear as a qualifier to the `cosAttribute` in the LDAP entry of the CoS definition. It can be defined as:

- `default` indicates that if there are two conflicting `cosQualifier` attributes assigned to the same user object, the one with the lowest priority number (0) takes precedence. (The priority level is set in the `cosPriority` attribute when a new CoS template entry is created for an organization or role. For more information, see “Conflicts and CoS,” on page 298 of Chapter , “.”)
- `override` indicates that the CoS template value overrides any value already present in the user entry; that is, CoS takes precedence over the user entry value.
- `merge-schemes` indicates that if there are two CoS templates assigned to the same user, then they are merged so that the values are combined and the user gets an aggregation of the CoS templates.

NOTE The URL Policy Agent service uses `merge-schemes` to obtain aggregated values for the Allow and Deny attributes. For example, if the Employee Role allows access to `*/employee.html` and the HR Role allows access to `*/hr.html`, a user possessing both of these roles is allowed access to both.

If this attribute is not defined, the default behavior is for the user entry value to override the CoS value in the organization or role. The default value is `default`. (The `operational` value is reserved for future use.)

any Attribute

The *any* attribute specifies whether the attribute for which it is defined will display in the Identity Server console. It has six possible values that can be multiply defined using the “|” (pipe) construct:

- *display* specifies that the attribute will display on the user profile page. The attribute is read/write for administrators and regular users.
- *adminDisplay* specifies that the attribute will display on the user profile page. It will not appear on an end user page; the attribute is read/write for administrators only.
- *userReadOnly* specifies that the attribute is read/write for administrators but is read only for regular users. It is displayed on the user profile pages as a non-editable label for regular users.
- *required* specifies that a value for the attribute is required in order for the object to be created. The attribute will display on the Create page with an asterisk.
- *optional* specifies that a value for the attribute is not required in order for the object to be created.
- *filter* specifies that the attribute will display on the Search page.

The *required* or *optional* keywords and the *filter* and *display* keyword can be specified with a pipe symbol separating the options (*any=required|display* or *any=optional|display|filter*). If the *any* attribute is set to *display*, the qualified attribute will display in Identity Server console when the properties for the Create page are displayed. If the *any* attribute is set to *required*, an asterisk will display in that attribute’s field, thus the administrator or user is required to enter a value for the object to be created in Identity Server console. If the *any* attribute is set to *optional*, it will display on the Create page, but users are not required to enter a value in order for the object to be created. If the *any* attribute is set to *filter*, the qualified attribute will display as a criteria attribute when Search is clicked from the User page.

%i18nIndex Attribute (i18nKey)

The *i18nKey* attribute, as defined in “*i18nFileName And i18nKey Attributes*,” on page 128, is referenced as an entity in the `sms.dtd`.

NOTE If the *i18nKey* value is blank (that is, “”), the Identity Server console will not display the attribute.

The amAdmin.dtd Structure

The `amAdmin.dtd` defines the data structure for all XML files which will be used to perform batch LDAP operations on the DIT using `amAdmin`. It is located in the `<identity_server_root>/SUNWam/dtd` directory. The command line operations include reads and gets on the attributes as well as creations and deletions of user objects (roles, organizations, users, people containers, and groups). The following sections discuss the elements and attributes of the `amAdmin.dtd` as well as the sample XML templates installed with Identity Server that use this structure. These samples can be found in `<identity_server_root>/SUNWam/samples/admin/cli/bulk-ops` and will be used to illustrate these sections.

Requests Element

The *Requests* element is the root element of the batch processing XML file. It must contain at least one child element which defines the Identity Server identity objects (Organization, Container, People Container, Role and Group) onto which the actual requests are performed. To enable batch processing, the root element can take more than one set of requests. The *Requests* element must contain at least one of the following sub-elements:

- `OrganizationRequests`
- `ContainerRequests`
- `PeopleContainerRequests`
- `RoleRequests`
- `GroupRequests`
- `SchemaRequests`
- `ServiceConfigurationRequests`

Based on the defined request, the corresponding Identity Server API will be called to perform the operation.

OrganizationRequests Element

The *OrganizationRequests* element consists of all requests that can be performed on Organization objects. The required XML attribute for this element is the LDAP Distinguished Name (DN) of the organization on which all of the sub-element requests will be performed. This element can have one or more sub-elements which perform their operations on the defined instance of the Organization object.

(Different *OrganizationRequests* elements can be defined in one document to modify more than one Organization DN.) Code Example 6-8 on page 140 defines a myriad of objects to be created from the top level organization, `o=isp`. The sub-elements of *OrganizationRequests* are:

- `CreateSubOrganization`
- `CreatePeopleContainer`
- `CreateRole`
- `CreateGroup`
- `CreatePolicy`
- `AssignPolicy`
- `UnAssignPolicy`
- `CreateServiceTemplate`
- `ModifySubOrganization`
- `ModifyServiceTemplate`
- `DeleteServiceTemplate`
- `ModifyPeopleContainer`
- `ModifyRole`
- `ModifyGroup`
- `ModifyPolicy`
- `GetSubOrganizations`
- `GetPeopleContainers`
- `GetRoles`
- `GetGroups`
- `GetUsers`
- `RegisterServices`
- `UnregisterServices`
- `GetRegisteredServiceNames`
- `GetNumberOfServices`
- `DeleteRoles`

- DeleteGroups
- DeletePolicy
- DeletePeopleContainers
- DeleteSubOrganizations

ContainerRequests Element

The *ContainerRequests* element consists of all requests that can be performed on Container objects. The required XML attribute for this element is the DN of the container on which the sub-element requests will be performed. This element can have one or more sub-elements which perform their operations on the same instance of the container. (Different *ContainerRequests* elements can be defined in one document to modify more than one Container DN.) Code Example 6-8 on page 140 illustrates how this element can be modeled. The sub-elements of *ContainerRequests* are:

- CreateSubContainer
- CreatePeopleContainer
- CreateRole
- CreateGroup
- CreatePolicy
- AssignPolicy
- UnAssignPolicy
- CreateServiceTemplate
- ModifyServiceTemplate
- ModifySubContainer
- ModifyPeopleContainer
- ModifyRole
- GetSubContainers
- GetPeopleContainers
- GetRoles
- GetGroups
- GetUsers

- RegisterServices
- UnregisterServices
- GetRegisteredServiceNames
- GetNumberOfServices
- DeleteRoles
- DeleteGroups
- DeletePolicy
- DeletePeopleContainers
- DeleteSubContainers

PeopleContainerRequests Element

The *PeopleContainerRequests* element consists of all requests that can be performed on People Container objects. The required XML attribute for this element is the DN of the container on which the sub-element requests will be performed. This element can have one or more sub-elements which perform their operations on the same instance of the people container. (Different *PeopleContainerRequests* elements can be defined in one document to modify more than one People Container DN.) Code Example 6-8 on page 140 illustrates how this element can be modeled. The sub-elements of *PeopleContainerRequests* are:

- CreateSubPeopleContainer
- ModifyPeopleContainer
- CreateUser
- ModifyUser
- GetNumberOfUsers
- GetUsers
- GetSubPeopleContainers
- DeleteUsers
- DeleteSubPeopleContainers

RoleRequests Element

The *RoleRequests* element consists of all requests that can be performed on roles. The required XML attribute for this element is the DN of the role on which the sub-element requests will be performed. This element can have one or more sub-elements which perform their operations on the same instance of the role. (Different *RoleRequests* elements can be defined in one document to modify more than one Role DN.) Code Example 6-8 on page 140 illustrates how this element can be modeled. The sub-elements of *RoleRequests* are:

- `CreateServiceTemplate`
- `ModifyServiceTemplate`
- `AssignPolicy`
- `UnAssignPolicy`
- `GetNumberOfUsers`
- `GetUsers`
- `AddUsers`

GroupRequests Element

The *GroupRequests* element consists of all requests that can be performed on group objects. The required XML attribute for this element is the DN of the group on which the sub-element requests will be performed. This element can have one or more sub-elements which perform their operations on the same instance of the group. (Different *GroupRequests* elements can be defined in one document to modify more than one Group DN.) Code Example 6-8 on page 140 illustrates how this element can be modeled. The sub-elements of *GroupRequests* are:

- `CreateSubGroup`
- `GetSubGroups`
- `GetNumberOfUsers`
- `GetUsers`
- `AddUsers`
- `DeleteSubGroups`

AttributeValuePair Element

The *AttributeValuePair* element can be a sub-element of many of the following batch processing requests. It can have two sub-elements, neither of which can themselves have sub-elements. The *Attribute* sub-element must be empty while the *Value* sub-element takes a default value to display in the Identity Server console. The *Attribute* sub-element takes a required XML attribute called `name`. The value of `name` is the attribute name which is equal to one string without spaces; no sub-elements are allowed. Code Example 6-13 on page 144 illustrates how an attribute/value pair would be added to a sub-organization.

Create<Object> Elements

The *CreateSubOrganization*, *CreateUser*, *CreateGroup*, *CreateSubContainer*, *CreatePeopleContainer*, *CreateSubGroup*, *CreateSubPeopleContainer* and *CreateRole* elements create a sub-organization, user, group, sub-container, people container, sub-group, sub-people container and role, respectively. The object is created in the DN that is defined in the second-level *<Object>Requests* element under which the *Create<Object>* element is defined. *AttributeValuePair* may be defined as a sub-element (or not). The required XML attribute for each element is `createdDN`; it takes the DN of the object to be created. Code Example 6-8 on page 140 illustrates an example of some of these elements.

Code Example 6-8 Portion of Batch Processing File `createRequests.xml`

```

...
<Requests>
<OrganizationRequests DN="dc=example,dc=com">

    <CreateSubOrganization
createdDN="o=suborg,dc=example,dc=com" />
    <CreatePeopleContainer
createdDN="ou=People,dc=example,dc=com" />
    <CreateRole createdDN="cn=ManagerRole,dc=example,dc=com" />
    <CreateRole createdDN="cn=EmployeeRole,dc=example,dc=com" />
    <CreateGroup
createdDN="ou=ContractorsGroup,dc=example,dc=com" />
    <CreateGroup createdDN="ou=EmployeesGroup,dc=example,dc=com" />

</Requests>
...

```

CreatePolicy Element

The *CreatePolicy* element creates one or more policy attributes. The *Policy* sub-element defines the named policy. The required XML attribute is `createdDN` which takes the DN of the organization where the policy will be created. This and the following nested elements are all illustrated in Code Example 6-9 on page 141.

Policy Element. The *Policy* sub-element defines the permissions or *rules* of the policy. It can take one or more of the *Rule* sub-elements. The required XML attribute is `name` which specifies the name of the policy. The `serviceName` attribute, which identifies the service to which the named policy applies, is an optional XML attribute.

Rule Element. The *Rule* sub-element defines a specific permission of the policy. *Rule* can take three sub-elements. The required XML attribute is `name` which defines a name for the rule. The three sub-elements are:

- **ServiceName Element**

The *ServiceName* sub-element defines the service for which a rule has been created. There are no sub-elements; the *ServiceName* element itself must be empty. The required XML attribute is `name` which takes a string value.

- **ResourceName Element**

The *ResourceName* sub-element defines the domain for which this permission is being defined. There are no sub-elements; the *ResourceName* element itself must be empty. The required XML attribute is `name` which takes a string value.

- **AttributeValuePair Element**

The *AttributeValuePair* sub-element defines the action names and corresponding action values of the rule. For additional information, see “Delete<Object> Elements,” on page 142.

Code Example 6-9 Portion of Batch Processing File `createPolicyOrg.xml`

```

...
<Requests>
<OrganizationRequests DN="o=isp">

<CreatePolicy createdDN="o=example.com,o=isp">
  <Policy name="urlpolicy" serviceName="iPlanetAMWebAgentService">
    <Rule name="Manager Rule">
      <ServiceName name="iPlanetAMWebAgentService"/>
      <ResourceName name="*.example.com"/>
      <AttributeValuePair>
        <Attribute name="permission"/>
      <Value>iplanet-am-web-agent-access-allow-list</Value>
    </AttributeValuePair>
  </Policy>
</CreatePolicy>

```

Code Example 6-9 Portion of Batch Processing File createPolicyOrg.xml

```

        </Rule>
        <Rule name="engManager Rule">
            <ServiceName name="iPlanetAMWebAgentService"/>
            <ResourceName name="*.example.com"/>
            <AttributeValuePair>
                <Attribute name="permission"/>
            <Value>iplanet-am-web-agent-access-allow-list</Value>
            </AttributeValuePair>
        </Rule>
    </Policy>
</CreatePolicy>
</OrganizationRequests>
</Requests>
...

```

CreateServiceTemplate Element

The *CreateServiceTemplate* element creates a service template for the organization defined in the second-level *Requests* element. There are no sub-elements; the *CreateServiceTemplate* element itself must be empty. The required XML attribute is *serviceName* which takes a string value. Code Example 6-10 on page 142 illustrates a service template being created for *sun.com*.

Code Example 6-10 Portion of Batch Processing File createServiceTemplates.xml

```

...
<Requests>
<OrganizationRequests DN="o=example.com,o=isp">

    <CreateServiceTemplate serviceName="sampleMailService"/>

</OrganizationRequests>
</Requests>
...

```

Delete<Object> Elements

The *DeleteSubOrganizations*, *DeleteUsers*, *DeleteGroups*, *DeleteSubContainers*, *DeletePeopleContainers*, *DeleteSubGroups*, *DeleteSubPeopleContainers*, and *DeleteRoles* elements delete a sub-organization, user, group, sub-container, people container, sub-group, sub-people container and role, respectively. The object is deleted from the DN that is defined in the second-level *<Object>Requests* element under which the *Delete<Object>* element is defined. *DeleteSubOrganizations*, *DeleteUsers*, *DeleteGroups*, *DeleteSubContainers*, *DeletePeopleContainers*, *DeleteSubGroups*, *DeleteSubPeopleContainers* and *DeleteRoles* take a sub-element *DN*; only six of the

listed elements have the XML attribute *deleteRecursively*. (*DeleteUsers* and *DeleteRoles* do not have this option; they have no qualifying XML attribute.) If *deleteRecursively* is set to *false*, accidental deletion of all sub-trees can be avoided; its default value is *false*. The DN sub-element takes a character value equal to the DN of the object to be deleted. Code Example 6-11 on page 143 illustrates an example of some of these elements.

Code Example 6-11 Portion of Batch Processing File `deleteOrgRequests.xml`

```

...
<Requests>
<OrganizationRequests DN="o=isp">

  <DeleteRoles>
    <DN>cn=ManagerRole,o=example.com,o=isp</DN>
    <DN>cn=EmployeeRole,o=example.com,o=isp</DN>
  </DeleteRoles>

  <DeleteGroups deleteRecursively="true">
    <DN>cn=EmployeesGroup,o=example.com,o=isp</DN>
    <DN>cn=ContractorsGroup,o=example.com,o=isp</DN>
  </DeleteGroups>

  <DeletePeopleContainers deleteRecursively="true">
    <DN>ou=People1,o=example.com,o=isp</DN>
  </DeletePeopleContainers>

  <DeleteSubOrganizations deleteRecursively="true">
    <DN>o=example.com,o=isp</DN>
  </DeleteSubOrganizations>

</OrganizationRequests>
</Requests>
...

```

DeletePolicy Element

The *DeletePolicy* element takes the sub-element *PolicyName*. The *PolicyName* element has no sub-elements; it must be empty. It has a required XML attribute *name* which takes a character value equal to the name of the policy. The *DeletePolicy* element itself takes a required XML attribute: `deleteDN`. It takes a value equal to the DN of the policy to be deleted.

DeleteServiceTemplate Element

The *DeleteServiceTemplate* element deletes the specified service template. There are no sub-elements; the *DeleteServiceTemplate* element itself must be empty. The required XML attributes are `serviceName` which takes a string value and `schemaType` which defines the attribute group (Global, Organization, Dynamic, User or Policy). Code Example 6-12 on page 144 illustrates how this element is formatted.

Code Example 6-12 Portion of Batch Processing File `deleteServiceTemplates.xml`

```

...
<Requests>
<OrganizationRequests DN="o=example.com,o=isp">
  <DeleteServiceTemplate
    serviceName="iPlanetAMAuthLDAPService"
    schemaType="organization">

    </DeleteServiceTemplate>
</OrganizationRequests>
</Requests>

```

Modify<Object> Elements

The *ModifyPeopleContainer*, *ModifySubContainer*, *ModifySubOrganization* and *ModifyRole*, *ModifyGroup* elements change the specified object. *AttributeValuePair* can be defined as a sub-element of the first four listed elements. (The *ModifyGroup* element can have no sub-elements; it must be empty.) The required XML attribute is `modifyDN` which takes the DN of the object to be modified. Code Example 6-13 on page 144 illustrates how these elements can be modeled.

Code Example 6-13 Portion of Batch Processing File `modifyRequests1.xml`

```

<Requests>
<OrganizationRequests DN="o=isp">
  <ModifySubOrganization modifyDN="o=sun.com,o=isp">
    <AttributeValuePair>
      <Attribute name="Description"/>
      <Value>DSAME Modify</Value>
    </AttributeValuePair>
  </ModifySubOrganization>

  <ModifyPeopleContainer modifyDN="ou=People,o=example.com">
    <AttributeValuePair>
      <Attribute name="Description"/>
      <Value>DSAME Modify</Value>

```

Code Example 6-13 Portion of Batch Processing File `modifyRequests1.xml`

```

    </AttributeValuePair>
  </ModifyPeopleContainer>

  <ModifyRole modifyDN="cn=ManagerRole,o=example.com">
    <AttributeValuePair>
      <Attribute name="iplanet-am-role-description"/>
      <Value>DSAME Modify</Value>
    </AttributeValuePair>
  </ModifyRole>

</OrganizationRequests>
</Requests>

```

ModifyServiceTemplate Element

The *ModifyServiceTemplate* element changes a specified service template.

AttributeValuePair must be defined as a sub-element of *ModifyServiceTemplate* to change the values. The required XML attribute is `serviceName` which takes a string value and `schemaType`. Code Example 6-14 on page 145 illustrates this element.

Code Example 6-14 Portion of Batch Processing File `modifyServiceTemplates.xml`

```

...
<Requests>
  <OrganizationRequests DN="o=example.com,o=isp">

    <ModifyServiceTemplate serviceName="sampleMailService">
      <AttributeValuePair>
        <Attribute
name="iplanet-am-sample-mail-sentmessages-folder"/>
        <Value>Hello Mail Sent</Value>
      </AttributeValuePair>
    </ModifyServiceTemplate>
  </OrganizationRequests>
</Requests>

```

Get<Object> Elements

The *GetSubOrganizations*, *GetUsers*, *GetGroups*, *GetSubContainers*, *GetPeopleContainers* and *GetRoles* elements get the specified object. A DN may be defined as a sub-element (or not). If none is specified, ALL of the specified objects at all levels within the organization defined in the second-level *Requests* element will be returned. The required XML attribute for all but *GetGroups* and *GetRoles* is `DNSOnly` and takes a `true` or `false` value. The required XML attribute of *GetGroups*

and *GetRoles* is `level` which takes a value of either `ONE_LEVEL` or `SUB_TREE`. `ONE_LEVEL` will retrieve just the groups at that node level; `SUB_TREE` gets groups at the node level and all those underneath it. Code Example 6-15 on page 146 illustrates how these elements can be modeled.

DNs Only Attribute

For all objects using the `DNsOnly` attribute, the *Get* elements work as stated below:

- If the element has the required XML attribute `DNsOnly` set to *true* and no sub-element DN is specified, only the DNs of the objects asked for will be returned.
- If the element has the required XML attribute `DNsOnly` set to *false* and no sub-element DN is specified, the entire object (a DN with attribute/value pairs) will be returned.
- If sub-element DNs are specified, the entire object will always be returned whether the required XML attribute `DNsOnly` is set to *true* or *false*.

Code Example 6-15 Portion of Batch Processing File `getRequests.xml`

```

...
<Requests>

  <OrganizationRequests DN="o=isp">

    <GetSubOrganizations DNsOnly="false">
      <DN>o=example1.com,o=isp</DN>
      <DN>o=example2.com,o=isp</DN>
    </GetSubOrganizations>

    <GetPeopleContainers DNsOnly="false">
      <DN>ou=People,o=example1.com,o=isp</DN>
      <DN>ou=People,o=example2.com,o=isp</DN>
    </GetPeopleContainers>

    <GetRoles level="SUB_TREE"/>

    <GetGroups level="SUB_TREE"/>

    <GetUsers DNsOnly="false">
      <DN>cn=puser,ou=People,o=example1.com,o=isp</DN>
    </GetUsers>

  </OrganizationRequests>

  ...

```

GetService Elements

The *GetRegisteredServiceNames* and *GetNumberOfServices* elements retrieve registered services and total number of registered services, respectively. The organization from which this information is retrieved is specified in the *OrganizationRequests* element. All three elements have no sub-elements or attributes; the elements themselves must be empty. Code Example 6-16 on page 147 illustrates how the *GetRegisteredServiceNames* element is modeled.

Code Example 6-16 Batch Processing File `getRegisteredServiceNames.xml`

```
...
<Requests>

  <OrganizationRequests DN="o=example.com,o=isp">
    <GetRegisteredServiceNames/>
  </OrganizationRequests>

</Requests>
```

ActionServices Elements

The *RegisterServices* and *UnregisterServices* elements perform the requested action on the service defined in the *OrganizationRequests* element. All elements take a sub-element *Service_Name* but have no XML attribute. The *Service_Name* element takes a character value equal to the name of the service. One or more *Service_Name* sub-elements can be specified.

Service Action Caveats

- The XML service file for the service must be loaded using the command line interface `amadmin` before a service can be acted upon.
- If no *Service_Name* element is specified or, in the case of *UnregisterServices*, the service was not previously registered, the request is ignored.
- If no *Service_Name* element is specified, the request will be ignored.

Code Example 6-17 on page 147 illustrates how these elements can be modeled.

Code Example 6-17 Portion of Batch Processing File `registerRequests.xml`

```
...
<Requests>
  <OrganizationRequests DN="o=example.com,o=isp">

    <RegisterServices>
      <Service_Name>sampleMailService</Service_Name>
    </RegisterServices>
```

Code Example 6-17 Portion of Batch Processing File `registerRequests.xml`

```

...
<Requests>

</OrganizationRequests>
</Requests>

```

AssignPolicy and UnAssignPolicy Elements

The *AssignPolicy* and *UnAssignPolicy* elements take the sub-element *PolicyName*. The *PolicyName* element has no sub-elements; it must be empty. It has a required XML attribute *name* which takes a character value equal to the name of the policy. The required XML attribute of *AssignPolicy* and *UnAssignPolicy* is `policyDN` which takes a value equal to the DN of the policy to be acted upon.

SchemaRequests Element

The *SchemaRequests* element consists of all requests that can be performed on the default values of the DSAME schema. It has two required XML attributes: *serviceName* and *SchemaType*. *serviceName* takes a value equal to the name of the service where the schema lives and *SchemaType* defines the attribute group (Global, Organization, Dynamic, User or Policy). This element can have zero or more sub-elements. The sub-elements of *SchemaRequests* are:

- `RemoveDefaultValues` Element
- `ModifyDefaultValues` Element
- `AddDefaultValues` Element
- `GetServiceDefaultValues`

RemoveDefaultValues Element

The *RemoveDefaultValues* element removes the default values from the schema specified in the parent *SchemaRequests* element. It takes a sub-element of *Attribute* which specifies the name of the attribute to be removed. The *Attribute* sub-element itself must be empty; it takes no sub-element. There is no required XML attribute.

Code Example 6-18 Portion of Batch Processing File `removeschemaRequests.xml`

```

...
<Requests>
<SchemaRequests serviceName="iPlanetAMUserService"
  SchemaType="dynamic">
<RemoveDefaultValues>
  <Attribute name="preferredlanguage"/>

```

Code Example 6-18 Portion of Batch Processing File `removeschemaRequests.xml`

```

...
<Requests>
  </RemoveDefaultValues>
</SchemaRequests>
</Requests>

```

AddDefaultValues and ModifyDefaultValues Elements

The *AddDefaultValues* and *ModifyDefaultValues* elements add or change the default values from the specified schema, respectively. They take a sub-element of *AttributeValuePair* which specifies the name of the attribute and the new default value; one or more attribute/value pairs can be defined. Code Example 6-19 on page 149 illustrates how this element can be modeled.

Code Example 6-19 Portion of Batch Processing File `addschemaRequests.xml`

```

...
<Requests>
  <SchemaRequests serviceName="iPlanetAMUserService"
    SchemaType="dynamic">
    <AddDefaultValues>
      <AttributeValuePair>
        <Attribute name="iplanet-am-user-auth-modules"/>
        <Value>Cert</Value>
      </AttributeValuePair>
    </AddDefaultValues>
  </SchemaRequests>
</Requests>

```

GetServiceDefaultValues Element

The *GetServiceDefaultValues* element retrieves the default values from the schema specified in the parent *SchemaRequests* element. There are no sub-elements; the *GetServiceDefaultValues* element itself must be empty. There is also no required XML attribute.

ServiceConfigurationRequests Element

The *ServiceConfigurationRequests* element is reserved for future use.

XML Files

Identity Server uses XML files to manage service attributes as well as perform service operations on service attributes. It does not implement any behavior or dynamically generate any code to interpret the attributes; it can only set or get attribute values. In addition to XML files that define service attributes, Identity Server also includes XML templates that can be used for batch processing. This section contains information on these types of XML files.

Default XML Service Files

Identity Server installs services that manage the attributes of its internal software components. The Identity Server console manages the attributes for these services; in addition, Identity Server provides code implementations to use them. These default XML service files are based on the `sms.dtd` and are located in `<identity_server_root>/SUNWam/config/xml`. They include:

- `amAdminConsole.xml`—Defines attributes for the Administration service.
- `amAuth.xml`—Defines attributes for the Core Authentication service.
- `amAuthAnonymous.xml`—Defines attributes for the Anonymous Authentication service.
- `amAuthCert.xml`—Defines attributes for the Certificate-based Authentication service.
- `amAuthLDAP.xml`—Defines attributes for the LDAP Authentication service.
- `amAuthRadius.xml`—Defines attributes for the Radius Authentication service.
- `amAuthSafeWord.xml`—Defines attributes for the SafeWord Authentication service.
- `amAuthSecurID.xml`—Defines attributes for the SecurID Authentication service.
- `amAuthUnix.xml`—Defines attributes for the Unix Authentication service.
- `amClientDetection.xml`—Defines attributes for the Client Detection service.
- `amDomainURLAccess.xml`—Defines attributes for the URL Access Policy service.
- `amDSS.xml`—Defines attributes for the Certificate Security service.

- `amEntrySpecific.xml`—Defines attributes for the displaying attributes on the Create, Properties and Search pages for a custom service.
- `amLogging.xml`—Defines attributes for the Logging service.
- `amMembership.xml`—Defines attributes for the Membership Authentication service.
- `amNaming.xml`—Defines attributes for the Naming service.
- `amPlatform.xml`—Defines attributes for the Platform service.
- `amPolicy.xml`—Defines attributes for the Policy service.
- `amSAML.xml`—Defines attributes for the SAML service.
- `amSession.xml`—Defines attributes for the Session service.
- `amUser.xml`—Defines attributes for the User service.
- `amWebAgent.xml`—Defines attributes for the web agents.

Modifying A Default XML Service File

Administrators can display and manage any attribute in the Identity Server console using XML service files. The new attribute(s) would need to be added to an existing XML service file. Alternately, they can be grouped into a new service by creating a new XML service file although the simplest way to add an attribute is just to extend an existing XML service file. For example, an administrator wants to manage the `nsaccountlock` attribute which will give users the option of locking the account it defines. To manage it through Identity Server, `nsaccountlock` must be described in a service. One option would be to add it to the `amUser.xml` service, `iPlanetAMUserService`. This is the service that, by default, includes many common attributes from the `inetOrgPerson` and `inetUser` object classes. Following is an example of how to add the `nsaccountlock` attribute to the `amUser.xml` service file.

1. Add the following code to the `SubSchema name=User` element in `<identity_server_root>/SUNWam/config/xml/amUser.xml`.

Code Example 6-20 `nsaccountlock` Example Attribute

```

...
<AttributeSchema name="nsaccountlock"
type="single_choice"
syntax="string"
any="filter"
isChangeableByUser="yes"
i18nKey="u13">
<ChoiceValues>

```

Code Example 6-20 nsaccountlock Example Attribute (Continued)

```

...
    <Value>true</Value>
    <Value>>false</Value>
</ChoiceValues>
<DefaultValues>
    <Value>>false</Value>
</DefaultValues>
</AttributeSchema>
...

```

2. Update the `<identity_server_root>/SUNWam/locale/en_US/amUser.properties` file with the new `i18nKey` tag `u13` including the text to be used for display.

Code Example 6-21 User Account Locked Example `i18nKey`

```

...
u13=User Account Locked
...

```

3. Remove the service `ou=iPlanetAMUserService,ou=services,dc=sun,dc=com` using the command line tool `amadmin`.
For information on the `amadmin` command line syntax, see *Sun ONE Identity Server Administration Guide*.
4. Reload the user service, `amUser.xml`, using the command line tool `amadmin`.
For information on the `amadmin` command line syntax, see *Sun ONE Identity Server Administration Guide*.

NOTE When modifying a default XML service file, be sure to also modify the Directory Server by extending the LDAP schema, if necessary. For more information, see “Service Definition,” on page 115.

Batch Processing XML Files

The `--data` or `-t` option of `amadmin` is used to perform batch processing using the command line. Batch processing XML templates have been installed and can be used to help an administrator to:

- Create, delete and read roles, users, organizations, groups, people containers and services.
- Get roles, people containers, and users.
- Get the number of users for groups, people containers, and roles.
- Import, register and un-register services.
- Get registered service names or the total number of registered services for an existing organization.
- Execute requests in multiple XML files.

The preferred way to perform most of these functions singularly is to use the Identity Server console. The batch processing templates have been provided for ease of use with bulk updates although they can also be used for single configuration updates. This section provides an overview of the batch processing templates which can be modified to perform batch updates on user objects (groups, users, roles, people containers, etc.) in the Directory Server.

NOTE Only XML files can be used as input for the `amadmin` tool. If an administrator wants to populate the DIT with user objects, or perform batch reads (gets) or deletes on the DIT, then the necessary XML input files, based on the `amadmin.dtd` or `sms.dtd`, must be written.

Batch Processing XML Templates

All of the batch processing XML files perform operations on the DIT; they create, delete, or get attribute information on user objects. The batch processing XML templates provided with Identity Server include:

- `ContCreateServiceTemplate.xml`—Creates a service template for a specific container object.
- `ContModifyRequests1.xml`—Adds new attributes for a sub-container object.
- `ContModifyRequests2.xml`—Adds new attributes for a people container object.
- `ContModifyRequests3.xml`—Adds new attributes for a sub-container object.

- `ContModifyRequests4.xml`—Adds new attributes to a role object.
- `ContassignPolicyRequests.xml`—Assigns policy to a specific container object.
- `ContunassignPolicyRequests.xml`—Removes an assigned policy from a specific container object.
- `PCModifyRequests1.xml`—Adds new attributes to a people container object.
- `PCModifyUserRequests.xml`—Adds new attributes to users in a people container object.
- `RoleCreateServiceTemplates.xml`—Creates a service template for a role object.
- `RoleassignPolicyRequests.xml`—Assigns policy to a role object.
- `RolemodifyServiceTemplates.xml`—Adds new attributes to a service template for a specific role object.
- `RoleunassignPolicyRequests.xml`—Removes policy from a specific role object.
- `addChoiceValuesRequest.xml`—Adds a selection of values the user can chose from to an existing service attribute.
- `addschemaRequests.xml`—Adds a default value to an existing service attribute.
- `addserviceConfigurationRequests.xml`—This is reserved for future use.
- `createPolicyOrg.xml`—Creates policy for an organization object.
- `createRequests.xml`—Creates a multitude of objects in the DS.
- `createServiceTemplates.xml`—Creates a service template for an organization object.
- `deleteGroupRequests.xml`—Deletes all objects under a specific group container.
- `deleteOrgRequests.xml`—Deletes a multitude of objects under a specific organization.
- `deletePCRequests.xml`—Deletes a multitude of objects under a specific people container.
- `deleteServiceTemplates.xml`—Deletes a service template under a specific organization.

- `deleteserviceConfigurationRequests.xml`—This is reserved for future use.
- `getNumOfServices.xml`—Passes a listing of an organization's total number of registered services.
- `getRegisteredServices.xml`—Passes a listing of an organization's registered services.
- `getRequests.xml`—Passes information about a multitude of objects in a specific organization.
- `modifyRequests1.xml`—Adds new attributes to a number of objects in a specific organization.
- `modifyRequests2.xml`—Adds new attributes to a people container object in a specific organization.
- `modifyRequests3.xml`—Adds new attributes to a role object in a specific organization.
- `modifyServiceTemplates.xml`—Modifies existing attributes in a service registered to a specific organization.
- `modifyschemaRequests.xml`—Adds new attributes to a number of objects in a specific organization.
- `registerRequests.xml`—Registers a service to an existing organization. (This service must have been previously imported.)
- `removeChoiceValueRequests.xml`—Removes the values a user can choose from in an existing attribute in a specific service.
- `removeschemaRequests.xml`—Removes the default value of an existing attribute in a specific service.
- `unassignPolicyRequests.xml`—Removes an assigned policy from a specific organization.
- `unregisterRequests.xml`—Unregisters a service from an existing organization. (This service must have been previously imported and registered.)

These XML templates follow the structure defined by the `amAdmin.dtd`. They are located in `<identity_server_root>/SUNWam/samples/admin/cli/bulk-ops`.

Modifying A Batch Processing XML Template

Any of the templates discussed above can be modified to best suit the desired operation. Choose the file that performs the request, modify the elements and attributes according to the service and use the `amadmin` executable to upload the changes to the Directory Server.

NOTE Be aware that creations of roles, groups, and organizations is a time-intensive operation.

Customizing User Pages

The User profile page and what attributes it displays will vary, depending on what the service developer defines. By default, every attribute in the `amUser.xml` file that has an `il8nKey` attribute specified and the `any` attribute set to `display` (`any=display`) will display in the Identity Server console. Alternately, if an attribute is specified to be of type `User` in another XML service file, the Identity Server console will also display it if the service is assigned to the user. Thus, User display pages in the Identity Server console can be modified to add new attributes in either of two ways:

- The `User` attribute schema definition in the specific XML service file can be modified.
- A new `User` schema attribute definition can be added to the User service (the `amUser.xml` service file).

For information on modifying XML service files, see “Modifying A Default XML Service File,” on page 151.

NOTE Any service can describe an attribute that is for a user only. The `amUser.xml` file is just the default placeholder for user attributes that are not tied to a particular service.

Service Management SDK

The Identity Server provides a Java API for service management. These interfaces can be used by developers to register services and applications, and manage their configuration data.

NOTE The Identity Server Javadocs can be accessed from any browser by copying the complete `<identity_server_root>/SUNWam/docs/` directory into the `<identity_server_root>/SUNWam/public_html` directory and pointing the browser to `http://<server_name.domain_name>:<port>/docs/index.html`.

Policy Service

The Sun™ One Identity Server includes a Policy Service that allows for the configuration and support of conditional policies for authorization and access control. It allows administrators to configure and administer these policies for applications, resources, and identities managed within the Identity Server deployment. This chapter explains the Policy Service and its architecture. It contains the following sections:

- What Is Policy?
- Policy Definition Type Document
- Java SDK For Policy
- C Library For Policy

What Is Policy?

A *policy* defines access control rules for an Identity Server deployment. These rules allow an administrator to assign security levels based on an organization's needs, and the conditions created within the policy, by assigning them to identities, groups or roles. This policy, when possessed by an object, defines which resources the object is able to access. A single policy can define either binary or non-binary decisions. A binary decision is *yes/no*, *true/false* or *allow/deny*; most policies are of this type. A binary decision might answer such questions as "Can user Mark execute the `changeSalary` method in the `PayCheck` class?" or "Can user Sally have access to the `PayCheck` application at all?" A non-binary decision represents the value of an attribute; for example, a mail service might include a `mailboxQuota` attribute with a maximum storage value set for each user. In general, a policy is configured to define what an object can do to which resource and under what conditions.

Policy Service

The Identity Server Policy Service provides for creation, administration and assignment of conditional policies. It allows administrators to define, modify, grant, revoke and delete policies for resources within the Identity Server deployment as well as query the Sun ONE Directory Server for stored policies.

Typically, a policy service includes a policy data store, a policy enforcer or *policy agent* and a library of interfaces that allows for the creation, administration and evaluation of policy. The components that are accessed when using the Identity Server Policy Service include:

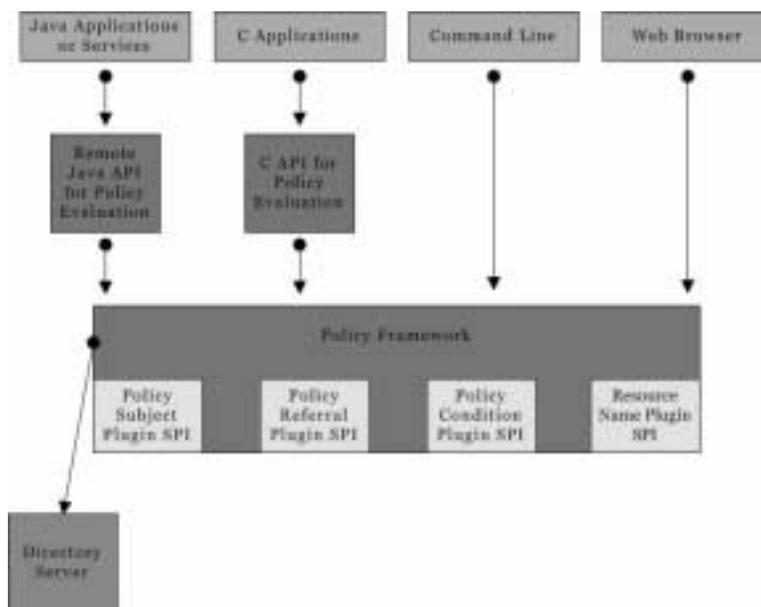
- Directory Server—is the data store which delivers an identity's authentication, and policy information. Additional information on this product and its functionality can be found in the Sun ONE Directory Server documentation.
- Identity Server—implements the Policy Service by providing policy administration and evaluation APIs. It also provides policy evaluation APIs written in C. Other Identity Server services accessed include authentication, session, logging and SSO.
- Policy Agent—is a Policy Enforcement Point (PEP) that protects an enterprise's resources on the remote web server on which the agent is installed. Policy agents are provided under separate cover from the Identity Server. The most current version of the Sun ONE Identity Server Policy Agent Pack can be downloaded from the iPlanet Developers Download Center located at <http://www.sun.com/software/download/developer/>. They are installed on proprietary web servers, remote from the Identity Server deployment web server(s). For example, an agent on a Human Resources web server would prevent personnel without the proper policies from viewing confidential salary information and other sensitive data. In order to make this example work, an Identity Server administrator must set up the policies that allow or deny users access to the remote web server's content.

NOTE Installing and administrating the policy agents is not in the scope of this documentation. Information on the currently available agents can be found in the *Sun ONE Identity Server Policy Agent Pack 1.1 Installation Guide* at http://docs.sun.com/?p=/coll/S1_s1IdServPolicyAgentPack_11.

Architecture

The Identity Server Policy Service allows for the protection of all types of applications and resources. Currently, though, only URL policy agents that protect an organization's web resources are available. Figure 7-1 illustrates the architecture of the Policy Service. As shown, custom agents or applications can be written to protect other types of resources including services or other applications.

Figure 7-1 Identity Server Policy Service Architecture



The architectural flow for protected web resources begins when a web browser requests a URL that resides on a protected web server; the web server's URL policy agent intercepts the request and checks for existing authentication credentials (an SSO token). If none exists or the existing authentication level or conditions are insufficient, the request is redirected to the Authentication Service. Once the user session is created or upgraded with a successful authentication, Identity Server responds to the browser request with a redirect to the original resource. The agent now finds a sufficient SSO token and issues a request to the Naming Service. (The Naming Service defines URLs for remote web servers to use for access to Identity Server's internal services.) The Naming Service returns locators for the Policy

Service which will check the user's policy, and for the Session Service which will begin the user's session upon authentication. Based on the aggregate of all policies assigned to the user, the individual is either allowed or denied access to the protected resource.

Policy Types

There are two types of policy that can be configured using Identity Server: a *normal* policy or a *referral* policy. A normal policy consists of *rules*, *subjects* and *conditions*. A referral policy consists of *rules* and *referrals* to organizations. These policy types are discussed below followed by the section "Policy Definition Type Document," on page 164 which expands on the terms.

Normal Policy

In Identity Server, a policy that defines access permissions is referred to as a *normal* policy. A normal policy consists of *rules*, *subjects* and *conditions*. A rule consists of a *resource*, and one or more sets of an *action* and a *value*. A resource defines the object that is being protected; an action is the name of an operation that can be performed on the resource and a value defines the permission.

NOTE It is acceptable to define an action without resources.

A subject defines who the policy affects. A condition defines the situations in which a policy is applicable; for instance, a 7 am to 10 am condition in a policy means that the policy is applicable only from 7 am to 10 am.

NOTE The terms referral, rule, resource, subject, condition, action and value correspond to the elements *Referral*, *Rule*, *ResourceName*, *Subject*, *Condition*, *Attribute* and *Value* in the `policy.dtd`. They are explained further in "Policy Definition Type Document," on page 164.

Referral Policy

A referral policy is used for policy delegation. If there is a top-level organization with a sub-organization in the Identity Server tree, there must be a referral policy configured at the top-level organization that points to the sub-organization, in effect, allowing the sub-organization to create normal policies. A *referral* policy controls this delegation for both policy creation and evaluation. It consists of one or more *rules* and one or more *referrals*. A rule defines the resource whose policy

creation or evaluation is being referred, while the referral defines the organization to which the policy creation or evaluation is being referred. For example, in the creation of policies for the sub-organization, the referral policy is configured at the top-level organization which states that the sub-organization can define policies for the resource whose URL is defined in the rule of the referral policy. Thus, the top-level organization is delegating policy creation and evaluation for the defined URL resource to the sub-level organization.

NOTE The referred-to organization can define or evaluate policies only for those resources (or sub-resources of those resources) that have been referred to it. This restriction, however, does not apply to the root organization. Therefore, an administrator must define management policies at the root level organization only.

There are two types of referral bundled with Identity Server: peer organization and sub-organization. They delegate to an organization on the same level and an organization on a sub-level, respectively. For example, consider a deployment whose root level organization is `dc=isp,dc=com` with sub-organizations `dc=sunone,dc=isp,dc=com` and `dc=suntwo,dc=isp,dc=com`. In order to define or evaluate policies at `dc=sunone,dc=isp,dc=com` or `dc=suntwo,dc=isp,dc=com`, two referral policies must first be created that point from `dc=isp,dc=com` to `dc=sunone,dc=isp,dc=com` and `dc=suntwo,dc=isp,dc=com`, respectively. Each referral policy contains the resource (or resource prefix) being managed. If `dc=sunone,dc=isp,dc=com` manages `http://www.sunone.com/`, the referral policy at `dc=isp,dc=com` contains `http://www.sunone.com/` in its rule and refers to policies created at the `dc=sunone,dc=isp,dc=com` organization. Only after creating root level referral policies can policies at the sub-organization be created.

Subjects

Policies are not explicitly assigned to identities rather, we assign *subjects* to policies. A subject is the identity object to which the policy is assigned and applied. The default subjects are:

- Identity Server Roles
- LDAP Groups
- LDAP Roles
- LDAP Users
- Organization

Policy Definition Type Document

Policy in Identity Server is most often configured using the Identity Server console. Information on how this is done can be found in the *Sun ONE Identity Server Administration Guide*. There is, however, a command line interface that can also be used for this purpose. (Information on how to use the `amadmin` interface can be found in *Sun ONE Identity Server Administration Guide*.) The `policy.dtd` defines the structure on which all policy XML files processed using the `amadmin` command line must be based. This section describes this structure.

Policy Element

Policy is the root element that defines the policy rule as an entity. It may contain one or more of the following sub-elements: *Rule*, *Conditions*, *Subjects*, or *Referrals*. The XML service attributes for the *Policy* element are the name of the policy, a description, a version number, and whether the policy type is referral or not.

NOTE When tagging a policy as *referral*, any Subjects and Conditions are ignored during policy evaluation. Conversely, when tagging a policy as *normal*, any Referrals are ignored during policy evaluation.

Rule Element

The *Rule* element defines the specifics of the policy rule. It defines the type of service or application for which a policy has been created as well as the name of the resource and the actions which are performed on the resource. It may contain one or more of the following elements: *ServiceName*, *ResourceName*, or *AttributeValuePair*. The XML service attribute for the *Rule* element is the name of the rule.

NOTE It is acceptable to have a defined policy that does not include a *ResourceName*.

ServiceName Element

The *ServiceName* element defines the name of the service that the policy applies to. This element represents the service type. It contains no other elements. The value is exactly as that defined in the service's XML file (which is based on the `sms.dtd`). The XML service attribute for the *ServiceName* element is the name of the service. Examples of a *ServiceName* might be Calendar Service, Mail Service or PayCheck application.

ResourceName Element

The *ResourceName* element defines the object that will be acted upon. The policy has been specifically configured to protect this object. It contains no other elements. The XML service attribute for the *ResourceName* element is the name of the object. Examples of a *ResourceName* might be `http://www.sunone.com:8080/images` on a web server or `ldap://sunone.com:389/dc=iplanet,dc=com` on a directory server. A more specific resource might be `salary://uid=jsmith,ou=people,dc=iplanet,dc=com` where the object being acted upon is the salary information of John Smith.

NOTE Currently, Identity Server 6.0 provides only web agents.

AttributeValuePair Element

The *AttributeValuePair* element defines an action and its values. It is used as a sub-element to *Subject Element*, *Referral Element* and *Condition Element*. It contains both the *Attribute* and *Value* elements and no XML service attributes.

Attribute Element

The *Attribute* element defines the name of the action. An action is an operation or event that is performed on a resource. POST or GET are actions performed on web server resources, READ or SEARCH are actions performed on directory server resources and `purchaseOptions` or `canUpdateCatalog` might be actions performed on a catalog service. The *Attribute* element must be paired with a *Value* element as described in the following section. The *Attribute* element itself contains no other elements. The XML service attribute for the *Attribute* element is the name of the action.

Value Element

The *Value* element defines the action itself. Allow/deny or yes/no are actions. Other action values can be either boolean, numeric, or strings. The value is defined in the service's XML file (based on the `sms.dtd`). The *Value* element contains no other elements and it contains no XML service attributes.

Subjects Element

The *Subjects* is the root element that defines a collection of *Subject* elements. The *Subjects* element contains one or more *Subject* elements and the name and description XML service attributes for each one. (The `includeType` attribute is not supported in this release.)

Subject Element

The *Subject* element identifies a collection of identities to whom the policy applies. For example, the action `canUpdateCatalog` for a catalog service might only be implemented by identities who possess the Marketing role. The *Subject* element contains the *AttributeValuePair* element and the name and type XML service attributes.

Referrals Element

The *Referrals* element defines a collection of referral elements. The *Referrals* element contains one or more *Referral* elements. The XML service attributes are the name and the description of the referral grouping.

Referral Element

The *Referral* element defines another identity to which the policy evaluation is delegated. The *Referral* element contains the *AttributeValuePair* element. The XML service attributes are the name of the referral and type.

Conditions Element

The *Conditions* element defines a collection of conditions, i.e. a group of restrictions on the defined policy. The *Conditions* element contains the one or more *Condition* elements. The XML service attributes are the name and description of the restriction grouping.

Condition Element

The *Condition* element defines a condition which specifies when a policy will be effective. For example, authentication level and time span might restrict a subject's access. The *Condition* element contains the *AttributeValuePair* element. It's XML service attributes are the condition name and type.

Java SDK For Policy

The crux of the Policy Service is the Policy SDK. It exposes the following Java API packages:

- `com.sun.identity.policy` provides the APIs that applications and services use to determine privileges. It is used by the Administration Console and/or the command line interface to manage, administer and evaluate policies.
- `com.sun.identity.policy.interfaces` provides the APIs that administrators use to manage policies and add plug-ins.
- `com.sun.identity.policy.client` provides the APIs that agents on a remote server use to evaluate policy.

Identity Server also has a C API library to allow C developers to integrate their applications with the Policy Service. In the following sections, a sampling of classes and the methods included with these API are discussed.

NOTE The Identity Server Javadocs can be accessed from any browser by copying the complete `<identity_server_root>/SUNWam/docs/` directory into the `<identity_server_root>/SUNWam/public_html` directory and pointing the browser to `http://<server_name.domain_name>:<port>/docs/index.html`.

Policy Evaluation Java APIs

The following APIs are used by Java developers to allow for the evaluation of policy privileges in their applications. This functionality is provided by the class `com.sun.identity.policy.client.PolicyEvaluator`, which provides support for both boolean and non-boolean type policies. A `PolicyEvaluator` must be created by calling the constructor with a service name. Public methods of this class include:

- `isAllowed`—evaluates the policy associated with the given resource and returns a boolean value indicating whether the policy evaluation resulted in an allow or deny.
 - Returns a boolean value of:
 - `true` if access is allowed.
 - `false` if access is denied.
 - Arguments:
 - `com.iplanet.sso.SSOToken`: The SSO Token associated with the principal for which the policy will be evaluated.
 - `java.lang.String resourceName`: A string representing the requested resource.
 - `java.lang.String actionName`: The action for which the policy will be evaluated. In a typical web application scenario, the action could be GET or POST.
 - `java.util.Map envParameters`: A map containing environment parameters that may be needed to successfully evaluate the associated policies.
 - Exceptions: throws `com.iplanet.sso.SSOException` if the given SSO token is not valid or has expired.
- `getPolicyDecision`—evaluates the policy and ascertains privileges for non-boolean decisions. It returns a decision that gives a user permission to perform a specific action on a specific resource. This method can also check permissions for multiple actions.
 - Returns `com.sun.identity.policy.PolicyDecision`.
 - Arguments:
 - `com.iplanet.sso.SSOToken`: The SSO token associated with the principal for which the policy will be evaluated.

- `java.lang.String resourceName`: A string representing the requested resource.
 - `java.util.Set actionName`: A collection of actions for which the policy will be evaluated.
 - `java.util.Map envParameters`: A map containing environment parameters that may be needed to successfully evaluate the associated policies.
- **Exceptions:** throws `com.iplanet.sso.SSOException` if the given SSO token is not valid or expired.

Policy Management Java APIs

The following APIs are used by systems administrators to allow for the management of policies in the Identity Server. The interfaces for this functionality are found in the `com.sun.identity.policy` package.

PolicyManager

`com.sun.identity.policy.PolicyManager` is the top level administrator class for policy management, providing methods that allow an administrator to create, modify or delete an organization's policies. The `PolicyManager` can be obtained from a specified organization or by checking a currently authenticated user's `SSOToken` for access privileges to the Directory Server. Some of this class' more widely used methods include:

- `getPolicyNames`—retrieves all named policies created for the organization for which the policy manager was instantiated. This method can also take a pattern (filter) as an argument.
- `getPolicy`—retrieves a policy when given the policy's name.
- `addPolicy`—adds a policy to the specified organization. If a policy with the same name already exists, it will be overwritten.
- `removePolicy`—removes a policy from the specified organization.

Policy

`com.sun.identity.policy.Policy` represents a policy definition with all its intended parts (rules, subjects, referrals or conditions). The policy object is saved in the data store only when the `store` method is called or if the `addPolicy` or `replacePolicy` methods from the `PolicyManager` class are invoked. This class contains methods to add, remove, replace or get any of the parts of a policy definition.

PolicyEvent

`com.sun.identity.policy.PolicyEvent` represents a happening in a policy that could potentially change the current access status. For example, `PolicyEvent` would be created and passed if a policy has been removed due to a timeout. This class works with the `PolicyListener` class in the `com.sun.identity.policy.interface` package.

Policy Plugin Java APIs

The following APIs are used by service developers and policy administrators who need to provide additional policy features as well as support for legacy policies. The package for these classes is `com.sun.identity.policy.interfaces`. The interfaces include:

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.

Subject—defines methods that can determine if an authenticated user (possessing an `SSOToken`) is a member of the given subject.

Referral—defines methods used to delegate the policy definition or evaluation of a selected resource (and its sub-resources) to another organization or policy server.

Condition—provides methods used to constrain a policy; for example, time of day or IP address. This interface allows the pluggable implementation of the conditions.

PolicyListener—defines an interface that allows the Policy Service to send and receive notifications when a policy is added, removed or changed.

C Library For Policy

Identity Server also provides a library of policy evaluation APIs for C applications to enable their integration into the Policy Service. The C library provides a comprehensive set of interfaces that query policy results for an authenticated user for a given action on a given resource. The result of the policy evaluation is called an *action value* and may not always be binary (allow/deny or yes/no); action values can also be non-boolean. For example, John Smith has a mailbox quota of 100MB. 100 is the value defined by a policy on a policy server. As policy evaluation results in string values only, the policy evaluation returned is 100 numeric not 100MB. It is up to the application developer to define metrics for the values obtained appropriately.

As the first step of policy implementation, the API abstracts how a resource is represented by mandating that any resource be represented in a string format. For example, on a web server, resources may be represented as URLs. The policy evaluation engine cares only about the relative relevance of one resource to other. There are five relative relevances defined between two resources, namely: *exact match*, *no match*, *subordinate match*, *superior match* or *exact pattern match*. Having represented the resources in string format, the service developer must provide interfaces that establish the relevant relationship between resources.

NOTE *Exact pattern match* is a special case where resources may be represented collectively as patterns. The information is abstracted from the Policy Service and the comparison operation must take a boolean parameter to trigger a pattern matched comparison. During the caching of policy information, the policy engine does not care about patterns, whereas during policy evaluation, the comparisons are pattern sensitive.

The service developer must also provide a method to extract the root of the given resource. For example, in a URL, the `protocol://server_name:port/` portion represents the root. The three functions (`has_patterns`, `get_resource_root` and `compare_urls`) are specializations of resource representations. The set of characteristics needed to define a resource is called a *resource trait*. Resource traits are taken as a parameter during service initialization in the `am_resource_traits_t` structure. Using the resource traits, the Policy Service constructs a resource graph for policy evaluation. In a web server policy sense, the relation between all the resources in the system spans out like a tree with the `protocol://server_name:port/` being the root of the tree.

NOTE The policy management system is generic and makes no assumptions about any particular policy definition requirement.

C APIs for Policy Evaluation

Two opaque data structures are defined: `am_map_t` and `am_properties_t`. `am_map_t` provides a key to multiple value mapping and `am_properties_t` provides a key to single value mapping. `am_properties_t` provides the additional functionality of loading a configuration file and getting values of specific data types. These are simple data structures that are only used for information exchange to and from the policy evaluation interfaces.

`am_map_t`

This data structure is an associative container with a key of type `const char *` and having multiple values of type `const char *`.

`am_map_create(am_map_t *map_ptr)`

Syntax

```
#include <am_map.h>

am_status_t am_map_create( am_map_t *map_ptr);
```

Parameters

This function takes the following parameters:

`map_ptr` Pointer to the `am_map_t` structure. This is an out parameter.

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the map structure was successfully created and assigned to `map_ptr`.
- `AM_NO_MEMORY` if there was an internal memory operation error.
- `AM_INVALID_ARGUMENT` if the pointer address of `map_ptr` was invalid.

Description

This function creates an instance of `am_map_t` structure and returns the pointer to the structure to the caller.

Memory Concerns

You should free the allocated structure by calling `am_map_destroy`.

```
am_map_copy(am_map_t source_map, am_map_t *map_ptr)
```

Syntax

```
#include <am_map.h>
```

```
am_status_t am_map_copy(am_map_t source_map, am_map_t *map_ptr);
```

Parameters

This function takes the following parameters:

<code>map_ptr</code>	After successful execution of this function, this pointer will be assigned a new instance of <code>am_map_t</code> structure and all the entries in <code>source_map</code> will be copied into this structure.
----------------------	---

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the copy operation was successfully performed.
- `AM_NO_MEMORY` if there was an internal memory operation error.
- `AM_INVALID_ARGUMENT` if the address of `map_ptr` or `source_map` is invalid.

Description

This function creates an instance of `am_map_t` structure, copies all the elements in `source_map` into the newly created structure and assigns it to `map_ptr`. It does not alter the contents of `source_map`.

Memory Concerns

The caller must make sure not to pass a `map_ptr` which as a valid `am_map_t` structure, otherwise the reference will be lost. The must destroy `map_ptr` after usage by calling `am_map_destroy`.

```
am_map_destroy(am_map_t *map_ptr)
```

Syntax

```
#include <am_map.h>
```

```
am_status_t am_map_destroy(am_map_t *map_ptr);
```

Parameters

This function takes the following parameters:

<code>map_ptr</code>	The map structure to be destroyed.
----------------------	------------------------------------

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the destroy operation was successfully performed.
- `AM_NO_MEMORY` if there was an internal memory operation error.
- `AM_INVALID_ARGUMENT` if the address of `map_ptr` or `source_map` is invalid.

Description

This function destroys an instance of `am_map_t` structure which is pointed by `map_ptr`.

Memory Concerns

Care must be taken that `map_ptr` was not freed before by calling `am_map_destroy` or by erroneously calling the system void free (void *) function.

```
am_map_clear(am_map_t map)
```

Syntax

```
#include <am_map.h>

am_status_t am_map_clear(am_map_t map);
```

Parameters

This function takes the following parameters:

<code>map</code>	The map structure in which all the keys and their values be removed.
------------------	--

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the destroy operation was successfully performed.
- `AM_INVALID_ARGUMENT` if the map argument is NULL.

Description

This function takes in a valid `am_map_t` structure and clears all the elements in it. After successful completion of this function `am_map_size` on this structure will return 0.

Memory Concerns

None.

am_map_size(const am_map_t map)

Syntax

```
#include <am_map.h>

size_t am_map_size(const am_map_t map);
```

Parameters

This function takes the following parameters:

map	The map whose size to be returned.
-----	------------------------------------

Returns

This function returns one of the following values:

- 0 or a positive number The number of key value pairs currently in the map.
- <0 if the map argument is NULL.

Description

This function takes in a valid `am_map_t` structure and returns its size.

Memory Concerns

None.

am_map_get_entries(am_map_t map, am_map_entry_iter_t *entry_iter_ptr)

Syntax

```
#include <am_map.h>

am_status_t am_map_get_entries(am_map_t map, am_map_entry_iter_t
*entry_iter_ptr);
```

Parameters

This function takes the following parameters:

map	The map for which iterator needs to be extracted.
entry_iter_ptr	The iterator pointer that must be assigned with the iterator of the map structure. This is an output parameter.

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the iterator was successfully assigned.
- `AM_NO_MEMORY` if there was an internal memory operation error.
- `AM_INVALID_ARGUMENT` if the address of `map_ptr` or `source_map` is invalid.
- `AM_NOT_FOUND` if the specified map contains no keys.

Description

This function extracts an iterator pointer that could be used to iterate over the key value pairs stored in this table.

Memory Concerns

The iterator pointer passed in must not have non destroyed iterators assigned to them. The caller, in future must call `am_map_entry_iter_destroy` to destroy the iterator instance.

```
am_map_insert(am_map_t map, const char *key, const char *value,
int replace)
```

Syntax

```
#include <am_map.h>
```

```
am_status_t am_map_insert(am_map_t map, const char *key, const char
*value, int replace);
```

Parameters

This function takes the following parameters:

<code>map</code>	The map to which the key-value pair must be added.
<code>key</code>	The key for the entry.
<code>value</code>	The value for the entry.
<code>replace</code>	Boolean to indicate whether to replace an existing value for the key or not.

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the operation is successful.
- `AM_INVALID_ARGUMENT` if the map, key or value argument, is NULL.
- `AM_NO_MEMORY` if there was an internal memory operation error.

Description

This function inserts a key-value pair into a map.

Memory Concerns

None.

```
am_map_erase(am_map_t map, const char *key)
```

Syntax

```
#include <am_map.h>

am_status_t am_map_erase(am_map_t map, const char *key);
```

Parameters

This function takes the following parameters:

map	The map to which the key-value pair must be added.
key	The key for the entry.

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the operation is successful.
- `AM_INVALID_ARGUMENT` if the map or key argument, is NULL.
- `AM_NOT_FOUND` if the specified key is not in the map.

Description

This function removes a key-value pair from a map.

Memory Concerns

None.

```
am_map_find(am_map_t map, const char *key,
am_map_value_iter_t *value_iter_ptr)
```

Syntax

```
#include <am_map.h>

am_status_t am_map_find(am_map_t map, const char *key,
am_map_value_iter_t *value_iter_ptr);
```

Parameters

This function takes the following parameters:

<code>map</code>	The map to which the key-value pair must be added.
<code>key</code>	The key for the entry.
<code>value_iter_ptr</code>	The iterator to which value iterator has to be assigned.

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the operation is successful.
- `AM_INVALID_ARGUMENT` if the map or key argument, is NULL.
- `AM_NOT_FOUND` if the specified key is not in the map.
- `AM_NO_MEMORY` if there was an internal memory operation error.

Description

This function takes a key and returns an iterator that iterates over the values associated with the key.

Memory Concerns

At the end of usage of `value_iter_ptr`, the caller must call `am_map_value_iter_destroy` with the iterators pointer.

```
am_map_find_first_value(am_map_t map, const char *key)
```

Syntax

```
#include <am_map.h>
```

```
const char *am_map_find_first_value(am_map_t map, const char *key);
```

Parameters

This function takes the following parameters:

<code>map</code>	The map to which the key-value pair must be added.
<code>key</code>	The key for the entry.

Returns

This function returns one of the following values:

- value if the operation is successful, returns the first associated value of this key in the map. The order of insertion does not guarantee the value returned.
- NULL if there is key is not present in the map.

Description

This function takes a key and returns the first value associated with the key.

Memory Concerns

Caller must not modify or free the return value.

```
am_map_entry_iter_destroy(am_map_entry_iter_t entry_iter)
```

Syntax

```
#include <am_map.h>

void am_map_entry_iter_destroy(am_map_entry_iter_t entry_iter);
```

Parameters

This function takes the following parameters:

`entry_iter` The iterator that must be destroyed.

Returns

None.

Description

This function destroys the `am_map_entry_iterator_t` passed to it.

Memory Concerns

Caller must be sure that this function is not called multiple times on the same `am_map_entry_iter_t`.

```
am_map_entry_iter_get_first_value(am_map_entry_iter_t entry_iter)
```

Syntax

```
#include <am_map.h>

const char * am_map_entry_iter_get_first_value(am_map_entry_iter_t
entry_iter);
```

Parameters

This function takes the following parameters:

`entry_iter` The iterator for which the first value is to be returned.

Returns

This function returns one of the following values:

- value if the operation is successful, returns the first associated value of this iterator. The order of insertion into the map does not guarantee the value returned.
- NULL if there iterator is NULL.

Description

This function destroys the `am_map_entry_iterator_t` passed to it.

Memory Concerns

Caller must be sure that this function is not called multiple times on the same `am_map_entry_iter_t`.

`am_map_entry_iter_get_key(am_map_entry_iter_t entry_iter)`

Syntax

```
#include <am_map.h>
```

```
const char * am_map_entry_iter_get_key(am_map_entry_iter_t  
entry_iter);
```

Parameters

This function takes the following parameters:

`entry_iter` The iterator for which the key needs to be returned.

Returns

This function returns one of the following values:

- key if the operation is successful, returns the key associated with the iterator.
- NULL if there iterator is NULL.

Description

This function returns the key of this key-value pair entry iterator.

Memory Concerns

Caller must not modify or free the return value.

```
am_map_entry_iter_get_values(am_map_entry_iter_t entry_iter,
am_map_value_iter_t *value_iter_ptr)
```

Syntax

```
#include <am_map.h>
```

```
am_status_t am_map_entry_iter_get_values(am_map_entry_iter_t
entry_iter, am_map_value_iter_t *value_iter_ptr);
```

Parameters

This function takes the following parameters:

<code>entry_iter</code>	The iterator that must be destroyed.
<code>value_iter_t</code>	The iterator that goes over the values associated with the key-value pair entry iterator.

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the operation is successful.
- `AM_INVALID_ARGUMENT` if the `entry_iter_ptr` or `value_iter_ptr` argument, is `NULL`.
- `AM_NOT_FOUND` if the specified iterator is `NULL` or does not reference a valid entry.
- `AM_NO_MEMORY` if there was an internal memory operation error.

Description

This function returns an `am_map_value_iter_t` that enumerates over the values associated with `am_map_entry_iter_t`.

Memory Concerns

After the use of `value_iter_t` the caller must call `am_map_value_iter_destroy`.

```
am_map_entry_iter_is_entry_valid(am_map_entry_iter_t entry_iter)
```

Syntax

```
#include <am_map.h>
```

```
int am_map_entry_iter_is_entry_valid(am_map_entry_iter_t
entry_iter);
```

Parameters

This function takes the following parameters:

`entry_iter` The iterator that must be destroyed.

Returns

This function returns one of the following values:

- `!0` if the specified `entry_iter` is valid.
- `0` if `entry_iter` is NULL or does not reference a valid entry.

Description

This function returns if the `entry_iter` passed in is valid.

Memory Concerns

None.

`am_map_entry_iter_entry_iter_next(am_map_entry_iter_t entry_iter)`

Syntax

```
#include <am_map.h>
```

```
int am_map_entry_iter_iter_next(am_map_entry_iter_t entry_iter);
```

Parameters

This function takes the following parameters:

`entry_iter` The iterator entry pointer to be advanced.

Returns

This function returns one of the following values:

- `!0` if the advancing operation was successful.
- `0` if `entry_iter` is NULL or does not reference a valid entry.

Description

Advances the specified iterator to the next entry in the map specified when the iterator was created.

Memory Concerns

None.

am_map_value_iter_destroy(am_map_value_iter_t iter)

Syntax

```
#include <am_map.h>

void

am_map_value_iter_destroy(am_map_value_iter_t value_iter);
```

Parameters

This function takes the following parameters:

`value_iter` The value iterator to be destroyed.

Returns

None.

Description

This function destroys a previously created `am_map_value_iter_t` structure.

Memory Concerns

Caller must make sure that previously destroyed instance of `am_map_value_iter_t` is not attempted to be destroyed again.

am_map_value_iter_get(am_map_value_iter_t value_iter)

Syntax

```
#include <am_map.h>

const char *

am_map_value_iter_get(am_map_value_iter_t value_iter);
```

Parameters

This function takes the following parameters:

`value_iter` The value iterator.

Returns

This function returns one of the following values:

- The value if the operation is successful.
- NULL if `value_iter` is NULL or does not reference a valid entry.

Description

Returns the value referenced by the value iterator.

Memory Concerns

None.

```
am_map_value_iter_is_value_valid(am_map_value_iter_t value_iter)
```

Syntax

```
#include <am_map.h>
```

```
int
```

```
am_map_value_iter_is_value_valid(am_map_value_iter_t value_iter);
```

Parameters

This function takes the following parameters:

`value_iter` The value iterator to be examined

Returns

This function returns one of the following values:

- `!0` if the `value_iter` is a valid reference.
- `0` if `value_iter` is `NULL` or does not reference a valid entry.

Description

Returns the validity of the `value_iter` iterator.

Memory Concerns

None.

```
am_map_value_iter_is_value_next(am_map_value_iter_t value_iter)
```

Syntax

```
#include <am_map.h>
```

```
int
```

```
am_map_value_iter_next(am_map_value_iter_t value_iter);
```

Parameters

This function takes the following parameters:

`value_iter` The value iterator to be advanced.

Returns

This function returns one of the following values:

- `!0` if the advance operation is performed successfully.
- `0` if `value_iter` is `NULL` or does not reference a valid entry.

Description

Returns the validity of the `value_iter` iterator.

Memory Concerns

None.

am_properties_t

This data structure is an associative container with a key of type `const char *` and having single value of type `const char *`. It also provides convenience methods to load and store property files (like the Java property file) and get methods to return specific data types. This structure enables the user of the policy evaluation library to bring in the required configuration from any configuration source.

```
am_properties_create(am_properties_t *properties_ptr)
```

Syntax

```
#include <am_properties.h>

am_status_t

am_properties_create(am_properties_t *properties_ptr);
```

Parameters

This function takes the following parameters:

`properties_ptr` The pointer to the `am_properties_t`.

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the operation was successful.
- `AM_NO_MEMORY` if there was an internal memory operation error.

- `AM_INVALID_ARGUMENT` if `properties_ptr` argument is `NULL`.

Description

Creates an instance of `am_properties_t` and assigns it to `properties_ptr`.

Memory Concerns

After the usage of the instance the caller must call `am_properties_destroy` to clean up the allocated memory.

```
am_properties_copy(am_properties_t source_ptr, am_properties_t
*properties_ptr)
```

Syntax

```
#include <am_properties.h>
```

```
am_status_t
```

```
am_properties_create(am_properties_t source_ptr, am_properties_t
*properties_ptr);
```

Parameters

This function takes the following parameters:

<code>source_ptr</code>	The <code>am_properties_t</code> instance whose data must be copied.
<code>properties_ptr</code>	The pointer to the <code>am_properties_t</code> to which the cloned instance must be assigned.

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the operation was successful.
- `AM_NO_MEMORY` if there was an internal memory operation error.
- `AM_INVALID_ARGUMENT` if `source_ptr` or `properties_ptr` argument is `NULL`.

Description

Creates an instance of `am_properties_t` and assigns it to `properties_ptr`. The function copies all the elements in the `source_ptr` to `properties_ptr`. The `source_ptr` is not affected during this operation.

Memory Concerns

After the usage of the instance `properties_ptr` the caller must call `am_properties_destroy` to clean up the allocated memory. The removal of any item in either structures do not affect the other.

```
am_properties_destroy(am_properties_t properties)
```

Syntax

```
#include <am_properties.h>

am_status_t

am_properties_destroy(am_properties_t properties);
```

Parameters

This function takes the following parameters:

`properties` The `am_properties_t` instance to be destroyed.

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the operation was successful.
- `AM_INVALID_ARGUMENT` if `properties` argument is `NULL`.

Description

Destroys an instance of `am_properties_t`.

Memory Concerns

Caller must make sure not to pass the same instance of `am_properties_t` to be destroyed more than once. After calling this function it is advised that the caller initializes `properties` to `NULL`.

```
am_properties_load(am_properties_t properties, const char
*file_name)
```

Syntax

```
#include <am_properties.h>

am_status_t

am_properties_load(am_properties_t properties, const char
*file_name);
```

Parameters

This function takes the following parameters:

<code>properties</code>	The <code>am_properties_t</code> instance to which the key-value pairs needs to be loaded into.
<code>file_name</code>	The name of the file from which the properties need to be loaded.

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the operation was successful.
- `AM_NSPR_ERROR` if an internal NSPR system operation failed.
- `AM_NO_MEMORY` if there was an internal memory operation error.
- `AM_INVALID_ARGUMENT` if properties argument is NULL.

Description

Loads property information from a specified file, an instance of `am_properties_t`.

Memory Concerns

None.

```
am_properties_store(am_properties_t properties, const char
*file_name)
```

Syntax

```
#include <am_properties.h>

am_status_t

am_properties_store(am_properties_t properties, const char
*file_name);
```

Parameters

This function takes the following parameters:

<code>properties</code>	The <code>am_properties_t</code> instance to which the key-value pairs needs to be saved.
<code>file_name</code>	The name of the file from which the properties need to be written to.

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the operation was successful.
- `AM_NSPR_ERROR` if an internal NSPR system operation failed.
- `AM_INVALID_ARGUMENT` if `properties` or `file_name` argument is `NULL` or `file_name` points to an empty string.

Description

Stores all the key-value pairs in this object into the file.

Memory Concerns

None.

```
am_properties_log(am_properties_t properties, am_log_module_id_t
module, am_log_level_t level)
```

Syntax

```
#include <am_properties.h>
```

```
void
```

```
am_properties_log(am_properties_t properties, am_log_module_id_t
module, am_log_level_t level);
```

Parameters

This function takes the following parameters:

<code>properties</code>	The <code>am_properties_t</code> instance to which the key-value pair needs to be saved.
<code>module</code>	Logging module use to log this property operations.
<code>level</code>	Logging level to use for the log messages. The levels of logging is defined in <code>am_log</code> section.

Returns

None.

Description

Sets the logging module to the property instance. All operations will be logged to the given log module using the level specified.

Memory Concerns

None.

```
am_properties_is_set(am_properties_t properties, const char *key)
```

Syntax

```
#include <am_properties.h>

int

am_properties_is_set(am_properties_t properties, const char *key);
```

Parameters

This function takes the following parameters:

properties	The <code>am_properties_t</code> instance whose contents need to be examined.
key	The key whose presence will be checked.

Returns

This function returns one of the following values:

- !0 if the key is present.
- 0 if the key is not present or the properties argument is NULL.

Description

This function checks if the key is present in the properties instance.

Memory Concerns

None.

```
am_properties_get(am_properties_t properties, const char *key,
const char **value_ptr)
```

Syntax

```
#include <am_properties.h>

am_status_t

am_properties_get(am_properties_t properties, const char *key, const
char **value_ptr);
```

Parameters

This function takes the following parameters:

<code>properties</code>	The <code>am_properties_t</code> instance from which the keys value needs to be extracted.
<code>key</code>	The key whose value will be returned.
<code>value_ptr</code>	The value pointer to which the value will be assigned to. This is an output parameter.

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the operation is successful.
- `AM_INVALID_ARGUMENT` if the `properties`, `key` or `value_ptr` argument is key is `NULL`.
- `AM_NOT_FOUND` if there was no occurrence of the key in this `am_properties_t` instance.
- `AM_NO_MEMORY` if there was an internal memory operation error.

Description

This function checks if the key is present in the properties instance and returns its value.

Memory Concerns

Caller must not modify the `value_ptr` structure or free the memory.

```
am_properties_get_with_default(am_properties_t properties, const
char *key, const char *default_value, const char **value_ptr)
```

Syntax

```
#include <am_properties.h>
```

```
am_status_t
```

```
am_properties_get(am_properties_t properties, const char *key, const
char *default_value, const char **value_ptr);
```

Parameters

This function takes the following parameters:

<code>properties</code>	The <code>am_properties_t</code> instance from which the keys value needs to be extracted.
<code>key</code>	The key whose value will be returned.

<code>default_value</code>	The value to be returned in case of any error condition.
<code>value_ptr</code>	The value pointer to which the value will be assigned to. This is an output parameter.

Returns

Return values may be ignored.

Description

This function checks if the key is present in the properties instance. If the key is not present, the function returns the default value passed in. Otherwise it returns the value of the key.

Memory Concerns

Caller must not modify the `value_ptr` structure or free the memory.

Information And Utility APIs

Following are the policy data structures and operations.

`am_policy_result_t`

Syntax

```
#include <am_policy.h>

typedef struct am_policy_result {
    const char *remote_user;
    const char *remote_IP;
    am_map_t advice_map;
    am_map_t attr_response_map;
} am_policy_result_t;
```

Parameters

This structure has the following components:

<code>remoteUser</code>	After policy evaluation, this variable is assigned the name of the remote user.
<code>advice_map</code>	On the server side some policies may have resulted in advices. For detailed discussion on advices, please refer to the policy service documentation.

`attr_response_map` After evaluation of policies, each policy may define pairs of keys and values. These values may provide more information about the user or about the policy evaluation itself. Apart from this, library may be requested to obtain user attributes. This is a configuration parameter set during service initialization in the service configuration properties structure.

Description

This structure unifies various components of policy evaluation, namely: name of the user try to perform an action on the resource, the advices as recommended by individual policies during evaluation and attribute responses providing specific values as set in policy definition or user attributes as requested during service initialization. The property `com.iplanet.am.policy.am.headerAttributes` specifies what all attributes in the users entry needs to be returned along with policy evaluation results.

Memory Concerns

Caller to `am_policy_evaluate` must call `am_policy_result_destroy` after using the results.

`am_resource_traits_t`

Syntax

```
typedef struct am_resource_traits {
    am_resource_match_t (*cmpFuncPtr)(const char *policyResName,
        const char *resourceName,
        am_bool_t usePatterns);
    am_bool_t (*hasPatterns)(const char *resourceName);
    am_bool_t (*getResourceRoot)(const char *resourceName,
        char *rootResourceName,
        size_t buflen);
} am_resource_traits_t;
```

Parameters

This structure has the following components:

<code>cmpFuncPtr</code>	The compare function takes two resource strings and a boolean value. The boolean value controls whether the compare function must use patterns defined in the resource name or not.
<code>hasPatterns</code>	The is function takes a resource name, examines it and returns true if it has patterns or not. Since the service writer can decide the implementation of pattern representations and so the symbols used as patterns, this function is a resource trait.
<code>getResourceRoot</code>	This function takes a given resource and finds a root resource name for that resource. The length is the size of the <code>rootResourceName</code> buffer. The minimum required length of the <code>rootResourceName</code> buffer is governed by the representation of the resource name and so up to the discretion of the service writer.

Description

This structure is an input parameter to `am_policy_init` function. This structure contains all the resource traits interfaces that are required during policy information maintenance and evaluation.

Memory Concerns

None.

am

The `am` methods are one-time library initialization and cleanup methods. These functions may be called once and only once for the entire life of the shared object.

`am_init(am_policy_t policy)`

Syntax

```
#include <am.h>

am_status_t

am_init(am_properties_t library_init_config);
```

Parameters

This function takes the following parameters:

`library_init_config` This structure contains configuration information required to initialize the shared object.

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the operation is successful.
- `AM_INVALID_ARGUMENT` if any argument is NULL or invalid.
- `AM_NSPR_ERROR` if there is an error while performing an NSPR operation.
- `AM_NO_MEMORY` if there was an internal memory operation error.
- `AM_FAILURE` if there was any unexpected error during initialization.

Description

This function initializes the shared object. This function must be called only once and must be the first function to be called in the library.

Memory Concerns

Caller must call ascertain that this function is not called more than once and its cleanup counterpart `am_cleanup` is called once and is the last function to be called in the library.

`am_init(am_policy_t policy)`

Syntax

```
#include <am.h>

am_status_t
am_cleanup(void);
```

Parameters

None.

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the operation is successful.
- `AM_INVALID_ARGUMENT` if any argument is NULL or invalid.
- `AM_NSPR_ERROR` if there is an error while performing an NSPR operation.
- `AM_NO_MEMORY` if there was an internal memory operation error.

- `AM_FAILURE` if there was any unexpected error during initialization.

Description

This function cleans up all the memory and resources allocated by the shared object. This function must be called only once and must be the last function to be called in the library.

Memory Concerns

None.

am_policy

The `am_policy` methods are service specific methods. These methods may be used only after `am_init` has been invoked successfully and may not be used after `am_cleanup` has been performed.

```
am_policy_init(const char *service_name, const char
*instance_name, am_resource_traits_t rsrcTraits, am_properties_t
service_config_properties, am_policy_t *policy_handle_ptr)
```

Syntax

```
#include <am_policy.h>

am_status_t

am_policy_init(const char *service_name, const char *instance_name,
am_resource_traits_t rsrcTraits, am_properties_t
service_config_properties, am_policy_t *policy_handle_ptr);
```

Parameters

This function takes the following parameters:

<code>service_name</code>	The name of the service about to be created.
<code>instance_name</code>	The instance name of the new service. Currently this parameter is unused.
<code>rsrcTraits</code>	The resource traits structure that contains the pointers to the actual implementations of the resource name operations.
<code>service_config_properties</code>	The <code>am_properties_t</code> instance that has the initialization values required during the initialization of the service.

`policy_handle_ptr` The pointer to the `am_policy_t` that will be initialized a `policy_handle` after successful completion of service creation.

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the operation is successful.
- `AM_INVALID_ARGUMENT` if any argument is `NULL` or invalid.
- `AM_NSPR_ERROR` if there is an error while performing an NSPR operation.
- `AM_NO_MEMORY` if there was an internal memory operation error.
- `AM_FAILURE` if there was any unexpected error during initialization.

Description

This function initializes a policy service instance.

Memory Concerns

Caller must call `am_policy_destroy` structure or free the memory.

`am_policy_destroy(am_policy_t policy)`

Syntax

```
#include <am_policy.h>

void
am_policy_destroy(am_policy_t policy);
```

Parameters

This function takes the following parameters:

`policy` The policy service which needs to be destroyed.

Returns

None.

Description

This function destroys a policy service instance.

Memory Concerns

Caller must call make sure the same service instance not be destroyed more than once.

```
am_policy_evaluate(am_policy_t policy_handle, const char
*sso_token, const char *resource_name, const char *action_name,
const am_map_t env_parameter_map, am_map_t
policy_response_map_ptr, am_policy_result_t *policy_result)
```

Syntax

```
#include <am_policy.h>
```

```
am_status_t
```

```
am_policy_evaluate(am_policy_t policy_handle, const char *sso_token,
const char *resource_name, const char *action_name, const am_map_t
env_parameter_map, am_map_t *policy_response_map_ptr,
am_policy_result_t *policy_result);
```

Parameters

This function takes the following parameters:

<code>policy_handle</code>	The policy service which needs to be destroyed.
<code>sso_token</code>	The single sign-on token of the user who must be evaluated for accessing the resource and perform a given action.
<code>resource_name</code>	The string form of the resource that the user wants to perform the action on.
<code>action_name</code>	The action the user wants to perform on the action.
<code>env_parameter_map</code>	The environment parameters like IP address the user is accessing the resource from.
<code>policy_response_map_ptr</code>	The response structure that will be populated after evaluation of the policies.
<code>policy_result</code>	Pointer to <code>am_policy_result_t</code> structure that will be populated during policy evaluation.

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the operation is successful.
- `AM_INVALID_ARGUMENT` if any argument is NULL or invalid.
- `AM_NSPPR_ERROR` if there is an error while performing an NSPPR operation.

- `AM_NO_MEMORY` if there was an internal memory operation error.
- `AM_FAILURE` if there was any unexpected error during initialization.

Description

This function destroys a policy service instance.

Memory Concerns

After using the results the caller must call `am_policy_result_destroy` on the `policy_result` to cleanup the memory allocated by the evaluation operation. `am_map_destroy` must also be called on response and `env_parameter_map` after their respective usage scope.

```
am_policy_is_notification_enabled(am_policy_t policy_handle)
```

Syntax

```
#include <am_policy.h>

am_bool_t

am_policy_is_notification_handled(am_policy_t policy_handle);
```

Parameters

This function takes the following parameters:

<code>policy_handle</code>	The policy service instance whose configuration needs to be examined.
----------------------------	---

Returns

This function returns one of the following values:

- `AM_TRUE` if notification is enabled for this service.
- `AM_FALSE` if notification is not enabled for this service.

Description

This function checks and returns the state of notification. This configuration is read by the service during initialization from the service configuration properties.

Memory Concerns

None.

```
am_policy_notify(am_policy_t policy_handle, const char
*notification_data, size_t notification_data_len)
```

Syntax

```
#include <am_policy.h>

am_status_t

am_policy_notify(am_policy_t policy_handle, const char
*notification_data, size_t notification_data_len);
```

Parameters

This function takes the following parameters:

<code>policy_handle</code>	The policy service instance whose configuration needs to be examined.
<code>notification_data</code>	The notification data that was received by the caller.
<code>notification_data_ len</code>	Notification data length.

Returns

This function returns one of the following values:

- `AM_SUCCESS` if the operation is successful.
- `AM_INVALID_ARGUMENT` if any argument is NULL or invalid.
- `AM_NSPR_ERROR` if there is an error while performing an NSPR operation.
- `AM_NO_MEMORY` if there was an internal memory operation error.
- `AM_FAILURE` if there was any unexpected error during initialization.

Description

When the configuration `com.iplanet.am.policy.am.notificationEnabled` is set to true, the library registers with the server to receive notification in case of any change of information (session or policy) on the server. The notification is sent to the URL again, set in the service configuration properties as a value of `com.iplanet.am.policy.am.notificationURL` during initialization.

Memory Concerns

None.

Specialization Methods

These functions are resource traits implementations for URLs. These are provided for the sake of convenience and as reference implementations of resource names.

```
am_policy_compare_urls(const char *policyResourceName, const
char *resourceName, am_bool_t usePatterns)
```

Syntax

```
#include <am_policy.h>
```

```
void
```

```
am_policy_compare_urls(const char *policyResourceName, const char
*resourceName, am_bool_t usePatterns);
```

Parameters

This function takes the following parameters:

<code>policyResourceName</code>	The resource name as defined in the policy. This is the only parameter that can have patterns it them.
<code>resourceName</code>	The name of the resource accessed by the user.
<code>usePatterns</code>	The patterns in the <code>policyResourceName</code> if present must be used if this parameter is set to <code>AM_TRUE</code> otherwise used as normal characters.

Returns

This function returns one of the following values:

- `EXACT_MATCH` if `policyResourceName` and `resourceName` match verbatim.
- `SUB_RESOURCE_MATCH` if `resourceName` is a subordinate resource of `policyResourceName`.
- `SUPER_RESOURCE_MATCH` if `policyResourceName` is a subordinate resource of `resourceName`.
- `NO_MATCH` if `policyResourceName` and `resourceName` do not match.
- `EXACT_PATTERN_MATCH` This value will be returned only if `usePatterns` argument is `AM_TRUE` and if `policyResourceName` has patterns which matches `resourceName` string.

Description

This function compares two given URLs. The `policyResourceName` is the URL defined in the policy definition. The policy definition can contain patterns. In the reference implementation, the comparison mechanism does only wildcard match, that is, it supports only `*` in a `policyResourceName` as a pattern. A service writer may replace this function with another implementation that supports complete regular expressions. But, the service writer must also take care to change the `hasPatterns` function to behave appropriately.

Memory Concerns

None.

```
am_policy_get_url_resource_root(const char *resourceName, char
*resourceRoot, size_t length)
```

Syntax

```
#include <am_policy.h>

am_bool_t

am_policy_get_url_resource_root(const char *resourceName, char
*resourceRoot, size_t length);
```

Parameters

This function takes the following parameters:

<code>resourceName</code>	The resource name for which the root resource must be extracted.
<code>resourceRoot</code>	The pointer where the resource root will be copied to.
<code>length</code>	The length of the <code>resourceRoot</code> buffer.

Returns

This function returns one of the following values:

- `AM_TRUE` if the operation was successful.
- `AM_FALSE` if `resourceName` is not a valid URL or either of the parameters were `NULL` or if the buffer was not enough to copy the `resourceRoot`.

Description

This function is takes a URL and extracts a root of the URL. For example, `http://www.sun.com/index.html` will return `http://www.sun.com/` and `http://www.sun.com:8080/index.html` will return `http://www.sun.com:8080/`.

Memory Concerns

In an implementation for a different resource other than URLs, the service writer implementing this function must make accurate judgement about the minimum size of `resourceRoot`.

```
am_policy_get_url_resource_has_patterns(const char
*resourceName)
```

Syntax

```
#include <am_policy.h>

am_bool_t

am_policy_get_url_resource_has_patterns(const char *resourceName);
```

Parameters

This function takes the following parameters:

`resourceName` The resource name to be examined for patterns.

Returns

This function returns one of the following values:

- `AM_TRUE` if the resource name has patterns.
- `AM_FALSE` if `resourceName` is not a valid URL or the parameter is `NULL` or if it does not have patterns.

Description

This function takes a URL and returns a boolean value reflecting its pattern content. For example, `http://www.sun.com/index.html` will return `AM_FALSE` and `http://www.sun.com/*.html` will return `AM_TRUE`. `http://www.sun.com/*` will return `AM_TRUE`.

Memory Concerns

None.

Initialization Variables

Following are explanations of the initialization variables:

com.iplanet.am.policy.am.cookieName The name of the cookie set by the authentication server after session creation.

com.iplanet.am.policy.am.namingURLThe URLs where the naming service is installed. Each URL must be separated with a space. If the current naming server does not respond, the next one is attempted and so on, in a round-robin fashion.

com.iplanet.am.policy.am.loginURLThe URLs where the login service is installed. Each URL must be separated with a space. If the current login server does not respond, the next one is attempted and so on, in a round-robin fashion.

com.iplanet.am.policy.am.logFileThe full path of the local log file name.

com.iplanet.am.policy.am.serverLogFileThe name of the log file in the server side.

com.iplanet.am.policy.am.logLevelsThe level of logging to be performed to the local log file.

com.iplanet.am.policy.am.usernameThe name of user as whom the client library will login to the policy server.

com.iplanet.am.policy.am.passwordThe password of user using which the client library will log on as the given user.

com.iplanet.am.policy.am.sslCertDirSSL directory where the certificate database is located.

com.iplanet.am.policy.am.trustServerCertsThis boolean value if true indicates that library must trust server certificate.

com.iplanet.am.policy.am.notificationEnabledSetting this variables to true indicates that the policy library must register for session and policy notifications to be delivered to the notificationURL.

com.iplanet.am.policy.am.agenturiprefixThe agenturiprefix is the URI under which the agent components are installed.

com.iplanet.am.policy.am.notificationURLThe notification URL to which the notifications must be sent. While using this library and having the notification enabled, the host program must listen as the notification URL and pass the received data to `am_policy_notify` function.

com.iplanet.am.policy.am.fetchHeadersThis is a boolean parameter when set to true, the policy evaluation request also requests for the users attribute values.

com.iplanet.am.policy.am.headerAttributesThe attributes whose values will be requested for during policy evaluation, if `fetchHeaders` parameters is set to true. This attribute has a syntax [`<ldap attribute name>` | `<local attribute name>`]*. The local attribute name and its values will be returned in the `attr_response_map` in `policy_result`.

Specialization Methods For Web Agents

`am_web_init(const char *config_file)`

Syntax:

```
#include <am_web.h>

am_status_t am_web_init(const char *config_file);
```

Parameters:

This function takes the following parameters:

<code>config_file</code>	A string representing the complete path to the config file used by the agent.
--------------------------	---

Returns:

This function returns one of the following values:

- `AM_SUCCESS` if the initialization was successful.
- `AM_INVALID_ARGUMENT` if any argument is NULL or invalid.
- `AM_NSPR_ERROR` if there is an error while performing an NSPR operation.
- `AM_NO_MEMORY` if there was an internal memory operation error.
- `AM_FAILURE` if the initialization failed due to unexpected error condition.

Description:

This function initializes the shared object. This function must be called only once and must be the first function to be called in the library.

Memory Concerns:

Caller must ascertain that this function is not called more than once and its cleanup counterpart `am_web_cleanup` is the last function to be called in the library.

`am_web_cleanup()`

Syntax:

```
#include <am_web.h>

am_status_t am_web_cleanup();
```

Parameters:

None.

Returns:

This function returns one of the following values:

- `AM_SUCCESS` if the operation was successful.
- `AM_INVALID_ARGUMENT` if any argument is NULL or invalid.
- `AM_NSPR_ERROR` if there is an error while performing an NSPR operation.
- `AM_NO_MEMORY` if there was an internal memory operation error.
- `AM_FAILURE` if the initialization failed due to unexpected error condition.

Description:

This function releases all the resources allocated by the library. This function must be called only once and must be the last function to be called in the library.

Memory Concerns:

Caller must ascertain that this function is not called more than once and its initialization counterpart `am_web_init` has been called before its invocation.

```
am_web_is_access_allowed(const char *sso_token, const char *url,  
const char *action_name, const char *client_ip, const am_map_t  
env_parameter_map, am_policy_result_t *result);
```

Syntax:

```
#include <am_web.h>
```

```
am_status_t
```

```
am_web_is_access_allowed(const char *sso_token, const char  
*url, const char *action_name, const char *client_ip, const am_map_t  
env_parameter_map, am_policy_result_t *result);
```

Parameters:

This function takes the following parameters:

<code>sso_token</code>	The <code>sso_token</code> from the Identity Server cookie. This parameter may be NULL if there is no cookie present.
<code>url</code>	The URL whose accessibility is being determined. This parameter may not be NULL.

<code>action_name</code>	The action (GET, POST, etc.) being performed on the specified URL. This parameter may not be NULL.
<code>client_ip</code>	The IP address of the client attempting to access the specified URL. If client IP validation is turned on, then this parameter may not be NULL.
<code>env_parameter_map</code>	A map containing additional information about the user attempting to access the specified URL. This parameter may not be NULL.
<code>result</code>	A data structure to store the result of this operation as passed back to the caller.

Returns:

This function returns one of the following values:

- `AM_SUCCESS` if the evaluation was performed successfully and access is to be allowed to the specified resource.
- `AM_NO_MEMORY` if the evaluation was not successfully completed due to insufficient memory being available.
- `AM_INVALID_ARGUMENT` if any of the `url`, `action_name`, `env_parameter_map`, or `result` parameters is NULL or if client IP validation is enabled and the `client_ip` parameter is NULL.
- `AM_INVALID_SESSION` if the specified `sso_token` does not refer to a currently valid session.
- `AM_ACCESS_DENIED` if the policy information indicates that the user does not have permission to access the specified resource or any error is detected other than the ones listed above.

Description:

This function evaluates the access control policies for a specified web-resource and action. The web-resource is identified as the URL that the user is trying to access and the action is the associated HTTP call made by the user agent such as GET or POST.

Memory Concerns:

The caller must free the memory associated with all the arguments passed into this function when they are no longer in use or applicable.

```
am_web_is_notification(const char *request_url)
```

Syntax:

```
#include <am_web.h>

am_bool_t

am_web_is_notification(const char *request_url);
```

Parameters:

This function takes the following parameters:

<code>request_url</code>	The URL of the web resource associated with this request.
--------------------------	---

Returns:

This function returns one of the following values:

- `AM_TRUE` if the associated request is Identity Server notification.
- `AM_FALSE` if the associated request is a regular request.

Description:

This function determines if the request is an Identity Server notification message intended for the policy SDK.

Memory Concerns:

None.

```
am_web_handle_notification(const char *data, size_t data_length);
```

Syntax:

```
#include <am_web.h>

void

am_web_handle_notification(const char *data, size_t data_length);
```

Parameters:

This function takes the following parameters:

<code>data</code>	A buffer containing data as obtained from the request.
<code>data_length</code>	The length of the data contained in the buffer.

Returns:

None

Description:

This function processes the notification data as sent by the Identity Server and updates the state of the Policy SDK accordingly. Any errors that may occur during the processing of the given data are logged in the appropriate log files.

Memory Concerns:

None.

```
am_web_get_redirect_url(am_status_t status, const am_map_t
advice_map, const char *goto_url, am_bool_t *allocated_ptr);
```

Syntax:

```
#include <am_web.h>
```

```
char *am_web_get_redirect_url(am_status_t status, const am_map_t
advice_map, const char *goto_url, am_bool_t *allocated_ptr);
```

Parameters:

This function takes the following parameters:

status	The return value obtained from the call to function <code>am_web_is_access_allowed()</code> .
advice_map	The advice map contained in the policy result as obtained from the call to function <code>am_web_is_access_allowed()</code> .
goto_url	The return URL to be used by the Identity Server to redirect the user after successful authentication.
allocated_ptr	A flag that indicates if any memory was allocated during the course of this operation which should be deallocated by the caller accordingly.

Returns:

A URL that may be used to redirect the user accordingly. This URL may be either the login URL or the access denied URL.

Description:

This function returns a string representing the URL for redirection that is appropriate to the provided status code and advice map returned by the Policy SDK. This may either redirect the user to the login URL or the access denied URL. If the redirection is to the login URL then the URL will include any existing information specified in the URL from the configuration file, like org value etc., followed by the specified goto parameter value, which will be used by Identity Server after the user has successfully authenticated.

Memory Concerns:

If the value returned in `allocated_ptr` parameter is `AM_TRUE`, then the caller is responsible for calling `am_web_free_memory()` function when done with the returned string. Using another free function may cause corruption of memory and result in fatal runtime errors. If `allocated_ptr` is `NULL`, then the function will always return a pointer to the access denied URL, which is not allocated.

```
am_web_do_result_attr_map_set(am_policy_result_t *result,
am_status_t (*setFunc)(char *, char *, void **), void **args);
```

Syntax:

```
#include <am_web.h>

am_status_t

am_web_do_result_attr_map_set(am_policy_result_t *result,
am_status_t (*setFunc)(char *, char *, void **), void **args);
```

Parameters:

This function takes the following parameters:

<code>result</code>	The result obtained from the call to function <code>am_web_is_access_allowed()</code> .
<code>setFunc</code>	A function pointer for the call back function to be used when iterating over various values of the result.
<code>args</code>	The optional arguments to be passed to the call back function during its invocation.

Return:

This function returns one of the following values:

- `AM_SUCCESS` if the operation was successful.
- `AM_FAILURE` if the operation failed due to unexpected error conditions.

- `AM_NO_MEMORY` if the operation failed due to lack of available memory required for the processing.

Description:

This function process `attr_response_map` of `am_policy_result_t` and performs the appropriate set action that caller pass in.

Memory Concerns:

None.

```
am_web_free_memory(void *memory)
```

Syntax:

```
#include <am_web.h>
```

```
void am_web_free_memory(void *memory);
```

Parameters:

This function takes the following parameter:

<code>memory</code>	A pointer to the previously allocated memory that should be released by this function.
---------------------	--

Return:

None.

Description:

This function releases previously allocated memory by any of the `am_web_*` functions.

Memory Concerns:

This function must be called in order to free any memory allocated by the `am_web_*` functions. Using other routines to free such memory may result in memory corruption and lead to fatal runtime error conditions.

```
am_web_get_cookie_name()
```

Syntax:

```
#include <am_web.h>
```

```
const char *am_web_get_cookie_name()
```

Parameters:

None.

Return:

This function returns the name of the Identity Server cookie.

Description:

This function returns the Identity Server cookie name as used by the Policy SDK. The value of this depends upon the value stored in the configuration file that was used to initialize the library.

Memory Concerns:

None

am_web_get_notification_url()

Syntax:

```
#include <am_web.h>

const char *am_web_get_notification_url()
```

Parameters:

None.

Return:

This function returns the notification URL used by the Agent.

Description:

This function returns the URL used by the Agent to receive Identity Server notifications. The value of this depends upon the value stored in the configuration file that was used to initialize the library.

Memory Concerns:

None

am_web_is_debug_on()

Syntax:

```
#include <am_web.h>

am_bool_t am_web_is_debug_on();
```

Parameters:

None.

Return:

This function returns one of the following values:

- AM_TRUE if debugging is turned on.
- AM_FALSE if debugging is turned off.

Description:

This function returns a flag which indicates if debugging has been enabled or disabled. When enabled, the Agent can use various convenience functions in order to log debug messages to the appropriate log files.

Memory Concerns:

None.

```
am_web_is_max_debug_on()
```

Syntax:

```
#include <am_web.h>

am_bool_t am_web_is_max_debug_on();
```

Parameters:

None.

Return:

This function returns one of the following values:

- AM_TRUE if debugging is turned on and is at the maximum level.
- AM_FALSE if debugging is either off or turned on to a level other than its maximum level.

Description:

This function returns a flag which indicates if debugging has been set at its maximum level. The caller can use this function to determine if it is appropriate to emit very verbose information to the debug logs.

Memory Concerns:

None.

```
am_web_log_always(const char *fmt, ...)
```

Syntax:

```
#include <am_web.h>
```

```
void am_web_log_always(const char *fmt, ...);
```

Parameters:

This function takes the following parameters:

fmt	A printf style format string along with associated variable list to be used to emit a formatted message.
-----	--

Return:

None.

Description:

This function allows logging of information regardless of the level at which the library has been configured through the specified values in the configuration file. It takes a printf style format string and argument list which is used to generate a formatted message that is then logged into the appropriate log file.

Memory Concerns:

None.

```
am_web_log_auth(am_web_access_t accessType, const char *fmt,
...)
```

Syntax:

```
#include <am_web.h>
```

```
am_bool_t am_web_log_auth(am_web_access_t accessType, const char
*fmt, ...);
```

Parameters:

This function takes the following parameters:

accesstype	An enumeration that indicates the type of access that is being logged. This value could be either LOG_DENY or LOG_ALLOW or LOG_BOTH or LOG_NONE.
fmt	A printf style format string along with associated variable list to be used to emit a formatted message.

Return:

This function returns one of the following values:

- AM_TRUE if the operation was successful

- `AM_FALSE` if the operation failed due to unexpected error conditions.

Description:

This function is used to log information to the remote Identity Server's log service. Depending upon the value specified in the configuration file, requests associated with certain accesstype values may or may not get logged by this function.

Memory Concerns:

None.

```
am_web_log_error(const char *fmt, ...)
```

Syntax:

```
#include <am_web.h>

void am_web_log_error(const char *fmt, ...);
```

Parameters:

This function takes the following parameters:

<code>fmt</code>	A printf style format string along with associated variable list to be used to emit a formatted message.
------------------	--

Return:

None.

Description:

This function formats and logs the given message as an error in the associated log file. This message will be logged if the associated logging level is greater than or equal to the error logging level.

Memory Concerns:

None.

```
am_web_log_warning(const char *fmt, ...)
```

Syntax:

```
#include <am_web.h>

void am_web_log_warning(const char *fmt, ...);
```

Parameters:

This function takes the following parameters:

`fmt` A printf style format string along with associated variable list to be used to emit a formatted message.

Return:

None.

Description:

This function formats and logs the given message as a warning in the associated log file. This message will be logged if the associated logging level is greater than or equal to the warning logging level.

Memory Concerns:

None.

`am_web_log_info(const char *fmt, ...)`

Syntax:

```
#include <am_web.h>
```

```
void am_web_log_info(const char *fmt, ...);
```

Parameters:

This function takes the following parameters:

`fmt` A printf style format string along with associated variable list to be used to emit a formatted message.

Return:

None.

Description:

This function formats and logs the given message as an informational message in the associated log file. This message will be logged if the associated logging level is greater than or equal to the informational logging level.

Memory Concerns:

None.

`am_web_log_debug(const char *fmt, ...)`

Syntax:

```
#include <am_web.h>
```

```
void am_web_log_debug(const char *fmt, ...);
```

Parameters:

This function takes the following parameters:

fmt	A printf style format string along with associated variable list to be used to emit a formatted message.
-----	--

Return:

None.

Description:

This function formats and logs the given message as a debug message in the associated log file. This message will be logged if the associated logging level is greater than or equal to the debug logging level.

Memory Concerns:

None.

```
am_web_log_max_debug(const char *fmt, ...)
```

Syntax:

```
#include <am_web.h>
```

```
void am_web_log_max_debug(const char *fmt, ...);
```

Parameters:

This function takes the following parameters:

fmt	A printf style format string along with associated variable list to be used to emit a formatted message.
-----	--

Return:

None.

Description:

This function formats and logs the given message as a debug message in the associated log file. This message will be logged if the associated logging level is greater than or equal to the maximum debug logging level.

Memory Concerns:

None.

Initialization Variables

The following table lists the initialization variables for the policy C APIs.

Table 7-1 Initialization Variables for C APIs

Initialization Variables	Explanation
<code>com.sun.am.policy.agents.accessDeniedURL</code>	The URL of the access denied page. If no value is specified, the agent will return an HTTP status of 403 (Forbidden).
<code>com.sun.am.policy.agents.unauthenticatedUser</code>	The user id to be used when the user is not authenticated and is trying to access a resource from the global allow list.
<code>com.sun.am.policy.agents.anonRemoteUserEnabled</code>	Enable or disable the processing of REMOTE_USER variable for anonymous users.
<code>com.sun.am.policy.agents.instanceName</code>	The unique identifier for this agent instance. This property is currently not used.
<code>com.sun.am.policy.agents.notenforcedList</code>	A list of URLs for which no authentication is required. The entries in this list can contain wild cards to represent zero or more characters and use a space as a separator between various entries.
<code>com.sun.am.policy.agents.cdsso-enabled</code>	A flag that indicates if cross-domain single sign on is enabled or disabled.
<code>com.sun.am.policy.agents.cdsso-server.loginURL</code>	A login URL to be used for authenticating users when the cross-domain single sign on is enabled.
<code>com.sun.am.policy.agents.logAccessType</code>	Server's remote logging service. Possible values are LOG_NONE, LOG_DENY, LOG_ALLOW, and LOG_BOTH.
<code>com.sun.am.policy.agents.client_ip_validation_enabled</code>	A flag that indicates if client IP address validation is enabled or disabled. When enabled, the agent checks to ensure that the IP address associated with a given request is the same as the IP address to which the associated session cookie was issued by Identity Server's session service.

Using The SAML Service

Sun™ One Identity Server uses the Security Assertion Markup Language (SAML) for exchanging security information. SAML defines an eXtensible Markup Language (XML) framework to achieve inter-operability across different vendor platforms that provide this type of information. This chapter explains SAML and defines how it is used within the Identity Server. It contains the following sections:

- Overview
- amSAML.xml
- SAML SDK
- SAML Service Samples

Overview

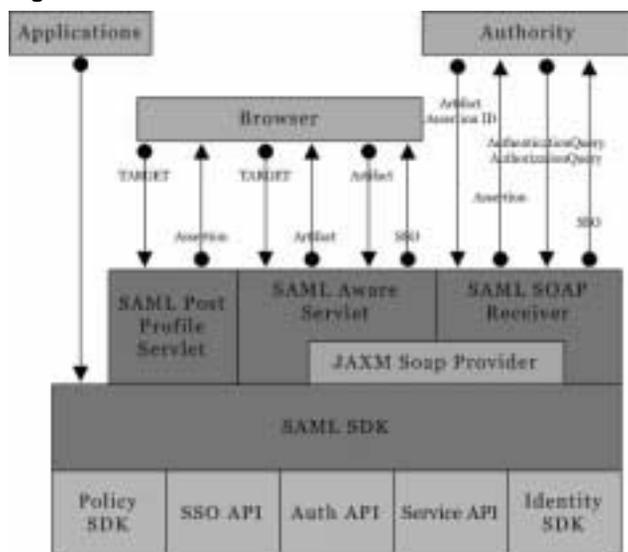
SAML is an open-standard protocol that uses an XML framework to exchange security information between an authority and a trusted partner site. The security information concerns itself with authentication status, access authorization decisions and subject attributes. The Organization for the Advancement of Structured Information Standards (OASIS) drives the development of the SAML specifications.

NOTE The latest SAML information and specifications can be found at <http://www.oasis-open.org/committees/security/>.

SAML security information is expressed in the form of an *assertion* about a *subject*. A *subject* is an entity in a particular domain, either human or computer, with which the security information concerns itself. (A person identified by an email address is a subject as might be a printer.) An *assertion* is a package of verified security

information that supplies one or more statements concerning a subject's authentication status, attributes or access authorization decisions. Assertions are issued by a SAML *authority*. (An authority is a platform or application that has been integrated with the SAML SDK, allowing it to relay security information.) The assertions are received by partner sites defined within the authority as *trusted*. SAML authorities use different sources to configure the assertion information including external data stores or assertions that have already been received and verified. Figure 8-1 illustrates how the SAML Service interacts with the other Identity Server components. (The blocks filled with solid color are components of the SAML Service.)

Figure 8-1 SAML Architecture



The SAML Service allows the Identity Server to work with external applications in the following ways:

- Users can authenticate against Identity Server and access trusted partner sites without having to re-authenticate. This is referred to as Single Sign-On.
- Identity Server acts as a policy decision point (PDP), allowing external applications to access user authorization information for the purpose of granting or denying access to their resources.
- Identity Server acts as both an attribute authority (allowing trusted partner sites to query a subject's attributes) and an authentication authority (allowing trusted partner sites to query a subject's authentication information.)

- Two parties in different security domains can validate each other for the purpose of performing business transactions.

NOTE The SAML service also allows Identity Server to take advantage of the open-source protocols being developed by the Liberty Alliance Project. More information on these specifications can be found at www.projectliberty.org and in Chapter 9, “Federation Management.”

Assertion Types

SAML assertions are represented as XML constructs based on a schema located at <http://www.oasis-open.org/committees/security/docs/cs-sstc-schema-assertion-01.xsd>. The SAML specification provides for several types of assertions that are also defined in the SAML Service:

- An *authentication assertion* declares that the specified subject has been authenticated by a particular means at a particular time. In Identity Server, the Authentication Service is the authentication authority. Code Example 8-1 illustrates an authentication assertion.

Code Example 8-1 Sample Authentication Assertion

```
<?xml version="1.0" encoding="UTF-8" ?>
<saml:Assertion
xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
MajorVersion="1"
MinorVersion="0" AssertionID="random-182726"
Issuer="sunserver.example.com"
IssueInstant="2001-11-05T17:23:00-02:00">
  <saml:AuthenticationStatement
AuthenticationMethod="urn:oasis:names:tc:SAML:1.0:am:password"
AuthenticationInstant="2001-11-05T17:22:00-02:00">
    <saml:Subject>
      <saml:NameIdentifier NameQualifier="sun.com">John
Doe</saml:NameIdentifier>
    </saml:Subject>
  </saml:AuthenticationStatement>
</saml:Assertion>
```

- An *attribute assertion* declares that the specified subject is associated with the specified attribute. In Identity Server, the Identity Management module is the attribute authority.

- An *authorization decision assertion* declares that the specified subject's request for access to a specified resource has been granted or denied. In Identity Server, the Policy Service is the authorization authority.

One assertion may contain many different statements made by the authority.

Profile Types

A set of rules describing how to embed and extract SAML assertions is called a *profile*. The profile describes how the assertions can be combined with other objects by an authority, transported from the authority to a trusted partner site and, subsequently, processed at the latter. Currently, Identity Server supports two profiles that use HTTP: the Web Artifact Profile and the Web POST profile. These profiles are used in the case of single sign-on when an authenticated user attempts to access resources from a trusted partner site. Both profiles can also be the receiver when accepting user single sign-on from a trusted partner site.

NOTE The profile methods can be initiated through a web browser or the SAML API. More information on this API can be found in "com.sun.identity.saml," on page 227.

Web Artifact Profile

The Web Artifact Profile defines interaction between three parties: a user equipped with a web browser, an authority site, and an trusted partner site. When an authenticated user attempts to access a trusted partner site (usually by clicking a link), they are redirected to a transfer service in the authority site. In Identity Server, the transfer service is the SAML Aware Servlet. The base of the transfer URL is `http(s)://<server:port>/<server_deploy_uri>/SAMLAwareServlet`; it is appended with the user's TARGET location (`?TARGET=URL_of_destination`). The SAML Aware Servlet then provides the following functions as part of the Web Artifact Profile:

1. It compares its list of Trusted Partner Sites against the user's TARGET location.

Only targets configured in the Trusted Partner Sites attribute of the SAML Service can access the SAML Service. More information on this attribute can be found in the *Sun ONE Identity Server Administration Guide*.

2. It looks for and validates the SSO Token in the inbound request.

Without a valid SSO Token, the Identity Server will not create an assertion.

3. It creates an *artifact* and a corresponding assertion.

An *artifact* is carried as part of the URL and points to an assertion and its source; it is not, and does not contain, the security information itself. The need to send an artifact rather than the assertion itself is dictated by the restrictions on URL size imposed by many web browsers.

4. It opens a connection to the Artifact Receiver URL and redirects the user to the TARGET location (trusted partner site) with a query string containing the artifact.

The Artifact Receiver URL is based on mapping configurations defined in the SAML Service. More information on this can be found in the *Sun ONE Identity Server Administration Guide*. Upon arriving at the TARGET location, the artifact is extracted and returned to the authority site in a query requesting the assertion to which the artifact points.

5. It accepts an artifact query from the trusted partner site and responds by sending the correct assertion.

The assertion is processed and the user is either granted or denied access to the trusted partner site. If access is granted an SSO token is generated, a cookie is set to the browser and the user is redirected to the TARGET location.

Web POST Profile

The Web POST Profile allows security information to be supplied to a trusted partner site without the use of an artifact. It consists of two interactions: the first between a user with a web browser and the Identity Server, and the second between the same user and a trusted partner site.

When an authenticated user attempts to access a trusted partner site using a web browser (usually by clicking a link), they are redirected to a transfer service in the authority site. In Identity Server, the transfer service is the SAML Post Profile Servlet. The base of the transfer URL is `http(s)://<server:port>/<server_deploy_uri>/SAMLPOSTProfileServlet`; it is appended with the user's TARGET location (`?TARGET=URL_of_destination`). The SAML POST Profile Servlet is what provides functions for the two Web POST Profile interactions. In the first interaction between the user and Identity Server:

1. It obtains the TARGET location from the request and retrieves the trusted partner site URL from the SAML Service.

Again, only targets configured in the Trusted Partner Sites attribute of the SAML Service can access the SAML Service. More information on this attribute can be found in the *Sun ONE Identity Server Administration Guide*.

2. It generates an assertion using the `AssertionManager` class of the SAML SDK.
3. It forms, signs and Base64 encodes a `SAMLResponse` containing the assertion.
4. It generates an HTML form, containing both the `SAMLResponse` and the `TARGET` as parameters, and posts the form as an HTTP response back to the user's browser.

In the second interaction between the user and the trusted partner site:

1. It obtains the `TARGET` and `SAMLResponse` from the request.
2. It Base64 decodes the `SAMLResponse`.
3. It verifies the signature on the `SAMLResponse` and obtains and verifies the SAML response itself.

It also verifies the assertion inside the `SAMLResponse` and enforces single-sign on policy.

4. It obtains or creates an `SSOToken` and redirects the authenticated user to the `TARGET` location.

An advantage of the Web POST profile is that, because it does not use SOAP, it easily moves through a firewall and/or proxy server. The Web POST profile function is provided by either of two means: an HTTP request using the `SAMLPOSTProfileServlet`, or an `SAMLClient` API call to a Java application.

NOTE According to the SAML specifications, the trusted partner site **MUST** ensure a single-use policy for SSO assertions communicated by the Web POST Profile. Thus, `SAMLPOSTProfileServlet` maintains a store of SSO assertion IDs and the time they expire. When an assertion is received, the servlet first checks for an entry in the map. If one exists, the servlet returns an error. If not, the assertion ID and expiration time is saved to the map. The `POSTCleanupThread` removes expired assertion IDs periodically.

SAML SOAP Receiver

Assertions are exchanged between Identity Server and inquiring parties using the `request` and `response` XML-based protocol defined in the SAML specification. These SAML assertions are then integrated into a standard communication protocol for transport purposes; Identity Server uses SOAP (Simple Object Access Protocol), a message communications specification integrating XML and HTTPS. *SOAP binding* defines how SAML `request` and `response` message exchanges are integrated into SOAP exchanges. The SAML SOAP Receiver is a servlet that

processes the message. It receives a SOAP message, extracts the SAML request and responds with another SOAP message containing the requested information. The SAML SOAP Receiver is the producer of SAML assertions. It responds to queries for authentication, attributes or authorization decisions as well as those that include an assertion identifier or artifact by returning assertions.

NOTE The access URL for the SAML SOAP Receiver is `http(s)://<server:port>/<server_deploy_uri>/SAMLSOAPReceiver`.

SOAP Messages

SOAP messages consist of three parts: an envelope, header data and a message body. (The SAML `request/response` elements are enclosed in the message body.) A client, acting as a SAML requestor, transmits a `<Request>` element within the body of a SOAP message to an entity acting as a SAML Receiver. In answer, the SAML Receiver **MUST** return either a `<Response>` element within the body of another SOAP message or a SOAP fault code (or error message).

NOTE The SAML requestor and the SAML Receiver **MUST NOT** include more than one SAML request or response per SOAP message or any additional XML elements in the SOAP body.

A SAML Request may contain queries for any of the following: authentication status, authorization decisions, attribute information and one or more assertion identifiers or artifacts. A SAML Response is sent back to the requesting party for every Request received.

NOTE The SAML SDK and the Java API for XML Messaging (JAXM) are used to construct SOAP messages and send them to the SOAP Receiver.

Protecting The SOAP Receiver

The Identity Server administrator has the option of protecting the SAML SOAP Receiver using authentication. There are five types:

- NOAUTH
- BASICAUTH
- SSL
- SSLWITHBASICAUTH

This option is configured in the Trusted Partner Sites attribute of the SAML Service in the form:

```
SourceID=sourceidofsite|SOAPUrl=urlofsite|AuthType=chosen_auth_option|User=userid
```

The default authentication type is NOAUTH. If SSL authentication is to be specified, it is configured in the `SOAPUrl` field with the `https` URL prefix. More information on the Trusted Partner Sites and other SAML Service attributes can be found in the *Identity Server Administration Guide*.

Accessing The SAML Service

The SAML Service can be accessed using a web browser or the SAML SDK. An end user would authenticate to the Identity Server through a web browser and, when authorized, access URLs from trusted partner sites. Developers, on the other hand, would integrate the APIs into their applications to enable them to exchange security information with the Identity Server. For example, a Java application can use the SAML API to accomplish single sign-on. After obtaining an SSO token from the Identity Server, the application can call the `doWebPOST()` method of the `SAMLClient` class which contacts the Identity Server and can redirect the application to the destination site.

amSAML.xml

`amSAML.xml` is the XML service file that defines the attributes for the SAML Service. All of the attributes in the SAML Service can be managed through either the Identity Server console or the XML service file except two. These attributes can only be managed through `amSAML.xml` using the `amadmin` command line interface.

- `iplanet-am-saml-cleanup-interval` is used to specify how often the internal thread is run in order to cleanup expired assertions from the internal data store. The default is 180 seconds.
- `iplanet-am-saml-assertion-max-number` is used to specify the maximum number of assertions the server can hold at one time. No new assertion will be created if the maximum number is reached. The default value is 0 which means there is no limit.

To change the values of these attributes, the `amSAML.xml` service file needs to be modified and then reloaded using `amadmin`. Information on how to use `amadmin` can be found in Chapter 6, “Service Management.” Information on the additional SAML Service attributes can be found in the Sun ONE Identity Server Administration Guide.

SAML SDK

Identity Server contains a SAML SDK made up of APIs and lower level packages. Administrators can use these packages to integrate the SAML functionality and XML messages into their applications and services. The SDK supports all types of assertions and operates with the Identity Server authorities to process external SAML requests and generate SAML responses. The packages include:

- `com.sun.identity.saml`
- `com.sun.identity.saml.assertion`
- `com.sun.identity.saml.common`
- `com.sun.identity.saml.plugins`
- `com.sun.identity.saml.protocol`
- `com.sun.identity.saml.xmlsig`

NOTE The Identity Server Javadocs can be accessed from any browser by copying the complete `<identity_server_root>/SUNWam/docs/` directory into the `<identity_server_root>/SUNWam/public_html` directory and pointing the browser to `http://<server_name.domain_name>:<port>/docs/index.html`.

`com.sun.identity.saml`

This package contains the `AssertionManager` and `SAMLClient` classes. The `AssertionManager` provides interfaces and methods to create and get assertions, authentication assertions and assertion artifacts; it is the connection between the SAML specification and the Identity Server. Some of the methods included are:

- `createAssertion`—creates an assertion with an authentication statement based on an Identity Server SSO Token ID.

- `createAssertionArtifact`—creates an artifact that references an assertion based on an Identity Server SSO Token ID
- `getAssertion`—returns an assertion based on the given parameter (given artifact, assertion ID or query).

The `SAMLClient`, on the other hand, provides methods to execute either the Web Artifact or Web POST profile from within an application as opposed to a web browser.

com.sun.identity.saml.assertion

This package contains the classes needed to stand for, transform, and integrate, an XML assertion into the application. For example, Code Example 8-2 illustrates how to use the `Attribute` class and `getAttributeValue` method to get the value of an attribute. From an `Assertion`, call the `getStatement()` method to retrieve a set of statements. If a statement is an `AttributeStatement`, call the `getAttribute()` method to get a list of attributes. From there, call `getAttributeValue()` to retrieve the `AttributeValue`.

Code Example 8-2 Sample Code To Get 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

This class defines a number of XML attributes (and some utility methods) common to all SAML elements. It also contains all SAML-related exceptions.

com.sun.identity.saml.plugins

Identity Server provides four SPIs, three of them with default implementations. The implementations of these SPIs can be altered, or brand new ones written, based on the specifications of a particular customized service. These can then be used to integrate the SAML service into the custom service. Currently, the APIs include the `AccountMapper`, `ActionMapper`, `AttributeMapper` and `SiteAttributeMapper`.

- `AccountMapper` is used to map Identity Server accounts from external partner sites to Identity Server for SSO purposes. A default account mapper implementation is provided. If a site-specific account mapper is not supplied, this default mapper is used.
- `AttributeMapper` is used in the `AttributeQuery` case. When a site receives an `AttributeQuery`, this mapper is called to obtain the `SSOToken` or an `Assertion` containing `AuthenticationStatement` from the query. It is also used to convert the attribute in the query to an attribute the site understands. A default attribute mapper is provided.
- `ActionMapper` is used to get SSO information and to map partner actions to Identity Server authorization decisions. A default action mapper implementation is provided. If a site-specific action mapper is not supplied, this default mapper is used.
- `SiteAttributeMapper` is also used for SSO. The default functionality of Identity Server is that when no mapper is specified and an assertion is created, either through web artifact or POST profile, it only contains `AuthenticationStatement(s)`. If a site wants to include `AttributeStatement(s)`, it can use this SPI to obtain the attributes. It creates `AttributeStatement(s)` from those attributes, and puts them inside the assertion.

NOTE The default behavior is that no attribute statements are returned unless specified in the plug-in.

com.sun.identity.saml.protocol

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

AuthenticationQuery

The `AuthenticationQuery` class represents an authentication query. An application sends a SAML request with an `AuthenticationQuery` inside. 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`, then the info in the `NameIdentifier` should be the same as the one in the SSOToken.

AttributeQuery

The `AttributeQuery` class represents a query concerning an identity's attributes. An application sends a SAML request with an `AttributeQuery` inside. The application develops an `AttributeMapper` to obtain either a SSOToken ID or an Assertion containing an `AuthenticationStatement` from the query and the mapper is then used to retrieve the attributes for the Subject. If no `AttributeMapper` for the querying site is found, then the `DefaultAttributeMapper` will be used. To use the `DefaultAttributeMapper`, the application should put either the SSOToken ID or an assertion containing an `AuthenticationStatement` in the `SubjectConfirmationData` element of the Subject in the query. If an SSOToken ID is used, then the `ConfirmationMethod` must be set to `urn:com:sun:identity:`. If an assertion is used, then this assertion should be issued by the Identity Server instance processing the query or a server that is trusted by the Identity Server instance processing the query.

NOTE In `DefaultAttributeMapper`, it is possible to query a subject's attributes using another subject's SSOToken as long as the SSOToken has the privilege of retrieving those attributes.

For a query using the `DefaultAttributeMapper`, any matching attributes found in the Identity Management module 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. `userServiceNameList`'s value is user service names separated by a comma.

AuthorizationDecisionQuery

The `AuthorizationDecisionQuery` class represents a query concerning an identity's authority to access protected resources. An application sends a SAML request with an `AuthorizationDecisionQuery` inside. The application develops an `ActionMapper` to obtain an SSOToken ID. The mapper is then used to retrieve the authentication decisions for the actions defined in the query.

If no `ActionMapper` for the querying site is found in the configuration, a `DefaultActionMapper` will be used. To use the `DefaultActionMapper`, the application should put the `SSOToken ID` in the `SubjectConfirmationData` element of the `Subject` in the query. If `SSOToken ID` is used, then the `ConfirmationMethod` must be set to `urn:com:sun:identity:.` If a `NameIdentifier` is present, then the info in the `SSOToken` must be the same as the one in the `NameIdentifier`.

NOTE The `DefaultActionMapper` handles actions in action namespace `urn:oasis:names:tc:SAML:1.0:ghpp` only. The `iPlanetAMWebAgentService` is used to serve the policy decisions for this action namespace.

The application may also pass in the authentication information through the `Evidence` element in the query. The `Evidence` could be an `AssertionIDReference` or an assertion containing an `AuthenticationStatement` issued by the Identity Server instance processing the query, or an assertion issued by a server that is trusted by the Identity Server instance processing the query. The `Subject` in the `AuthenticationStatement` as the evidence should be the same as the one in the query.

NOTE Policy conditions can be passed in through `AttributeStatements` of `Assertion(s)` inside the `Evidence` of the query. If the value of an attribute contains `TEXT` node only, then the condition is set as `attributeName=attributeValueString`; otherwise, the condition is set as `attributename=attributeValueElement`.

com.sun.identity.saml.xmlsig

All SAML assertions, requests and responses may be signed using this signature API. This package contains the classes needed to sign and verify.

SAML Service Samples

There are several samples that can be accessed from the Identity Server installation. These samples illustrate how the SAML service can be used in different ways. They include:

- A sample that serves as the basis for using the SAML client API. This sample is located in `<identity_server_root>/SUNWam/samples/SAML/client`.

- A sample that illustrates how to form a Query, and write an AttributeMapper as well as how to send and process a SOAP message using the SAML SDK. This sample is located in
`<identity_server_root>/SUNWam/samples/SAML/query.`
- A sample application for achieving SSO using the Web Artifact profile or the Web POST profile. This sample is located in
`<identity_server_root>/SUNWam/samples/SAML/sso.`
- A sample that illustrates how to use the XMLSIG API. It is located in
`<identity_server_root>/SUNWam/samples/SAML/xmlsig.`

Federation Management

Sun™ One Identity Server 6.0 contains a Federation Management module which implements the open standards for federated network identity being developed by the Liberty Alliance Project. This chapter explains the Liberty Alliance Project and the concept of federated network identity as well as describing how it is integrated within the Identity Server. It contains the following sections:

- Overview
- Federation Management Process
- Federation Management API
- Customizing The Module
- Federation Management Samples

Overview

On the Internet, one person might have a multitude of accounts set up to access various business, community and personal service providers; for example, the person might have used different names, user IDs, passwords or preferences to set up accounts for 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 service provider. These attributes serve to uniquely identify the individual with that provider and may include a name, phone number, social security number, address, credit records, bank balances or bill payment information.

Because the Internet is fast becoming the prime vehicle for business, community and personal interactions, it has become necessary to fashion a system for online users to aggregate their local identities, enabling them to have one *network identity*. This system is *identity federation*. Identity federation allows a user to associate,

connect or bind multiple Internet service providers' local identities. A network identity allows users to login at one service provider's site and then go to an affiliated site without having to re-authenticate or re-establish their identity. The Liberty Alliance Project was implemented to make identity federation a reality.

The Liberty Alliance Project

The goal of the Liberty Alliance Project is to enable individuals and organizations to more easily conduct transactions while protecting the individual's identity. To accomplish this, the Alliance has established specifications for identity federation that enables:

- Opt-in account linking where users can choose to federate different internet service provider accounts.
- Simplified single sign-on where a user can log in and authenticate with one provider's federated account and navigate to another account without having to log in again.
- Authentication context where organizations with linked accounts communicate the type and level of authentication that should be used when the user logs in.
- Global log-out where a user logs out of the site to which they initially logged in and is automatically logged out of all sites that maintain a live session.
- A client feature which can be implemented in fixed and wireless devices to facilitate use of the Liberty specifications.

These capabilities can be achieved when commercial or non-commercial organizations join together into a 'circle of trust' based on Liberty-enabled technology and operational agreements. This 'circle of trust' is referred to as an *authentication domain*. The authentication domain includes service providers (who offer web-based services to users), identity providers (service providers who also offer federated authentication), and the users themselves. Once an authentication domain is established, users can federate any or all identities they might have with the service providers that have joined this domain, enabling them to make use of the federated authentication capabilities.

Liberty Specification Concepts

The Federation Management module built into the Identity Server is designed to be compatible with the Liberty Alliance Project's Version 1.0 specifications. A number of concepts are derived from these specifications. They include:

Service Provider

Service providers are commercial or not-for-profit organizations that offer web-based services. This broad category can include internet portals, retailers, transportation providers, financial institutions, entertainment companies, and governmental agencies.

Identity Provider

Identity providers are service providers that specialize in providing authentication services. In the Liberty context, authentication done by an identity provider is honored by all service providers with whom it is affiliated.

Authentication Domain

An Authentication Domain is a group of affiliated service providers consisting of one or more identity providers. The group is also referred to as a 'circle of trust'. Once established, single sign-on is enabled within the authentication domain.

Trusted Provider

A Trusted Provider is one of a group of service and identity providers, affiliated together based on the Liberty architecture and operational agreements, with whom users can transact and communicate in a secure environment.

Account Federation (Identity Federation)

Account federation occurs when a user unites accounts that were initially set up with distinct service and identity providers. Users retain their individual accounts with each provider in the Authentication Domain while, simultaneously, establishing a link that allows the exchange of user information between them.

Federated Identity

A federated identity refers to the amalgamation of a user's distinct service provider's account attributes (personal data, online configurations, buying habits and history, shopping preferences, etc.). The information is still administered by the user, yet it is securely shared with the organizations of their choosing.

Federation Termination (Defederation)

Users have the ability to terminate federations. Federation termination results in the cancellation of affiliations established between the user's identity provider account and federated service provider accounts.

Single Sign-on

Single sign-on (SSO) is established when a user with a federated identity authenticates to an identity provider and is then able to access affiliated service providers without having to authenticate again.

Single Logout

When a user logs out from an identity provider or a service provider, they will effectively be logged out from all service providers or identity providers in that authentication domain.

Common Domain

When authenticated, an identity provider writes a cookie stating the user's preferred identity provider (itself). However, due to the constraints in cookie standards, there is no way for an identity provider in one DNS domain to write a cookie that a service provider in another DNS domain can read. To work around this situation, the Liberty specification advocates the use of a Common Domain (also known as third level domain).

Name Identifier

Identity federation maps a user's account information across a number of service and identity provider organizations. The user's identity is exchanged between the identity and service providers as a *name identifier*, and is stored in the Directory Server data store.

Federation Management Process

Out of the box, Identity Server has two options for user or application authentication. The first is the Identity Server Authentication Service and the second is the Liberty-enabled Federation Management Service. In an Identity Server scenario when a user or application tries to access a resource protected by the Identity Server, the user is redirected to the Authentication Service via a Login page for access authorization. When the user provides credentials, the authentication module verifies them and either allows or denies access.

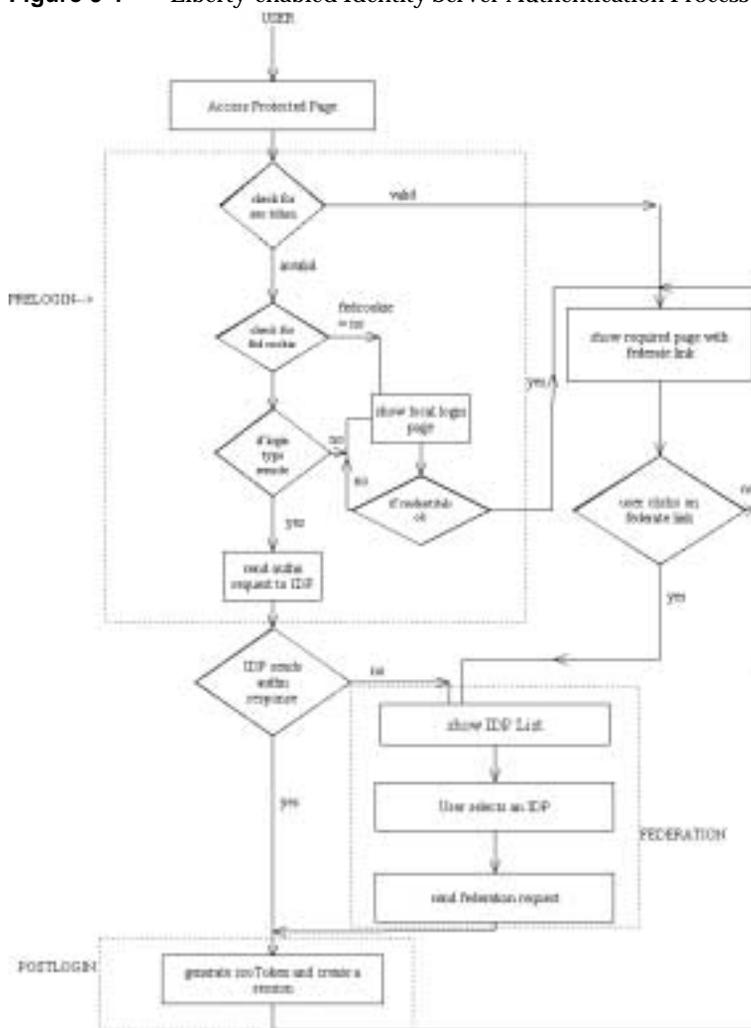
NOTE More information on the Authentication Service can be found in Chapter 3, “Authentication Service.”

In a scenario where the Identity Server is Liberty-enabled and a user or application attempts to access a protected resource, the user is redirected to a Pre-Login page which invokes the Federation Management Service’s Pre-Login servlet. This servlet searches for either a valid Identity Server single sign-on token or a valid Federation Cookie (which indicates that a user has federated his account using this Identity Server provider). If an SSO token is found, the user’s Federation information is retrieved, and the user is authenticated; a Federation Cookie is also set and the user is returned to the target resource.

NOTE The federation cookie is different from the Identity Server cookie discussed in “Cookies and Session Tokens,” on page 81. By default, the federation cookie is a persistent cookie and there is currently no option to disable this.

If a Federation Cookie is found, the user is directed to the Federation Single Sign-On Service which provides an Authentication Assertion allowing the user access to the target resource. If neither of these items is found, the user is redirected to the Identity Server Authentication Service where, upon successful authentication, they are directed to the Post-Login page which invokes the Post-Login servlet. This servlet processes the user’s Identity Server authentication and initiates the Federation Management Single Sign-On Service which, once again, provides an Authentication Assertion to allow the user access to the target resource. Figure 9-1 on page 238 illustrates this flow.

Figure 9-1 Liberty-enabled Identity Server Authentication Process Flow



Federation Management Protocols

In order to enable the federation process, the Liberty Alliance Project's Phase I Specifications define the following protocols that are implemented by the Identity Server:

Single Sign-on and Federation Protocol

This is the protocol used to federate a user's identity for a service provider with their identity for an identity provider, thus enabling single sign-on. It also specifies the means by which a service provider obtains an Authentication Assertion from an identity provider to provide single sign-on to the user. There are two types which either the identity or service provider can implement:

- SOAP - based Single Sign On and Federation Protocol relies on a SOAP call from the service provider to the identity provider.
- Form POST - based Single Sign On and Federation Protocol relies on a form POST to communicate between the service provider and the identity provider.

Federation Termination Notification Protocol

This is the protocol used to notify providers when a user's existing federated identity is terminated. The termination can be initiated at either the identity or service provider. The provider will notify all other providers in the Authentication Domain when a user defederates their identity. There are two types of notification which either the identity or service provider can implement:

- SOAP - based Federation Termination Notification Protocol relies on a SOAP call from the service provider to the identity provider.
- Form POST - based Federation Termination Notification Protocol relies on a form POST to communicate between the service provider and the identity provider.

Name Registration Protocol

At the time of federating a user account, the identity provider generates a name identifier that serves as the term the identity provider and the service provider use in referring to the user when communicating. This is the *IDP Provided NameIdentifier*. Subsequent to federation, however, the service provider may register a different name identifier with the identity provider. This is the *SP Provided NameIdentifier*. The identity provider must use the *SP Provided NameIdentifier* when communicating with the service provider about the user until after federation when they will both use the *IDP Provided NameIdentifier*.

Single Log-Out Protocol

This is the protocol used to synchronize the session log-out functionality across all sessions that were authenticated and opened by a particular identity provider. There are two types which either the identity or service provider can implement:

- SOAP-based Single Log-Out Protocol relies on asynchronous SOAP messaging between service providers and identity providers.
- Form POST-based Single Log-Out Protocol relies on a form POST to communicate between service providers and identity providers.

IDP Introduction Protocol

In federation networks having more than one identity provider, the service providers need a way to determine which identity provider(s) is the user's preferred identity provider. The Liberty specification defines a protocol which relies on a cookie written in a domain that is common between identity providers and service providers. This predetermined domain is the *common domain* and the cookie containing the preferred identity provider is known as the *common domain cookie*. The service provider can read this cookie value to identify a user's preferred identity provider and get authentication assertions from that identity provider. Both identity providers and service providers implement this protocol.

Federation Management API

The `LibertyManager` class forms the basis of the Federation Management APIs. This interface is instantiated by web applications that want to access the Federation Management module. It contains the methods needed by the module JSPs for account federation, session termination, log in, log out and other actions. These methods include:

- `getSPList()`—which returns a list of all trusted service providers.
- `getSPList(String hostedProviderID)`—which returns a list of all trusted service providers for the specified hosted provider.
- `getIDPList()`—which returns a list of all trusted identity providers.
- `getIDPList(String hostedProviderID)`—which returns a list of all trusted identity providers for the specified hosted provider.
- `getSPFederationStatus(String user, String provider)`—which retrieves the federations status of a user with a service provider. This method assumes that the user is already federated with the provider.
- `getIDPFederationStatus(String user, String provider)`—which retrieves the federation status of a user with an identity provider. This method assumes that the user is already federated with the provider.

- `getFederatedProviders(String userName)`—which returns a specific user's federated providers.
- `getProvidersToFederate(String providerID, String userName)`—which returns the list of all trusted identity providers to which the specified user is not already federated.
- `getListOfCOTs(String providerID)`—which returns a list of authentication domains for the given provider.

NOTE The Identity Server Javadocs can be accessed from any browser by copying the complete `<identity_server_root>/SUNWam/docs/` directory into the `<identity_server_root>/SUNWam/public_html` directory and pointing the browser to `http://<server_name.domain_name>:<port>/docs/index.html`.

Customizing The Module

The Federation Management module uses JSP files to define the look and feel of its pages. An administrator can customize the JSPs by changing the tags accordingly. The JSPs can be found in the `<identity_server_root>/SUNWam/web-apps/services/config/federation/default/` directory and include:

- `CommonLogin.jsp`—displays links to the login pages of the trusted identity providers as well as the local login link. It is displayed when the user is not locally logged in or not logged in at the identity provider site. The list of trusted identity providers is obtained by the `getIDPList(hostedProviderID)` method.
- `Error.jsp`—displays an error page when one has occurred.
- `Federate.jsp`—is displayed when the user clicks the Federate link in the `index.jsp`. It displays a drop-down menu that lists all providers with which the user is not yet federated. This list is constructed from the `getProvidersToFederate(userName, providerID)` method which returns all active providers to which the user is not yet federated.
- `FederationDone.jsp`—displays the status of federation (success or cancelled). It checks this status using the `isFederationCancelled(request)` method.
- `Footer.jsp`—displays a branded footer.
- `Header.jsp`—displays a branded header.

- `ListOfCOTs.jsp`—displays multiple authentication domains (or circles of trust) when the service provider belongs to more than one. When a user is authenticated by an identity provider and the provider belongs to more than one authentication domain, they will be shown the `ListOfCOTs.jsp` to select one domain as the preferred domain. In the case that the provider belongs to only one domain, then this page will not display as, by default, the one domain is the preferred domain. The list of authentication domains is obtained by using the `getListOfCOTs(providerID)` method.
- `LogoutDone.jsp`—displays the status of the local logout.
- `Termination.jsp`—is displayed when the user clicks the defederate link. It shows a drop-down menu of all providers to which the user has already federated; from this list, the user can choose to defederate. The list is constructed using 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). It checks this status using the `isTerminationCancelled(request)` method.

The files in this directory provide a default GUI for the module. To customize it for a specific organization, this `default` directory can be copied and renamed to reflect the name of the organization (or any value). It would then be placed at the same level as the `default` directory and the files within this directory could then be modified as needed.

Federation Management Samples

There are a number of samples included with the Identity Server that demonstrate the different protocols used in the Federation Management module. They are located in the `<identity_server_root>/SUNWam/samples/liberty/` directory. Instructions on how to implement the samples can be found in the `README` file.

- Sample 1 illustrates a scenario with one Service Provider and one Identity Provider configured on two separate Identity Server installations. Two server machines are required.
- Sample 2 illustrates a scenario with one Service Provider whose resources are deployed on a Sun ONE Web Server protected by an Identity Server Policy Agent and one Identity Provider. At least two server machines are required for this sample also.

- Sample 3 illustrates a multiple hosted providers scenario with two Service Providers and two Identity Providers. This sample scenario requires only one server machine and one Identity Server installation. Four hosted providers (two Service Providers and two Identity Providers) are created on the same Identity Server Installation.

Logging Service

The Sun™ One Identity Server provides a Logging Service to record information such as user activity, traffic patterns, and authorization violations. In addition, Identity Server includes a Logging API to allow external applications to take advantage of the Logging Service. This chapter explains the service and the API. It contains the following sections:

- Overview
- Log Message Formats
- Logging API
- Logging SPI
- Debug Files
- Secure Logging

Overview

The Logging Service enables all Identity Server services to record information that might be useful to the administrator in one centralized location. The recorded information may include access denials and approvals, authorization violations and code exceptions. This information allows administrators to analyze user activity, Identity Server traffic patterns and authorization violations. As with all Identity Server services, the Logging Service uses a global configuration file, named `amLogging.xml`, to define its attributes (such as maximum log size and log location), or whether the log information is written to a flat file or a relational database.

NOTE The directory location for all logs is `/var/opt/SUNWam`.

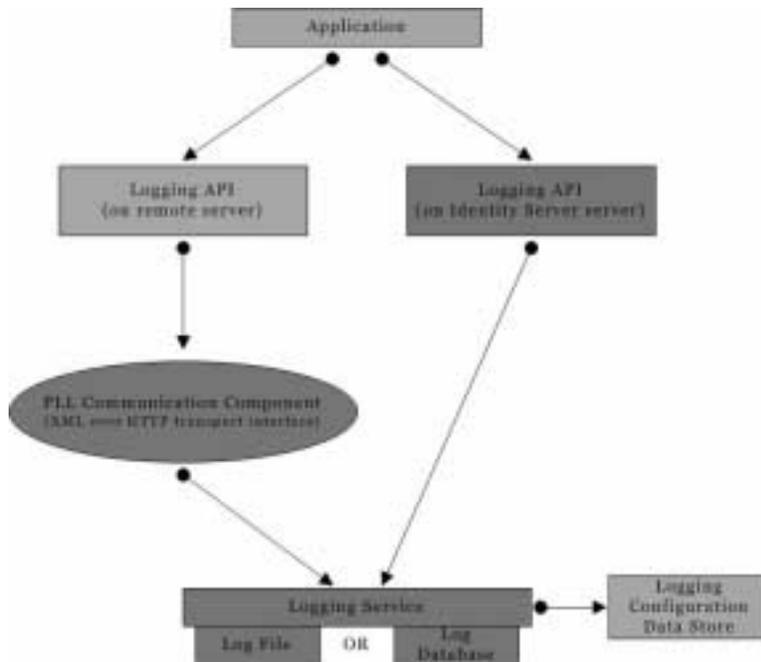
Logging Architecture

External Java applications use the Logging API to access the Logging Service. These interfaces may reside on a remote server or on the same server as the Logging Service. If the APIs live remotely, the PLL Communication Component, an XML over HTTP interface, is used to send the logging request to the Logging Service.

NOTE The logging architecture extends the Java™ 1.4 Logging API specifications.

An application accesses the Logging Service by calling the Logging API. Upon receiving a request, the Logging Service loads the configuration data stored in the Directory Server using the Identity Server SDK. Any exception message will be logged, based on these configuration values. On any error, a `LoginException` is thrown. Figure 10-1 illustrates the architecture of the Logging Service.

Figure 10-1 Logging Service Architecture



Logging Service XML File

The Logging Service holds the attributes and values for the logging function. These attributes and values are defined in the `amLogging.xml` service file located in `<identity_server_root>/SUNWam/config/xml`. The values defined in `amLogging.xml` are applied across the Identity Server deployment and are inherited by every configured organization. More information on the Logging Service and its attributes can be found in the *Sun ONE Identity Server Administration Guide*.

Log Security

An optional logging feature adds additional security to the log files in terms of tamper detection. No special coding is required to leverage this feature. Please refer to the *Sun ONE Identity Server Administration Guide* for steps to turn on and configure this secure logging feature.

Log Message Formats

Identity Server supports logging messages stored in both, a text file and a relational database. The following sections explain the data storage formats used in these formats.

Flat File Format

The default flat file format is the W3C Extended Log Format (ELF). In leveraging this format, the Logging Service records time, Data, HostName, LoginID, LogLevel, Domain and IPAddr fields in each log record.

- `time` is the date (yyyy/mm/dd) and time (hh:mm:ss) at which the log message was recorded.
- `Data` is the description of the user activity, errors or other useful information which the application wants to log.
- `HostName` is the hostname from which the operation was performed.
- `LoginID` is the ID of the user attempting to access the application.
- `LogLevel` corresponds to the JDK1.4 | LogLevel of the log record.
- `Domain` is the Identity Server domain to which the user belongs.

- IPAddr is the IP address from which the operation was performed.

Code Example 10-1 illustrates a log record formatted for a flat file.

Code Example 10-1 Flat File Formatted Log Record Sample

```
#Version: 1.0
#Fields: time Data HostName LoginID LogLevel Domain IPAddr
"13-11-2002 18:34:50" "Login Success
UserId->uid=amAdmin,ou=People,dc=example,dc=com
UserDomain->dc=sun,dc=com service->adminconsole.service
testmachine.example.com "cn=user,ou=Users,dc=example,dc=com" INFO
dc=example,dc=com testmachine.example.com/134.135.134.135
```

Relational Database Format

For applications using a relational database to log messages, the message is stored in a database table. Identity Server uses Java Database Connectivity (JDBC) to access data from Java programs in an Oracle® environment. The database schema is as follows:

Table 10-1 Relational Database Log Format

Column Name	Data Type	Description
TIME	VARCHAR2(30)	Date of the log in the format yyyy/mm/dd hh:mm:ss.
DATA	VARCHAR2(1024)	The log message itself.
HOSTNAME	VARCHAR2(300)	Host name of machine from which the logged operation was performed.
LOGINID	VARCHAR2(300)	Login ID of the user who performed the logged operation.
LOGLEVEL	VARCHAR2(300)	JDK 1.4 log level of the log record.
DOMAIN	VARCHAR2(300)	Identity Server domain of the user.
IPADDR	VARCHAR2(300)	IP Address of the machine from which the logged operation was performed.

NOTE There is a limitation in the log name length for Oracle JDBC logging: the length of the log name cannot exceed 30 characters. Oracle does not support longer names.

Logging API

The Logging API provides log management tools for all Identity Server services as well as providing a set of Java classes for external applications to create, retrieve, submit, or delete log information. These API extend the JDK 1.4 API. The main classes are `Logger` and `LogRecord`. They are contained in the package `com.sun.identity.log`.

NOTE The Identity Server Javadocs can be accessed from any browser by copying the complete `<identity_server_root>/SUNWam/docs/` directory into the `<identity_server_root>/SUNWam/public_html` directory and pointing the browser to `http://<server_name.domain_name>:<port>/docs/index.html`.

Logger Class

This `Logger` class provides the methods for applications to use in creating log files and writing log information to them.

- The `getLogger()` method returns a logger object and simultaneously creates a log in the designated logging location.
- The `log()` method records a single piece of log information or a `LogRecord`. It allows an application to submit a logging message to a predetermined log.

LogRecord Class

The `LogRecord` class provides the means to represent the information that needs to be logged. Each instance represents a single piece of log information or `LogRecord` that comes from the application.

Logging Exceptions

There are a number of exceptions that can be thrown using the Logging APIs. The generic `LogException` is probably the most common. It signals an error condition while logging a message. Other exceptions include:

- `ConnectionException`—This exception is thrown when the connection to the database fails.
- `DriverLoadException`—This exception is thrown when the JDBC driver load fails.
- `InvalidLogNameException`—This exception is thrown when the log name is invalid.
- `LogAlreadyExistException`—This exception is thrown when the log already exists.
- `LogCreateException`—This exception is thrown when log creation fails.
- `LogDeleteException`—This exception is thrown when the log deletion fails.
- `LogException`—A `LogException` is thrown when applications are denied log access because they don't have the privileges or a valid session.
- `LogFatalException`—This exception is thrown when a fatal error occurs.
- `LogHandlerException`—A `LogException` is thrown when a log handler error is encountered.
- `LogInactiveException`—A `LogException` is thrown when the log is in inactive status. (Inactive/active status is not currently supported.)
- `LogInvalidSessionException`—This exception is thrown when an application accesses a log which does not exist.
- `LogNotFoundException`—This exception is thrown when an application accesses a log which does not exist.
- `LogPrivDeniedException`—A `LogException` is thrown when the access privilege is denied.
- `LogProfileException`—A `LogException` is thrown when access privilege is denied.
- `LogReadExceedsMaxException`—A `LogException` is thrown when the log size exceeds the maximum size defined in the Logging service.
- `LogReadException`—A `LogException` is thrown when an error is encountered in retrieving the log information.

- `LogTypeException`—This exception is thrown when a log type error occurs.
- `LogWriteException`—This exception is thrown when the log record submission fails.
- `NullLocationException`—This exception is thrown when the location is null.

Sample Logging Code

Code Example 10-2 provides sample code to illustrate uses for the Identity Server logging classes.

Code Example 10-2 Logging API Samples

```

Logger logger = Logger.getLogger("SampleLogFile");
// Creates the file or table in the LogLocation specified in the
amLogging.xml and returns the Logger object.

LogRecord lr = new LogRecord(Level.INFO, "SampleData", ssoToken);
// Creates the LogRecord filling details from ssoToken.

logger.log(lr, ssoToken);
// Writes the info into the backend file, db or remote server.

```

Logging SPI

The Logging Service framework allows a customer to plug in a class which can decide whether a `LogRecord` should be retained or discarded based on the authorization of the owner of the `SSOToken` to perform predefined log operations. For using this facility, the customer must define a logging policy using the policy framework and use it from his plugin to take the decision whether the owner of the `SSOToken` has permissions to perform the requested logging operation.

Plugin Log Verifier

If secure logging is enabled, the log files are verified periodically to detect any attempt of tampering. The customers can customize the action taken if a tampering is detected, by following the steps below.

1. Implement the `com.sun.identity.log.spi.IVerifierOutput` interface, programming it for the desired functionality.

2. Add the implementing class in the classpath of Identity Server.
3. Modify the property `iplanet-am-logging-verifier-action-class` in the `<identity_server_root>/SUNWam/config/xml/amLogging.xml` file with the name of the new class.

Plugin Authorization Mechanism

The logging framework allows the customer to plugin a class which decides whether a `LogRecord` should be logged or discarded based on the authorization of the owner of the `SSOToken` to perform predefined log operations. For using this facility, the customer must define his logging policy using the policy framework and use it from his plugin to take the decision whether the owner of the `SSOToken` has permissions to perform the requested logging operation.

1. Implement the `com.sun.identity.log.spi.IAuthorizer` interface programming it for the desired functionality.
2. Add the implementing class in the classpath of Identity Server.
3. Modify the property `iplanet-am-logging-authz-class` in the `<identity_server_root>/SUNWam/config/xml/amLogging.xml` file with the name of the new class.

Log Files

By default, Identity Server currently records events in four logs. These files should be monitored by the administrator on a regular basis. The directory for the log files can be found in `var/opt/SUNWam`.

NOTE The policy agents are responsible for logging exceptions related to resource access or denial; in other words, policy-related issues. For more information on this function, see the *Sun ONE Identity Server Policy Agent Guide*.

SSO-related Logs

The Logging Service logs the following events for the SSO component:

- Login
- Logout

- Session Idle TimeOut
- Session Max TimeOut
- Failed To Login
- Session Reactivation
- Session Destroy

The log file is called `amSSO` and is stored in the `var/opt/SUNWam/logs` directory.

Console-related Logs

The Identity Server console logs record the creation, deletion and modification of identity-related objects, policies and services including, among others, organizations, organizational units, users, roles, policies and groups. It also records modifications of user attributes including passwords and the addition or removal of users to or from roles and groups. The log is named `amConsole` and is stored in the `var/opt/SUNWam/logs` directory.

Authentication-related Logs

The Logging component logs user logins and logouts. The log is named `amAuthentication` and is stored in the `var/opt/SUNWam/logs` directory.

Federation-related Logs

The Federation component logs federation-related events including, but not limited to, the creation of an Authentication Domain and the creation of a Hosted Provider. The log is named `amFederation` and is stored in the `var/opt/SUNWam/logs` directory.

Debug Files

Debug files are stored in `var/opt/SUNWam/debug`. This location, along with the level of the debug information, is configurable in the `AMConfig.properties` file, located in the `<identity_server_root>/SUNWam/lib/` directory. The debug files may be monitored in the event of, for example, a product crash. The administrator can try to understand the reason for an error situation from these files.

Secure Logging

Secure Logging enables the detection of unauthorized changes or tampering with the security logs. The Identity Server administrator can enable secure logging by following this procedure:

1. Create a web server certificate with the name *Logger* and install it in the Sun ONE Web Server running the Identity Server.

Refer to the Sun ONE Web Server documentation for instructions on this detailed procedure.

2. Select the Logging Service under the Service Configuration module, turn on Secure Logging and click Save.
3. Create a file in the `<identity_server_root>/SUNWam/config` directory named `.wtpass` which contains the Web Server administrator password.

Ensure that read permission is given only to the user running the Web Server process. The administrator can configure the log sign interval and log verification interval from the Identity Server console.

4. Restart the Web Server after making these changes.

Client Detection

The Sun™ One Identity Server may be accessed using multiple clients types, whether HTML-based, WML-based or other protocols. In order for this function to work, Identity Server must be able to identify the client type. The client detection API is used for this purpose. This chapter offers information on the API, and how it can be used to recognize the client type. It contains the following sections:

- Overview
- Client Data
- Client Detection API

Overview

Identity Server has the capability to process requests from multiple client type browsers. The client detection API can be used to determine the protocol used by the requesting client browser and retrieve the correctly formatted pages for the particular client type.

NOTE Currently, Identity Server only defines client data for supported HTML client browsers including Internet Explorer and Netscape Communicator.

Since any browser type requesting access to the Identity Server must first be successfully authenticated, client detection is accomplished within the Authentication Service. When a client's HTTP request is passed to the Identity Server, it is directed to the Authentication Service. Within this framework, the first step in user validation is to identify the browser type using information stored in the HTTP string request. The Authentication Service then uses this information to retrieve the browser type's characteristics. The characteristics are configured and

stored in the `amClientDetection.xml` file and are referred to as the *client data*. Based on this client data, correctly formatted authentication pages are sent back to the client browser (for example, HTML or WML pages). Once the user is validated, the client type is added to the session token (as the key `clientType`) where it can be retrieved and used by other Identity Server services.

NOTE The client detection mechanism is disabled by default which assumes the client to be of the `genericHTML` type. All client data associated with `genericHTML`, as explained in “Client Data,” on page 256, will be used.

Client Data

In order to recognize client types, Identity Server stores their identifying characteristics in its Directory Server data store. This *client data* identifies the features of all of the particular deployment’s supported client browsers. Client data for supported client types are defined in the `amClientDetection.xml` file. The attribute in which it is defined is `iplanet-am-client-detection-client-types`. The different aspects of the client data are separated by a pipe (“|”) as follows:

```
clientType=<value>|userAgent=<value>|contentType=<value>|cookieSupport=<value>|fileIdentifier=<value>|filePath=<value>|charset=<value>.
```

The fields defined are:

- `clientType`—an arbitrary string which uniquely identifies the client. The default is `genericHTML`.
- `UserAgent`—a search filter used to compare/match the user-agent defined in the HTTP header. The default is `Mozilla/4.0`.
- `contentType`—defines the content type of the HTTP request. The default is `text/html`.
- `cookieSupport`—defines whether cookies are supported by the client browser. The default is `true`.
- `fileIdentifier`—is not used at this time.
- `filePath`—is used to locate the client type files (templates and JSP files). The default is `html`.

- `charset`—defines the character encoding used by Identity Server to send a response to the browser. The default value is `UTF-8`. The character set can be configured for any given locale by adding `charset_locale=codeset` where the code set name is based on the Internet Assigned Numbers Authority (IANA) standard.

NOTE In order to enable client detection for the Identity Server deployment, the `iplanet-am-auth-client-detection-enabled` attribute, also defined in the `amClientDetection.xml` file, must be set to `true`.

Client Detection API

By default, Identity Server only includes client detection functionality for browsers that use HTML. But, it is packaged with an API for writing proprietary client detectors that can retrieve any client data. The client detection API are in a package called `com.ipplanet.services.cdm`. This package provides the interfaces and classes to detect any client browser types. The procedure would include defining the client type characteristics for the new module (as stated in “,” on page 256) as well as implementing the client detection API within the external application. Identity Server services can be accessed by multiple client browser types. For example, a client accessing Identity Server may be a HTML client type or a WML client type. As any client browser requesting access to an Identity Server service must be successfully authenticated, client detection is accomplished as part of the Authentication Service. This service identifies the client type from its incoming `HttpRequest` for access, using the `getClientType` method in the `ClientDetectionInterface` interface. Upon successful authentication, the client type is then added to the user’s session token where other applications can find it and use the client detection API to retrieve it.

NOTE The Identity Server Javadocs can be accessed from any browser by copying the complete `<identity_server_root>/SUNWam/docs/` directory into the `<identity_server_root>/SUNWam/public_html` directory and pointing the browser to `http://<server_name.domain_name>:<port>/docs/index.html`.

Client Detection Module Interface

Client detection capability is provided by the `ClientDetectionInterface` interface. It contains a `getClientType` method which is called by the Authentication Service when a new login request is received. The Authentication Service executes the retrieval of the value of the `iplanet-am-auth-client-detection-class` attribute to determine the name of the implementing class of the `ClientDetectionInterface`. The service then passes the `HttpRequest` to the `getClientType` method which does the actual client detection and returns the `clientType` as a string. The default implementation will assume the client type to be the defined default type. An error condition will be handled by the `ClientDetectionException` class. Code Example 11-1 below is an example implementation of the `ClientDetectionInterface`.

Code Example 11-1 Implementation of the `ClientDetectionInterface`

```

/**
 * $Id: ClientDetectionDefaultImpl.java,v 1.2 2002/05/09 01:27:40
denz Exp $
 * Copyright 2001 Sun Microsystems, Inc. Some preexisting
 * portions Copyright 2001 Netscape Communications Corp.
 * All rights reserved. Use of this product is subject to
 * license terms. Federal Acquisitions: Commercial Software --
 * Government Users Subject to Standard License Terms and
 * Conditions.
 *
 * Sun, Sun Microsystems, the Sun logo, and iPlanet are
 * trademarks or registered trademarks of Sun Microsystems, Inc.
 * in the United States and other countries. Netscape and the
 * Netscape N logo are registered trademarks of Netscape
 * Communications Corporation in the U.S. and other countries.
 * Other Netscape logos, product names, and service names are
 * also trademarks of Netscape Communications Corporation,
 * which may be registered in other countries.
 */

package com.iplanet.services.cdm;

/* iPlanet-PUBLIC-CLASS */

import javax.servlet.http.HttpServletRequest;
import java.util.HashMap;
import com.iplanet.am.util.*;

/**
 * The <code>ClientDetectionInterface</code> interface needs to
 * be implemented by services and applications serving multiple
 * clients, to determine the client from which the request has
 * originated. This interface detects the clientType from the
 * client request.
 */

```

Code Example 11-1 Implementation of the ClientDetectionInterface

```

public class ClientDetectionDefaultImpl implements
ClientDetectionInterface
{
    /** Detects the client type based on the request object
     * @param request Http Servlet Request
     * @return a String representing the client type
     * @exception ClientDetectionException when there is an error
     * retrieving client data
     */

    /* For caching purpose, HashMap is used */
    private static HashMap agentTable = new HashMap (10);
    private static Debug debug =
Debug.getInstance("amClientDetection");

    public String getClientType(HttpServletRequest request)
        throws ClientDetectionException {

        String lstr = request.getHeader("User-Agent");
        if (debug.messageEnabled() ) {
            debug.message("DefaultImpl Agent = "+lstr);
        }

        String result;

        /* Check if it is in the cache */

        result = (String)agentTable.get(lstr);
        if (result != null && result.length() != 0) {
            return result;
        }
        result = "genericHTML"; // Set the default value

        /* Known formats if User-Agents are
        Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0) ==>
MSIE
        Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 4.0) ==>
MSIE
        Mozilla/4.76 [en] (X11; U; SunOS 5.8 sun4u) ==>
NSCP_UNIX
        Mozilla/5.0 (X11; U; Linux i686; en-US; rv:0.9.2.1)
Gecko/20010901 ==> NSCP_UNIX
        Mozilla/4.79 [en] (Windows NT 5.0; U) ==> NSCP_WIN32
        Mozilla/4.78 [ja/[Vine,RedHat]] (X11; U; Linux 2.4.7-10
i686) ==> NSCP6
        Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0) ==>
MSIE6
        Mozilla/4.78 [en] (WinNT; U) ==> NSCP_WIN32
        */

        char[] str = lstr.toCharArray();

```

Code Example 11-1 Implementation of the ClientDetectionInterface

```

String tokens ;

/* Skip leading space */
int idx = 0;
int st;
try {
    /* Skip the preceding white spaces */
    while (Character.isWhitespace(str[idx])) {
        idx++;
    }
    st = idx;
    /* Get the first token */
    while (!Character.isWhitespace(str[idx])) {
        idx++;
    }
    String agent = new String (str,st,idx);

    /* Look for compatibilty */
    while ((str[idx]!='(')) {
        idx++;
    }

    st = idx+1;
    while ((str[idx]!=')')) {
        idx++;
    }
    tokens = new String(str,st,idx-st);
    if (tokens.indexOf ("MSIE")!= -1) {
        if (tokens.indexOf ("6.0")!= -1)
            return "MSIE6";
        else
            result= "MSIE";
    } else if (agent.indexOf("Mozilla/4")!= -1) {
        if ( (tokens.indexOf ("X11;") != -1) &&
(tokens.indexOf ("U;")!= -1))
            result= "NSCP_UNIX";
        if (tokens.indexOf ("Windows") != -1 ||
tokens.indexOf("WinNT") != -1)
            result= "NSCP_WIN32";
        } else if ( (agent.indexOf ("Mozilla/5") != -1) )
            result="NSCP6";
    } catch (IndexOutOfBoundsException ex) {
        // Unable to parse the User-Agent or unknown Agent.
        Fall back to
        // Generic HTML
        if (debug.messageEnabled() ) {
            debug.message("DefaultImpl AgentException in parsing
= "+ex);
        }
    }

    if (debug.messageEnabled() ) {
        debug.message("DefaultImpl result = "+result);
    }
}

```

Code Example 11-1 Implementation of the `ClientDetectionInterface`

```
synchronized (agentTable) {  
    agentTable.put (lstr, result);  
}  
return result;  
}  
}
```


Identity Server Utilities

The Sun ONE Identity Server provides scripts to backup and restore data as well as application programming interfaces (API) that are used by the server itself or by external applications. This chapter explains the scripts and the API. It contains the following sections:

- Backup And Restore
- Utility API

Backup And Restore

The Backup and Restore function of Identity Server allows businesses to keep their data safe by backing it to up and recovering it following an unexpected loss. Backed-up information includes all configuration, customization and identity data that has been modified or added since the initial installation of the Identity Server. Log and debug files are also backed up. (Identity Server will not backup anything that remains unchanged from the installation state.)

The Restore function re-configures a freshly installed Identity Server to a former state, reflected by the data that was last backed up. It restores all the configuration, customization and identity data that was last backed up as well as the log and debug files. Both the Backup and Restore functions are initiated through the use of scripts provided with Identity Server.

NOTE The backup and restore functions are performed on the Identity Server data stored in the Directory Server.

Backup Script

Following is the script used for backing up data. The utility is named `am2bak` and can be found in the `<identity_server_root>/SUNWam/bin` directory. `am2bak` takes command-line parameters and creates a `backup.inf` file containing information pertinent to the backup. A tar file is then created consisting of all the data.

Usage

The script is:

- `./am2bak [-v | --verbose] [-k | --backup <backup-name>] [-l | --location <location>] [[-c | --config] | [-b | --debug] | [-g | --log] | [-t | --cert] | [-d | --ds] | [-a | --all]]*`
- `./am2bak -h | --help`
- `./am2bak -n | --version`

The options are defined as:

- `-v | --verbose` — runs the script in verbose mode.
- `-k | --backup <backup-name>` — defines the name of the backup file. The default filename is `ambak`.
- `-l | --location <location>` — defines the location of the backup file. The default is `<identity_server_root>/backup`.
- `-c | --config` — confines the backup to only configuration files. This also includes the service configuration data (updated service schema files and the service configurations for various organizations).
- `-b | --debug` — confines the backup to only debug files.
- `-g | --log` — confines the backup to only log files.
- `-t | --cert` — confines the backup to only the certification database.
- `-d | --ds` — confines the backup to the Directory Server.
- `-a | --all` — defines a complete backup of the Identity Server. This is the default option.
- `--help` — accesses the script's help feature.
- `--version` — prints the version of the backup script being used to the screen.

Backup Procedure

1. Login as root.

The user running this script must have root access.

2. Run the script ensuring that the correct path is used, if necessary.

The script will backup the following Solaris™ Operating Environment files:

- **Configuration and Customization Files:**
 - `<identity_server_root>/SUNWam/config/`
 - `<identity_server_root>/SUNWam/locale/`
 - `<identity_server_root>/SUNWam/servers/httpacl`
 - `<identity_server_root>/SUNWam/lib/*.properties` (**Java property files**)
 - `<identity_server_root>/SUNWam/bin/amserver.<instance-name>`
 - `<identity_server_root>/SUNWam/servers/https-<all_instances>`
 - `<identity_server_root>/SUNWam/servers/web-apps-<all_instances>`
 - `<identity_server_root>/SUNWam/web-apps/services/WEB-INF/config`
 - `<identity_server_root>/SUNWam/web-apps/services/config`
 - `<identity_server_root>/SUNWam/web-apps/applications/WEB-INF/classes`
 - `<identity_server_root>/SUNWam/web-apps/applications/console`
 - `/etc/rc3.d/K55amserver.<all_instances>`
 - `/etc/rc3.d/S55amserver.<all_instances>`
 - `<directory_server_root>/slapd-<host>/config/schema/`
 - `<directory_server_root>/slapd-<host>/config/slapd-collations.conf`
 - `<directory_server_root>/slapd-<host>/config/dse.ldif`
- **Log And Debug Files:**
 - `var/opt/SUNWam/logs` (**Identity Server log files**)
 - `var/opt/SUNWam/install` (**Identity Server installation log files**)

- `var/opt/SUNWam/debug` (Identity Server debug files)
- **Certificates:**
 - `<identity_server_root>/SUNWam/servers/alias`
 - `<directory_server_root>/alias`

The script will also backup the following Microsoft® Windows 2000 operating system files:

- **Configuration and Customization Files:**
 - `<identity_server_root>/web-apps/services/WEB-INF/config/*`
 - `<identity_server_root>/locale/*`
 - `<identity_server_root>/web-apps/applications/WEB-INF/classes/*.properties` (java property files)
 - `<identity_server_root>/servers/https-<host>/config/jvm12.conf`
 - `<identity_server_root>/servers/https-<host>/config/magnus.conf`
 - `<identity_server_root>/servers/https-<host>/config/obj.conf`
 - `<directory_server_root>/slapd-<host>/config/schema/*.ldif`
 - `<directory_server_root>/slapd-<host>/config/slapd-collations.conf`
 - `<directory_server_root>/slapd-<host>/config/dse.ldif`
- **Log And Debug Files:**
 - `var/opt/logs` (Identity Server log files)
 - `var/opt/debug` (Identity Server debug files)
- **Certificates:**
 - `<identity_server_root>/servers/alias`
 - `<identity_server_root>/alias`

Restore Script

Following is the script used for restoring backed-up data to a freshly reinstalled Identity Server. The utility is named `bak2am` and can be found in the `<identity_server_root>/SUNWam/bin` directory. `bak2am` takes the backup file name as a command-line parameters, reads the `backup.inf` file, untars the data file and performs the restoration accordingly.

NOTE The Restore script will stop the Identity Server if it is running when the script is activated.

Usage

The script is:

- `./bak2am [-v | --verbose] -z | --gzip <tar.gz-file> or -t | --tar <tar-file>`
- `./bak2am -h | --help`
- `./bak2am -n | --version`

The options are defined as:

- `-v | --verbose` — runs the script in verbose mode.
- `-z | --gzip <tar.gz file-name>` — defines the location of a gzipped data backup tar file. A full path name must be used if the file is not located in the default `<identity_server_root>/backup` directory.
- `-t | --tar <tar-file>` — defines the location of a data backup tar file. A full path name must be used if the file is not located in the default `<identity_server_root>/backup` directory.
- `--help` — accesses the script's help feature.
- `--version` — prints the version of the backup script being used to the screen.

Restore Procedure

1. Login as root.

The user running this script must have root access.

2. Untar the input tar file.

This was generated when the backup script was run.

Utility API

The utilities package is called `com.iplanet.am.util`. It contains utility programs that can be used by external applications accessing Identity Server. The API include:

- `StatsListener`
- `AdminUtils`
- `Debug`
- `Locale`
- `Stats`
- `SystemProperties`
- `ThreadPool`

NOTE The Identity Server Javadocs can be accessed from any browser by copying the complete `<identity_server_root>/SUNWam/docs/` directory into the `<identity_server_root>/SUNWam/public_html` directory and pointing the browser to `http://<server_name.domain_name>:<port>/docs/index.html`.

API Summary

Following is a summary of the utility API and their functions.

AMPasswordUtil

The `AMPasswordUtil` interface can be used to encrypt and decrypt passwords.

AdminUtils

This class contains the methods used to retrieve `TopLevelAdmin` information. The information comes from the server configuration file (`serverconfig.xml`).

Debug

Debug allows an interface to file debug and exception information in a uniform format. It supports different levels of filing debug information (in the ascending order): OFF, ERROR, WARNING, MESSAGE and ON. A given debug level is enabled if it is set to at least that level. For example, if the debug state is ERROR, only errors will be filed. If the debug state is WARNING, only errors and warnings will be filed. If the debug state is MESSAGE, everything will be filed. MESSAGE and ON are the same level except MESSAGE writes to a file, whereas ON writes to `System.out`.

NOTE Debugging is an intensive operation and may hurt performance when abused. Java evaluates the arguments to `message()` and `warning()` even when debugging is turned off. It is recommended that the debug state be checked before invoking any `message()` or `warning()` methods to avoid unnecessary argument evaluation and to maximize application performance.

Locale

This class is a utility that provides the functionality for applications and services to internationalize their messages.

SystemProperties

This class provides functionality that allows single-point-of-access to all related system properties. First, the class tries to find `AMConfig.class`, and then a file, `AMConfig.properties`, in the CLASSPATH accessible to this code. The class takes precedence over the flat file. If multiple servers are running, each may have their own configuration file. The naming convention for such scenarios is `AMConfig_serverName`.

ThreadPool

ThreadPool is a generic thread pool that manages and recycles threads instead of creating them when a task needs to be run on a different thread. Thread pooling saves the virtual machine the work of creating brand new threads for every short-lived task. In addition, it minimizes the overhead associated with getting a thread started and cleaning it up after it dies. By creating a pool of threads, a single thread from the pool can be reused any number of times for different tasks. This reduces response time because a thread is already constructed and started and is simply waiting for its next task.

Another characteristic of this thread pool is that it is fixed in size at the time of construction. All the threads are started, and then each goes into a wait state until a task is assigned to it. If all the threads in the pool are currently assigned a task, the pool is empty and new requests (tasks) will have to wait before being scheduled to run. This is a way to put an upper bound on the amount of resources any pool can use up. In the future, this class may be enhanced to provide support growing the size of the pool at runtime to facilitate dynamic tuning.

AMConfig.properties File

`AMConfig.properties` is the resource configuration file for the Sun™ One Identity Server. It provides instructions for the Identity Server set up. This chapter explains the elements of the `AMConfig.properties`. It contains the following sections:

- Overview
- Deployment Directives
- Configuration Directives
- Read-Only Directives

Overview

Identity Server is configured by placing directives in plain text configuration files. The main configuration file is `AMConfig.properties`. `AMConfig.properties` is located in `<identity_server_root>/SUNWam/lib`. Changes to this configuration files are only recognized when Identity Server is started or restarted.

Identity Server configuration files contain one directive per line and each directive has a corresponding value. Directives and their values are case-sensitive. Indentation of the directives is consistent throughout the file. Lines which begin with the characters “/” are considered comments, and are ignored by the application. Comments are completed with a last line that contains the closing characters “*/”.

NOTE The Identity Server must be restarted for any change in `AMConfig.properties` to take effect.

Deployment Directives

There are a number of deployment-specific attributes configured in `AMConfig.properties`. These are defined in this section.

Identity Server Directives

The following section describe the directives that define the Identity Server.

Installation

The following directives are Identity Server-specific. They are defined during installation.

- `com.ipplanet.am.server.host=sunbox1.red.ipplanet.com`
The value of this directive is the DNS domain name of the machine on which the Identity Server is located.
- `com.ipplanet.am.server.port=58080`
The value of this directive is the port number of the Identity Server.
- `com.ipplanet.am.console.protocol=http`
The value of this directive is the protocol used to communicate with the Identity Server.
- `com.ipplanet.am.jdk.path=/export/SUNWam/java`
The value of this directive is the path to the JDK used by the Identity Server.
- `com.sun.identity.authentication.super.user=uid=amAdmin,ou=People,dc=madisonparc,dc=com`
This directive identifies the super user for the Sun ONE Identity Server deployment. This user is `amadmin` by default but may be any user in the Directory Server. The value of the directive is the full DN of the user. This user must always login using LDAP authentication as they will always be authenticated against the Directory Server.

Console

The following directives are specific to the Identity Server console.

- `com.ipplanet.am.console.host=sunbox1.red.ipplanet.com`

The value of this directive is the DNS domain name of the machine on which the Identity Server console is located.

- `com.ipplanet.am.console.port=58080`

The value of this directive is the port number of the Identity Server console.

Cookies

The following directives are specific to Identity Server cookies.

- `com.ipplanet.am.cookie.name=iPlanetDirectoryPro`

The value of this key is the name of the cookie.

- `com.ipplanet.am.pcookie.name=DProPCookie`

The value of this key is the name of the persistent cookie if that function is enabled.

Miscellaneous Directives

The following directives define miscellaneous values necessary for the Identity Server.

- `com.ipplanet.am.daemons=unix`
- `com.ipplanet.am.locale=en_US`
- `com.ipplanet.am.logstatus=ACTIVE`
- `com.ipplanet.am.version=6.0`

Directory Server

The following directives are Directory Server-specific.

Installation

This information is defined during installation for the Directory Server to which the Identity Server points.

- `com.ipplanet.am.directory.host=sunbox1.red.ipplanet.com`

The value of this directive is the DNS domain name of the machine on which the Directory Server is located.

- `com.ipplanet.am.directory.port=389`

The value of this directive is the port number of the Directory Server.

- `com.ipplanet.am.server.protocol=http`

The value of this directive is the protocol used to communicate with the Directory Server.

Directory Server Tree

The values of these directives is the top-level organization defined during the installation process.

- `com.ipplanet.am.defaultOrg=dc=madisonparc,dc=com`
- `com.ipplanet.am.rootsuffix=dc=madisonparc,dc=com`
- `com.ipplanet.am.domaincomponent=dc=madisonparc,dc=com`

Configuration Directives

There are a number of services configured in `AMConfig.properties` that can not be configured using the Identity Server console. These back-end services are defined in this section.

Debug Service

The following directives are used to configure the Debug Service, which logs developer information in the case of application errors. (The Logging Service writes logs to be monitored by the application administrator.)

- `com.ipplanet.services.debug.level=error`

The possible values for this directive are: `off` | `error` | `warning` | `message`. They indicate the amount of information that would be recorded in the debug files.

- `com.ipplanet.services.debug.directory=/var/opt/SUNWam/debug`

The value of this directive specifies the output directory for the debug files. This directory should be writable by the server process.

NOTE In defining values for the Debug Service, remember that trailing spaces are significant. On a Windows® system, use forward slashes “/” to separate directories. Spaces in the file name are also allowed on a Windows system.

Stats Service

The following keys are used to configure the Stats Service for recording service statistics. Currently, this service is used by the Identity Server SDK and the Session Service. Code Example 12-1 is a portion of the stats file which also illustrates the information that is recorded. The file is named `amSDKStats` by default.

Code Example 12-1 Portion of amSDKStats File

```
11/26/2002 01:46:18:592 PM PST: Thread[Thread-10,5,main]
SDK Cache Statistics
-----
Interval: 214
Hits during interval: 38
Hit ratio for this interval: 0.17757009345794392
Total number of requests: 214
Total number of Hits: 38
Overall Hit ratio: 0.17757009345794392
Total Cache Size: 72
```

- `com.ipplanet.am.stats.interval=3600`

The statistics interval should be at least 5 seconds to avoid CPU saturation. Identity Server will assume that any value less than that is 5 seconds.

- `com.ipplanet.services.stats.state=off`

Possible values for this directive are: `off` | `file` | `console`. `file` will write to a file named `amSDKStats` under the specified directory and `console` will write into Web Server log files.

- `com.ipplanet.services.stats.directory=/var/opt/SUNWam/debug`

This directive specifies the output directory for the statistics files, the debug directory by default.

NOTE

In defining values for the Stats Service, remember that trailing spaces are significant. On a Windows® system, use forward slashes “/” to separate directories. Spaces in the file name are also allowed on a Windows system.

SAML

These directives identify the SAML XML signature keystore file, the keystore password file and the key password file, respectively.

- `com.sun.identity.saml.xmlsig.keystore=/export/SUNWam/lib/keystore.jks`
- `com.sun.identity.saml.xmlsig.storepass=/export/SUNWam/config/.storepass`
- `com.sun.identity.saml.xmlsig.keypass=/export/SUNWam/config/.keypass`
- `com.sun.identity.saml.xmlsig.certalias=test`

The value of this key is the name of the certificate alias.

Miscellaneous Services

The following directives define the URIs for the Profile, Naming and Notification services.

- `com.iplanet.am.profile.host=sunbox1.red.iplanet.com`
- `com.iplanet.am.profile.port=58080`
- `com.iplanet.am.naming.url=http://sunbox1.red.iplanet.com:58080/amserver/naming-service`
- `com.iplanet.am.notification.url=http://sunbox1.red.iplanet.com:58080/amserver/notification-service`

SDK Caching

Each SDK cache entry stores a set of attributes and values for a user. Because the size of each object is dependent upon the number of attributes it has, modifying this property will affect the performance of Identity Server.

- `com.iplanet.am.sdk.cache.maxSize=10000`

This directive is used to configure SDK caching; it specifies the size of the cache when caching is enabled. The value of this directive refers to the number of objects cached and should be an integer greater than 0; if not, the default 10000 will be used.

- `com.iplanet.am.session.maxSessions=5000`

This directive is used to specify maximum number of concurrent sessions. Logging in would give a Maximum Sessions error if the maximum concurrent sessions exceeds the defined number.

Simple Mail Transfer Protocol (SMTP)

The following directives can be set to any valid SMTP server and port.

- `com.ipplanet.am.smtphost=localhost`
- `com.sun.identity.sm.smtpport=25`

Identity Object Processing

This directive has a value equal to the implementation class of a module used for processing user creates, deletes, and modifies.

- `com.ipplanet.am.sdk.userEntryProcessingImpl=`

SSL

This directive enables Secure Socket Layers (SSL).

- `com.ipplanet.am.directory.ssl.enabled=false`

Certificate Database

These directives are used by the command line utilities, the SDK and the LDAP and Certificate-based authentication modules when initiating SSL connections to the Directory Server. It is also used when opening https connections from within the servlet container in the Sun ONE Web Server.

- `com.ipplanet.am.admin.cli.certdb.dir=/export/SUNWam/servers/alias`
The value of this key is the name of the path to the certificate database.
- `com.ipplanet.am.admin.cli.certdb.prefix=https-sunbox1.red.ipplanet.com-sunbox1-`
The value of this key is the certificate database prefix.
- `com.ipplanet.am.admin.cli.certdb.passfile=/export/SUNWam/config/.wtpass`
The value of this key is the name of the file that contains the password for the certificate database.

OCSP Configuration

These directives define configurations for the OCSP (Online Certificate Server Protocol). If set, the CA cert must be presented in the Web Server's cert database. If the OCSP URL is set, the OCSP responder nickname must be set also or they both will be ignored. If not set, the OCSP responder URL presented in the user's certificate will be used for OCSP validation. If the OCSP responder URL is not presented in user's cert, then no OCSP validation will be performed.

- `com.sun.identity.authentication.ocsp.responder.url=`
This directive defines the OCSP responder URL for this instance of Identity Server (for example, `http://ocsp.example.com/ocsp`).
- `com.sun.identity.authentication.ocsp.responder.nickname=`
This directive defines the OCSP responder nickname, the Certificate Authority cert nickname for the responder defined above (for example, Certificate Manager - example).

Replication

These two properties are not required to support replication but they may be helpful in limiting errors due to latency. Enabling them may have a negative impact on performance but, if replication has significant latency, the retries may be enough to prevent Entry Not Found errors. For example, let's assume an Identity Server console is pointing to a read-only consumer configured to refer writes to a master. If a new organization is created, all write requests are referred to the master and then replicated back to the consumer. If Identity Server reads the organization back before it has been replicated to the consumer, it will get an Entry Not Found error.

NOTE It is not recommended to run the Identity Server console against a read-only consumer. The exception to this rule is when operating against user entries whose creations and modifications do not have the same latency problems as the SDK has special behavior to prevent such problems for these entries.

- `com.ipланet.am.replica.num.retries=0`
This key specifies the number of times to retry. When an Entry Not Found error is returned to the SDK, it will retry *n* times where *n* is the value of this directive.
- `com.ipланet.am.replica.delay.between.retries=1000`
This key specifies the delay time (in milliseconds) between the retries defined above.

Event Connection And LDAP Connection

These two sets of SDK properties are implemented when load balancers are used between the Identity SDK and the Directory Server. When the SDK performs an operation which fails, it will retry the operation as long as the exception is one defined in the `ldap.error.codes` property. These properties are necessary for failover configuration when the failover is done via a load balancer and not through the Identity SDK. They are also important since not all load balancers return the same error codes.

Event Connection

- `com.ipplanet.am.event.connection.num.retries=3`
This directive specifies the number of time to retry an event connection.
- `com.ipplanet.am.event.connection.delay.between.retries=3000`
This directive specifies the delay time (in milliseconds) between retries.
- `com.ipplanet.am.event.connection.ldap.error.codes.retries=80,81,91`
This directive specifies the LDAPException errors for which the retries will occur. The value is any valid LDAP error code.

LDAP Connection

The following keys are used to configure LDAP connection for add, delete modify, read and search.

- `com.ipplanet.am.ldap.connection.num.retries=3`
This directive specifies the number of times to retry a LDAP connection.
- `com.ipplanet.am.ldap.connection.delay.between.retries=1000`
The directive specifies the delay time (in milliseconds) between retries.
- `com.ipplanet.am.ldap.connection.ldap.error.codes.retries=80,81,91`
This directive specifies the LDAPException error codes for which the retries will occur.

Read-Only Directives

The following properties are read-only and should not be modified. Any changes to these directives may render the Identity Server unusable.

Base Directory

The following directives identify the base directory as defined during the installation process.

- `com.ipplanet.am.installdir=/export/SUNWam`
- `com.ipplanet.am.install.basedir=/export/SUNWam/web-apps/services/WEB-INF`

Shared Secret

The following directive is the shared secret for the Authentication module.

- `com.ipplanet.am.service.secret=AQIC5wM2LY4SfczLlj6134qMTx0nkE5XiFMg`

Deployment Descriptors

The following directives are used to identify the deployment descriptors (URIs) for Identity Server services and agents.

- `com.ipplanet.am.services.deploymentDescriptor=/amserver`
- `com.ipplanet.am.console.deploymentDescriptor=/amconsole`
- `com.ipplanet.am.policy.agents.url.deploymentDescriptor=AGENT_DEPLOY_URI`

Session Properties

The following directives are configurations for the Session Service.

- `com.ipplanet.am.session.failover.enabled=false`

This directive is used to enable or disable the session failover feature.

NOTE Session failover is an unsupported option in Identity Server 6.0.

- `com.ipplanet.am.naming.failover.url=`

This directive can be used by any remote SDK application that wants failover in, for example, session validation or getting the service URLs.

- `com.ipplanet.am.session.httpSession.enabled=true`
This directive is used to enable or disable the use of a `httpSession`.
- `com.ipplanet.am.session.invalidsessionmaxtime=3`
This directive is used to keep the invalid session in the session table for this period. The value is in minutes (for example, 3 minutes).

NOTE This value should always be greater than the time-out value in your authentication module properties file.

- `com.ipplanet.am.session.client.polling.enable=false`
- `com.ipplanet.am.session.client.polling.period=180`
The two above directives are used to enable session client side notification. The default polling period is 180 seconds.
- The following is used in the Unix authentication module.
`unixHelper.port=58946`
- The following key is used to check whether the Sun ONE Identity Server is running on the Application Server or not. This key is modified only by the installer so do not change it.
`com.ipplanet.am.iASConfig=false`

Cross Domain Single Sign-On Support

The following directives are used for Cross Domain SSO support.

- `com.ipplanet.services.cdssso.CDCURL=http://sunbox1.red.ipplanet.com:58080/amserver/cdcservlet`
This directive points to the `cdcservlet` running with the instance of Identity Server.
- `com.ipplanet.services.cdssso.cookiesdomain=`
This directive specifies a comma separated list of domains for which the `cdssso` servlet will set a `SSOToken` (for example: `.sun.com,example.com`).
- `com.ipplanet.services.cdc.authLoginUrl=http://sunbox1.red.ipplanet.com:58080/amserver/UI/Login`

The value of this directive is the URL with which a user can login.

SecureRandom Properties

This directive specifies the factory class name for SecureRandomFactory.

- `com.ipplanet.security.SecureRandomFactoryImpl=com.ipplanet.am.util.JSSSecureRandomFactoryImpl`

The available implementation classes are:

- a. `com.ipplanet.am.util.JSSSecureRandomFactoryImpl` (uses JSS)
- b. `com.ipplanet.am.util.SecureRandomFactoryImpl` (pure Java)

SocketFactory properties

This directive specifies the factory class name for LDAPSocketFactory.

- `com.ipplanet.security.SSLSocketFactoryImpl=com.ipplanet.services.ldap.JSSSocketFactory`

Available classes are:

- a. `com.ipplanet.services.ldap.JSSSocketFactory` (uses JSS)
- b. `netscape.ldap.factory.JSSESocketFactory` (pure Java)

Encryption

This directive specifies the encrypting class implementation.

- `com.ipplanet.security.encryptor=com.ipplanet.services.util.JSSEncryption`

Available classes are:

- a. `com.ipplanet.services.util.JCEEncryption`
- b. `com.ipplanet.services.util.JSSEncryption`.

Remote Installation

This directive defines whether the console is installed on a remote machine or a local machine. It is used by the Authentication Service and the Identity Server console.

- `com.ipplanet.am.console.remote=false`

IP Address Checking

This directive specifies whether the IP address of the client will be checked in SSO Token creations and validations.

- `com.ipplanet.am.clientIPCheckEnabled=false`

Remote Policy API Directives

The properties listed below are defined for the Remote Policy API.

Username

This directive specifies the username for the Application authentication module.

- `com.sun.identity.agents.app.username=UrlAccessAgent`

Log File Name

This directive specifies the name of the log file to use for logging remote policy messages. The directory where this file is located is determined by Logging Service settings.

- `com.sun.identity.agents.server.log.file.name=amRemotePolicyLog`

Resource Result Cache Size

This directive specifies the size of the cache created on the server where the policy agent resides.

- `com.sun.identity.agents.cache.size=1000`

Polling Interval

The polling interval is the duration of time for refreshing the cache.

- `com.sun.identity.agents.polling.interval=3`

Resource Name Comparison

This directive indicates whether to use wildcard for resource name comparison.

- `com.sun.identity.agents.use.wildcard=true`

Returned Policy Attributes

This directive defines the policy attributes to be returned by policy evaluator. The specification is of the format `a[,...]` where `a` is the attribute in the data store that will be fetched.

- `com.sun.identity.agents.header.attributes=cn,ou,o,mail,employeenumber,c`

Resource Comparator Class Name

- `com.sun.identity.agents.resource.comparator.class=com.sun.identity.policy.plugins.PrefixResourceName`

Resource Name's Wildcard

- `com.sun.identity.agents.resource.wildcard=*`

Resource Name's Delimiter

- `com.sun.identity.agents.resource.delimiter=/`

Case Sensitivity

This is to indicator whether case sensitivity is turned on or off during policy evaluation. The default value is false or off.

- `com.sun.identity.agents.resource.caseSensitive=false`

Policy Action True Value

This value is ignored if the application does not access the method `PolicyEvaluator.isAllowed`.

- `com.sun.identity.agents.true.value=allow`
- `com.sun.identity.federation.fedCookieName=fedCookie`

Federation Signing

This directive defines whether federation requests and responses will be signed before sending and whether federation requests and responses that are received will be verified for signature validity. The default is false; requests and responses that are sent and received will not be verified for signature.

- `com.sun.identity.federation.services.signingOn=false`

FQDN Map

The FQDN Map is a simple map that enables Identity Server Authentication service to take corrective action in the case where the users may have typed in an incorrect URL such as by specifying partial hostname or using an IP address to access protected resources.

Valid Values

Valid values must comply with the syntax of this property which represent invalid FQDN values mapped to their corresponding valid counterparts. The format for specifying this property is as follows:

```
com.sun.identity.server.fqdnMap[<invalid-name>]=<valid-name>
```

where <invalid-name> is a possible invalid FQDN host name that may be used by the user, and the <valid-name> is the FQDN host name the filter will redirect the user to.

CAUTION Ensure that there are no invalid or overlapping values for the same invalid FQDN name. Failing to do so may lead to the application becoming inaccessible.

This directive can be used for creating a mapping for more than one hostname. This may be the case when the applications hosted on a server are accessible by more than one hostname. It may also be used to configure Identity Server to NOT take corrective action for certain hostname URLs. For example, if no corrective action (such as a redirect) is desired for users who access application resources using a raw IP address, the map entry would look like:

```
com.sun.identity.server.fqdnMap[<IP>]=<IP>
```

Any number of such properties may be specified as long as they are valid and conform to the above stated requirements.

Examples of FQDN mapping might be:

- `com.sun.identity.server.fqdnMap[issserver]=issserver.mydomain.com`
- `com.sun.identity.server.fqdnMap[issserver.mydomain]=issserver.mydomain.com`
- `com.sun.identity.server.fqdnMap[<IP address>]=issserver.mydomain.com`
- `com.sun.identity.server.fqdnMap[<invalid-name>]=<valid-name>`

Directory Server Concepts

Sun™ One Identity Server uses Sun ONE Directory Server to store its data. Certain features of the LDAP-based Directory Server are also used by Identity Server to help manage the data. This chapter contains information on these Directory Server features and how they are used. It contains the following sections:

- Overview
- Roles
- Access Control Instructions (ACIs)
- Class Of Service

Overview

Because Identity Server needs an underlying directory server to function, it has been built to work with Sun ONE Directory Server. They are complementary in architecture and design data. Use of the directory, though, may not be exclusive to Identity Server and therefore, needs to be treated as a completely separate deployment. For more information on Directory Server deployment, see the Sun ONE Directory Server documentation.

This appendix explains three Directory Server functions that are used by the Identity Server. *Roles* are an identity grouping mechanism, *access control instructions* define a type of permission and *class of service* are an attribute grouping mechanism. They are more fully defined below. For more specific information on these features, see the Sun ONE Directory Server documentation.

Roles

Roles are a Directory Server entry mechanism similar to the concept of a *group*. A group has members; a role has members. A role's members are LDAP entries that are said to *possess* the role. The criteria of the role itself is defined as an LDAP entry with attributes, identified by the Distinguished Name (DN) attribute of the entry. Directory Server has a number of different types of roles but Identity Server can only manage one of them: the managed role.

NOTE The other Directory Server role types can still be used in a directory deployment; they just can not be managed by Identity Server.

Users can possess one or more roles. For example, a contractor role which has attributes from the Session Service and the URL Policy Agent Service might be created. Thus, when new contractors start, the administrator can assign them this role rather than setting separate attributes in the contractor entry. If the contractor were then to become a full-time employee, the administrator would just re-assign the user a different role.

Managed Roles

With a managed role, membership is defined in each member entry and not in the role definition entry. An attribute which designates membership is placed in each LDAP entry that possesses the role. This is in sharp contrast to a traditional static group which centrally lists the members in the group object entry itself.

NOTE By inverting the membership mechanism, the role will scale better than a static group. In addition, the referential integrity of the role is simplified, and the roles of an entry can be easily determined.

An administrator assigns the role to a member entry by adding the `nsRoleDN` attribute to it. The value of `nsRoleDN` is the DN of the role definition entry. The following apply to managed roles:

- Multiple managed roles can be created for each organization or sub-organization.
- A managed role can be enabled with any number of services.

- Any user that possesses a role with a service will inherit the service attributes from that role.

NOTE All Identity Server roles can only be configured directly under organization or sub-organization entries.

Definition Entry

A role's definition entry is a LDAP entry in which the role's characteristic attributes are defined. These attributes are passed onto the member entry. Below is a sample LDAP entry that represents the definition entry of a manager role.

Code Example 12-2 LDAP Definition Entry

```
dn: cn=managerrole,dc=siroe,dc=com
   objectclass: top
   objectclass: LDAPsubentry
   objectclass: nsRoleDefinition
   objectclass: nsSimpleRoleDefinition
   objectclass: nsManagedRoleDefinition
   cn: managerrole
   description: manager role within company
```

The `nsManagedRoleDefinition` object class inherits from the `LDAPsubentry`, `nsRoleDefinition` and `nsSimpleRoleDefinition` object classes.

Member Entry

A role's member entry is a LDAP entry to which the role is applied. An LDAP entry that contains the attribute `nsRoleDN` and its value DN indicates that the entry has the characteristics defined in the value DN entry. In Code Example 12-3 below, the DN identifies Code Example 12-2 above as the role definition entry:

```
cn=managerrole,dc=siroe,dc=com.
```

Virtual Attribute

When a member entry that contains the `nsRoleDN` attribute is returned by a Directory Server search, `nsRoleDN` will be duplicated as the `nsRole` attribute in the same entry. `nsRole` will carry a value of any managed, filtered or nested roles assigned to the user (such as `ContainerDefaultTemplateRole`). Code Example 12-3 on page 290 includes this virtual attribute when returned by Directory Server only.

Code Example 12-3 LDAP Member Entry

```

dn: uid=managerperson,ou=people,dc=siroe,dc=com
   objectclass: top
   objectclass: person
   objectclass: inetorgperson
   uid: managerperson
   gn: manager
   sn: person
   nsRoleDN: cn=managerrole,ou=people,dc=siroe,dc=com
   nsRole: cn=managerrole,ou=people,dc=siroe,dc=com
   nsRole:
cn=containerdefaulttemplaterole,ou=people,dc=siroe,dc=com
   description: manager person within company

```

How Identity Server Uses Roles

Identity Server uses roles to apply access control instructions. When first installed, the Identity Server configures access control instructions (ACIs) that define administrator permissions. These ACIs are then designated in roles (such as Organization Admin Role and Organization Help Desk Admin Role) which, when assigned to a user, define the user's access permissions. For a list of roles created for each Identity Server object configured, see "Access Control Instructions (ACIs)," on page 292.

Role Creation

When a role is created, it contains the auxiliary LDAP object class `iplanet-am-managed-role`. This object class, in turn, contains the following allowed attributes:

- `iplanet-am-role-managed-container-dn` contains the DN of the identity-related object that the role was created to manage.
- `iplanet-am-role-type` contains a value used by the Identity Server console for display purposes. After authentication, the console gets the user's roles and checks this attribute for the correct page to display based on which of the following three values it has:
 - 1 for top-level administrator only.
 - 2 for all other administrators.
 - 3 for user.

If the user has no administrator roles, the User profile page will display. If the user has an administrator role, the console will start the user at the top-most administrator page based on which value is present.

NOTE When Identity Server attempts to process two templates that are set to the same priority level, Directory Server arbitrarily picks one of the templates to return. For more information, see the Sun ONE Directory Server documentation.

Role Location

All roles in an organization are viewed from the organization's top-level. For example, if an administrator wants to add a user to the administrator role for a people container, the administrator would go to the organization above the people container, look for the role based on the people container's name, and add the user to the role.

NOTE Alternately, an administrator might go to the user profile and add the role to the user.

Displaying The Correct Login Start Page

The attribute `iplanet-am-user-admin-start-dn` can be defined for a role or a user; it would override the `iplanet-am-role-type` attribute by defining an alternate display page URL. Upon a user's successful authentication:

1. Identity Server checks the `iplanet-am-user-admin-start-dn` for the user.

This attribute is contained in the User service. If it is set, the user is started at this point. If not, Identity Server goes to step 2.

NOTE The value of `iplanet-am-user-admin-start-dn` can override the administrator's start page. For example, if a group administrator has read access to the top-level organization, the default starting page of the top-level organization, taken from `iplanet-am-role-type`, can be overridden by defining `iplanet-am-user-admin-start-dn` to display the group's start page.

2. Identity Server checks the user for the value of `iplanet-am-role-type`.

If the attribute defines an administrator-type role, the value of `iplanet-am-role-managed-container-dn` is retrieved and the highest point in the directory tree is displayed as a starting point. For more information on the `iplanet-am-role-type` attribute, see “Role Creation,” on page 290.

NOTE If the attribute has no value, a search from Identity Server root is performed for all container-type objects; the highest object in the directory tree that corresponds to the `iplanet-am-role-type` value is where the user starts. Although rare, this step is memory-intensive in very large directory trees with many container entries.

Access Control Instructions (ACIs)

Access control in Identity Server is implemented using Directory Server roles. Users inherit access permissions based on their role membership and parent organization. Identity Server installs pre-configured administrator roles that define access permissions for administrators; these roles are dynamically created when a group, organization, container or people container object is configured. They are:

- `Organization Admin`
- `Organization Help Desk Admin`
- `Group Admin`
- `Container Admin`
- `Container Help Desk Admin`
- `People Container Admin`.

NOTE This section refers to ACIs as they are applied to administrative roles only. There are other ACIs which are created and used in Identity Server but do not apply to this topic or to roles.

These default roles, when possessed by a user entry, apply a set of default access control instructions (ACIs) that define read and write access to the entries in the object for which the roles were created. For example, when an organization is created, the Identity Server SDK creates an `Organization Admin` role and an `Organization Help Desk Admin` role. The permissions are read and write access to all organization entries and read access to all organization entries, respectively.

NOTE The Identity Server SDK gets the ACIs from the attribute `iplanet-am-admin-console-dynamic-aci-list` (defined in the `amAdminConsole.xml` service file) and sets them in the roles after they have been created.

Defining ACIs

ACIs are defined in the Identity Server console administration XML service file, `amAdminConsole.xml`. This file contains two global attributes that define ACIs for use in Identity Server: `iplanet-am-admin-console-role-default-acis` and `iplanet-am-admin-console-dynamic-aci-list`.

`iplanet-am-admin-console-role-default-acis`

This global attribute defines which *Access Permissions* are displayed in the Create Role screen of the Identity Server console. By default, `Organization Admin`, `Organization Help Desk Admin` and `No Permissions` are displayed. If other default permissions are desired, they must be added to this attribute.

`iplanet-am-admin-console-dynamic-aci-list`

This global attribute is where all of the defined administrator-type ACIs are stored. For information on how ACIs are structured, see “Format of Predefined ACIs,” on page 293.

NOTE Because ACIs are stored in the role, changing the default permissions in `iplanet-am-admin-console-dynamic-aci-list` after a role has been created will not affect it. Only roles created after the modification has been made will be affected.

Format of Predefined ACIs

ACIs defined using Identity Server for use in administrator-type roles follow a different format than those defined using the Directory Server. The format of the predefined Identity Server ACI is `permissionName | ACI Description | DN:ACI ## DN:ACI ## DN:ACI where:`

- `permissionName`—The name of the permission which generally includes the object being controlled and the type of access. For example, `Organization Admin` is an administrator that controls access to an organization object.

- **ACI Description**—A text description of the access these ACIs allow.
- **DN:ACI**—There can be any number of DN:ACI pairs separated by the ## symbols. The SDK will get and set each pair in the entry named by DN. This format also supports tags which can be dynamically substituted when the role is created. Without these tags, the DN and ACI would be hard-coded to specific organizations in the directory tree which would make them unusable as defaults. For example, if there is a default set of ACIs for every Organization Admin, the organization name should not be hard-coded in this role. The supported tags are ROLENAME, ORGANIZATION, GROUPNAME, and PCNAME. These tags are substituted with the DN of the entry when the corresponding entry type is created. See the “Default ACIs,” on page 294 for examples of ACI formats. Additionally, more complete ACI information can be found in the Sun ONE Directory Server documentation.

NOTE If there are duplicate ACIs within the default permissions, the SDK will print a debug message.

Default ACIs

Following are the default ACIs installed by Identity Server. They are copied from a Identity Server configuration whose top-level organization is configured as o=isp.

- Top Level Admin|Access to all entries|o=isp:aci:
(target="ldap:///o=isp")(targetattr="*)(version 3.0; acl "Proxy user rights"; allow (all) roledn = "ldap:///ROLENAME";)
- Organization Admin|Read and Write access to all organization entries|o=isp:aci:(target="ldap://(\$dn),o=isp")(targetfilter=(!(|(nsroledn=cn=Top Level Admin Role,o=isp)(nsroledn=cn=Top Level Help Desk Admin Role,o=isp))))(targetattr = "*)(version 3.0; acl "Organization Admin Role access allow"; allow (all) roledn = "ldap:///cn=Organization Admin Role,[\$dn],o=isp";)##o=isp:aci:(target="ldap:///cn=Organization Admin Role,(\$dn),o=isp")(targetattr="*)(version 3.0; acl "Organization Admin Role access deny"; deny (write,add,delete,compare,proxy) roledn = "ldap:///cn=Organization Admin Role,(\$dn),o=isp";)
- Organization Help Desk Admin|Read access to all organization entries|ORGANIZATION:aci:(target="ldap:///ORGANIZATION")(targetfilter=(!(|(nsroledn=cn=Top Level Admin Role,o=isp)(nsroledn=cn=Top Level Help Desk Admin Role,o=isp)(nsroledn=cn=Organization Admin

- ```
Role,ORGANIZATION))))(targetattr = "*") (version 3.0; acl
"Organization Help Desk Admin Role access allow"; allow
(read,search) roledn = "ldap:///ROLENAME");##ORGANIZATION:aci:
(target="ldap:///ORGANIZATION")(targetfilter=(!(|(nsroledn=cn=To
p Level Admin Role,o=isp)(nsroledn=cn=Organization Admin
Role,ORGANIZATION))))(targetattr = "userPassword") (version 3.0;
acl "Organization Help Desk Admin Role access allow"; allow
(write)roledn = "ldap:///ROLENAME");
```
- Container Admin|Read and Write access to all organizational unit entries|o=isp:aci:(target="ldap://(\$dn),o=isp")(targetfilter=(!(|(nsroledn=cn=Top Level Admin Role,o=isp)(nsroledn=cn=Top Level Help Desk Admin Role,o=isp))))(targetattr = "\*") (version 3.0; acl "Container Admin Role access allow"; allow (all) roledn = "ldap:///cn=Container Admin Role,[\$dn],o=isp");o=isp:aci:(target="ldap:///cn=Container Admin Role,(\$dn),o=isp")(targetattr="\*") (version 3.0; acl "Container Admin Role access deny"; deny (write,add,delete,compare,proxy) roledn = "ldap:///cn=Container Admin Role,(\$dn),o=isp");)
  - Container Help Desk Admin|Read access to all organizational unit entries|ORGANIZATION:aci:(target="ldap:///ORGANIZATION")(targetfilter=(!(|(nsroledn=cn=Top Level Admin Role,o=isp)(nsroledn=cn=Top Level Help Desk Admin Role,o=isp)(nsroledn=cn=Container Admin Role,ORGANIZATION))))(targetattr = "\*") (version 3.0; acl "Container Help Desk Admin Role access allow"; allow (read,search) roledn = "ldap:///ROLENAME");##ORGANIZATION:aci:(target="ldap:///ORGANIZATION")(targetfilter=(!(|(nsroledn=cn=Top Level Admin Role,o=isp)(nsroledn=cn=Container Admin Role,ORGANIZATION))))(targetattr = "userPassword") (version 3.0; acl "Container Help Desk Admin Role access allow"; allow (write) roledn = "ldap:///ROLENAME");)
  - Group Admin|Read and Write access to all group members|ORGANIZATION:aci:(target="ldap:///GROUPNAME")(targetattr = "\*") (version 3.0; acl "Group and people container admin role"; allow (all) roledn = "ldap:///ROLENAME");##ORGANIZATION:aci:(target="ldap:///ORGANIZATION")(targetfilter=(!(|(!FILTER)(|(nsroledn=cn=Top Level Admin Role,o=isp)(nsroledn=cn=Top Level Help Desk Admin Role,o=isp)(nsroledn=cn=Organization Admin Role,ORGANIZATION)(nsroledn=cn=Container Admin Role,ORGANIZATION))))(targetattr != "iplanet-am-web-agent-access-allow-list ||

```
iplanet-am-web-agent-access-not-enforced-list ||
iplanet-am-domain-url-access-allow ||
iplanet-am-web-agent-access-deny-list")(version 3.0;acl "Group
admin's right to the members"; allow (read,write,search) roledn =
"ldap:///ROLENAME";)
```

- People Container Admin|Read and Write access to all users|ORGANIZATION:aci:(target="ldap:///PCNAME")(targetfilter=(!(|(nsroledn=cn=Top Level Admin Role,o=isp)(nsroledn=cn=Top Level Help Desk Admin Role,o=isp)(nsroledn=cn=Organization Admin Role,ORGANIZATION)(nsroledn=cn=Container Admin Role,ORGANIZATION))))(targetattr != "iplanet-am-web-agent-access-allow-list || iplanet-am-web-agent-access-not-enforced-list || iplanet-am-domain-url-access-allow || iplanet-am-web-agent-access-deny-list") (version 3.0; acl "People container admin role"; allow (all) roledn = "ldap:///ROLENAME";)

---

**NOTE** Identity Server generates a Top Level Admin and Top Level Help Desk Admin during installation. These roles can not be dynamically generated for any other identity-type objects but the top-level organization.

---

## Class Of Service

Both dynamic and policy attributes use *class of service* (CoS), a feature of the Directory Server that allows attributes to be created and managed in a single central location, and dynamically added to user entries as the user entry is called. Attribute values are not stored within the entry itself; they are generated by CoS as the entry is sent to the client browser. Dynamic and policy attributes using CoS consist of the following two LDAP entries:

- **CoS Definition Entry**—This entry identifies the type of CoS being used (Classic CoS). It contains all the information, except the attribute values, needed to generate an entry defined with CoS. The scope of the CoS is the entire sub-tree below the parent of the CoS definition entry.
- **Template Entry**—This entry contains a list of the attribute values that are generated when the target entry is displayed. Changes to the attribute values in the Template Entry are automatically applied to all entries within the scope of the CoS.

The CoS Definition entry and the Template entry interact to provide attribute information to their target entries; any entry within the scope of the CoS. Only those services which have dynamic or policy attributes use the Directory Server CoS feature; no other services do.

---

**NOTE** For additional information on the CoS feature, see the Sun ONE Directory Server documentation.

---

## CoS Definition Entry

CoS definition entries are stored as LDAP subentries under the organization level but can be located anywhere in the DIT. They contain the attributes specific to the type of CoS being defined. These attributes name the *virtual* CoS attribute, the template DN and, if necessary, the specifier attribute in target entries. By default, the CoS mechanism will not override the value of an existing attribute with the same name as the CoS attribute. The CoS definition entry takes the `cosSuperDefinition` object class and also inherits from the following object class that specifies the type of CoS:

### `cosClassicDefinition`

The `cosClassicDefinition` object class determines the attribute and value that will appear with an entry by taking the base DN of the template entry from the `cosTemplateDN` attribute in the definition entry and combining it with the target entry specifier as defined with the `cosSpecifier` attribute, also in the definition entry. The value of the `cosSpecifier` attribute is another LDAP attribute which is found in the target entry; the value of the attribute found in the target entry is appended to the value of `cosTemplateDN` and the combination is the DN of the template entry. Template DNs for classic CoS must therefore have the following structure `cn=specifierValue,baseDN`.

## CoS Template Entry

CoS Template entries are an instance of the `cosTemplate` object class. The CoS Template entry contains the value or values of the virtual attributes that will be generated by the CoS mechanism and displayed as an attribute of the target entry. The template entries are stored under the definition entries.

---

**NOTE** When possible, definition and template entries should be located at the same level for easier management.

---

## Conflicts and CoS

There is the possibility that more than one CoS can be assigned to a role or organization, thus creating conflict. When this happens, Identity Server will display either the attribute value based on a pre-determined template priority level or the aggregate of all attribute values defined in the `cosPriority` attribute. For example, an administrator could create and load multiple services, register them to an organization, create separate roles within the organization and assign multiple roles to a particular user. When Identity Server retrieves this user entry, it sees the CoS object classes, and adds the virtual attributes. If there are any priority conflicts, it will look at the `cosPriority` attribute for a priority level and return the information with the lowest priority number (which is the highest priority level). For more information on CoS priorities, see “cosQualifier Attribute,” on page 133 of Chapter 6, “Service Management” or the Sun ONE Directory Server documentation.

---

**NOTE** Conflict resolution is decided by the Directory Server before the entry is returned to Identity Server. Identity Server allows only the definition of the priority level and CoS type.

---

## Application Schema

If a customer is using an existing application and wants to manage its attributes using the Identity Server console, a LDAP schema is probably defined and has been loaded into the Directory Server. If Directory Server does not already have the existing application’s attributes and object classes loaded, then it needs to be updated using the Directory Server console or the `ldapmodify` command line interface. The schema update needs to be completed before loading the application’s created XML service file. Other options for adding or modifying Directory Server schema can be found in the Sun ONE Directory Server documentation or in the *Sun ONE Identity Server Installation and Configuration Guide*.

## A

- abstract objects 98
  - update 124
- access
  - SAML service 226
- access control instructions (ACIs) 292
  - default 294
  - defined 293
  - format 293
- account federation 235
- account locking
  - and authentication service 41
- ACIs 292
  - default 294
  - defined 293
  - format 293
- amAdmin.dtd 135
- AMagent.properties 84
- AMConfig.properties 271
  - configuration directives 274
  - deployment directives 272
  - overview 271
  - read-only directives 279
- amEntrySpecific.xml 107
- amLogging.xml 247
- amSAML.xml 226
- anonymous authentication module 48
- API
  - authentication
    - java 62
  - client detection
    - java 257
  - federation management
    - java 240
  - identity management SDK 102
    - caching 106
    - remote installation 107
  - logging service
    - java 249
    - sample code 251
  - policy SDK 167
    - C applications 171
    - policy evaluation API 168
    - policy management API 169
    - policy plugin API 170
  - SAML service SDK 227
  - service management SDK 156
  - SSO
    - java 85
    - non-web-based applications 85
    - overview 86
    - sample 90
    - utility
      - java 268
- architecture
  - console 28
  - logging service 246
  - policy service 161
- assertion types
  - and SAML 221
- assign
  - authentication methods 48
    - by authentication level 54
    - by module 54

- by organization 49
  - by role 50
  - by service 52
  - by user 54
- attribute inheritance 120
- attributes
  - and service files 117
- authentication
  - C applications 63
    - examples 73
  - remote-auth.dtd 63
- authentication domain 235
- authentication level authentication 54
- authentication methods
  - assign 48
  - assign by authentication level 54
  - assign by module 54
  - assign by organization 49
  - assign by role 50
  - assign by service 52
  - assign by user 54
- authentication modules
  - anonymous 48
  - certificate 48
  - core 47
  - create 55
  - credential requirements 57
    - and Auth\_Module\_Properties.dtd 57
  - custom 54
  - LDAP 48
  - membership 48
  - modify amAuth.xml 61
  - NT 48
  - proprietary 47
  - RADIUS 48
  - SafeWord 48
  - Unix 48
- authentication service 37
  - and C applications 63
    - examples 73
  - API
    - java 62
  - authentication methods
    - assign 48
    - assign by authentication level 54
    - assign by module 54
  - assign by organization 49
    - assign by role 50
    - assign by service 52
    - assign by user 54
  - authentication modules
    - anonymous 48
    - Auth\_Module\_Properties.dtd 57
    - certificate 48
    - core 47
    - create 55
    - credential requirements 57
    - custom 54
    - LDAP 48
    - membership 48
    - modify amAuth.xml 61
    - NT 48
    - proprietary 47
    - RADIUS 48
    - SafeWord 48
    - Unix 48
  - FQDN mapping 41
  - localization properties
    - configure 56
  - overview 37
    - accessing 38
    - account locking 41
    - authenticating 39
    - client detection 40
  - remote-auth.dtd 63
  - samples
    - login module 77
    - remote client API 77
  - SPI
    - java 71
  - URL parameters 71
  - user interface 42
    - customization 43
    - JSP templates 45
- authentication user interface
  - customization 43
  - JSP templates 45
- authentication-related logs 253

## B

- background colors customization 34
- backup and restore 263
  - backup script 264
  - restore script 267
- backup script 264

## C

- C applications
  - and policy 171
  - authentication 63
    - examples 73
- CDSSO component 82
- certificate authentication module 48
- Certificate Server
  - documentation 14
- class of service 296
  - and dynamic attributes 118
  - conflicts 298
  - definition entry 297
  - template entry 297
- client data
  - in client detection 256
- client detection 255
  - and authentication service 40
  - API
    - java 257
  - client data 256
  - overview 255
- common domain 236
- configuration directives
  - in AMConfig.properties 274
- configure
  - localization properties 56
- console
  - and authentication user interface 42
  - architecture 28
  - customization 29
    - background colors 34
    - creating custom GUI 30
    - default console directory 30
  - display container objects 35
  - localization 34
  - plug-in modules 35
  - precompile JSP 32
  - user profile view 32
- graphical user interface 28
- overview 27
- plug-in modules 28
- sample 36

- console-related logs 253
- container objects
- displaying 35
- ContainerDefaultTemplateRole 120
- cookies
- and session tokens 81
- core authentication module 47
- CoS 296
- conflicts 298
- definition entry 297
- template entry 297
- create
- custom authentication modules 55
- custom GUI 30
- cross domain controller 82
- cross-domain
- 82, 84
- cross-domain SSO 81
- configure 83
- enable 82
- custom authentication modules 54
- amAuth.xml
  - modify 61
- create 55
- credential requirements 57
  - and Auth\_Module\_Properties.dtd 57
- customization
- authentication and JSP templates 45
- authentication user interface 43
- background colors 34
- console 29
- creating custom GUI 30
- default console directory 30
- display container objects 35
- federation management module 241
- localizing console 34
- plug-in modules 35

- precompile JSP 32
- user profile view 32

## D

- debug files 253
- default console directory 30
- deployment directives
  - in AMConfig.properties 272
- developer information 15
- Directory Server 287
  - ACIs 292
    - default 294
    - defined 293
    - format 293
  - class of service 296
    - conflicts 298
    - definition entry 297
    - template entry 297
  - documentation 14
  - extend schema 120
  - overview 287
  - roles 288
    - Identity Server and 290
    - managed roles 288
  - schema 298
- documentation
  - Certificate Server 14
  - Directory Server 14
  - overview 12
  - Proxy Server 14
  - related iPlanet products 14
  - terminology 13
  - Web Server 14
- downloads
  - Sun ONE software 14
- DTD files 125
  - policy.dtd 164
- dynamic attributes
  - and service files 118

## F

- federated identity 235
- federation management 233
  - API
    - java 240
  - Liberty Alliance Project 234
  - Liberty concepts
    - account federation 235
    - authentication domain 235
    - common domain 236
    - defined 235
    - federated identity 235
    - federation termination 236
    - identity provider 235
    - name identifier 236
    - service provider 235
    - single logout 236
    - single sign-on 236
    - trusted provider 235
  - module
    - customization 241
  - overview 233
  - process 236
  - protocols 238
    - federation termination notification 239
    - IDP introduction 240
    - name registration 239
    - single log-out 239
    - single sign-on and federation 239
  - samples 242
- federation termination 236
- federation termination notification protocol 239
- federation-related logs 253
- FQDN mapping
  - and authentication service 41

## G

- global attributes
  - and service files 117
- graphical user interface. See also console

## I

- identity management 97
  - amEntrySpecific.xml 107
  - object templates 99, 101
    - creation template 100
    - structure template 100
  - ums.xml 100
- overview 97
  - abstract objects 98
  - marker object classes 98
- samples
  - SDK 109
- SDK 102
  - caching 106
  - remote installation 107
- ums.xml
  - modify 101
- identity provider 235
- Identity Server
  - file system 24
  - overview 17
    - application management services 19
    - data management components 17
    - extending 21
    - Java packages 22
    - managing access 20
  - related product information 14
- Identity Server Console. See console
- IDP introduction protocol 240
- inheritance
  - attributes 120

## J

- java
  - API
    - authentication 62
    - client detection 257
    - federation management 240
    - logging service 249
    - SSO 85
    - SSO sample 90
    - utility 268

- identity management SDK 102
  - caching 106
  - remote installation 107
- policy SDK 167
  - policy evaluation API 168
  - policy management API 169
  - policy plugin API 170
- SAML service SDK 227
- service management SDK 156
- SPI
  - authentication 71
  - logging service 251
- Java packages 22

## L

- LDAP authentication module 48
- LDAP schema 298
- Liberty Alliance Project 234
- Liberty concepts
  - account federation 235
  - authentication domain 235
  - common domain 236
  - defined 235
  - federated identity 235
  - federation termination 236
  - identity provider 235
  - name identifier 236
  - service provider 235
  - single logout 236
  - single sign-on 236
  - trusted provider 235
- localization
  - console 34
- localization properties
  - configure 56, 123
- log message formats 247
- log security 247
- log types
  - authentication-related logs 253
  - console-related logs 253
  - debug files 253
  - federation-related logs 253
  - SSO-related logs 252

- logging
  - amLogging.xml 247
- logging service 245
  - API
    - java 249
    - sample code 251
  - architecture 246
  - log message formats 247
  - log security 247
  - log types
    - authentication-related logs 253
    - console-related logs 253
    - debug files 253
    - federation-related logs 253
    - SSO-related logs 252
  - overview 245
  - secure logging 254
  - SPI 251

## M

- managed roles 288
- marker object classes 98
- membership authentication module 48
- module authentication 54

## N

- name identifier 236
- name registration protocol 239
- normal policy 162
- NT authentication module 48

## O

- object templates 99
  - ums.xml 100
- organization attributes
  - and service files 118

- organization authentication 49
- overview
  - AMConfig.properties 271
  - API
    - SSO 86
  - application management services 19
  - authentication service 37
    - accessing 38
    - account locking 41
    - authenticating 39
    - client detection 40
    - user interface 42
  - client detection 255
  - console 27
  - cross-domain SSO 81
  - data management components 17
  - Directory Server 287
  - extending Identity Server 21
  - federation management 233
  - identity management 97
    - abstract objects 98
    - marker object classes 98
  - Identity Server 17
    - file system 24
  - Java packages 22
  - logging service 245
  - managing access 20
  - policy service 159
  - SAML service 219
  - service files 114
    - DTD files 114
  - service management 113
  - SSO 79
    - policy agents 80
    - session tokens 80
    - user credentials 80

## P

- plug-in module customization 35
- plug-in modules 28
- policy
  - and subjects 163
  - SDK 167

- C applications 171
  - policy evaluation API 168
  - policy management API 169
  - policy plugin API 170
- types 162
  - normal 162
  - referral 162
- policy attributes
  - and service files 119
- policy evaluation API 168
- policy management API 169
- policy plugin API 170
- policy service 159
  - architecture 161
  - defined 160
  - overview 159
  - policy
    - and subjects 163
  - policy types 162
    - normal 162
    - referral 162
  - policy.dtd 164
- policy.dtd 164
- precompile JSP 32
- process
  - federation management 236
- Professional Services 14
- profile types
  - and SAML 222
  - web artifact profile 222
  - web POST profile 223
- proprietary authentication modules 47
- protocols
  - federation management 238
  - federation termination notification 239
  - IDP introduction 240
  - name registration 239
  - single log-out 239
  - single sign-on and federation 239
- Proxy Server
  - documentation 14

## R

- RADIUS authentication module 48
- read-only directives
  - in AMConfig.properties 279
- referral policy 162
- remote-auth.dtd 63
- restore and backup 263
- restore script 267
- role authentication 50
- roles 288
  - Identity Server
    - roles and 290
  - Identity Server and 290
  - managed roles 288

## S

- SafeWord authentication module 48
- SAML service 219
  - access to 226
  - amSAML.xml 226
  - assertion types 221
  - overview 219
  - profile types 222
    - web artifact profile 222
    - web POST profile 223
  - SAML SOAP receiver 224
    - SOAP messages 225
  - samples 231
    - SDK 227
- SAML SOAP receiver 224
  - SOAP messages 225
- samples
  - authentication
    - login module 77
    - remote client API 77
  - console 36
  - federation management 242
  - identity management SDK 109
  - logging service
    - code 251
  - SAML 231
  - SSO 94

- command line SSO 95
  - remote SSO 95
  - SSO servlet 94
- search template 101
- secure logging 254
- service attributes
  - inheritance 120
  - virtual attributes 118
- service authentication 52
- service definition 115
  - amAdmin.dtd 135
  - Directory Server
    - extend schema 120
  - DTD files 125
  - localization properties 123
  - service registration 124
  - sms.dtd 126
- service file
  - import 122
- service files
  - attribute inheritance 120
  - attributes 117
    - dynamic 118
    - global 117
    - organization 118
    - policy 119
    - user 119
  - batch processing
    - batch processing service files 153
  - ContainerDefaultTemplateRole 120
  - create 117
  - modify 151
  - naming conventions 117
  - overview 114
    - DTD files 114
  - ums.xml 99, 100
  - user pages
    - customize 156
- service management 113
  - overview 113
  - SDK 156
  - service definition 115
  - service files
    - create 117
    - naming conventions 117
- service provider 235
- service registration 124
- services
  - overview
    - authentication 37
    - federation management 233
    - SSO 79
- session service see SSO
- session tokens
  - and cookies 81
- single logout 236
- single log-out protocol 239
- single sign-on 236
- single sign-on and federation protocol 239
- single sign-on See SSO
- sms.dtd 126
- SOAP messages 225
- Solaris
  - patches 14
  - support 14
- SPI
  - authentication
    - java 71
  - logging service 251
- SSO 79
  - API
    - java 85
    - non-web-based applications 85
    - overview 86
    - sample 90
  - cookies and session tokens 81
  - cross-domain
    - AMagent.properties 84
    - CDSSO component 82
    - configure 83
    - cross domain controller 82
    - enable 82
  - cross-domain support 81
  - overview 79
    - policy agents 80
    - session tokens 80
    - user credentials 80
  - samples 94
    - command line SSO 95
    - remote SSO 95
    - SSO servlet 94

- SSO-related logs 252
- subjects
  - policy 163
- Sun ONE
  - support 14
- support
  - Professional Services 14
  - Solaris 14
  - Sun ONE 14

## T

- trusted provider 235

## U

- ums.xml
  - creation template 100
  - modify 101
  - object templates 99
  - search template 101
  - structure template 100
- Unix authentication module 48
- URL parameters 71
  - authentication 71
- user attributes
  - and service files 119
- user authentication 54
- user pages
  - customize 156
- user profile view
  - customization 32
- utilities 263
  - backup and restore 263
    - backup script 264
    - restore script 267
- utility
  - API
    - java 268

## V

- virtual attributes
  - and dynamic attributes 118

## W

- web artifact profile 222
- web POST profile 223
- Web Server
  - documentation 14

## X

- XML
  - abstract objects
    - update 124
  - amEntrySpecific.xml 107
  - amSAML.xml 226
  - Directory Server
    - schema 120
  - service definition
    - amAdmin.dtd 135
    - DTD files 125
    - localization properties 123
    - service registration 124
    - sms.dtd 126
  - service file
    - import 122
  - service files
    - amLogging.xml 247
    - attribute inheritance 120
    - attributes 117, 118, 119
    - batch processing 153
    - ContainerDefaultTemplateRole 120
    - create 117
    - DTD files 114
    - modify 151
    - naming conventions 117
    - overview 114
    - user pages 156
  - ums.xml 99, 100

- creation template 100
- modify 101
- search template 101
- structure template 100
- virtual attributes 118