Customization and API Guide

Sun™ ONE Identity Server

Version 6.1

816-6774-10
November 2003

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at http://www.sun.com/patents and one or more additional patents or pending patent applications in the U.S. and in other countries.

THIS PRODUCT CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF SUN MICROSYSTEMS, INC. USE, DISCLOSURE OR REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN PERMISSION OF SUN MICROSYSTEMS, INC.

U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

This distribution may include materials developed by third parties.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and in other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, Java, Solaris, JDK, Java Naming and Directory Interface, JavaMail, JavaHelp, J2SE, iPlanet, the Duke logo, the Java Coffee Cup logo, the Solaris logo, the SunTone Certified logo and the Sun ONE logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon architecture developed by Sun Microsystems, Inc.

Legato and the Legato logo are registered trademarks, and Legato NetWorker, are trademarks or registered trademarks of Legato Systems, Inc. The Netscape Communications Corp logo is a trademark or registered trademark of Netscape Communications Corporation.

The OPEN LOOK and Sun(TM) Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

Products covered by and information contained in this service manual are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

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.

Sun Microsystems, Inc. détient les droits de propriété intellectuels relatifs à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et ce sans limitation, ces droits de propriété intellectuelle peuvent inclure un ou plus des brevets américains listés à l'adresse http://www.sun.com/patents et un ou les brevets supplémentaires ou les applications de brevet en attente aux Etats - Unis et dans les autres pays.

CE PRODUIT CONTIENT DES INFORMATIONS CONFIDENTIELLES ET DES SECRETS COMMERCIAUX DE SUN MICROSYSTEMS, INC. SON UTILISATION, SA DIVULGATION ET SA REPRODUCTION SONT INTERDITES SANS L AUTORISATION EXPRESSE, ECRITE ET PREALABLE DE SUN MICROSYSTEMS, INC.

Cette distribution peut comprendre des composants développés par des tierces parties.

Des parties de ce produit pourront être dérivées des systèmes Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, Java, Solaris, JDK, Java Naming and Directory Interface, JavaMail, JavaHelp, J2SE, iPlanet, le logo Duke, le logo Java Coffee Cup, le logo Solaris, le logo SunTone Certified et le logo Sun[tm] ONE sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

Legato, le logo Legato, et Legato NetWorker sont des marques de fabrique ou des marques déposées de Legato Systems, Inc. Le logo Netscape Communications Corp est une marque de fabrique ou une marque déposée de Netscape Communications Corporation.

L'interface d'utilisation graphique OPEN LOOK et Sun(TM) a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une license non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui, en outre, se conforment aux licences écrites de Sun.

Les produits qui font l'objet de ce manuel d'entretien et les informations qu'il contient sont regis par la legislation americaine en matiere de controle des exportations et peuvent etre soumis au droit d'autres pays dans le domaine des exportations et importations. Les utilisations finales, ou utilisateurs finaux, pour des armes nucleaires, des missiles, des armes biologiques et chimiques ou du nucleaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou reexportations vers des pays sous embargo des Etats-Unis, ou vers des entites figurant sur les listes d'exclusion d'exportation americaines, y compris, mais de maniere non exclusive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une facon directe ou indirecte, aux exportations des produits ou des services qui sont regi par la legislation americaine en matiere de controle des exportations et la liste de ressortissants specifiquement designes, sont rigoureusement interdites.

LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFACON.

Contents

List of Figures

List of Tables

List of Procedures

List of Code Examples

About This Guide

The *Sun*™ ONE *Identity Server Customization and API Guide* offers information on how to customize Sun ONE Identity Server and integrate its functionality into an organization's current technical infrastructure. It also contains details about the programmatic aspects of the product and its API. This preface contains the following sections:

- Audience for This Guide
- Identity Server 6.1 Documentation Set
- Documentation Conventions Used in This Guide
- Related Information

Audience for This Guide

This *Customization and API Guide* is intended for use by IT administrators and software developers who implement an integrated identity management and web access platform using Sun ONE servers and software. It is recommended that administrators understand the following technologies:

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

Because Sun ONE Directory Server is used as the data store in an Identity Server deployment, administrators should also be familiar with the documentation provided with that product. The latest Directory Server documentation can be accessed online.

Identity Server 6.1 Documentation Set

The Identity Server documentation set is separated into two core sets of manuals: the Sun ONE Identity Server 6.1 core application manuals and the Sun ONE Identity Server Policy Agents books.

Identity Server Core Documentation

The Identity Server documentation set contains the following titles:

- *Product Brief* provides an overview of the Identity Server application and its features and functions.
- *Migration Guide* provides details on how to migrate existing data and Sun ONE product deployments to the latest version of Identity Server. For instructions on installing Identity Server, see the *Sun Java Enterprise System 2003Q4 Installation Guide*.
- *Administration Guide* describes how to use the Identity Server console as well as manage user and service data via the command line.
- *Customization and API Guide* documents how to customize an Identity Server installation. It also includes instructions on how to augment the application with new services using the public APIs.
- *Deployment Guide* provides information on planning an Identity Server deployment within an existing information technology infrastructure.
- The *Release Notes* will be available online after the product is released. They gather an assortment of last-minute information, including a description of what is new in this current release, known problems and limitations, installation notes, and how to report issues with the software or the documentation.

Updates to the *Release Notes* and links to modifications of the core documentation can be found on the Identity Server page at the Sun ONE documentation web site. Updated documents will be marked with a revision date.

Identity Server Policy Agent Documentation Set

Policy agents for Identity Server are available on a different schedule than the server product itself. Therefore, the documentation set for the policy agents is available outside the core set of Identity Server documentation. The following titles are included in the set:

- *Web Policy Agents Guide* documents how to install and configure an Identity Server policy agent on various web and proxy servers. It also includes troubleshooting and information specific to each agent.
- *J2EE Policy Agents Guide* documents how to install and configure an Identity Server policy agent that can protect a variety of hosted J2EE applications. It also includes troubleshooting and information specific to each agent.
- The *Release Notes* will be available online after the set of agents is released. There is generally one *Release Notes* file for each agent type release. The *Release Notes* gather an assortment of last-minute information, including a description of what is new in this current release, known problems and limitations, installation notes, and how to report issues with the software or the documentation.

Updates to the *Release Notes* and modifications to the policy agent documentation can be found on the Policy Agents page at the Sun ONE documentation web site. Updated documents will be marked with a revision date.

Your Feedback on the Documentation

Sun Microsystems and the Identity Server technical writers are interested in improving the documentation and welcome any comments and suggestions. Please email these comments to `docfeedback@sun.com`.

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, emphasis, and words used in the literal sense.
- `Monospace font` is used for sample code and code listings, API and language elements (such as function names and class names), filenames, pathnames, directory names, HTML tags, and any text that must be typed on the screen.
- *Italic serif font* is used within code and code fragments to indicate variable placeholders. For example, the following command uses *filename* as a variable placeholder for an argument to the `gunzip` command:

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

Terminology

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

- *Identity Server* refers to Identity Server and any installed instances of the Identity Server software.
- *Policy and Management services* refers to the collective set of Identity Server components and software that are installed and running on a dedicated deployment container such as a web server.
- *Directory Server* refers to an installed instance of Sun ONE Directory Server.
- *Application Server* refers to an installed instance of Sun ONE Application Server.
- *Web Server* refers to an installed instance of Sun ONE Web Server.
- *IdentityServer_base* is a variable place holder for the home directory where you have installed Identity Server.
- *DirectoryServer_base* is a variable place holder for the home directory where you have installed Sun ONE Directory Server.

- *ApplicationServer_base* is a variable place holder for the home directory where you have installed Sun ONE Application Server.
- *WebServer_base* is a variable place holder for the home directory where you have installed Sun ONE Web Server.
- *Web container that runs Identity Server* refers to the dedicated J2EE container (such as Web Server or Application Server) where the Policy and Management Services are installed.

Related Information

In addition to the documentation provided with Identity Server, there are several other sets of documentation that might be helpful. Table 0-1 lists these and additional sources of information.

Table 0-1 Where to Find Related Sun ONE Resources

Information or Resource	Internet Location
Directory Server documentation	`http://docs.sun.com/coll/S1_DirectoryServer_52`
Web Server documentation	`http://docs.sun.com/coll/S1_websvr61_en`
Web Proxy Server documentation	`http://docs.sun.com/prod/s1.webproxys#hic`
Sun ONE Download Center	`http://wwws.sun.com/software/download/`
Sun ONE Technical Support	`http://www.sun.com/service/sunone/software/index.html`
Sun ONE Professional Services Information	`http://www.sun.com/service/sunps/sunone/index.html`
Sun Enterprise Services, Solaris Patches and Support	`http://sunsolve.sun.com/`
Developer Information	`http://developers.sun.com/prodtech/index.html`

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

Chapter 1

Introduction

The *Sun™ ONE Identity Server Customization and API Guide* describes the programmatic and customization details 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™ application programming interfaces (APIs) for integration and the JavaServer Pages™ (JSP) for customization. This introductory chapter contains the following sections:

- Identity Server Overview
- Extending Identity Server
- Identity Server File System
- Client Browser Support

Identity Server Overview

Sun ONE Identity Server integrates identity management with the ability to create and enforce authentication processes and access to directory data and corporate resources. These capabilities enable organizations to deploy a comprehensive system that helps to secure and protect their assets and information, as well as deliver their web-based applications. Towards this end, Identity Server contains components and application management utilities or *services.*

NOTE An *identity* is a representation of an object used in a network environment. The identity, which can be internal (an employee, a printer) or external (a customer, a vendor), contains a set of attributes that uniquely identifies it. The simplest identity might contain user name (or object identifier) and password attributes. More complex identities might contain attributes for a phone number, social security number, building location, or address.

Data Management Components

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

- Service Configuration—provides a solution for customizing and registering configuration parameters or *attributes* into a service; the service can then be integrated into, and managed using, Identity Server. The solution includes a Document Type Definition (DTD) that defines the structure for creating a service's XML file, Java APIs that are used to integrate the XML file into the deployment and the Identity Server console which is used to manage the service.

- Identity Management—provides a solution for managing identities. It includes an API for creating, modifying and removing Identity-related Objects (users, roles, groups, containers, organizations, sub-organizations, etc.) as well as an XML template that defines each object's Lightweight Directory Access Protocol (LDAP) attributes. This template allows for the object's storage in the Sun ONE Directory Server, the data store for Identity Server.

- Policy Management—provides a solution for defining and retrieving access privilege settings (or *policy*) to protect an enterprise's resources. It includes an API that applications can use to retrieve an identity's policy. The policy is then used to determine an identity's right to access the requested resource.

- Federation Management—provides a solution for defining authentication domains, service providers and identity providers in order to give users the functionality of *federation*. Federation allows a user to aggregate multiple digital identities allowing single sign-on to affiliated sites. This module is based on the Liberty Alliance Project's Version 1.1 specifications.

- Current Sessions—provides a solution for an Identity Server administrator to view and manage user session information. It keeps track of 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 identity data as well as configured policies. The majority of the data is stored in the Directory Server using LDAP; certain of it is stored as XML.

Identity Server Management Services

When Identity Server is installed, a number of utilities (or *services)* are installed to help manage the deployment. A *service* is actually a grouping of configuration parameters (or *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,” in this manual and the *Sun ONE Identity Server Administration Guide*. The current installed services include:

- Administration Service—provides properties for the configuration of the Identity Server as well as attributes to customize the application specific to each configured organization. Information on the Administration Service attributes can be found in the Administration Service attributes chapter of the *Sun ONE Identity Server Administration Guide*.
- Authentication Service—provides an interface for gathering user credentials and issuing single sign-on (session) tokens. It also contains an SDK to write plug-ins in order to integrate token validation and authentication credential storage functionality for proprietary authentication servers. For information on this service, see Chapter 3, “Authentication Service” of this manula and the chapter on the Authentication Service attributes in the *Sun ONE Identity Server Administration Guide*.
- Client Detection Service—allows Identity Server to detect the client type of an accessing browser. Information on this service can be found in Chapter 11, “Client Detection Service,” in this manual and the chapter on the Client Detection Service attributes in the *Sun ONE Identity Server Administration Guide*.
- Globalization Settings—contains properties to configure Identity Server for different character sets. More information on this service, see the chapter on the Globalization Settings attributes in the *Sun ONE Identity Server Administration Guide*.
- Auditing Features—provides a record-keeping functionality. Both file-based logs and logs stored in a relational database are supported. Information on this service can be found in Chapter 10, “Auditing Features,” in this manual and the chapter on the Logging Service attributes in the *Sun ONE Identity Server Administration Guide*.
- Naming Service—allows client browsers to locate the URL for services in a deployment that is running more than one Identity Server ensuring that the URL returned for the service is the one for the host on which the user session was created. More information on this service can be found in the Naming Service attributes chapter of the *Sun ONE Identity Server Administration Guide*.

- Password Reset Service—contains properties that can be configured per organization to implement the Password Reset Service. For information on this service, see the chapter on the Password Reset Service attributes in the *Sun ONE Identity Server Administration Guide*.
- Platform Service—provides configurable attributes for the Identity Server deployment. For information on this service, see the chapter on the Platform Service attributes in the *Sun ONE Identity Server Administration Guide*.
- Policy Configuration Service—provides properties for configuring the policy function as well as attributes to configure the Policy Service for each configured organization. For information on this service, see Chapter 7, "Policy Service," in this manual and the chapter on the Policy Configuration Service attributes in the *Sun ONE Identity Server Administration Guide*.
- Security Assertion Markup Language (SAML) Service—provides an interface integrating SAML Service, Simple Object Access Protocol (SOAP) and `https` for sending and receiving security information. This service encrypts data passed between different security entities. An API is provided to this end. For information on this service, see Chapter 8, "SAML Service," in this manual and the chapter on the SAML Service attributes in the *Sun ONE Identity Server Administration Guide*.
- Session Service—provides attributes to configure session properties for all authorized sessions in each configured organization. For information on this service, see Chapter 4, "Single Sign-On And Sessions," in this manual and the chapter on the Session Service attributes in the *Sun ONE Identity Server Administration Guide*.
- User Service—provides attributes to configure the user properties for all users in each configured organization. For information on this service, see Chapter 5, "Identity Management," in this manual or the chapter on the User Service attributes in the *Sun ONE Identity Server Administration Guide*.

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 console is built using the Sun ONE Application Framework and can be called by all users, from top level administrator to end users. The console can be customized for each configured organization by modifying and integrating a set of JSP and related files. Information on console customization can be found in Chapter 2, "The Identity Server Console," in this manual. Identity Server also offers data backup, restoration and other software utilities. Information on these functionalities can be found in Chapter 12, "Identity Server Utilities," in this manual. Information on command-line executables can be found in the *Sun ONE Identity Server Administration Guide*.

Managing Access

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

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 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 verify the credentials. (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) or *session* token which holds the user’s identity. The API also generates a *token ID*, a random identification string associated with the session token. The session token is then sent back to the requesting browser in the form of a cookie while the authentication component directs the user to the requested secure application or page. Additional information on the Authentication Service can be found in Chapter 3, “Authentication Service,” in this manual.

NOTE Web access might also include an additional security measure to evaluate a user’s access privileges. This includes installed policy agents. Additional information can be found in the *Sun ONE Identity Server Web Policy Agents Guide* and *J2EE Policy Agents Guide.*

Application Access

External applications can access Identity Server to request user information using the Identity Server SDK. For example, a mail service might store its users’ mailbox size information in Identity Server and the SDK can be used to retrieve this information. To process the request, the system running the application must have the Identity Server SDK installed. Additional information on both the C and Java APIs can be found throughout this manual in the respective chapters.

Extending Identity Server

One of the architectural goals of Identity Server is to provide an extensible interface. This 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 JSP.
3. Default services can be implemented using a set of Java API.

Service Definition With XML

As discussed in "Identity Server Overview" on page 35, the application contains a number of management services. All Identity Server services are written using the XML. Administrators or service developers can modify the internal XML service files installed with Identity Server or configure new XML service files to customize the application 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," of this manual.

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

Console Customization

The Identity Server console is used for managing and monitoring identities, services and protected resources throughout the Identity Server deployment. The framework uses XML files, JSP templates and Cascading Style Sheets (CSS) to control the look and feel of the console screens. These files can be duplicated and then modified to make changes to the design for each configured 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," of this manual.

Identity Server SDK

The Identity Server SDK contains public interfaces to implement the behavior of Identity Server's default or customized services. Both Java and C interfaces are provided. The packages include:

Identity Management SDK

Identity Server provides the framework to create and manage users, roles, groups, containers, organizations, organizational units, and sub-organizations. The Java package name is `com.iplanet.am.sdk`. There are currently no comparable C interfaces.

Service Management SDK

The service management interfaces can be used by developers to register services and applications, and manage their configuration data. The Java package name is `com.sun.identity.sm`. There are currently no comparable C interfaces.

Authentication Programming Interfaces

Identity Server provides interfaces to extend the functionality of the Authentication Service in two ways. The API provides interfaces that can be used remotely by either Java or C applications to utilize the authentication features of Identity Server. The SPI can be used to plug new authentication modules, written in Java, into the Identity Server authentication framework.

Utility API

This API provides a number of Java classes that can be used to manage system resources. It includes thread management and debug data formatting. The Java package name is `com.iplanet.am.util`. There are currently no comparable C interfaces.

Logging API And Logging SPI

The Logging Service records, among other things, access approvals, access denials and user activity. The Logging API can be used to enable logging for external Java applications. The package names begin with `com.sun.identity.log`. The Logging SPI are Java packages that can be used to develop plug-ins for customized features. The package names begin with `com.sun.identity.log.spi`. There are currently no comparable C interfaces.

Client Detection API

Identity Server can detect the type of client browser that is attempting to access its resources and respond with the appropriately formatted pages. The Java package used for this purpose is `com.iplanet.services.cdm`. There are currently no comparable C interfaces.

SSO API

Identity Server provides Java interfaces for validating and managing SSO tokens, and for maintaining the user's authentication credentials. All applications wishing to participate in the SSO solution can use this API. The Java package name is `com.iplanet.sso`. The Session Service also includes an API for C applications.

Policy SDK

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

SAML SDK

Identity Server uses the SAML API to exchange acts of authentication, authorization decisions and attribute information. The Java package names begin with `com.sun.identity.saml`. There are currently no comparable C interfaces.

Federation Management API

Identity Server uses the Federation Management API to add functionality based on the Liberty Alliance Project specifications. The Java package name is `com.sun.liberty`. There are currently no comparable C interfaces.

Identity Server File System

Identity Server installs its packages and files in a directory named `SUNWam`. The complete file system layout for Identity Server can be found in the *Sun ONE Identity Server Deployment Guide*.

Client Browser Support

Identity Server 6.1 is supported on the following client browsers:

- Netscape™ Communciator 7.0
- Netscape Communicator 6.2.1
- Netscape Navigator™ 4.79
- Microsoft® Internet Explorer 6.0
- Microsoft Internet Explorer 5.5

Chapter 2

The Identity Server Console

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

- Overview
- Customizing The Console
- Console API
- Precompiling The Console JSP
- Console Samples

Overview

The Identity Server console is a web interface that allows administrators with different levels of access to, among other things, create organizations, create (and delete) users to (and from) those organizations, and establish enforcement policies that protect and limit access to the organization’s resources. In addition, administrators can view and terminate current user sessions and manage their federation configurations (create, delete and modify authentication domains and providers). Users without administrative privileges, on the other hand, can manage personal information (name, e-mail address, telephone number, etc.), change their password, subscibe and unsubscribe to groups, and view their roles. All of these functionalities are accomplished using a web browser.

NOTE The client web browser accessing the console must support JavaScript, version 1.2 and cookies.

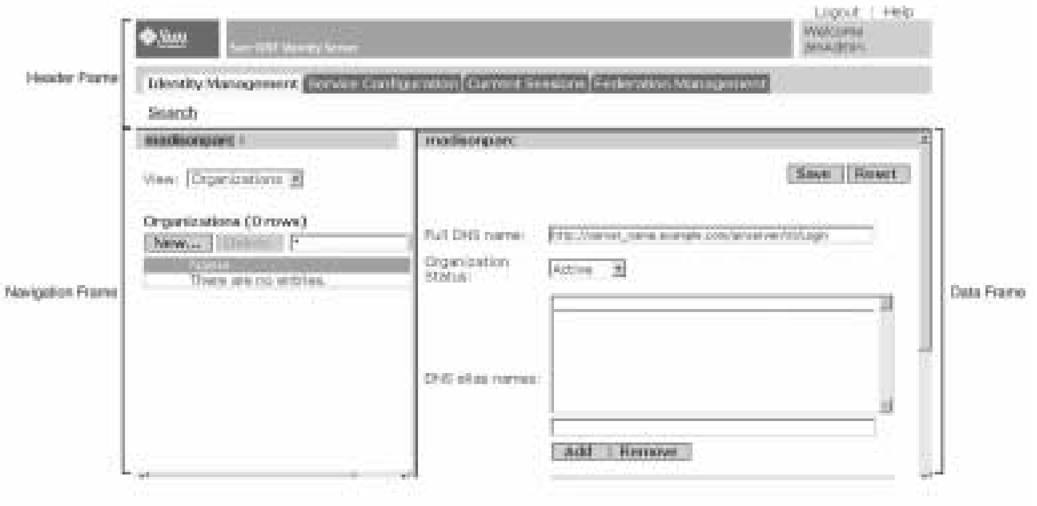
The console ships with four modules: Identity Management (including user and policy management), Service Configuration, Current Sessions (including session management) and Federation Management. Customization of these modules and the Identity Server console can be achieved, in varying degrees, by modifying the JSP and XML files that define the interface as well as extending the Sun ONE Application Framework ViewBeans.

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

Console Interface

The console is divided into three frames as pictured in Figure 2-1: Header, Navigation and Data. The Header frame displays corporate branding information as well as the first and last name of the currently logged-in user as defined in their profile. It also contains a set of tabs to allow the user to switch between the management modules, a hyperlink to the Identity Server Help system, a Search function and a Logout link. The Navigation frame on the left displays the object hierarchy of the chosen management module, and the Data frame on the right displays the attributes of the object selected in the Navigation frame.

Figure 2-1 The Identity Server Console



Generating The Console Interface

When the Identity Server console receives an HTTP(S) request, it first determines whether the requesting user has been authenticated. If not, the user is redirected to the Identity Server login page supplied by the Authentication Service. After successful authentication, the user is redirected back to the console which reads all of the user's available roles, and extracts the applicable permissions and behaviors. The console is then dynamically constructed for the user based on this information. For example, users with one or more administrative roles will see the administration console view while those without any administrative roles will see the end user console view. Roles also control the actions a user can perform and the identity objects that a user sees. Pertaining to the former, the organization administrator role allows the user read and write access to all objects within that organization while a help desk administrator role only permits write access to the users' passwords. With regards to the latter, a person with a people container administrator role will only see users in the relevant people container while the organization administrator will see all identity objects. Roles also control read and write permissions for service attributes as well as the services the user can access.

Plug-In Modules

An external application can be plugged-in to the 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. The application developer would create the JSPs for both left and right frames, and all view beans, and models associated with them. Information on how to define a module tab can be found in "To Add A Module Tab" on page 58.

Accessing The Console

The Naming Service defines URLs used to access the internal services of Identity Server. The URL used to access the Administration Console web application is:

`http://`*identity_server_host.domain_name:port*`/amconsole`

The first time Administration Console (`amconsole`) is accessed, it brings the user to the Authentication web application (`amserver`) for authentication and authorization purposes. After login, `amserver` redirects the user to the configured success login URL as discussed in "The User Interface Login URL" on page 74 of Chapter 3, "Authentication Service." The default successful login URL is `http(s)://`*identity_server_host.domain_name:port*`/amconsole/base/AMAdminFrame`.

Customizing The Console

The Identity Server console uses JSP and CSS to define the look and feel of the pages used to generate its frames. A majority of the content is generated dynamically—based on where, and at what, the user is looking. In that regard, the modification of the content is somewhat restricted. Within the Navigation frame, the layout of the controls (the view menu), the action buttons, and the table with current objects in each JSP can be changed. In the Data frame, the content displayed is dynamically generated based on the XML service file being accessed but the layout, colors, and fonts are controlled by the `adminstyle.css` style sheet.

The Default Console Files

An administrator can modify the console by changing tags in the JSP and CSS. All of these files can be found in the *IdentityServer_base*/`SUNWam/web-apps/applications/console` directory. The files in this directory provide the default Sun ONE interface. Out of the box, it contains the following sub-directories:

- `base` contains JSP that are not service-specific.
- `css` contains the `adminstyle.css` which defines styles for the console.
- `federation` contains JSP related to the Federation Management module.
- `html` contains miscellaneous HTML files.
- `images` contains images referenced by the JSP.
- `js` contains JavaScript™ files.
- `policy` contains JSP related to the Policy Service.
- `service` contains JSP related to the Service Management module.
- `session` contains JSP related to the Current Sessions (session management) module.
- `user` contains JSP related to the Identity Management module.

NOTE Console-related JSP contain HTML and custom library tags. The tags are defined in tag library descriptor files (`.tld`) found in the *IdentityServer_base*/`SUNWam/web-apps/WEB-INF` directory. Each custom tag corresponds to a view component in its view bean. While the tags in the JSP can be removed, new tags can not be added. For more information, see the Sun ONE Application Framework documentation.

Creating Custom Organization Files

To customize the console for use by a specific organization, the *IdentityServer_base*/`SUNWam/web-apps/applications/console` directory should first be copied, renamed and placed on the same level as the default directory. The files in this new directory can then be modified as needed.

NOTE There is no standard to follow when naming the new directory. The new name can be any arbitrarily chosen value.

For example, customized console files for the organization `dc=new_org, dc=com` might be found in the *IdentityServer_base*`/SUNWam/web-apps/applications/custom_directory` directory.

To Create Custom Organization Files

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

   `cd` *IdentityServer_base*`/SUNWam/web-apps/applications`

2. Make a new directory at that level.

   The directory name can be any arbitrary value. For this example, it is named *IdentityServer_base*`/SUNWam/web-apps/applications/custom_directory/`.

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

   IdentityServer_base`/SUNWam/web-apps/applications/console` contains the default JSP for Identity Server. Ensure that any image files are also copied into the new directory.

4. Customize the files in the new directory.

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

5. Modify the `AMBase.jsp` file.

   In our example, this file is found in *IdentityServer_base*`/SUNWam/web-apps/applications/custom_directory/ba`SE. 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 this 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. Change the value of the JSP Directory Name attribute in the Administration Service to match that of the directory created in Step 2 on page 50.

The JSP Directory Name attribute points the Authentication Service to the directory which contains an organization's customized console interface. Using the console itself, display the services registered to the organization for which the console changes will be displayed. If the Administration Service is not visible, it will need to be registered. For information on registering services, see Chapter 6, "Service Management" in this manual or the *Sun ONE Identity Server Administration Guide*.

Once the new set of console files have been modified, the user would need to log into the organization where they were made in order to see any changes. Elaborating on our example, if changes are made to the JSP located in the *IdentityServer_base*`/SUNWam/web-apps/applications/custom_directory` directory, the user would need need to login to that organization using the URL `http://`*server_name.domain_name:port*`/`*service_deploy_uri*`/UI/Login?org=`*custom_directory_organization*.

NOTE	More information on this login URL and authentication URL parameters can be found in Chapter 3, "Authentication Service" in this manual.

Alternate Customization Procedure

The console can also be modified by simply replacing the default images in *IdentityServer_base*`/SUNWam/web-apps/applications/console/images`, with new, similarly named images.

Miscellaneous Customizations

Included in this section are procedures for several specific customizations available to administrators of the Identity Server console.

To Modify The Service Configuration Display

A *service* is a group of attributes that are managed together by the Identity Server console. Out-of-the-box, Identity Server loads a number of services it uses to manage its own features. For example, the configuration parameters of the Logging Service are displayed and managed in the Identity Server console, while code implementations within Identity Server use the attribute values to run the service. There is a defined procedure for adding Identity Server services to the console. For information on this procedure, see "Defining A Custom Service" on page 243 of

Chapter 6, “Service Management.” Chapter 6 also contains information on how to extend existing services, add or remove a service name from the Navigation frame using the “serviceHierarchy Attribute” and change the default service display using the “propertiesViewBeanURL Attribute”

To Modify The User Profile View

The Identity Server console creates a default User Service view based on information defined in the `amUser.xml` service file.

NOTE Attributes defined as User attributes in each service’s specific XML file can also be displayed in the User Service. More information on how this is done can be found in “Customizing User Pages” on page 292 of Chapter 6, “Service Management” in this manual.

A modified 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. For example, an organization might want User attributes to be formatted differently than the default vertical listing provided. Another customization option might be to break up complex attributes into smaller ones. Currently, the server names are listed in one text field as:

protocol://*Identity Server_host.domain*:*port*

Instead, the display can be customized with three text fields:

protocol_chooser_field://*server_host_field*:*port_number_field*

A third customization option might be to add JavaScript to the ViewBean to dynamically update attribute values based on other defined input. The custom JSP would be placed in the *IdentityServer_base*/`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`. More information on this procedure can be found in “Console Samples” on page 60.

Display Options For The User Profile Page

There are a number of attributes in the Administration Service that can be selected to display certain objects on the User Profile page. Display User's Roles, Display User's Groups and User Profile Display Options specify whether to display the roles assigned to a user, the groups to which a user is a member and the schema attributes, respectively. More information on these service attributes can be found in the *Sun ONE Identity Server Administration Guide*.

To Localize The Console

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

To Display Service Attributes

Service attributes are defined in XML service files based on the `sms.dtd`. In order for a particular service attribute to be displayed in the console, it must be configured with the `any` XML attribute. The any attribute specifies whether the service attribute for which it is defined will display in the Identity Server console. More information on this attribute can be found in "any Attribute" on page 264 of Chapter 6, "Service Management" in this manual.

To Customize Interface Colors

All the colors of the console are configurable using the Identity Server style sheet `adminstyle.css` located in the *IdentityServer_base*`/SUNWam/web-apps/applications/console/css` directory. For instance, to change the background color for the navigation frame, modify the `BODY.navFrame` tag; or to change the background color for the data frame, modify the `BODY.dataFrame`. The tags take 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 respective frame after the page is reloaded in the browser. Code Example 2-2 details the tag in `adminstyle.css`.

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

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

To Change The Default Attribute Display Elements

The console auto-generates Data frame pages based on the definition of a service's attributes in an XML service definition file. As documented in "The sms.dtd Structure" in Chapter 6, "Service Management" in this manual, each service attribute is defined with the XML attributes `type`, `uitype` and `syntax`. `Type` specifies the kind of value the attribute will take. `uitype` specifies the HTML element displayed by the console. `syntax` defines the format of the value. The values of these attributes can be mixed and matched to alter the HTML element used by the console to display the values of the attributes. For example, by default, an attribute of the `single_choice` type displays its choices as a drop down list in which only one choice can be selected. This list can also be presented as a set of radio buttons if the value of the `uitype` attribute is changed to `radio`. Code Example 2-3 illustrates this concept.

Code Example 2-3 `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>Daily</Value>
                   </DefaultValues>
               </AttributeSchema>
```

Table 2-1 is a listing of the possible values for each attribute, and the corresponding HTML element that each will display based on the different groupings.

Table 2-1 Service Attribute Values and Corresponding Display Elements

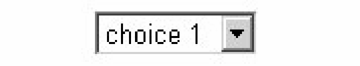
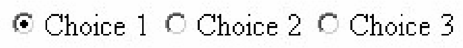
type Value	syntax Value	uitype Value	Element Displayed In Console
single_choice	string	No value defined	pull-down menu choices 
		radio	radio button choices 

Table 2-1 Service Attribute Values and Corresponding Display Elements *(Continued)*

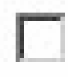
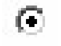
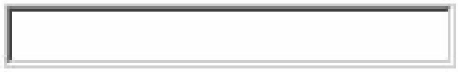
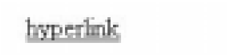
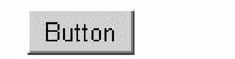
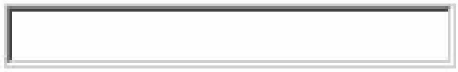
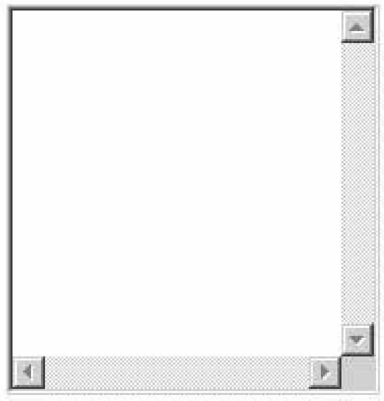
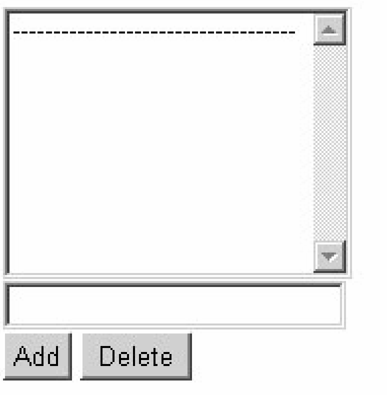
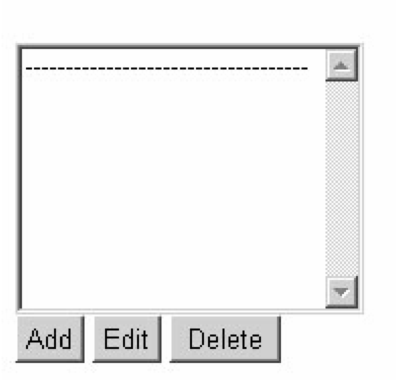
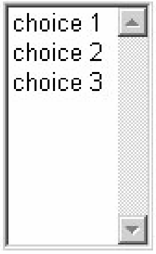
type Value	syntax Value	uitype Value	Element Displayed In Console
Single	boolean	No value defined	checkbox 
		radio	radio button 
	string	No value defined	text field 
		link	hyperlink 
		button	clickable button 
	password	No value defined	text field 
	paragraph	No value defined	scrolling text field 

Table 2-1 Service Attribute Values and Corresponding Display Elements *(Continued)*

type Value	syntax Value	uitype Value	Element Displayed In Console
list	string	No value defined	Add/Delete name list 
		name_value_list	Add/Edit/Delete name list 
multiple_choice	string	No value defined	choice list 

To Add A Module Tab

"Plug-In Modules" on page 48 mentions the capability to plug-in external applications as modules. Once this is accomplished, the module needs to be accessible via the console by adding a new module tab. Label information for module tabs are found in the `amAdminModuleMsgs.properties` console properties file located in *IdentityServer_base*/`SUNWam/locale/`. To add label information for a new module, add a key and value pair similar to `module105_NewTab=My New Tab`. Code Example 2-4 illustrates the default pairs in the file.

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
```

The module name and and a URL for the external application also need to be added to the View Menu Entries attribute in the Administration Service (or `iplanet-am-admin-console-view-menu` in the `amAdminConsole.xml` service file). When a module tab in the Header frame is clicked, this defined URL is displayed in the Navigation frame. For example, to define the display information for the tab sample, an entry similar to `module105_NewTab|/amconsole/`*custom_directory*`/`*custom_NavPage* would be added to the View Menu Entries attribute in the Administration Service.

NOTE The console retrieves all the entries from this attribute and sorts them by i18n key. This determines the tab display order in the Header frame .

After making these changes and restarting Identity Server, a new tab will be displayed with the name My New Tab. For information on the sample that explains how to add a new tab, see "Console Samples" on page 60.

To Display Container Objects

In order to create and manage LDAP organizational units (referred to as *containers* in the console), the following attributes need to be enabled (separately or together) in the Administration Service.

- Display Containers In Menu—Containers are organizational units as viewed using the Identity Server console. If this option is selected, the menu choice Containers will be displayed in the View menu for top-level Organizations, Sub-Organizations and other containers.
- Show People Containers—People containers are organizational units containing user profiles. If this option is selected, the menu choice People Containers will be displayed in the View menu for Organizations, Containers and Sub-Organizations.
- Show Group Containers—Group containers are organizational units containing groups. If this option is selected, the menu choice Group Containers will be displayed in the View menu for Organizations, Containers and Group Containers.

Viewing any of these display options is also dependent on whether the Enable User Management attribute is selected in the Administration Service. (This attribute is enabled by default after a new installation.) More information on these attributes can be found in the *Sun ONE Identity Server Administration Guide*.

Console API

The public console API package is named `com.iplanet.am.console.base.model`. It contains interfaces that can be used to monitor and react to events that occur in the console. This *listener* can be called when the user executes an action on the console that causes an event. An event can have multiple listeners registered on it. Conversely, a listener can register with multiple events. Events that might be used to trigger a listener include:

- Displaying a tab in the Header frame.
- Creating or deleting identity-related objects.
- Modifying the properties of an identity-related object.
- Sending attribute values to the console ViewBean for display purposes.

When a listener is created all the methods of that interface must be implemented thus, the methods in the `AMConsoleListener` interface must be implemented. The `AMConsoleListenerAdapter` class provides default implementations of those methods and can be used instead. Creating a console event listener includes the following:

1. Write a console event listener class (or implement the default methods in the `AMConsoleListenerAdapter` class).

2. Compile the code.
3. Register the listener in the Administration Service.

Identity Server includes a sample implementation of the ConsoleEventListener. See "ConsoleEventListener" on page 61 for more information. The Identity Server Javadocs also contains more detailed information on the listener interfaces and class.

Precompiling The Console JSP

Each JSP is compiled when it is first accessed. Because of this, there is a delay when displaying the HTML page on the browser. To avoid this delay, the system administrator can precompile the JSP by running the following command:

WebServer_install_directory`/servers/bin/https/bin/jspc -webapp` *IdentityServer_base*`/SUNWam/web-apps/applications`

where, by default, *WebServer_install_directory* is `/opt/SUNWwbsvr`.

Console Samples

Sample files have been included to help understand how the Identity Server console can be customized. The samples include instructions on how to:

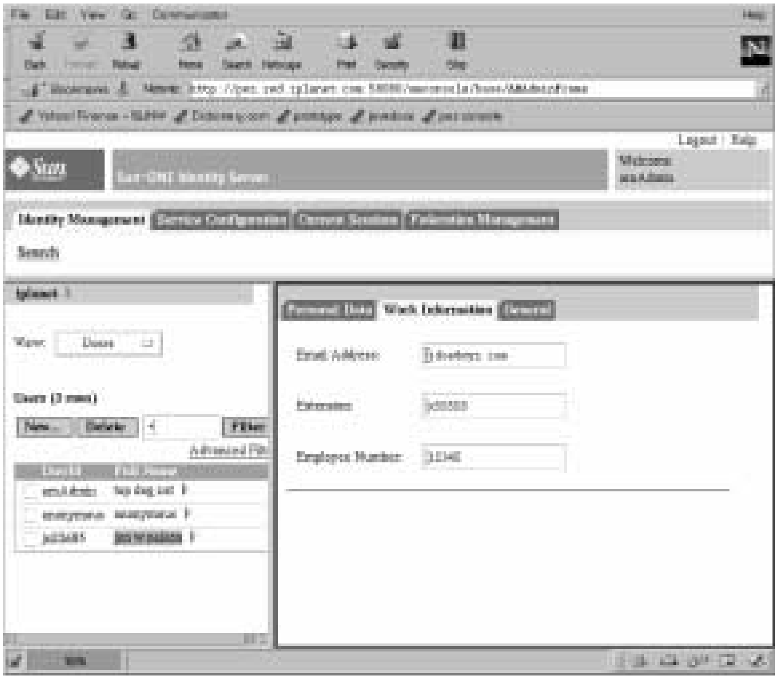
Modify User Profile Page

This sample modifies the user interface by adding a hyperlink that allows an existing user to change their configured password. It is in the `ChangeUserPassword` directory.

Create A Tabbed Identity Management Display

This sample creates a custom user profile which displays the profile with three tabs. Figure 2-2 contains a screenshot of a tabbed user profile. It is in the `UserProfile` directory.

Figure 2-2 Console With Three Tabs



Email Address: [illegible]

Extension: [illegible]

Employee Number: [illegible]

ConsoleEventListener

This sample displays the parameters passed to `AMConsoleListener` class in the `amConsole` debug file. It is in the `ConsoleEventListener` directory.

Add Administrative Function

This sample adds functionality to the Identity Management module that allows an administrator to move a user from one organization to other. It is in the `MoveUser` directory.

Add A New Module Tab

This sample adds a new tab into the Header frame. This tab will connect to an external application and can be configured using the console. It is in the `NewTab` directory.

Create A Custom User Profile View

This sample creates a custom user profile view to replace the default user profile view. A different user profile view can be created for each configured organization. A custom class would need to be written that extends the default user profile view bean. This class would then be registered in the User Profile Display Class attribute of the Administration Service. There is an example of how to do this in the samples directory. This sample is in the `UserProfile` directory.

These samples are located in *IdentityServer_base*/`SUNWam/samples/console`. Open the `README` file in this directory for general instructions. Each specific sample directory also contains a `README` file with instructions relevant to that sample.

NOTE The console samples are only available when Identity Server is installed on the Solaris™ operating system.

Chapter 3

Authentication Service

The Authentication Service is the point of entry for Sun™ ONE Identity Server. A user or client application must pass an authentication process before being allowed access to the console or any resource that is secured by it. This chapter explains the authentication process, custom authentication modules and clients, and other related features. It contains the following sections:

- Overview
- Authentication Service Modules
- Authentication Service User Interface
- Authentication Methods
- Authentication Features
- Authentication DTD Files
- Custom Authentication Modules
- Authentication Programming Interfaces
- Authentication Samples

Overview

Identity Server provides secure access to web-based (or non-web-based) applications and the data that they store. When a user or application attempts to access a protected resource, it is directed to submit credentials to one (or more) authentication modules; for instance, the LDAP module requires authentication against a Sun ONE Directory Server while the SecurID® module requires authentication against RSA ACE/Server® software. Gaining access to any of these resources requires that the requesting entity be given permission based on the submitted credentials. The Authentication Service is the *authority*, granting or denying access upon completion of the required authentication process. After a successful authentication, the requestor would be directed to the requested resource or the Identity Server console.

The Authentication Service may be accessed in one of the following ways:

- End users authenticate to Identity Server via a web browser in order to gain access to either the Identity Server console OR a protected resource based on a configured redirection URL.
- Java™ applications authenticate to Identity Server using the Java Authentication API.
- C applications authenticate to Identity Server usingthe C authentication API.

Authentication Via A Web Browser

A user with a web browser can authenticate to Identity Server using the Authentication Service User Interface. This interface is accessed by entering the Authentication Service User Interface login URL in the browser's location bar. After entering the URL, the user is prompted to submit verifying credentials based on the invoked authentication module(s). Once the credentials have been passed back to Identity Server (assuming successful authentication), the user can gain access based on their privileges:

- Administrators can access the administration portion of the Identity Server console to manage their organization's identity data.
- Users can access their own profiles to modify personal data.
- A user can access a resource defined as a redirection URL parameter appended to the login URL. For more information on redirection URLs, see "Authentication Methods" on page 103.
- A user can access the resource protected by a policy agent.

NOTE An initial step in the authenticating process is to identify the type of client making the `HTTP(S)` request. The URL information is used to retrieve the browser's characteristics and, based on these characteristics, the correct authentication pages are returned; for example, HTML pages. Once the user is validated, the client type is added to the session token. More information on client detection can be found in Chapter 11, "Client Detection Service" in this manual.

Authentication Via The Java API

External Java applications can authenticate to Identity Server using the Authentication API For Java Applications. This API provides interfaces to initiate the authentication process and communicate authentication credentials to the Authentication Service; the Identity Server API are based on the Java Authentication and Authorization Service (JAAS) specification. JAAS are Java packages that enable services to authenticate and enforce access controls upon users. They implement a version of the standard Pluggable Authentication Module (PAM) framework, and support user-based authorization.

NOTE The JAAS packages, starting with `javax.security.auth`, can be found in the Javadocs for Java 2 Platform, Standard Edition (J2SE), version 1.4.2.

Developers can incorporate the API classes and methods into their Java applications to allow communication with the Authentication Service. The external application's Java request is converted to an XML message format and passed to Identity Server over `HTTP(S)`. Once received, the XML message is converted back into a Java request which can be interpreted by the Authentication Service.

NOTE All request and subsequent return messages are passed in the XML format. The messages are structured according to the `remote-auth.dtd` discussed in "The remote-auth.dtd Structure" on page 135.

The Authentication Service returns login requirement screens based on the authentication module being accessed. The user returns authentication credentials and is allowed or denied access based on these credentials. The Java API package is discussed further in "Authentication API For Java Applications" on page 154.

Authentication Via The C API

Identity Server also includes an Authentication API for C applications to authenticate to the Identity Server. This API provides functions to initiate the authentication process and communicate authentication credentials to the Authentication Service. After passing the authentication process, a validated session token is sent back to the C application. The C API are discussed further in "Authentication API For C Applications" on page 156.

Redirection URLs

Upon a successful or failed authentication, Identity Server looks for information on where to redirect the user. There is an order of precedence in which the application will look for this information based on the authentication method and whether the authentication has been successful or has failed. Because these redirection URLs are based on the method of authentication, this order (and related information) is detailed in "Authentication Methods" on page 103.

Authentication Service Modules

Identity Server is installed with a set of default authentication services. The following sections describe the installed *modules* and, where applicable, provide an image of the module's login requirement screen. (The login requirement screens are discussed in more detail in "Authentication Service User Interface" on page 74.)

NOTE See the *Sun ONE Identity Server Administration Guide* for more information on how to register the authentication services and modules using the Identity Server console.

Authentication Configuration Service

The Authentication Configuration Service allows for the configuration of authentication modules based on roles or organizations. It is also the service where the Login Succes URL and the Login Failed URL attributes are defined. This is not a module to which a user can login. More information on this service can be found in the Authentication Configuration Service Attributes chapter of the *Sun ONE Identity Server Administration Guide*.

Core Authentication Service

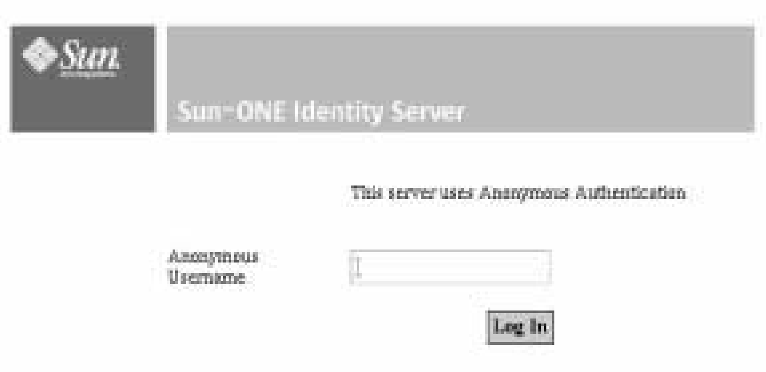
The Core Authentication Service is the configuration base for all authentication modules. It must be registered as a service to an organization before any user can log in using the other authentication modules. It allows the Identity Server administrator to define default values for an organization's authentication parameters. These values can then be picked up if no overriding value is defined in the chosen authentication module. The default values for the Core Authentication

Service are defined in the amAuth.xml file and stored in Directory Server after installation. This is not a module to which a user can login. More information on this service can be found in Core Authentication Service Attributes chapter of the *Sun ONE Identity Server Administration Guide*.

Anonymous Authentication Module

This module allows a user to log in without specifying a user name and/or password. Additionally, an Anonymous user can be created. Logging 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. More information on this module can be found in the Anonymous Authentication Attributes chapter of the *Sun ONE Identity Server Administration Guide*.

Figure 3-1 Anonymous Authentication Login Requirement Screen



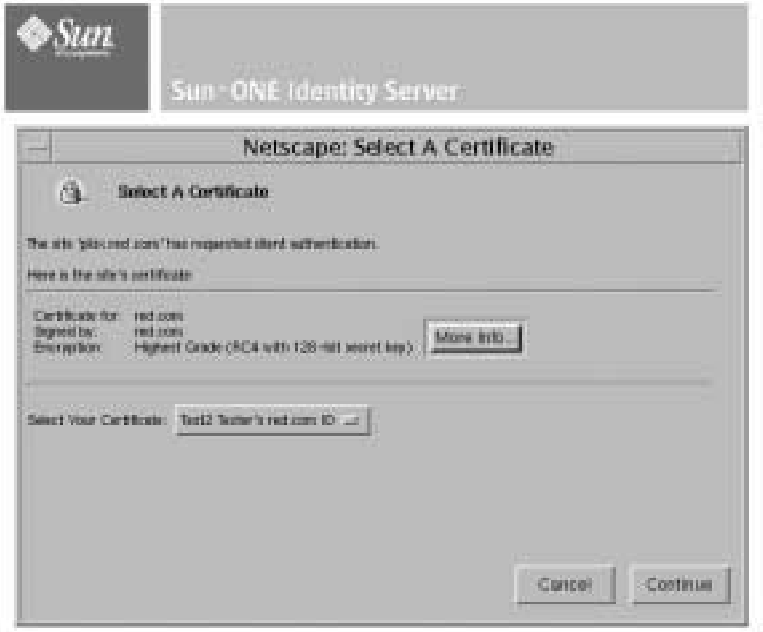
This server uses Anonymous Authentication

Anonymous Username

Certificate Authentication Module

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. More information on this module can be found in the Certificate Authentication Attributes chapter of the *Sun ONE Identity Server Administration Guide*.

Figure 3-2 Certificate-based Authentication Login Requirement Screen



The site 'piox.red.com' has requested client authentication.

Here is the site's certificate

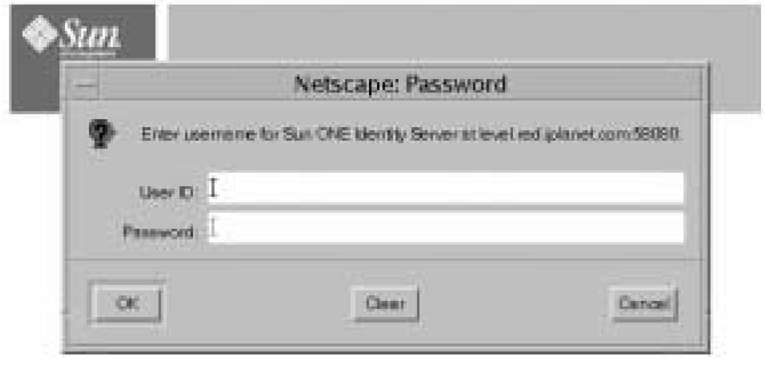
Certificate for: red.com
Signed by: red.com
Encryption: Highest Grade (RC4 with 128-bit secret key)

Select Your Certificate: Test2 Tester's red.com ID

HTTP Basic Authentication Module

This module allows login using the HTTP's basic authentication with no data encryption. A user name and password are requested via the web browser. Credentials are validated internally using the LDAP authentication module. More information on this module can be found in the HTTP Basic Authentication Attributes chapter of the *Sun ONE Identity Server Administration Guide*.

Figure 3-3 HTTP Basic Authentication Login Requirement Screen



Netscape: Password

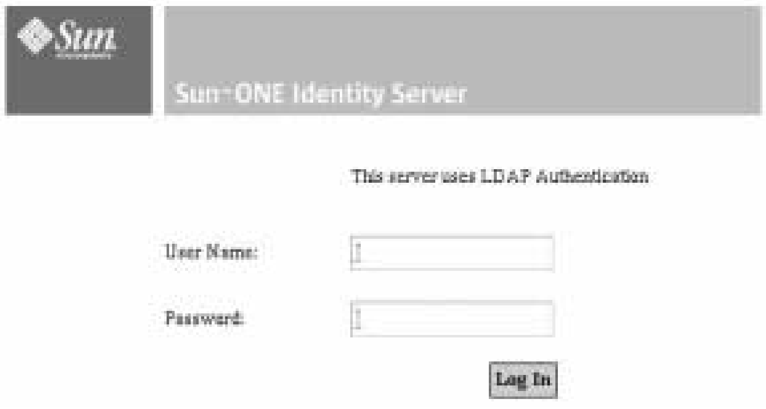
Enter username for Sun ONE Identity Server at level.red.iplanet.com:58080

LDAP Authentication Module

This module allows for authentication using LDAP bind, an operation which associates a user ID password with a particular LDAP entry. The authentication module enabled after Identity Server is installed is LDAP. More information on this module can be found in the LDAP Authentication Attributes chapter of the *Sun ONE Identity Server Administration Guide*.

NOTE An administrator can define different multiple LDAP configurations for an organization. More information on this can be found in “Multi-LDAP Authentication Module Configuration” on page 126.

Figure 3-4 LDAP Authentication Login Requirement Screen



Sun ONE Identity Server

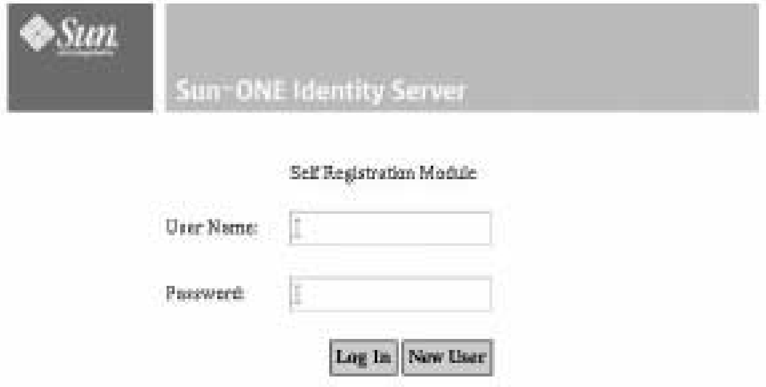
This server uses LDAP Authentication

Membership Authentication Module

Membership authentication is implemented similarly to personalized sites such as `my.site.com`, or `mysun.sun.com`. When this service is enabled, a user can create an account, personalize it without the aid of an administrator, and access it as a registered user. More information on this module can be found in the Membership Authentication Attributes chapter of the *Sun ONE Identity Server Administration Guide*.

NOTE The Membership Authentication Module is used in "Configuring The Authentication Module" on page 144 as a sample module.

Figure 3-5 Membership Authentication Login Requirement Screen

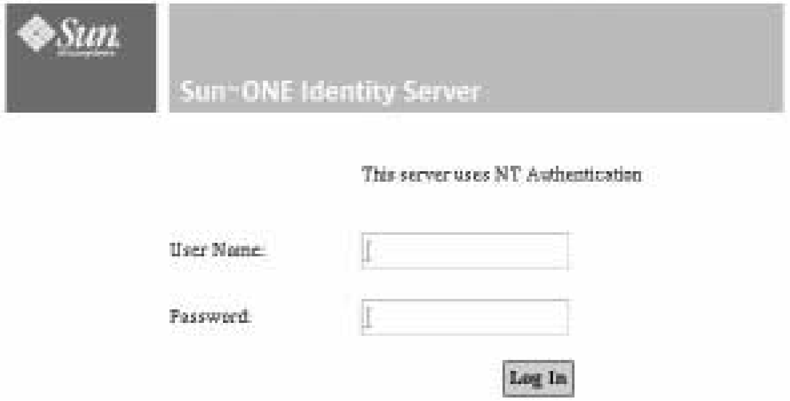


NT Authentication Module

This module allows for authentication against a Microsoft Windows® NT server. More information on this module can be found in the NT Authentication Attributes chapter of the *Sun ONE Identity Server Administration Guide*.

NOTE In order to actualize the NT authentication module, Samba 2.2.2 must be downloaded and installed. Samba is a file and print server for blending Windows and UNIX® machines together without requiring a separate Windows NT/2000 Server. More information, and the download itself, can be accessed at `http://wwws.sun.com/software/download/products/3e3af224.html`.

Figure 3-6 NT Authentication Login Requirement Screen



This server uses NT Authentication

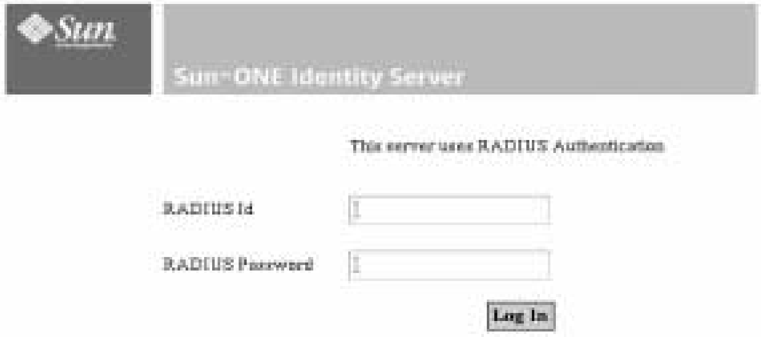
User Name:

Password:

RADIUS Authentication Module

This module allows for authentication using an external Remote Authentication Dial-In User Service (RADIUS) server. More information on this module can be found in the RADIUS Authentication Attributes chapter of the *Sun ONE Identity Server Administration Guide*.

Figure 3-7 RADIUS Authentication Login Requirement Screen



This server uses RADIUS Authentication

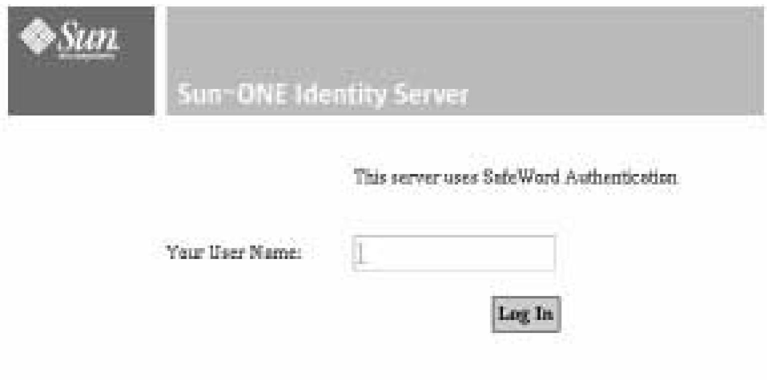
RADIUS Id

RADIUS Password

SafeWord Authentication Module

This module allows for authentication using Secure Computing's SafeWord® PremierAccess™ server software and SafeWord tokens. More information on this module can be found in the SafeWord Authentication Attributes chapter of the *Sun ONE Identity Server Administration Guide*.

Figure 3-8 SafeWord Authentication Login Requirement Screen



Sun-ONE Identity Server

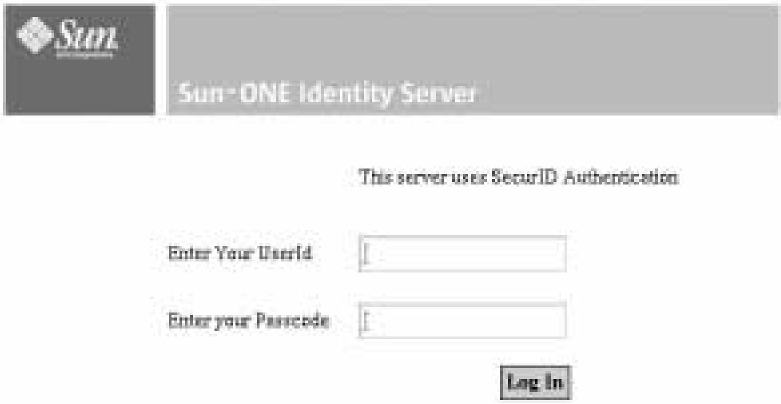
This server uses SafeWord Authentication

Your User Name:

SecurID Authentication Module

This module allows for authentication using RSA ACE/Server software and RSA SecurID authenticators. More information on this module can be found in the SecurID Authentication Attributes chapter of the *Sun ONE Identity Server Administration Guide*.

Figure 3-9 SecurID Authentication Login Requirement Screen



This server uses SecurID Authentication

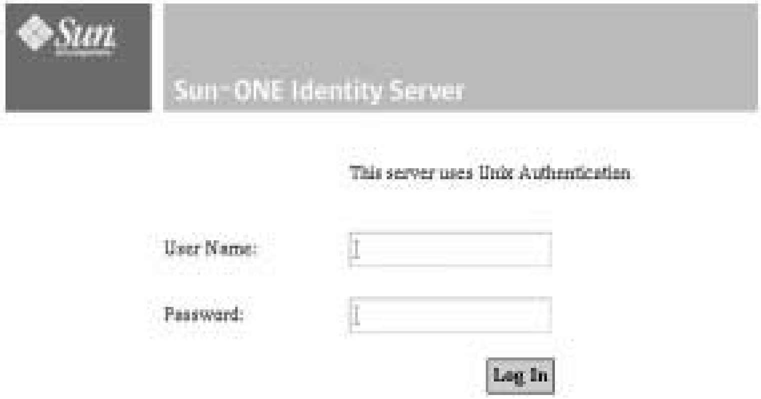
Enter Your Userid

Enter your Passcode

UNIX® Authentication Module

This Solaris only module allows for authentication using a user's UNIX identification and password. More information on this module can be found in the Unix Authentication Attributes chapter of the *Sun ONE Identity Server Administration Guide*.

Figure 3-10 UNIX Authentication Login Requirement Screen



This server uses Unix Authentication

User Name:

Password:

Authentication Service User Interface

The Authentication Service provides a web-based user interface for all out-of-the-box authentication modules installed in the Identity Server deployment. This interface provides a dynamic and customizable means for gathering authentication credentials by displaying the login requirement screens (based on the invoked authentication module) to a user requesting access. The interface is built using Sun ONE Application Framework (sometimes referred to as *JATO*), a Java 2 Enterprise Edition (J2EE) presentation framework used to help developers build functional web applications.

NOTE Information on Sun ONE Application Framework 2.0 can be found at `http://docs.sun.com/coll/S1_appframe20_en`.

The User Interface Login URL

The Authentication Service user interface is accessed by entering a login URL into the Location Bar of a web browser. This URL is:

`http://`*identity_server_host.domain_name:port*`/`*service_deploy_uri*`/UI/Login`

NOTE During installation, the *service_deploy_uri* is configured as `amserver`. This default service deployment URI will be used throughout this document.

The user interface login URL can also be appended with Login URL Parameters to define specific authentication methods or successful/failed authentication redirection URLs. Additional information on redirection URLs can be found in "Authentication Methods" on page 103.

Login 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`. A number of parameters can be combined in one login URL as in `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?module=LDAP&locale=ja&goto=http://www.sun.com`. If more than one parameter exists, they are separated by an ampersand (&). The combinations though must adhere to the following guidelines:

- Each parameter can occur only once in one URL. For example, `module=LDAP&module=NT` is not computable.
- Both the `org` parameter and the `domain` parameter determine the login organization. In this case, only one of the two parameters should be used in the login URL. If both are used and no precedence is specified, only one will take effect.
- The parameters `user`, `role`, `service`, `module` and `authlevel` are for defining authentication modules based on their respective criteria. Due to this, only one of them should be used in the login URL. If more than one is used and no precedence is specified, only one will take effect.

The following sections describe parameters that, when appended to the The User Interface Login URL and typed in the Location bar of a web browser, achieve various authentication functionalities.

TIP To simplify an authentication URL and parameters for distribution throughout an organization, an administrator might configure an HTML page with a simple URL that possesses links to the more complicated login URLs for all configured authentication methods.

goto Parameter

A `goto=`*successful_authentication_URL* parameter overrides the value defined in the Login Success URL of the Authentication Configuration Service. It will link to the specified URL when a successful authentication has been achieved. A `goto=`*logout_URL* parameter can also be used to link to a specified URL when the user is logging out. An example goto on a successful authentication URL might be `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?goto=http://www.sun.com/homepage.html`. An example `goto` logout URL might be `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Logout?goto=http://www.sun.com/logout.html`.

NOTE There is an order of precedence in which Identity Server looks for successful authentication redirection URLs. Because these redirection URLs and their order are based on the method of authentication, this order (and related information) is detailed in “Authentication Methods” on page 103.

gotoOnFail Parameter

A `gotoOnFail=`*failed_authentication_URL* parameter overrides the value defined in the Login Failed URL of the Authentication Configuration Service. It will link to the specified URL if a user has failed authentication. An example `gotoOnFail` URL might be `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?gotoOnFail=http://www.sun.com/auth_fail.html`.

NOTE There is an order of precedence in which Identity Server looks for failed authentication redirection URLs. Because these redirection URLs and their order are based on the method of authentication, this order (and related information) is detailed in "Authentication Methods" on page 103.

org Parameter

The `org=`*orgName* parameter allows a user to authenticate as a user in the specified organization.

TIP A user who is not already a member of the specified organization will receive an error message when they attempt to authenticate with the `org` parameter. A user profile, though, can be dynamically created in the Directory Server if all of the following are TRUE:

- The User Profile attribute in the Core Authentication Service must be set to `Dynamically Created`.
- The user must successfully authenticate to the required module.
- The user does not already have a profile in Directory Server.

From this parameter, the correct login page (based on the organization and its locale setting) will be displayed. If this parameter is not set, the default is the top-level organization. An example `org` URL might be `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?org=sun`.

NOTE See the *Sun ONE Identity Server Administration Guide* for information on how to configure authentication for an organization.

user Parameter

The `user=`*userName* parameter forces authentication based on the module configured in User Authentication Configuration attribute of the user's profile. For example, one user's profile can be configured to authenticate using the Certification module while another user might be configured to authenticate using the LDAP module. Adding this parameter sends the user to their configured authentication process rather than the method configured for their organization. An example `user` URL might be `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?user=jsmith`.

NOTE See the *Sun ONE Identity Server Administration Guide* for information on how to configure authentication for a user.

role Parameter

A `role=`*roleName* parameter sends the user to the authentication process configured for the specified role. A user who is not already a member of the specified role will receive an error message when they attempt to authenticate with this parameter. An example `role` URL might be `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?role=manager`.

NOTE See the *Sun ONE Identity Server Administration Guide* for information on how to configure authentication for a role.

locale Parameter

Identity Server has the capability to display localized screens (translated into languages other than English) for the authentication process as well as for the console itself. The `locale=`*localeName* parameter allows the specified locale to take precedence over any other defined locales. The login locale is displayed by the client after searching for the configuration in the following places, order-specific:

1. Value of locale parameter in Login URL

   The value of the `locale=`*localeName* parameter takes precedence over all other defined locales.

2. Locale defined in user's profile

   If there is no URL parameter, the locale is displayed based on the value set in the User Preferred Language attribute of the user profile.

3. Locale defined in the HTTP header

   This locale is set by the web browser.

4. Locale defined in Core Authentication Service

   This is the value of the Default Auth Locale attribute in the Core Authentication Service. More information can be found in the *Sun ONE Identity Server Administration Guide*..

5. Locale defined in Platform Service

   This is the value of the Platform Locale attribute in the Platform Service. More information can be found in the *Sun ONE Identity Server Administration Guide*.

6. Operating system locale

The locale derived from this pecking order is stored in the user's session token and Identity Server uses it for loading the localized authentication module only. After successful authentication, the locale defined in the User Preferred Language attribute of the user's profile is used. If none is set, the locale used for authentication will be carried over. An example `module` URL might be `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?locale=ja`.

NOTE Information on how to localize the screen text and error messages can be found in "Configuring Authentication Localization Properties" on page 151 and "Configuring Console Localization Properties" on page 251 of Chapter 6, "Service Management" in this manual.

module Parameter

The `module=`*moduleName* parameter allows authentication via the specified authentication module. Any of the modules listed in "Authentication Service Modules" on page 66 can be specified although they must first be registered under the organization to which the user belongs and selected as one of that organization's authentication modules in the Core Authentication Service. An example `module` URL might be `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?module=Unix`.

NOTE The authentication module names are case-sensitive when used in a URL parameter.

service Parameter

The `service=`*serviceName* parameter allows a user to authenticate via a service's configured authentication scheme. Different authentication schemes can be configured for different services using the Authentication Configuration Service. For example, an online paycheck application might require authentication using the more secure Certificate Authentication Module while an organization's employee directory application might require only the LDAP Authentication Module. An authentication scheme can be configured, and named, for each of these services. An example `service` URL might be `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?service=sv1`.

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

arg Parameter

The `arg=newsession` parameter is used to end a user's current session and begin a new one. The Authentication Service will destroy a user's existing session token and perform a new login in one request. 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` URL might be `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?arg=newsession`.

authlevel Parameter

An `authlevel=`*value* parameter tells the Authentication Service to call a module with an authentication level equal to or greater than the specified authentication level value. Each authentication module is defined with a fixed integer authentication level. An example `authlevel` URL might be `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?authlevel=1`.

NOTE The Authentication Level is set in each module's specific profile. More information on this module can be found in the *Sun ONE Identity Server Administration Guide*.

domain Parameter

This parameter allows a user to login to an organization identified as the specified domain. The specified domain must match the value defined in the Domain Name attribute of the organization's profile. An example `domain` URL might be `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?domain=sun.com`.

TIP A user who is not already a member of the specified domain/organization will receive an error message when they attempt to authenticate with the `org` parameter. A user profile, though, can be dynamically created in the Directory Server if all of the following points are TRUE:

- The User Profile attribute in the Core Authentication Service must be set to `Dynamically Created`.
- The user must successfully authenticate to the required module.
- The user does not already have a profile in Directory Server.

iPSPCookie Parameter

The `iPSPCookie=yes` 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 Authentication Service. 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 value of the Persistent Cookie Max Time attribute specified in the Core Service elapses. An example `iPSPCookie` URL might be `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?org=example&iPSPCookie=yes`.

IDTokenN Parameters

Identity Server has included this parameter option to enable a user to pass authentication credentials using a URL or HTML forms. With the `IDTokenN=`*value* parameters, a user can be authenticated without accessing the Authentication Service User Interface. This process is called *Zero Page Login*. Zero page login works only for authentication modules that use one login page. The values of `IDToken0`, `IDToken1`, `...`, `IDTokenN` map to the fields on the authentication module's login page. For example, the LDAP authentication module might use `IDToken1` for the userID information, and `IDToken2` for password information. In this case, the LDAP module IDTokenN URL would be: `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?module=LDAP&IDToken1=`*userID*`&IDToken2=`*password*. (`module=LDAP` can be omitted if LDAP is the default authentication module.) For Anomymous authentication, the login URL parameter might be `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?module=Anonymous&IDToken1=`*anonymousUserID*.

NOTE The token names `Login.Token0`, `Login.Token1`, `...`, `Login.TokenN` (from previous releases) are still supported but will be deprecated in a future release. It is recommended to use the new `IDTokenN` parameters.

File Types Of The User Interface

The Authentication Service User Interface uses JavaServer Pages™, XML for Authentication Module Configuration Files, JavaScript Files, Cascading Style Sheets, Image Files, and Localization Properties Files to convey graphical-based representations of the login requirement screens, logout screens, and error messages for each authentication module. Table 3-1 lists these file types, how they are used and in what directory they can be found.

Table 3-1 Authentication Service User Interface File Types

File Extension	Description	Location
`.jsp`	JavaServer Pages are HTML files with JATO tags that define Login and error message pages.	*IdentityServer_base*`/SUNWam/web-apps/services/config/auth/default`
`.xml`	Authentication Module Configuration Files define login requirements.	*IdentityServer_base*`/SUNWam/web-apps/services/config/auth/default`
`.js`	JavaScript Files contains server-side code for parsing.	*IdentityServer_base*`/SUNWam/web-apps/services/js`
`.css`	Cascading Style Sheets maintain consistency in color and text.	*IdentityServer_base*`/SUNWam/web-apps/services/css`
`.gif/.jpg`	Image Files to convey the look and feel of the interface.	*IdentityServer_base*`/SUNWam/web-apps/services/login_images`
`.properties`	Localization Properties Files for internationalization.	*IdentityServer_base*`/SUNWam/locale`

JavaServer Pages

All Authentication Service user interface screens are JavaServer Pages (JSP). Simply put, JSP are HTML files that contain additional code to generate dynamic content. More specifically, they contain HTML code to display static text and graphics, as well as application code to generate information. When the page is displayed in a web browser, it will contain both the static HTML content and dynamic content retrieved via the application code. There is one login page common to all the authentication modules in Identity Server; it is called `Login.jsp`.

NOTE Notwithstanding the above, the Membership authentication module and the Self-registration authentication module each have their own login pages, `membership.jsp` and `register.jsp`, respectively.

The common login page dynamically displays the invoked authentication module's required elements at run time. For example, when a user invokes the LDAP authentication module, the LDAP module header, user name field and password field are displayed.

CAUTION When modifying JSP pages, all <, & and > characters must be escaped.

JSP Files

Identity Server includes a number of JSP for use by the Authentication Service user interface. Table 3-2 contains a list of them. They are located in *IdentityServer_base*/`SUNWam/web-apps/services/config/auth/default`.

Table 3-2 List of Customizable JSP Templates

File Name	Purpose
`account_expired.jsp`	Informs the user that their account has expired and should contact the system administrator.
`auth_error_template.jsp`	Informs the user when an internal authentication error has occurred.
`authException.jsp`	Informs the user that an error has occurred during authentication.
`configuration.jsp`	Informs the user that there has been a configuration error.
`disclaimer.jsp`	This is a sample, customizable disclaimer page used in the Self-registration authentication module.
`Exception.jsp`	Informs the user that an error has occurred.

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

File Name	Purpose
`invalidPCookieUserid.jsp`	Informs the user that a persistent cookie user name does not exist in the persistent cookie domain.
`invalidPassword.jsp`	Informs the user that the password entered does not contain enough characters.
`invalid_domain.jsp`	Informs the user that there is no such domain.
`Login.jsp`	This is a Login/Password template.
`login_denied.jsp`	Informs the user that no profile has been found in this domain.
`login_failed_template.jsp`	Informs the user that authentication has failed.
`Logout.jsp`	Informs the user that they have logged out.
`maxSessions.jsp`	Informs the user that the maximum sessions have been reached.
`membership.jsp`	A login page for the Self-registration module.
`Message.jsp`	A generic message template for a general error not defined in one of the other error message pages.
`missingReqField.jsp`	Informs the user that a required field has not been completed.
`module_denied.jsp`	Informs the user that the chosen authentication module has been denied.
`module_template.jsp`	A customizable module page.
`new_org.jsp`	This page is displayed when a user witha valid session in one organization wants to login to another organization.
`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.

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

File Name	Purpose
`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.
`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 file is an XML file that defines the requirements that each module seeks from a user for authentication. In other words, this file defines the login registration screens that a user might see when directed to authenticate (i.e. user name / password screen, change password screen, etc.). Each authentication module has its own configuration file located in the same directory as the JavaServer Pages, *IdentityServer_base*/`SUNWam/web-apps/services/config/auth/default`. Modifying elements in this file will automatically customize the authentication module's interface. The file name follows the format *modulename*`.xml`; for example, `SafeWord.xml` or `LDAP.xml` where *modulename* is the name of the class without the package name. The syntax of the file itself follows the DTD format described in "Auth_Module_Properties.dtd" on page 131.

NOTE More information on the authentication module configuration file can be found in "Configuring The Authentication Module" on page 144.

XML Files

Identity Server defines an authentication module configuration file for each of the default Authentication Service Modules. The included XML files, located in *IdentityServer_base*/`SUNWam/web-apps/services/config/auth/default`, are:

Table 3-3 List of Authentication Module Configuration Files

File Name	Purpose
`Anonymous.xml`	For anonymous authentication although there are no specific credentials required to authenticate.
`Application.xml`	For Identity Server internal use only. Do not remove or modify this file.
`Cert.xml`	For certificate-based authentication although there are no specific credentials required to authenticate.
`HTTPBasic.xml`	Defines one screen with a header only as credentials are requested via the user's web browser.
`LDAP.xml`	Defines a Login screen, a Change Password screen and two error message screens (Reset Password and User Inactive).
`Membership.xml`	Default data interface which can be used to customize for any domain.
`NT.xml`	Defines a Login screen.
`RADIUS.xml`	Defines a Login screen and a RADIUS Password Challenge screen.
`SafeWord.xml`	Defines two Login screens: one for User Name and the next for Password.
`SecurID.xml`	Defines five Login screens including UserID and Passcode, PIN mode, and Token Passcode.
`Unix.xml`	Defines a Login screen and an Expired Password screen.

JavaScript Files

Identity Server includes JavaScript files which are parsed within the `Login.jsp`. They can be found in *IdentityServer_base*/`SUNWam/web-apps/services/js`. Other, more customer-specific JavaScript files can be coded and accessed from this location.

JS Files

The JavaScript files used by the Authentication Service are detailed in Table 3-4. They can be found in *IdentityServer_base*/`SUNWam/web-apps/services/js`.

Table 3-4 List of JavaScript Files

File Name	Purpose
`auth.js`	Used by `Login.jsp` for parsing all module files to display login requirement screens.
`browserVersion.js`	Used by `Login.jsp` to detect the client type.

Cascading Style Sheets

Identity Server uses cascading style sheets (CSS) to define the look and feel of the user interface. Characteristics like fonts and font weights, background colors and link colors are specified in the CSS. They are located in *IdentityServer_base*/`SUNWam/web-apps/services/css`.

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

CSS Files

There are a number of browser-based CSS included with Identity Server. These CSS, detailed in Table 3-5, can be found in *IdentityServer_base*/`SUNWam/web-apps/services/css`.

Table 3-5 List of Cascading Style Sheets

File Name	Purpose
`css_generic.css`	Configured for generic web browsers.
`css_ie5win.css`	Configured specifically for Microsoft® Internet Explorer v.5 for Windows®.
`css_ns4sol.css`	Configured specifically for Netscape™ Communicator v. 4 for Solaris™.
`css_ns4win.css`	Configured specifically for Netscape Communicator v.4 for Windows.
`styles.css`	Used in JSP pages as a default style sheet.

Image Files

The default user interface is branded with Sun Microsystems, Inc. logos and images. Identity Server has included these GIF files in *IdentityServer_base/*`SUNWam/web-apps/services/login_images`. These images can be replaced with images relevant to other organizations. (JPG files can also be used.)

GIF Files

Table 3-6 contains a listing of the GIF images used for the user interface. These files can be found in *IdentityServer_base*/`SUNWam/web-apps/services/css`.

Table 3-6 List of Sun Microsystems Branded GIF Images

File Name	Purpose
`Identity_LogIn.gif`	Sun ONE Identity Server banner across the top.
`Registry_Login.gif`	No longer used.
`bannerTxt_registryServer.gif`	No longer used.
`logo_sun.gif`	Sun Microsystems logo in the upper right corner.
`spacer.gif`	A one pixel clear image used for layout purposes.
`sunOne.gif`	Sun ONE logo in the lower right corner.

Localization Properties Files

A localization properties file, also referred to as an *i18n (internationalization) properties file* specifies the screen text and error messages that an administrator or user will see when directed to an authentication module's attribute configuration page. Each authentication module has its own properties file that follows the naming format `amAuth`*modulename*`.properties`; for example, `amAuthLDAP.properties`. They are located in *IdentityServer_base*/`SUNWam/locale/`. The default character set is ISO-8859-1 so all values are in English, but Java applications can be adapted to various languages without code changes by translating the values in the localization properties file.

PROPERTIES Files

Table 3-7 contains a listing of the localization properties files configured for each module. These files can be found in *IdentityServer_base*/`SUNWam/locale`.

Table 3-7 List of Localization Properties Files

File Name	Purpose
amAuth.properties	Defines the parent Core Authentication Service.
amAuthAnonymous.properties	Defines the Anonymous Authentication Module.
amAuthApplication.properties	For Identity Server internal use only. Do not remove or modify this file.
amAuthCert.properties	Defines the Certificate Authentication Module.
amAuthConfig.properties	Defines the Authentication Configuration Service.
amAuthContext.properties	Defines the localized error messages for the AuthContext Java class. For more information, see Authentication API For Java Applications.
amAuthContextLocal.properties	For Identity Server internal use only. Do not remove or modify this file.
amAuthHTTPBasic.properties	Defines the HTTP Basic Authentication Module.
amAuthLDAP.properties	Defines the LDAP Authentication Module.
amAuthMembership.properties	Defines the Membership Authentication Module.
amAuthNT.properties	Defines the NT Authentication Module.
amAuthRadius.properties	Defines the RADIUS Authentication Module.
amAuthSafeWord.properties	Defines the SafeWord Authentication Module.
amAuthSecurID.properties	Defines the SecurID Authentication Module.
amAuthUI.properties	Defines labels used in the authentication user inteface.
amAuthUnix.properties	Defines the UNIX® Authentication Module.

Customizing The Authentication User Interface

Many of the File Types Of The User Interface can be modified to bring a custom look and feel to the Authentication Service. The changes can be made on a number of levels, for example, to reflect authentication to different organizations via branding or to reflect different types of client applications. More specifically, one organization might customize their files to reflect their own logo and corporate colors while another might configure their service applications to have different authentication methods for security reasons. The JSP and properties files can be customized at the following levels:

- Top-level Organization—files can be customized for a user authenticating to the top-level organization.
- Organizations—files can be customized for a user authenticating to any sub-organization be it sub to the top-level organization or any specific sub-level organization.
- Locale—files can be customized to display a translated user interface at the organization level.
- Service—files can be customized for a user authenticating to a specific service.
- Client Type—files can be customized to support multiple clients (web browsers, wireless browsers, etc.).

The following sections contain the procedure for modifying JSP and properties files to create a customized Authentication Service user interface.

NOTE Anytime one of the File Types Of The User Interface are modified, the web application archive (WAR) `services.war` needs to be redeployed. The first step is to manually jar the services directory, then undeploy the old WAR and redeploy the new WAR based on the deployed web container. For information on how to do this, see Appendix C, "WAR Files."

To Create New Directories For Custom Console Files

All custom JSP and XML properties files should be stored in directories based on their level of customization. The JSP and XML properties files stored in *IdentityServer_base*`/SUNWam/web-apps/services/config/auth/default` define the Authentication Service user interface for the top-level organization configured during installation. Modifying these files would change the interface that, for example, `amadmin`, the top level administrator user, would see when logged in.

NOTE The directory that contains the Authentication Service user interface files for the top-level organization is configured during installation as `default`. This directory name can be changed to the actual name of the top-level organization for purposes of consistency with the Identity Server console.

The path to all directories containing customized user interface files begins from *IdentityServer_base*`/SUNWam/web-apps/services/config/auth/default/....` Assuming this, the generic directory path, depending on the level of customization, is:

`...`*organization_name* OR *organization_name_locale* / *any_sub_organization_directories* OR *any_sub_organization_directories_locale* / *any_client_type_directories* / *any_service_directories*

Table 3-8 lists some levels of customization with the corresponding path to the directory where the modified files would be found.

Table 3-8 Directory Paths Based On Customization Level

Level	Custom Directory Path Where Modified Files Live
Top-level Organization Customization	JSPs and XML files stored in `default`
Top-level Organization Customization Locale	JSPs and XML files stored in newly created `default`_*locale*
Sub-level Organization Customization	`...`*2nd_level_organization_name*1 OR *2nd_level_organization_name_locale*1`/`*sub_organization_name*2 OR *sub_organization_name_locale*2`/...`
Service Customization	`...`*2nd_level_organization_name*1 OR *2nd_level_organization_name_locale*1`/`*sub_organization_name*2 OR *sub_organization_name_locale*2`/`*service_name*
Client Type (HTML, WML, etc.) Customization	`...`*2nd_level_organization_name*1 OR *2nd_level_organization_name_locale*1`/`*sub_organization_name*2 OR *sub_organization_name_locale*2`/`*client_type*

Table 3-8 defines directory paths based on a simple configuration (customize an organization's branding or add WML files for multiple client types). Often, though, customization projects are not that straightforward. In the case that an organization wants to customize authentication for itself, a sub-organization and maybe a service, there is a specific path that these directories must follow. For example, let's assume the following:

- Top-level Organization: default
- Sub-organization: Sun

  Locales: en (English) and ja (Japanese)

  - Sub-organization: SunONE

    Locales: en (English) and ja (Japanese)

    Client Types: HTML for web browsers and WML for Nokia and Motorola cell phones

    Service: paycheck

NOTE The `default` directory in *IdentityServer_base*/`SUNWam/web-apps/services/config/auth/` will be left untouched.

As there are two locales to be configured for Sun, modified files will reside in the directory based on their respective locale:

1. *IdentityServer_base*/`SUNWam/web-apps/services/config/auth/default/Sun_en/...`
2. *IdentityServer_base*/`SUNWam/web-apps/services/config/auth/default/Sun_ja/...`

Each of the two defined localized directories will contain the following three sub-directories storing English and Japanese files for the client types and service:

`...sunONE/html/paycheck`

`...sunONE/wml/nokia/paycheck`

`...sunONE/wml/motorola/paycheck`

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

To Create A Custom Login Interface

All of the files in the *IdentityServer_base*/`SUNWam/web-apps/services/config/auth/default` directory need to be copied into the new directory(s) created in the prior section, “To Create New Directories For Custom Console Files.” Customizing these files pertains to the actual interface for the organization as opposed to customizing the directories which pertains to the custom files storage directory. The authentication module configuration files are XML files based on the `Auth_Module_Properties.dtd`; the syntax of this DTD should be followed when customizing these files. Modifying elements in these files customize the authentication interface. More information on modifying this file can be found in “Configuring The Authentication Module” on page 144. See Table 3-2 for a list of the available JSP templates.

Customizing The Default Login Page

`Login.jsp` and the authentication module configuration files contain certain elements that can be modified. Strong HTML skills and an understanding of web servers are a prerequisite for modifying these files.

NOTE Although the JSP contain embedded JATO tags, it is not necessary to understand the Sun ONE Application Framework in order to customize them. For those who might be interested, though, an overview of the Application Framework can be found at http://docs.sun.com/source/817-0447-10/s1afovew.html.

They are located in *IdentityServer_base*/`SUNWam/web-apps/services/config/auth/default/`. `Login.jsp` is the common login page for all authentication modules, except Membership (`membership.jsp`) and self-registration (`register.jsp`), which dynamically displays the required user interface elements from the invoked authentication module's credentials file at run time. For example, if the authentication method is LDAP, the appropriate module header, user name field and password field will be displayed.

NOTE The processing logic is defined in the JATO ViewBean, `com.sun.identity.authentication.UI.LoginViewBean`.

styles.css The module header text and prompts used in the Authentication Service User Interface are formatted based on styles defined in `styles.css` (copied in Code Example 3-1). `loginText` defines the font formatting for `Login.jsp`. The style of these text fields can be changed by modifying `styles.css`.

Code Example 3-1 styles.css Style Sheet

```
a:hover {  text-decoration: underline}
footerText {  font-family: Helvetica, Arial, Geneva, sans-serif; font-size:
9pt;
color: #333333}
loginText {  font-family: Helvetica, Arial, Geneva, sans-serif}
mastheadLinks {  font-family: Helvetica, Arial, Geneva, sans-serif;
font-size: 9pt;
color: #4D59AB; text-decoration: none}
mastheadUsername {  font-family: Helvetica, Arial, Geneva, sans-serif;
font-size:
10pt; color: #333333; font-weight: bold}
input.buttonblue{  cursor: hand; font-family: verdana; background: #594fbf;
color:
```

Code Example 3-1 styles.css Style Sheet *(Continued)*

```
#ffffff; font-weight: bold; font-size: 10pt; padding: 1px 1px; margin: 0px
0px;
border: 0px}
doubleArrow {  font-family: Arial, Helvetica, sans-serif; font-size: 10pt;
font-weight: bold; color: #594FBF}
mastheadSeparators { font-family: Helvetica, Arial, Geneva, sans-serif;
color:
#A2A2A2; text-decoration: none; font-size: 12pt}
```

Module Header Text Each authentication module contains header text that displays the name of the module on the interface. In Figure 3-10 on page 73, the module header text reads *This server uses Unix Authentication*. This field is defined as the `StaticTextHeader` in `Login.jsp`.

Code Example 3-2 Module Header Text Definition in Login.jsp

```
<!-- display authentication scheme -->
    <tr>
    <td colspan="2" width="140">&nbsp;</td>
    <td>
    <jato:content name="ContentStaticTextHeader">
        <jato:getDisplayFieldValue name='StaticTextHeader'
    defaultValue='Authentication' fireDisplayEvents='true'
    escape='false'/>
    </jato:content>
    </td>
    </tr>
```

The JATO ViewBean picks up the value for `StaticTextHeader` from the Authentication Module Configuration Files. If there is no value defined in this file, the `defaultValue` defined in `Login.jsp` (*Authentication*) is picked up.

Code Example 3-3 is the authentication module configuration file for Unix Authentication, `Unix.xml`. The value of `header`, defined in the Callbacks Element, is the module header text picked up by the JATO ViewBean. This field can be customized per module.

Code Example 3-3 Unix.xml Authentication Module Configuration File

```
<!DOCTYPE ModuleProperties PUBLIC "=//iPlanet//Authentication Module
Properties XML
Interface 1.0 DTD//EN"
"jar://com/sun/identity/authentication/Auth_Module_Properties.dtd">

<ModuleProperties moduleName="Unix" version="1.0" >
    <Callbacks length="2" order="1" timeout="60" header="This
server uses Unix Authentication" >
        <NameCallback>
            <Prompt> User Name: </Prompt>
        </NameCallback>
        <PasswordCallback echoPassword="false" >
            <Prompt> Password: </Prompt>
        </PasswordCallback>
    </Callbacks>
    <Callbacks length="0" order="2" timeout="120" header=" Your password has
expired.
Please contact service desk to reset your password" error="true" />
</ModuleProperties>
```

Name Prompt and Input Field Each authentication module contains text to display next to the first credential input field, generally a prompt for the user's name. In the Unix authentication interface (Figure 3-10 on page 73), the name prompt text reads *User Name*. The name prompt field is defined by the `txtPrompt` in the first table data (`td`) tag (under `<!---- text box display ----->`) in `Login.jsp` as detailed in Code Example 3-4. The input field itself is defined by the second table data (`td`) tag.

Code Example 3-4 Name Prompt And Field Definition in Login.jsp

```
<jato:content name="textBox">
    <!---- text box display ----->
    <tr>
    <form name="frm<jato:text name="txtIndex" />" action="blank"
        onSubmit="defaultSubmit(); return false;">

    <td class="loginText" width="120">
           <label for="IDToken<jato:text name="txtIndex" />">
                <jato:text name="txtPrompt" defaultValue="User name:"
escape="false" />
                <jato:content name="isRequired">
                    <font color="#0000ff">*</font>
                </jato:content>
            </label>
    </td>
```

Code Example 3-4 Name Prompt And Field Definition in Login.jsp *(Continued)*

```
    <td class="loginText" width="20">&nbsp;</td>
    <td class="loginText">
        <input type="text" name="IDToken<jato:text name="txtIndex" />"
                id="IDToken<jato:text name="txtIndex" />"
    value="" size="20">
    </td>
    </form>
    </tr>

    <tr><td colspan="3">&nbsp;</td></tr>
    <!---- end of textBox ---->
    </jato:content>
```

The JATO ViewBean picks up the actual value for `txtPrompt`, not from `Login.jsp` but, from the Authentication Module Configuration Files. If there is no value defined in this file, the `defaultValue` defined in `Login.jsp` (*User name:*) is picked up.

Code Example 3-3 is the authentication module configuration file for Unix Authentication, `Unix.xml`. The value of `prompt`, defined in the NameCallback Element, is the text picked up by the JATO ViewBean to define the input field. In this case, the name prompts are defined similarly: *User name:* in `Login.jsp` and *User Name:* in `Unix.xml`. This field can be customized per module.

Password Prompt and Input Field Each authentication module contains text to display next to the second credential field, generally a prompt for the user's password. In the Unix authentication interface (Figure 3-10 on page 73), the password prompt text reads *Password*. The password prompt field is defined by the `txtPrompt` in the first table data (`td`) tag (under `<!---- password display ----->`) in `Login.jsp` as detailed in Code Example 3-5. The input field itself is defined by the second table data (`td`) tag.

Code Example 3-5 Password Prompt And Field Definition in Login.jsp

```
<jato:content name="password">
    <!---- password display ---->
    <tr>
    <form name="frm<jato:text name="txtIndex" />" action="blank"
        onSubmit="defaultSubmit(); return false;">

    <td class="loginText" width="120">
            <label for="IDToken<jato:text name="txtIndex" />">
                <jato:text name="txtPrompt" defaultValue="Password:"
escape="false" />
```

Code Example 3-5 Password Prompt And Field Definition in Login.jsp *(Continued)*

```
            <jato:content name="isRequired">
                <font color="#0000ff">*</font>
            </jato:content>
        </label>
</td>

<td class="loginText" width="20">&nbsp;</td>
<td class="loginText">
    <input type="password" name="IDToken<jato:text name="txtIndex" />"
            id="IDToken<jato:text name="txtIndex" />"
value="" size="20">
</td>
</form>
</tr>

<tr><td colspan="3">&nbsp;</td></tr>
<!---- end of password ---->
</jato:content>
```

The JATO ViewBean picks up the value for `txtPrompt` from the Authentication Module Configuration Files. If there is no value defined in this file, the `defaultValue` defined in `Login.jsp` (*Password:*) is picked up.

Code Example 3-3 is the authentication module configuration file for Unix Authentication, `Unix.xml`. The value of `prompt`, defined in the PasswordCallback Element, is the prompt text picked up by the JATO ViewBean. In this case, the name prompts are the same: *Password:* in `Login.jsp` and *Password:* in `Unix.xml`. This field can be customized per module.

Choice Prompt and Value Fields Figure 3-11 on page 116 pictures an interface where multiple authentication modules are displayed and the user is prompted to choose one. There are a number of reasons that choices are displayed; for example, the Membership authentication module defines choices when a user is self-registering with a user name that already exists. They are then offered a sample list of other user names from which to choose or given the option to create a new one. The choice prompt field for Membership authentication is defined by the `txtPrompt` in the first table data (`td`) tag (under `<!---- choice value display ----->`) in `membership.jsp` as detailed in Code Example 3-6. The input field type is defined by the second table data (`td`) tag as radio buttons. This option can be changed to a check box by switching the input type value of “`radio`” (in both the selectedChoice and unselectedChoice tags) with the value “`checkbox`”.

Code Example 3-6 Choice Prompt And Value Fields Definition in membership.jsp

```
        <!-- choice value display -->
        <tr>
        <form name="frm<jato:text name="txtIndex" />" action="blank"
            onSubmit="defaultSubmit(); return false;">

        <td class="loginText">
            <label for="IDToken<jato:text name="txtIndex" />">
                <jato:text name="txtPrompt" defaultValue="RadioButton:"
escape="false" />
                <jato:content name="isRequired">
                    <font color ="#0000ff">*</font>
                </jato:content>
            </label>
        </td>

        <td class="loginText" width="20">&nbsp;</td>
        <td class="loginText">

            <jato:tiledView name="tiledChoices"
type="com.sun.identity.authentication.UI.CallBackChoiceTiledView">

            <jato:content name="selectedChoice">
                <input type="radio"
                    name="IDToken<jato:text name="txtParentIndex" />"
                    id="IDToken<jato:text name="txtParentIndex" />"
                    value="<jato:text name="txtIndex" />"
                    checked><jato:text name="txtChoice" /><br>
            </jato:content>

            <jato:content name="unselectedChoice">
                <input type="radio"
                    name="IDToken<jato:text name="txtParentIndex" />"
                    id="IDToken<jato:text name="txtParentIndex" />"
                    value="<jato:text name="txtIndex" />"
                    ><jato:text name="txtChoice" /><br>
            </jato:content>

            </jato:tiledView>
        </td>
        </form>
        </tr>
        <tr><td colspan="3">&nbsp;</td></tr>
        <!-- end of choice -->
```

The JATO ViewBean picks up the value for `txtPrompt` from the Authentication Module Configuration Files. If there is no value defined in this file, the `defaultValue` defined in `membership.jsp` (*RadioButton:*) is picked up.

CAUTION If changing the radio buttons to checkboxes, remember to change the value of `txtPrompt` in `membership.jsp` also.

Code Example 3-7 is an extract of the authentication module configuration file for Membership Authentication, `Membership.xml`. The value of `prompt`, defined in the ChoiceCallback Element, is the prompt text picked up by the JATO ViewBean. In this case, the name prompts are different: *RadioButton:* in `membership.jsp` and *A user already exists with the user name you entered. Please choose one of the following user names, or create your own:* in `Membership.xml`. This field can be customized per module.

Code Example 3-7 Membership.xml Configuration File Extract

```
...
    <Callbacks length="2" order="17" timeout="120" header="Self 
Registration" >
        <ChoiceCallback attribute="uid" >
            <Prompt>A user already exists with the user name you entered. 
&lt;BR&gt;Please choose one of the following user names, or create your 
own:</Prompt>
            <ChoiceValues>
              <ChoiceValue>
                  <Value>Create My Own</Value>
              </ChoiceValue>
            </ChoiceValues>
        </ChoiceCallback>
        <ConfirmationCallback>
            <OptionValues>
                <OptionValue>
                    <Value> Submit </Value>
                </OptionValue>
            </OptionValues>
        </ConfirmationCallback>
    </Callbacks>
   ...
```

Logos And Branding Images Custom images can be used for corporate logos and branding. Any new GIF or JPG images must be placed in *IdentityServer_base*`/SUNWam/web-apps/services/login_images`. To access this new image, edit the appropriate section of the authentication module configuration file. There are a number of places where this can be changed. Code Example 3-8 contains the image source attributes from `Membership.xml`, the Membership authentication module configuration file. In this code, `logo_sun.gif` and

`Identity_LogIn.gif` would be replaced with a custom logo and title, respectively. The `spacer.gif` tags can be deleted or modified as needed. The default Sun Microsystems logo in the upper left corner and the default *Sun*™ ONE *Identity Server* title across the top of the interface are pictured in Figure 3-4 on page 69.

Code Example 3-8 Image Source Attributes in Membership.xml Extract

```
...
<!-- branding -->
    <tr>
    <td width="110"><img src="<%= ServiceURI %>/login_images/logo_sun.gif"
width="110"
        height="82" alt="Sun Microsystems Logo"></td>
    <td><img src="<%= ServiceURI %>/login_images/spacer.gif" width="9"
height="1"
        alt=""></td>
    <td valign="bottom" bgcolor="#ACACAC" width="100%"><img
        src="<%= ServiceURI %>/login_images/Identity_LogIn.gif" width="300"
        height="30" alt="Sun ONE Identity Server"></td>
    </tr>
    <tr>
    <td colspan="3"><img src="<%= ServiceURI %>/login_images/spacer.gif"
width="1"
        height="39" alt=""></td>
    </tr>
...
```

The final image attribute is in `Login.jsp`. Code Example 3-9 can be moved around the JSP and the image will be displayed wherever in the code it is placed.

Code Example 3-9 Image Source Attribute in Login.jsp

```
<jato:content name="ContentImage">
<!-- customized image defined in properties file -->
<p><img name="IDImage"
src="<jato:getDisplayFieldValue name='Image'/>" alt=""></p>
</jato:content>
```

Code Example 3-9 refers to an image file defined in the authentication module configuration file. The XML code for this image would be written as:

`<Callbacks length="3" order="1" timeout="120" header="Sample Module" image="MyCustomImage.gif">`.

Password Reset Link It is possible put add a link on the LDAP Authentication module login page to the Password Reset Service. This allows a user to reset their password prior to authentication. The following line can be placed below the Submit Button Code in Code Example 3-10.

<a
“href=<console_proto>://*console_host*.*console_domain*:*console_port*/*password_reset_URI* >Forgot your password?</a>

Code Example 3-10 Submit Button Code From Login.jsp

```
<jato:content name="ContentButtonLogin">
    <!-- Submit button -->

    <jato:content name="hasButton">

    <script language="javascript">
        defaultBtn = '<jato:text name="defaultBtn" />';
    </script>

    <tr>
    <td colspan="2">&nbsp;</td>
    <td align="right">
        <table border="0" cellpadding="2" cellspacing="0">
        <tr>

        <jato:tiledView name="tiledButtons"
    type="com.sun.identity.authentication.UI.ButtonTiledView">
        <td>
    <script language="javascript">
        markupButton(
         '<jato:text name="txtButton" />',
         "javascript:LoginSubmit('<jato:text name="txtButton" />')");
    </script>
        </td>
        </jato:tiledView>
        </tr>
        </table>
    </td>
    </tr>

    <!---- end of hasButton ---->
    </jato:content>


    <jato:content name="hasNoButton">
    <tr>
    <td colspan="2">&nbsp;</td>
    <td align="right">
        <script language="javascript">
    markupButton(
        '<jato:text name="lblSubmit" />',
         "javascript:LoginSubmit('<jato:text name="lblSubmit" />')");
        </script>
```

Code Example 3-10 Submit Button Code From Login.jsp *(Continued)*

```
    </td>
    </tr>
    <!---- end of hasNoButton ---->
    </jato:content>

    <!---- end of ContentButtonLogin ---->
    </jato:content>
```

Log In Button The following modifications can be made to the default Log In button displayed on each authentication interface screen. The button color and/or background can be customized in any of the following stylesheets located in *IdentityServer_base*`/SUNWam/web-apps/services/css`.

- `css_ie5win.css` defines styles for Netscape 6 and Internet Explorer 5.
- `css_ns4sol.css` defines styles for Netscape 4 on Solaris.
- `css_ns4win.css` defines styles for Netscape 4 on Windows.
- `css_generic.css` defines styles for all other browsers.

Code Example 3-11 is extracted from `css_ns4sol.css`. To change the background color, in the appropriate style sheet, change the color values for `.button-content-enabled { background-color: #CCC; }` and for `a.button-link:link, a.button-link:visited { color: #000; background-color: #CCC; text-decoration: none; }`.

Code Example 3-11 css_ns4sol.css Extraction

```
/* BUTTONS */

/* Regular Button - Enabled */
.button-frame-enabled { background-color: #000; }
.button-content-enabled { background-color: #CCC; }
.button-link-enabled-text { color: #000; margin: 4px 0px; font-weight: bold;
}
a.button-link:link, a.button-link:visited { color: #000; background-color:
#CCC;
text-decoration: none; }
a.button-link:active { color: #000; background-color: #999; text-decoration:
none;}

/* Regular Button - Disabled */
.button-frame-disabled { background-color: #999; }
```

Code Example 3-11 css_ns4sol.css Extraction *(Continued)*

```
.button-content-disabled { background-color: #CCC; }
.button-link-disabled-text { color: #999; margin: 4px 0px; font-weight:
bold; }
```

By default, the login button reads `Log In`. To change this text on a global level, the `amAuthUI.properties` file, the authentication service's console localization properties file, would be modified.

NOTE `amAuthUI.properties` is defined in "Configuring Authentication Localization Properties" on page 151. More information can be found in "Configuring Console Localization Properties" on page 251 of Chapter 6, "Service Management," in this manual.

Code Example 3-12 is an extraction from `amAuthUI.properties`. Notice the default value is `LogIn=Log In`. For an example, the value `Log In` can be changed to `Submit`.

Code Example 3-12 amAuthUI.properties Extract

```
...
Submit=Submit
LogIn=Log In
NewUser=New User
Reset=Reset Form
Cancel=Cancel
Agree=Agree
Disagree=Disagree
Yes=Yes
No=No
Continue=Continue
...
```

The button text can also be changed per module. If each authentication module needs to display different text, each authentication module configuration file needs to be modified with a ConfirmationCallback Element. The value for the Callbacks length can be increased, if necessary. Code Example 3-17 on page 146 is the authentication module configuration file for Membership, `Membership.xml`. It contains an example of the Confirmation Callback.

Authentication Methods

The Authentication Service provides different ways in which authentication can be applied. These different authentication methods can be accessed by specifying Login URL Parameters, or through the Authentication Programming Interfaces. Access using parameters and a URL is discussed in "The User Interface Login URL" on page 74. Access via the API is discussed in "Authentication Programming Interfaces" on page 153.

NOTE Specific information on how to assign an authentication method using the Identity Server console can be found in the *Sun ONE Identity Server Administration Guide*.

The following sections detail the different authentication methods with configured login URLs and redirection URL order of precedence for each. The authentication methods are:

- Organization-based Authentication
- Role-based Authentication
- Service-based Authentication
- User-based Authentication
- Authentication Level-based Authentication
- Module-based Authentication

For each of these methods, the user can either pass or fail the authentication. Once the determination has been made, each method follows this procedure. Step 1 through Step 3 follows a successful authentication; Step 4 follows both successful and failed authentication.

1. Identity Server confirms whether the authenticated user(s) is defined in the Directory Server data store and whether the profile is active.

   The User Profile attribute in the Core Authentication Service can be defined as `Required`, `Dynamically Configured` or `Ignored`. Following a succesful authentication, Identity Server confirms whether the authenticated user(s) is defined in the Directory Server data store and, if the User Profile value is `Required`, confirms that the profile is active. (This is the default case.) If the User Profile is `Dynamically Configured`, the Authentication Service will create the user profile in the Directory Server data store. If the User Profile is set to `Ignore`, the user validation will not be done.

2. Execution of the Authentication Post Processing SPI is accomplished.

   The Core Authentication Service contains an Authentication PostProcessing Class attribute which may contain the authentication post-processing class name as its value. `AMPostAuthProcessInterface` is the post-processing interface. It can be executed on either successful or failed authentication or on logout. More information on this interface can be found in “Implementing Authentication Post Processing” on page 182.

3. The following properties are added to, or updated in, the session token and the user’s session is activated.

   - `Organization`—This is the DN of the organization to which the user belongs.
   - `Principal`—This is the DN of the user.
   - `Principals`—This is a list of names to which the user has authenticated. (This property may have more then one value defined as a pipe separated list.)
   - `UserId`—This is the user’s DN as returned by the module, or in the case of modules other than LDAP or Membership, the user name. (All `Principals` must map to the same user. The UserID is the user DN to which they map.)
   - `UserToken`—This is a user name. (All `Principals` must map to the same user. The UserToken is the user name to which they map.)
   - `Host`—This is the host name or IP address for the client.
   - `authLevel`—This is the highest level to which the user has authenticated.
   - `AuthType`—This is a pipe separated list of authentication modules to which the user has authenticated (i.e.: `module1|module2|module3`).
   - `clientType`—This is the device type of the client browser.
   - `Locale`—This is the locale of the client.
   - `CharSet`—This is the determined character set for the client.
   - `Role`—Applicable for role-based authentication only, this is the role to which the user belongs.
   - `Service`—Applicable for service-based authentication only, this is the service to which the user belongs.
   - `loginURL`—This is the client’s login URL.

4. Identity Server looks for information on where to redirect the user after either a successful or failed authentication.

   URL redirection can be to either an Identity Server page or a URL. The redirection is based on an order of precedence in which Identity Server looks for redirection based on the authentication method and whether the authentication has been successful or has failed. This order is detailed in the URL redirection portions of the following authentication methods sections.

TIP One more authentication method is a functionality of the Policy Service. "Policy-Based Resource Management" on page 311 of Chapter 7, "Policy Service" in this manual contains information on resource-based authentication.

Organization-based Authentication

This method of authentication allows a user to authenticate to an organization or sub-organization. It is the default method of authentication for Identity Server. The authentication method for an organization is set by registering the Core Authentication Service to the organization and defining the Organization Authentication Configuration attribute. More information on how this is done can be found in the Authentication Options chapter of the *Sun ONE Identity Server Administration Guide*.

Organization-based Authentication Login URLs

The organization for authentication can be specified in the The User Interface Login URL by defining the org Parameter or the domain Parameter. The organization of a request for authentication is determined from the following, in order of precedence:

1. The `domain` parameter.
2. The `org` parameter.
3. The value of the `DNS Alias Names` (Organization alias names) attribute in the Administration Service.

After calling the correct organization, the authentication module(s) to which the user will authenticate are retrieved from the Organization Authentication Configuration attribute in the Core Authentication Service. The login URLs used to specify and initiate organization-based authentication are:

- `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login`
- `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?domain=`*domain_name*

- `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?org=`*org_name*

If there is no defined parameter, the organization will be determined from the server host and domain specified in the login URL. More information on these parameters can be found in "Login URL Parameters" on page 74.

Organization-based Authentication Redirection URLs

Upon a successful or failed organization-based authentication, Identity Server looks for information on where to redirect the user. Following is the order of precedence in which the application will look for this information.

Successful Organization-based Authentication Redirection URLs

The redirection URL for successful organization-based authentication is detemined by checking the following places in order of precedence:

1. A URL set by the authentication module.
2. A URL set by a `goto` Login URL parameter.
3. A URL set in the `clientType` custom files for the `iplanet-am-user-success-url` attribute of the user's profile (`amUser.xml`).
4. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute of the user's role entry.
5. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute of the user's organization entry.
6. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute as a global default.
7. A URL set in the `iplanet-am-user-success-url` attribute of the user's profile (`amUser.xml`).
8. A URL set in the `iplanet-am-auth-login-success-url` attribute of the user's role entry.
9. A URL set in the `iplanet-am-auth-login-success-url` attribute of the user's organization entry.
10. A URL set in the `iplanet-am-auth-login-success-url` attribute as a global default.

Failed Organization-based Authentication Redirection URLs

The redirection URL for failed organization-based authentication is detemined by checking the following places in the following order:

1. A URL set by the authentication module.
2. A URL set by a `gotoOnFail` Login URL parameter.
3. A URL set in the `clientType` custom files for the `iplanet-am-user-failure-url` attribute of the user’s entry (`amUser.xml`).
4. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute of the user’s role entry.
5. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute of the user’s organization entry.
6. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute as a global default.
7. A URL set for the `iplanet-am-user-failure-url` attribute in the user’s entry (`amUser.xml`).
8. A URL set for the `iplanet-am-auth-login-failure-url` attribute of the user’s role entry.
9. A URL set for the `iplanet-am-auth-login-failure-url` attribute of the user’s organization entry.
10. A URL set for the `iplanet-am-auth-login-failure-url` attribute as the global default.

Role-based Authentication

This method of authentication allows a user to authenticate to a role (either static or filtered) within an organization or sub-organization.

NOTE The Authentication Configuration Service must first be registered to the organization before it can be registered as an instance to the role. More information on how this is done can be found in the Authentication Configuration section of Chapter 7 in the *Sun ONE Identity Server Administration Guide*.

For authentication to be successful, the user must belong to the role and they must authenticate to each module defined in the Authentication Configuration Service instance configured for that role. For each instance of role-based authentication, the following attributes can be specified:

- Conflict Resolution Level—sets a priority level for the Authentication Configuration Service instance defined for different roles that both may contain the same user. For example, if User1 is assigned to both Role1 and Role2, a higher conflict resolution level can be set for Role1 so when the user attempts authentication Role1 will have the higher priority for success or failure redirects and post-authentication processes.
- Authentication Configuration—defines the authentication modules configured for the role's authentication process.
- Login Success URL—defines the URL to which a user is redirected on successful authentication.
- Login Failed URL—defines the URL to which a user is redirected on failed authentication.
- Authentication Post Processing Classes—defines the post-authentication interface. More information can be found in "Implementing Authentication Post Processing" on page 182.

Role-based Authentication Login URLs

Role-based authentication can be specified in the The User Interface Login URL by defining a role Parameter. After calling the correct role, the authentication module(s) to which the user will authenticate are retrieved from the Authentication Configuration Service instance defined for the role.

The login URLs used to specify and initiate this role-based authentication are:

- `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?role=`*role_name*
- `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?org=`*org_name*`&role=`*role_name*

If there is no configured org Parameter, the organization to which the role belongs will be determined from the server host and domain specified in the login URL itself. More information on these parameters can be found in "Login URL Parameters" on page 74.

Role-based Authentication Redirection URLs

Upon a successful or failed role-based authentication, Identity Server looks for information on where to redirect the user. Following is the order of precedence in which the application will look for this information.

Successful Role-based Authentication Redirection URLs

The redirection URL for successful role-based authentication is detemined by checking the following places in the following order:

1. A URL set by the authentication module.
2. A URL set by a `goto` Login URL parameter.
3. A URL set in the `clientType` custom files for the `iplanet-am-user-success-url` attribute of the user's profile (`amUser.xml`).
4. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute of the role to which the user has authenticated.
5. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute of another role entry of the authenticated user. (This option is a fallback if the previous redirection URL fails.)
6. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute of the user's organization entry.
7. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute as a global default.
8. A URL set in the `iplanet-am-user-success-url` attribute of the user's profile (`amUser.xml`).
9. A URL set in the `iplanet-am-auth-login-success-url` attribute of the role to which the user has authenticated.
10. A URL set in the `iplanet-am-auth-login-success-url` attribute of another role entry of the authenticated user. (This option is a fallback if the previous redirection URL fails.)
11. A URL set in the `iplanet-am-auth-login-success-url` attribute of the user's organization entry.
12. A URL set in the `iplanet-am-auth-login-success-url` attribute as a global default.

Failed Role-based Authentication Redirection URLs

The redirection URL for failed role-based authentication is detemined by checking the following places in the following order:

1. A URL set by the authentication module.

2. A URL set by a `goto` Login URL parameter.
3. A URL set in the `clientType` custom files for the `iplanet-am-user-failure-url` attribute of the user's profile (`amUser.xml`).
4. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute of the role to which the user has authenticated.
5. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute of another role entry of the authenticated user. (This option is a fallback if the previous redirection URL fails.)
6. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute of the user's organization entry.
7. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute as a global default.
8. A URL set in the `iplanet-am-user-failure-url` attribute of the user's profile (`amUser.xml`).
9. A URL set in the `iplanet-am-auth-login-failure-url` attribute of the role to which the user has authenticated.
10. A URL set in the `iplanet-am-auth-login-failure-url` attribute of another role entry of the authenticated user. (This option is a fallback if the previous redirection URL fails.)
11. A URL set in the `iplanet-am-auth-login-failure-url` attribute of the user's organization entry.
12. A URL set in the `iplanet-am-auth-login-failure-url` attribute as a global default.

Service-based Authentication

This method of authentication allows a user to authenticate to a specific service or application registered to an organization or sub-organization. The service is configured as a Service Instance within the Authentication Configuration Service and is associated with an Instance Name. For authentication to be successful, the user must authenticate to each module defined in the Authentication Configuration Service instance configured for the service. For each instance of service-based authentication, the following attributes can be specified:

- Authentication Configuration—defines the authentication modules configured for the service's authentication process.
- Login Success URL—defines the URL to which a user is redirected on successful authentication.
- Login Failed URL—defines the URL to which a user is redirected on failed authentication.
- Authentication Post Processing Classes—defines the post-authentication interface. More information can be found in "Implementing Authentication Post Processing" on page 182.

Service-based Authentication Login URLs

Service-based authentication can be specified in the The User Interface Login URL by defining a service Parameter. After calling the service, the authentication module(s) to which the user will authenticate are retrieved from the Authentication Configuration Service instance defined for the service.

The login URLs used to specify and initiate this service-based authentication are:

- `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?service=`*service_name*
- `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?org=`*org_name*`&service=`*service_name*

If there is no configured `org` parameter, the organization will be determined from the server host and domain specified in the login URL itself. More information on these parameters can be found in "Login URL Parameters" on page 74.

Service-based Authentication Redirection URLs

Upon a successful or failed service-based authentication, Identity Server looks for information on where to redirect the user. Following is the order of precedence in which the application will look for this information.

Successful Service-based Authentication Redirection URLs

The redirection URL for successful service-based authentication is detemined by checking the following places in the following order:

1. A URL set by the authentication module.
2. A URL set by a `goto` Login URL parameter.
3. A URL set in the `clientType` custom files for the `iplanet-am-user-success-url` attribute of the user's profile (`amUser.xml`).

4. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute of the service to which the user has authenticated.
5. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute of the user's role entry.
6. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute of the user's organization entry.
7. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute as a global default.
8. A URL set in the `iplanet-am-user-success-url` attribute of the user's profile (`amUser.xml`).
9. A URL set in the `iplanet-am-auth-login-success-url` attribute of the service to which the user has authenticated.
10. A URL set in the `iplanet-am-auth-login-success-url` attribute of the user's role entry.
11. A URL set in the `iplanet-am-auth-login-success-url` attribute of the user's organization entry.
12. A URL set in the `iplanet-am-auth-login-success-url` attribute as a global default.

Failed Service-based Authentication Redirection URLs

The redirection URL for failed service-based authentication is detemined by checking the following places in the following order:

1. A URL set by the authentication module.
2. A URL set by a `goto` Login URL parameter.
3. A URL set in the `clientType` custom files for the `iplanet-am-user-failure-url` attribute of the user's profile (`amUser.xml`).
4. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute of the service to which the user has authenticated.
5. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute of the user's role entry.

6. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute of the user's organization entry.
7. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute as a global default.
8. A URL set in the `iplanet-am-user-failure-url` attribute of the user's profile (`amUser.xml`).
9. A URL set in the `iplanet-am-auth-login-failure-url` attribute of the service to which the user has authenticated.
10. A URL set in the `iplanet-am-auth-login-failure-url` attribute of the user's role entry.
11. A URL set in the `iplanet-am-auth-login-failure-url` attribute of the user's organization entry.
12. A URL set in the `iplanet-am-auth-login-failure-url` attribute as a global default.

User-based Authentication

This method of authentication allows a user to authenticate to an authentication process configured specifically for them. The process is configured as a value of the User Authentication Configuration attribute in the user's profile. For authentication to be successful, the user must authenticate to each module defined.

User-based Authentication Login URLs

User-based authentication can be specified in the The User Interface Login URL by defining a user Parameter. After calling the correct user, the authentication module(s) to which the user will authenticate are retrieved from the User Authentication Configuration instance defined for them.

The login URLs used to specify and initiate this role-based authentication are:

- `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?user=`*user_name*
- `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?org=`*org_name*`&user=`*user_name*

If there is no configured org Parameter, the organization to which the role belongs will be determined from the server host and domain specified in the login URL itself. More information on these parameters can be found in "Login URL Parameters" on page 74.

User Alias List Attribute

On receiving a request for user-based authentication, the Authentication Service first verifies that the user is a valid user and then retrieves the Authentication Configuration data for them. In the case where there is more then one valid user profile associated with the value of the user Login URL parameter, all profiles must map to the specified user. The User Alias Attribute (`iplanet-am-user-alias-list`) in the User profile is where other profiles belonging to the user can be defined. If mapping fails, the user is denied a valid session. The exception would be if one of the users is a Super Admin whereby the user mapping validation is not done and the user is given Super Admin rights.

User-based Authentication Redirection URLs

Upon a successful or failed user-based authentication, Identity Server looks for information on where to redirect the user. Following is the order of precedence in which the application will look for this information.

Successful User-based Authentication Redirection URLs

The redirection URL for successful user-based authentication is detemined by checking the following places in order of precedence:

1. A URL set by the authentication module.
2. A URL set by a `goto` Login URL parameter.
3. A URL set in the `clientType` custom files for the `iplanet-am-user-success-url` attribute of the user’s profile (`amUser.xml`).
4. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute of the user’s role entry.
5. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute of the user’s organization entry.
6. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute as a global default.
7. A URL set in the `iplanet-am-user-success-url` attribute of the user’s profile (`amUser.xml`).
8. A URL set in the `iplanet-am-auth-login-success-url` attribute of the user’s role entry.
9. A URL set in the `iplanet-am-auth-login-success-url` attribute of the user’s organization entry.

10. A URL set in the `iplanet-am-auth-login-success-url` attribute as a global default.

Failed User-based Authentication Redirection URLs

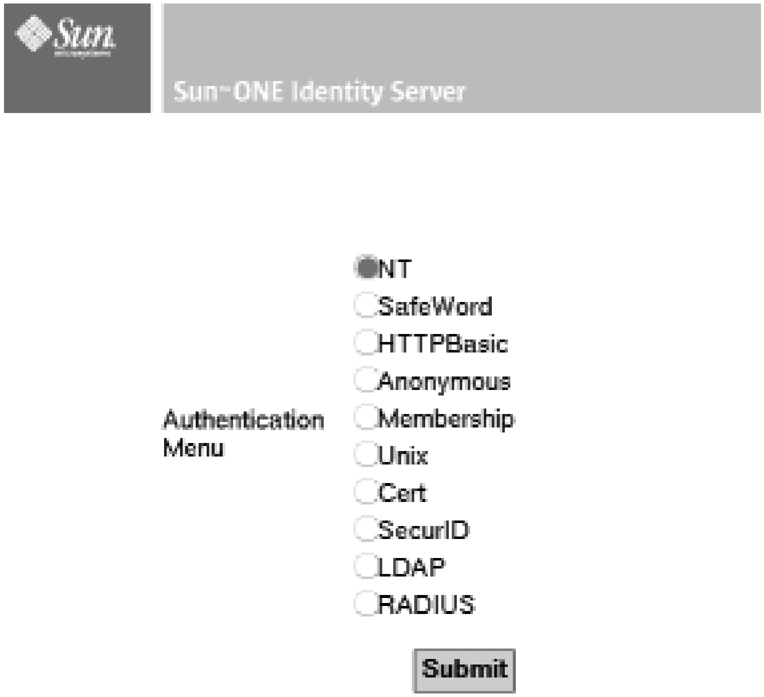
The redirection URL for failed user-based authentication is detemined by checking the following places in the following order:

1. A URL set by the authentication module.
2. A URL set by a `gotoOnFail` Login URL parameter.
3. A URL set in the `clientType` custom files for the `iplanet-am-user-failure-url` attribute of the user's entry (`amUser.xml`).
4. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute of the user's role entry.
5. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute of the user's organization entry.
6. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute as a global default.
7. A URL set for the `iplanet-am-user-failure-url` attribute in the user's entry (`amUser.xml`).
8. A URL set for the `iplanet-am-auth-login-failure-url` attribute of the user's role entry.
9. A URL set for the `iplanet-am-auth-login-failure-url` attribute of the user's organization entry.
10. A URL set for the `iplanet-am-auth-login-failure-url` attribute as the global default.

Authentication Level-based Authentication

This method of authentication allows an administrator to specify the security level of the modules to which identities can authenticate. Each authentication module has a separate Authentication Level attribute and the value of this attribute can be defined as any valid integer. With Authentication Level-based authentication, the Authentication Service displays a module login page with a menu containing the authentication modules that have authentication levels equal to or greater then the value specified in the Login URL parameter. Users can select a module from the presented list. Figure 3-11 illustrates this list of modules based on authentication level. Once the user selects a module, the remaining process is based on Module-based Authentication as discussed on page 118.

Figure 3-11 Authentication Level-based Authentication Login Screen



Authentication Level-based Authentication Login URLs

Authentication level-based authentication can be specified in The User Interface Login URL by defining a authlevel Parameter. After calling the login screen with the relevant list of modules, the user must choose one with which to authenticate. The login URLs used to specify and initiate authentication level-based authentication are:

- `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?authlevel=`*authentication_level*
- `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?org=`*org_name*`&authlevel=`*authentication_level*

If there is no configured `org` parameter, the organization to which the user belongs will be determined from the server host and domain specified in the login URL itself. More information on these parameters can be found in “Login URL Parameters” on page 74.

Authentication Level-based Authentication Redirection URLs

Upon a successful or failed authentication level-based authentication, Identity Server looks for information on where to redirect the user. Following is the order of precedence in which the application will look for this information.

Successful Authentication Level-based Authentication Redirection URLs

The redirection URL for successful authentication level-based authentication is detemined by checking the following places in order of precedence:

1. A URL set by the authentication module.
2. A URL set by a `goto` Login URL parameter.
3. A URL set in the `clientType` custom files for the `iplanet-am-user-success-url` attribute of the user’s profile (`amUser.xml`).
4. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute of the user’s role entry.
5. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute of the user’s organization entry.
6. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute as a global default.
7. A URL set in the `iplanet-am-user-success-url` attribute of the user’s profile (`amUser.xml`).
8. A URL set in the `iplanet-am-auth-login-success-url` attribute of the user’s role entry.
9. A URL set in the `iplanet-am-auth-login-success-url` attribute of the user’s organization entry.
10. A URL set in the `iplanet-am-auth-login-success-url` attribute as a global default.

Failed Authentication Level-based Authentication Redirection URLs

The redirection URL for failed authentication level-based authentication is detemined by checking the following places in the following order:

1. A URL set by the authentication module.
2. A URL set by a `gotoOnFail` Login URL parameter.
3. A URL set in the `clientType` custom files for the `iplanet-am-user-failure-url` attribute of the user's entry (`amUser.xml`).
4. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute of the user's role entry.
5. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute of the user's organization entry.
6. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute as a global default.
7. A URL set for the `iplanet-am-user-failure-url` attribute in the user's entry (`amUser.xml`).
8. A URL set for the `iplanet-am-auth-login-failure-url` attribute of the user's role entry.
9. A URL set for the `iplanet-am-auth-login-failure-url` attribute of the user's organization entry.
10. A URL set for the `iplanet-am-auth-login-failure-url` attribute as the global default.

Module-based Authentication

This method of authentication allows a user to specify the module to which they will authenticate. The specified module must be registered to the organization or sub-organization that the user is accessing. This is configured in the Organization Authentication Modules attribute of the organization's Core Authentication Service. On receiving this request for module-based authentication, the Authentication Service verifies that the module is correctly configured as noted, and if the module is not defined, the user is denied access.

NOTE See the *Sun ONE Identity Server Administration Guide* for more information on how to register the authentication services and modules using the Identity Server console.

Module-based Authentication Login URLs

Module-based authentication can be specified in The User Interface Login URL by defining a module Parameter. The login URLs used to specify and initiate module-based authentication are:

- `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?module=`*authentication_module_name*
- `http://`*server_name.domain_name*`:`*port*`/amserver/UI/Login?org=`*org_name*`&module=`*authentication_module_name*

If there is no configured `org` paramter, the organization to which the user belongs will be determined from the server host and domain specified in the login URL itself. More information on these parameters can be found in "Login URL Parameters" on page 74.

Module-based Authentication Redirection URLs

Upon a successful or failed module-based authentication, Identity Server looks for information on where to redirect the user. Following is the order of precedence in which the application will look for this information.

Successful Module-based Authentication Redirection URLs

The redirection URL for successful module-based authentication is detemined by checking the following places in order of precedence:

1. A URL set by the authentication module.
2. A URL set by a `goto` Login URL parameter.
3. A URL set in the `clientType` custom files for the `iplanet-am-user-success-url` attribute of the user's profile (`amUser.xml`).
4. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute of the user's role entry.
5. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute of the user's organization entry.
6. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-success-url` attribute as a global default.
7. A URL set in the `iplanet-am-user-success-url` attribute of the user's profile (`amUser.xml`).
8. A URL set in the `iplanet-am-auth-login-success-url` attribute of the user's role entry.

9. A URL set in the `iplanet-am-auth-login-success-url` attribute of the user's organization entry.
10. A URL set in the `iplanet-am-auth-login-success-url` attribute as a global default.

Failed Module-based Authentication Redirection URLs

The redirection URL for failed module-based authentication is detemined by checking the following places in the following order:

1. A URL set by the authentication module.
2. A URL set by a `gotoOnFail` Login URL parameter.
3. A URL set in the `clientType` custom files for the `iplanet-am-user-failure-url` attribute of the user's entry (`amUser.xml`).
4. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute of the user's role entry.
5. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute of the user's organization entry.
6. A URL set in the `clientType` custom files for the `iplanet-am-auth-login-failure-url` attribute as a global default.
7. A URL set for the `iplanet-am-user-failure-url` attribute in the user's entry (`amUser.xml`).
8. A URL set for the `iplanet-am-auth-login-failure-url` attribute of the user's role entry.
9. A URL set for the `iplanet-am-auth-login-failure-url` attribute of the user's organization entry.
10. A URL set for the `iplanet-am-auth-login-failure-url` attribute as the global default.

Authentication Features

This section defines a number of features that can be enabled and/or configured. They include the following:

- Account Locking
- Authentication Module Chaining

- Fully Qualified Domain Name Mapping
- Persistent Cookie
- Multi-LDAP Authentication Module Configuration
- Session Upgrade
- Validation Plug-in Interface.

Account Locking

The Authentication Service provides a feature where a user will be *locked out* from authenticating after *n* failures. This feature is turned off by default, but can be enabled using the Identity Server console.

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

The Core Authentication Service contains attributes for enabling and customizing this feature including (but not limited to):

- **Login Failure Lockout Mode** which enables account locking.
- **Login Failure Lockout Count** which defines the number of tries that a user may attempt to authenticate before being locked out. This count is valid per user ID only; the same user ID needs to fail for the given count after which that user ID would be locked out.
- **Login Failure Lockout Interval** defines (in minutes) the amount of time in which the Login Failure Lockout Count value must be completed before a user is locked out.
- **Email Address to Send Lockout Notification** specifies an email address to which user lockout notifications will be sent.
- **Warn User After N Failure** specifies the number of authentication failures that can occur before a warning message will be displayed to the user. This allows an administrator to set additional login attempts after the user is warned about an impending lockout.
- **Login Failure Lockout Duration** defines (in minutes) how long the user will have to wait before attempting to authenticate again after lockout.

- **Lockout Attribute Name** defines which LDAP attribute in the user's profile will be set to `inactive` for Physical Locking.
- **Lockout Attribute Value** defines to what the LDAP attribute specified in **Lockout Attribute Name** will be set: `inactive` or `active`.

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 For special instructions when using this feature on a Microsoft® Windows 2000 operating system, see "Simple Mail Transfer Protocol (SMTP)" in Appendix A, "AMConfig.properties File," in this manual.

Identity Server supports two types of account locking are supported: Physical Locking and Memory Locking. These are defined in the next sections.

Physical Locking

This is the default locking behavior for Identity Server. The locking is initiated by changing the status of a LDAP attribute in the user's profile to inactive. The `Lockout Attribute Name` attribute defines the LDAP attribute used for locking purposes. See the *Sun ONE Identity Server Administration Guide* for more information on configuring physical locking.

NOTE An aliased user is one that is mapped to an existing LDAP user profile by configuring the User Alias List Attribute (`iplanet-am-user-alias-list` in `amUser.xml`) in the LDAP profile. Aliased users can be verified by adding `iplanet-am-user-alias-list` to the Alias Search Attribute Name field in the Core Authentication Service. That said, if an aliased user is locked out, the actual LDAP profile to which the user is aliased will be locked. This pertains only to physical lockout with authentication modules other than LDAP and Membership.

Memory Locking

Memory locking is enabled by changing the `Login Failure Lockout Duration` attribute to a value greater then 0. The user's account is then locked in memory for the number of minutes specified. The account will be unlocked after the time period has passed. Following are some special considerations when using the memory locking feature:

- If Identity Server is restarted, all accounts locked in memory are unlocked.

- If a user's account is locked in memory and the administrator changes the account locking mechanism to physical locking (by setting the lockout duration back to 0), the user's account will be unlocked in memory and the lock count reset.
- After memory lockout, when using authentication modules other than LDAP and Membership, if the user attempts to login with the correct password, a *User does not have profile in this organization error.* is returned rather than a *User is not active.* error.

NOTE If the Failure URL attribute is set in the user's profile, neither the lockout warning message nor the message indicating that their account has been locked will not be displayed; the user will be redirected to the defined URL.

Authentication Module Chaining

One or more authentication modules can be configured so a user must pass authentication credentials to all of them. This is referred to as *authentication chaining*. Authentication chaining in Identity Server is achieved using the JAAS framework integrated in the Authentication Service. Module chaining is configured under the Authentication Configuration Service. Each registered module is assigned one of the following four values:

- Required—Authentication to this module is required to succeed before the user can proceed down the chain.
- Requisite—The LoginModule is required to succeed. If it succeeds, authentication continues down the LoginModule list. If it fails, control immediately returns to the application (authentication does not proceed down the LoginModule list).
- Sufficient—The LoginModule is not required to succeed. If it does succeed, control immediately returns to the application (authentication does not proceed down the LoginModule list). If it fails, authentication continues down the LoginModule list.
- Optional—The LoginModule is not required to succeed. If it succeeds or fails, authentication still continues to proceed down the LoginModule list.

Once authentication to all modules in the chain is successful, control is returned to the Authentication Service (from the JAAS framework) which validates all the user IDs used to authenticate and maps them to one user. The mapping is achieved by configuring the `User Alias List` attribute in the user's profile. A valid session token is issued to the user if all the maps are correct; if not, the user is denied a valid session token. The following properties would represent the single authenticated user to which the other users are aliased:

- Principal (would contain the DN of the user in the case that the user has one)
- UserToken
- UserId

NOTE In authentication chaining, if all user IDs do not map to one single user, the failure redirection URL will be picked up from the last failed authentication module or none if all individual modules succeed (with different user ID). If case of user-based authentication, no matter what user ID is given in the authentication page, the failure redirection URL will always be picked up from the user parameter in the login URL.

Fully Qualified Domain Name Mapping

Fully Qualified Domain Name (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 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 (as illustrated in Code Example 3-13) as long as they conform to the stated requirements. If this property is not set, the user would be sent to the default server name configured in the `com.iplanet.am.server.host=`*server_name* property also found in the `AMConfig.properties` file.

:

Code Example 3-13 FQDN Mapping Attribute In `AMConfig.properties`

```
com.sun.identity.server.fqdnMap[isserver]=isserver.mydomain.com
com.sun.identity.server.fqdnMap[isserver.mydomain]=isserver.mydomain.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 a 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 address*`]=`*IP address*.

CAUTION 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.

Persistent Cookie

A persistent cookie is one that continues to exist after the web browser is closed, allowing a user to login with a new browser session without having to reauthenticate. The name of the cookie is defined by the `com.iplanet.am.pcookie.name` property in `AMConfig.properties`; the default value is `DProPCookie`. The cookie value is a 3DES-encrypted string containing the userDN, organization name, authentication module name, maximum session time, idle time, and cache time. To enable persistent cookies:

1. Turn on the `Persistent Cookie Mode` in the Core Authentication Service.
2. Configure a time value for the `Persistent Cookie Maximum Time` attribute in the Core Authentication Service.

3. Append the iPSPCookie Parameter with a value of `yes` to The User Interface Login URL.

   Once the user authenticates using this URL, if the browser is closed, they can open a new browser window and will be redirected to the console without reauthentication. This will work until the time defined in Step 2 elapses.

Persistent Cookie Mode can be turned on using the Authentication SPI method:

`AMLoginModule.setPersistentCookieOn().`

Multi-LDAP Authentication Module Configuration

As a form of failover or to configure multiple values for an attribute when the Identity Server console only provides one value field, an administrator can define multiple LDAP authentication module configurations under one organization. Although these additional configurations are not visible from the console, they work in conjunction with the primary configuration if an initial search for the requesting user's authorization is not found. For example, one organization can define a search through LDAP servers for authentication in two different domains or it can configure multiple user naming attributes in one domain. For the latter, which has only one text field in the console, if a user is not found using the primary search criteria, the LDAP module will then search using the second scope. Following are the steps to configure additional LDAP configurations.

To Add An Additional LDAP Configuration

1. Write an XML file including the complete set of attributes and new values needed for second(or third) LDAP authentication configuration.

   The available attributes can be referenced by viewing the `amAuthLDAP.xml` located in `etc/opt/SUNWam/config/xml`. This XML file created in this step though, unlike the `amAuthLDAP.xml`, is based on the structure of the `amadmin.dtd` as defined in Chapter 6, "Service Management," in this manual. Any or all attributes can be defined for this file. Code Example 3-14 is an example of a subconfiguration file that includes values for all attributes available to the LDAP authentication configuation.

Code Example 3-14 Sample XML File To Add An LDAP SubConfiguration

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--
  Copyright (c) 2002 Sun Microsystems, Inc. All rights reserved.
  Use is subject to license terms.
-->
```

Code Example 3-14 Sample XML File To Add An LDAP SubConfiguration *(Continued)*

```
<!DOCTYPE Requests
    PUBLIC "-//iPlanet//Sun ONE Identity Server 6.0 Admin CLI DTD//EN"
    "jar://com/iplanet/am/admin/cli/amAdmin.dtd"
>
<!--
  Before adding subConfiguration load the schema with
GlobalConfiguration defined and replace corresponding
 serviceName and subConfigID in this sample file OR load
 serviceConfigurationRequests.xml before loading this sample
-->
<Requests>
<OrganizationRequests DN="dc=iplanet,dc=com">
    <AddSubConfiguration subConfigName = "ssc"
        subConfigId = "serverconfig"
        priority = "0" serviceName="iPlanetAMAuthLDAPService">

        <AttributeValuePair>
            <Attribute name="iplanet-am-auth-ldap-server"/>
            <Value>newvalue</Value>
        </AttributeValuePair>
        <AttributeValuePair>
            <Attribute name="iplanet-am-auth-ldap-server"/>
            <Value>vbrao.red.iplanet.com:389</Value>
        </AttributeValuePair>
        <AttributeValuePair>
            <Attribute name="iplanet-am-auth-ldap-base-dn"/>
            <Value>dc=iplanet,dc=com</Value>
        </AttributeValuePair>
        <AttributeValuePair>
            <Attribute name="planet-am-auth-ldap-bind-dn"/>
            <Value>cn=amldapuser,ou=DSAME Users,dc=iplanet,dc=com</Value>
        </AttributeValuePair>
        <AttributeValuePair>
            <Attribute name="iplanet-am-auth-ldap-bind-passwd"/>
            <Value>copy encrypted password</Value>
        </AttributeValuePair>
        <AttributeValuePair>
            <Attribute name="iplanet-am-auth-ldap-user-naming-attribute"/>
            <Value>uid</Value>
        </AttributeValuePair>
        <AttributeValuePair>
            <Attribute name="iplanet-am-auth-ldap-user-search-attributes"/>
            <Value>uid</Value>
        </AttributeValuePair>
        <AttributeValuePair>
            <Attribute name="iplanet-am-auth-ldap-search-scope"/>
            <Value>SUBTREE</Value>
        </AttributeValuePair>
        <AttributeValuePair>
            <Attribute name="iplanet-am-auth-ldap-ssl-enabled"/>
            <Value>false</Value>
        </AttributeValuePair>
        <AttributeValuePair>
            <Attribute name="iplanet-am-auth-ldap-return-user-dn"/>
            <Value>true</Value>
```

Code Example 3-14 Sample XML File To Add An LDAP SubConfiguration *(Continued)*

```
          </AttributeValuePair>
          <AttributeValuePair>
              <Attribute name="iplanet-am-auth-ldap-auth-level"/>
              <Value>0</Value>
          </AttributeValuePair>
          <AttributeValuePair>
              <Attribute name="iplanet-am-auth-ldap-server-check"/>
              <Value>15</Value>
          </AttributeValuePair>

      </AddSubConfiguration>

  </OrganizationRequests>
  </Requests>
```

2. Get an encrypted value for the amadmin password using the `ampassword` utility shipped with Identity Server.

   `ampassword -e` *administrator_password*

   More information on this utility can be found in the *Sun ONE Identity Server Administration Guide.*

3. Copy the encrypted password as the value for the `iplanet-am-auth-ldap-bind-passwd` in the XML file created in Step 1.

   The value of this attribute is formatted in bold in Code Example 3-14 on page 126.

4. Load the XML file using the `amadmin` command line tool.

   `./amadmin -u amadmin -w` *administrator_password* `-v -t` *name_of_XML_file*.

Be aware that this second LDAP configuration can not be seen or modified using the Identity Server console.

TIP There is a sample available for multi-LDAP configuration. See the `serviceAddMultipleLDAPConfigurationRequests.xml` command line template in */IdentityServer_base/*`SUNWam/samples/admin/cli/bulk-ops/`. Instructions can be found in `Readme.html` at */IdentityServer_base/*`SUNWam/samples/admin/cli/`.

Session Upgrade

The Authentication Service allows for the upgrading of a valid session token based on a second, successful authentication performed by the same user to one organization. If a user with a valid session token attempts to authenticate to a resource secured by his current organization and this second authentication request is successful, the session is updated with the new properties based on the new authentication. If the authentication fails, the user's current session is returned without an upgrade. If the user with a valid session attempts to authenticate to a resource secured by a different organization, the user will receive a message asking whether they would like to authenticate to the new organization. The user can, at this point, maintain the current session or attempt to authenticate to the new organization. Successful authentication will result in the old session being destroyed and a new one being created.

During session upgrade, if a login page times out, redirection to the original success URL will occur. Timeout values are determined based on:

- The page timeoutvalue set for each module (default is 1 minute)
- `com.iplanet.am.invalidMaxSessionTime` property in `AMConfig.properties` (default is 10 minutes)
- `iplanet-am-max-session-time` (default is 120 minutes)

The values of `com.iplanet.am.invalidMaxSessionTimeout` and `iplanet-am-max-session-time` should be greater than the page timeout value, or the valid session information during session upgrade will be lost and URL redirection to the previous successful URL will fail.

Validation Plug-in Interface

An Administrator can write username or password validation logic suitable to their organization, and plug this into the Authentication Service. (This functionality is supported only by the LDAP and Membership authentication modules.) Before authenticating the user or changing the password, Identity Server will invoke this plugin. If the validation is successful, authentication continues; if it fails, an authentication failed page will be thrown. The plugin extends the `com.iplanet.am.sdk.AMUserPasswordValidation` class which is part of the Service Management SDK. Information on this SDK can be found in the `com.iplanet.am.sdk` package in the Identity Server Javadocs and in Chapter 6, "Service Management," in this manual. The steps below document how to write and configure a validation plugin for Identity Server.

1. The new plugin class will extend the `com.iplanet.am.sdk.AMUserPasswordValidation` class and implement the `validateUserID()` and `validatePassword()` methods. `AMException` should be thrown if validation fails.
2. Compile the plugin class and place the `.class` file in the desired location. Update the classpath so that it is accessible by the Identity Server during runtime.
3. Login to the Identity Server console as top-level administrator. Click on the Service Management tab, and get to the attributes for the Administration Service. Type the name of the plugin class (including the package name) in the `UserID & Password Validation Plugin Class` field.
4. Logout and login.

Authentication DTD Files

The Authentication Service uses document type definition (DTD) files to define the structure for the XML files it uses. The DTDs are located in *IdentityServer_base*/`SUNWam/dtd` and, for use within the Authentication Service, include:

- `Auth_Module_Properties.dtd`—defines the structure for XML files used by each authentication module to specify the properties for their particular Authentication Service interface. These files are detailed in "Authentication Module Configuration Files" on page 84. Information on this document can be found in "Auth_Module_Properties.dtd" on page 131.
- `remote-auth.dtd`—defines the structure for XML files used by the "Authentication Programming Interfaces" on page 153. Information on this document can be found in "The remote-auth.dtd Structure" on page 135.

NOTE Other DTD files are discussed in Chapter 6, "Service Management," in this manual.

Auth_Module_Properties.dtd

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 user. The DTD is located in *IdentityServer_base*/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 3-15 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 3-15 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>
```

Code Example 3-15 LDAP.xml *(Continued)*

```
     </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 define the information a module needs to gather from the client requesting authentication. Each *Callbacks* element signifies a separate screen that can be called during the authentication process. It can contain one or more of four sub-elements: *NameCallback*, *PasswordCallback*, *ChoiceCallback* or *ConfirmationCallback*. The required XML attributes of *Callbacks* are:

- `length`—takes a value equal to the number of callback requests for the defined element.
- `order`—takes a value equal to the number this particular screen would be in the sequence of callbacks. (The value of `order` starts with the number ‘1’.)

The optional XML attributes are:

- `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 (JSP or HTML) used as a display template for this screen.
- `image`—defines a custom image to be displayed on the screen at a specific location.
- `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 3-15 on page 131 defines three screen callback elements that can be called by the LDAP Authentication module. The first asks the requestor for a name and password. The second is an optional screen that allows the requestor to change their password. The final screen sends a message informing the user that it is time to reset their password.

NOTE The Callbacks element can also be used for error handling. An error message can be sent to the user interface using the header and error attributes and formatting the element as:

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

Or one of the pre-defined error JSPs, located in *IdentityServer_base*/`SUNWam/web-apps/services/config/auth/default`, can be sent by formatting the element as:

```
<Callbacks length="0" order="n" timeout="120"
template="jsp_name" />
```

NameCallback Element

The *NameCallback* element is used to request data from the user; for example, a user identification. It can contain one sub-element, *Prompt,* which defines the text that precedes the input field. 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 text defined in *Prompt*.) `attribute` takes a character value of the corresponding LDAP attribute of this value.

PasswordCallback Element

The *PasswordCallback* element is used to request password data to be entered by the user. It can contain one sub-element, *Prompt,* which defines the text that precedes the input field. The XML attributes are `echoPassword`, `isRequired` and `attribute`. `echoPassword` is required, takes a value of `true` or `false` and defines whether the password should be displayed on the screen. `isRequired` is optional, 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 text defined in *Prompt*.) `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*. *Prompt* defines the text that precedes the input field. *ChoiceValues* defines the values from which the user may choose. 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 text defined in *Prompt*.) `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 to the authentication interface (such as 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). In future versions of Identity Server, this element will support additional features. It can contain one sub-element, *OptionValues*, which provides a list of text information. There are no XML attributes.

NOTE When a custom authentication module XML service file is configured without the *ConfirmationCallback*, button properties are picked up from the global i18n authentication properties file, `amAuthUI.properties`.

Prompt Element

The *Prompt* element is used to set the text that will display beside a text input field on the browser screen. It has no sub-elements or XML attributes.

ChoiceValues and ChoiceValue Element

The *ChoiceValues* element provides a list of choices 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 is selected when the choices are 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 is selected when the choices are 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.

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 Identity Server console receives these XML-based messages which provide definitions to initiate the collection of credentials and perform authentication. It is located in the *IdentityServer_base*`/SUNWam/dtd` directory. An explanation of the elements defined by the `remote-auth.dtd` follows. More information can be found in the file itself.

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 of the DTD.

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 that is used to keep track of the authentication session. Table 3-9 shows the Request sub-elements and the possible Responses for each.

Table 3-9 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

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 being 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 defined authentication module plug-ins configured for the 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*. 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`. See "Authentication Methods" on page 103 for more information.

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 level or the service level, or the module level. The `IndexType` attribute defines `authLevel`, `role`, `user`, `moduleInstance` and `service`. The `IndexName` element takes a value equal to that of the name of the level at which the authentication method has been defined. For example, if `IndexType` is `user` or `role` then, `IndexName` might be `joe` or `administrator`, respectively. IndexName 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.

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.

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

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.

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.

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.

Custom Authentication Modules

The Authentication Service allows an organization to write and plug-in custom modules for authentication providers not currently implemented by Identity Server. The first step is to code the module in Java using the Service Programming Interfaces. Instructions on how to do this and which classes to use can be found in "Implementing A Custom Authentication Module" on page 171.

NOTE To write a custom authentication module, knowledge of the JAAS API is necessary, especially for defining the module's configuration properties. For more information, 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/`.

Integrating A Custom Authentication Module

The following steps describe the procedure to integrate a custom authentication module into the Identity Server deployment. In detailing this procedure, it also explains what files are needed in order to make the integration work. For information on how to implement a custom module, see "Service Programming Interfaces" on page 170.

1. Create an authentication module configuration file.

   An authentication module configuration file specifies, at a minimum, the information required from an identity (user, service, or application) for authentication. It is located in *IdentityServer_base*/`SUNWam/web-apps/services/config/auth/default`. The required credentials might include, but are not limited to, user name and password. Based on this file, the Authentication Service will dynamically generate the login screens. Additional information on the file itself can be found in "Authentication Module Configuration Files" on page 84. Information on how to create it can be found in "Configuring The Authentication Module" on page 144.

TIP Creating the authentication module configuration file first allows time to plan the module requirements as each Callbacks Element defined corresponds to a login state. When an authentication process is invoked, a `Callback()` is generated from the module for each state. For more information, see "Auth_Module_Properties.dtd" on page 131 and "Implement The LoginModule Interface" on page 174.

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 *IdentityServer_base/*`SUNWam/locale/`. Information on this file can be found in "Configuring Authentication Localization Properties" on page 151. Additional information on how to modify its contents can be found in "Configuring Console Localization Properties" on page 251 of Chapter 6, "Service Management," in this manual.

3. Create an XML service file for the new authentication module and import it into Identity Server.

   The XML service file is written and imported to allow the management of the authentication module's parameters using the console. The name of the XML service file follows the format `amAuth`*modulename*`.xml` (for example, `amAuthSafeWord.xml` or `amAuthLDAP.xml`). The file is located in `/etc/opt/SUNWam/config/xml`. Information on the steps to create and import an XML service file can be found in Chapter 6, "Service Management," in this manual.

4. Modify attributes in the Core Authentication Service.

   The new module needs to be recognized by the Identity Server framework. Information on how this is done can be found in "Modifying The Core Authentication Service" on page 152. Additional information on these attributes can be found in the Core Authentication Attributes chapter of the *Sun ONE Identity Server Administration Guide*.

5. Restart Identity Server.

Configuring The Authentication Module

The authentication module configuration file is an XML file that defines the requirements that each module seeks for authentication. In other words, this file defines the screens that a user might see when directed to authenticate (i.e.: user name screen, password screen, change password screen, etc.). Modifying elements in this 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`. (*modulename* is the same name as the class without the package.) Each authentication module has its own configuration file, located in *IdentityServer_base*`/SUNWam/web-apps/services/config/auth/default`.

NOTE	An authentication module configuration file needs to be created and stored for each custom module, as discussed, even if the module itself has no requirements for authentication.

If there is more than one organization in the Identity Server deployment, each organization should have its own authentication directory named *IdentityServer_base*`/SUNWam/web-apps/services/config/auth/default/`*org_name*. If an organization has more than one locale, the files are stored separately, in directories appended with a locale, as in *IdentityServer_base*`/SUNWam/web-apps/services/config/auth/default/`*org_name_locale*. Additionally, with service authentication, there might be an authentication directory under the organization's tree that corresponds to the service. More information on this directory hierarchy can be found in "To Create New Directories For Custom Console Files" on page 89.

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.

Elements Of The Authentication Module Configuration File

Each login page in the authentication module is defined by a Callback element which is generally a request for authentication information.

NOTE To understand more on how these authentication module configuration files are defined and constructed, refer to "Auth_Module_Properties.dtd" on page 131.

Code Example 3-16 defines two login pages: the first asks for a first and last name while the second asks for a password. The order in which the pages are displayed to the user are defined in the `order` attribute of the Callbacks element. All login requests begin with the Callback defined as 1. From that point, the module takes control of the login process and decides on the next page for display based on user interaction.

Code Example 3-16 Sample Authentication Module Configuration File

```
<ModuleProperties moduleName="LoginModuleSample" version="1.0" >
                   <Callbacks length="2" order="1" timeout="60"
                       header="This is a sample login page" >
                   <NameCallback>
                       <Prompt> User Name </Prompt>
                   </NameCallback>
                   <NameCallback>
                       <Prompt> Last Name </Prompt>
                   </NameCallback>
                   </Callbacks>
                   <Callbacks length="1" order="2" timeout="60"
                       header="You made it to page 2" >
                       <PasswordCallback echoPassword="false" >
                           <Prompt> Enter any password </Prompt>
                       </PasswordCallback>
                   </Callbacks>
               </ModuleProperties>
```

In the sample module configuration shown above, page one has two requests for information, the first is for User Name and the second is for Last Name. When the user responds, the information is sent back to the module, where the module writer validates it and returns the next page. Page two has one request for a user password. When the user responds, the password is returned. If the module writer throws an exception, an `Authentication failed.` page is sent to the user. If no exception is thrown, the user will be redirected to their default page based on the information on redirection URLs in "Authentication Methods" on page 103.

Customizing Membership.xml

The Membership Authentication Module contains a self-registration functionality. Membership authentication is implemented for users to configure personalized portals. The user can create an account, personalize it without the aid of an administrator, and access it as a registered user. Code Example 3-17 contains `Membership.xml`, the authentication module configuration file for the Membership Authentication module. Modifying this file allows an administrator to add customer-specific user profile data fields onto the self-registration page. In `Membership.xml`, Callbacks 1 through 15 pertain to the Membership portion of the module; users who have already registered for personalized site profiles can login with a user name and password. Callback 16 and up refer to the Self-Registration portion; users can register and a profile will be created. The customization example begun after Code Example 3-17 refers to Callback 16 and customizing the self-registration functionality.

Code Example 3-17 Membership.xml Configuration File

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

    <Callbacks length="3" order="1" timeout="120" header="Self Registration
Module" template="membership.jsp" >
        <NameCallback>
            <Prompt> User Name: </Prompt>
        </NameCallback>
        <PasswordCallback echoPassword="false" >
            <Prompt> Password: </Prompt>
        </PasswordCallback>
        <ConfirmationCallback>
            <OptionValues>
                <OptionValue>
                    <Value> Log In </Value>
                </OptionValue>
                <OptionValue>
                    <Value> New User </Value>
                </OptionValue>
             </OptionValues>
        </ConfirmationCallback>
    </Callbacks>

    <Callbacks length="4" order="2" timeout="240" header="Password Expiring
Please Change&lt;BR/&gt;#REPLACE#&lt;BR/&gt;" >
        <PasswordCallback echoPassword="false" >
            <Prompt> Current Password: </Prompt>
        </PasswordCallback>
        <PasswordCallback echoPassword="false" >
            <Prompt> New Password: </Prompt>
        </PasswordCallback>
        <PasswordCallback echoPassword="false" >
            <Prompt> Confirm New Password: </Prompt>
        </PasswordCallback>
```

Code Example 3-17 Membership.xml Configuration File *(Continued)*

```
        <ConfirmationCallback>
            <OptionValues>
                <OptionValue>
                    <Value> Submit </Value>
                </OptionValue>
                <OptionValue>
                    <Value> Cancel </Value>
                </OptionValue>
            </OptionValues>
        </ConfirmationCallback>
    </Callbacks>

    <Callbacks length="0" order="3" timeout="120"
template="wrongPassword.jsp" />

    <Callbacks length="0" order="4" timeout="120"
template="noUserProfile.jsp" />

    <Callbacks length="0" order="5" timeout="120" template="noUserName.jsp"
/>

    <Callbacks length="0" order="6" timeout="120" template="noPassword.jsp"
/>

    <Callbacks length="0" order="7" timeout="120"
template="noConfirmation.jsp" />

    <Callbacks length="0" order="8" timeout="120"
template="passwordMismatch.jsp" />

    <Callbacks length="0" order="9" timeout="120"
template="configuration.jsp" />

   <Callbacks length="0" order="10" timeout="120" template="userExists.jsp"
/>

    <Callbacks length="0" order="11" timeout="120"
template="profileException.jsp" />

    <Callbacks length="0" order="12" timeout="120"
template="missingReqField.jsp" />

    <Callbacks length="0" order="13" timeout="120"
template="userPasswordSame.jsp" />

    <Callbacks length="0" order="14" timeout="120"
template="invalidPassword.jsp" />

   <Callbacks length="0" order="15" timeout="120" header="Your password has
expired. Please contact service desk to reset your password" error="true" />
```

Code Example 3-17 Membership.xml Configuration File *(Continued)*

```
    <Callbacks length="8" order="16" timeout="300" header="Self 
Registration" template="register.jsp" >
        <NameCallback isRequired="true" attribute="uid" >
            <Prompt> User Name: </Prompt>
        </NameCallback>
        <PasswordCallback echoPassword="false" isRequired="true" 
attribute="userPassword" >
            <Prompt> Password: </Prompt>
        </PasswordCallback>
        <PasswordCallback echoPassword="false" isRequired="true" >
            <Prompt> Confirm Password: </Prompt>
        </PasswordCallback>
        <NameCallback isRequired="true" attribute="givenname" >
            <Prompt> First Name: </Prompt>
        </NameCallback>
        <NameCallback isRequired="true" attribute="sn" >
            <Prompt> Last Name: </Prompt>
        </NameCallback>
        <NameCallback isRequired="true" attribute="cn" >
            <Prompt> Full Name: </Prompt>
        </NameCallback>
        <NameCallback attribute="mail" >
            <Prompt> Email Address: </Prompt>
        </NameCallback>
        <ConfirmationCallback>
            <OptionValues>
                <OptionValue>
                    <Value> Register </Value>
                </OptionValue>
                <OptionValue>
                    <Value> Cancel </Value>
                </OptionValue>
             </OptionValues>
        </ConfirmationCallback>
    </Callbacks>

    <Callbacks length="2" order="17" timeout="120" header="Self 
Registration" >
        <ChoiceCallback attribute="uid" >
            <Prompt>A user already exists with the user name you entered. 
&lt;BR&gt;Please choose one of the following user names, or create your 
own:</Prompt>
            <ChoiceValues>
              <ChoiceValue>
                  <Value>Create My Own</Value>
              </ChoiceValue>
            </ChoiceValues>
        </ChoiceCallback>
        <ConfirmationCallback>
            <OptionValues>
                <OptionValue>
                    <Value> Submit </Value>
                </OptionValue>
            </OptionValues>
```

Code Example 3-17 Membership.xml Configuration File *(Continued)*

```
        </ConfirmationCallback>
    </Callbacks>

   <Callbacks length="1" order="18" timeout="120" template="disclaimer.jsp"
>
        <ConfirmationCallback>
            <OptionValues>
                <OptionValue>
                    <Value> Agree </Value>
                </OptionValue>
                <OptionValue>
                    <Value> Disagree </Value>
                </OptionValue>
            </OptionValues>
        </ConfirmationCallback>
    </Callbacks>
</ModuleProperties>
```

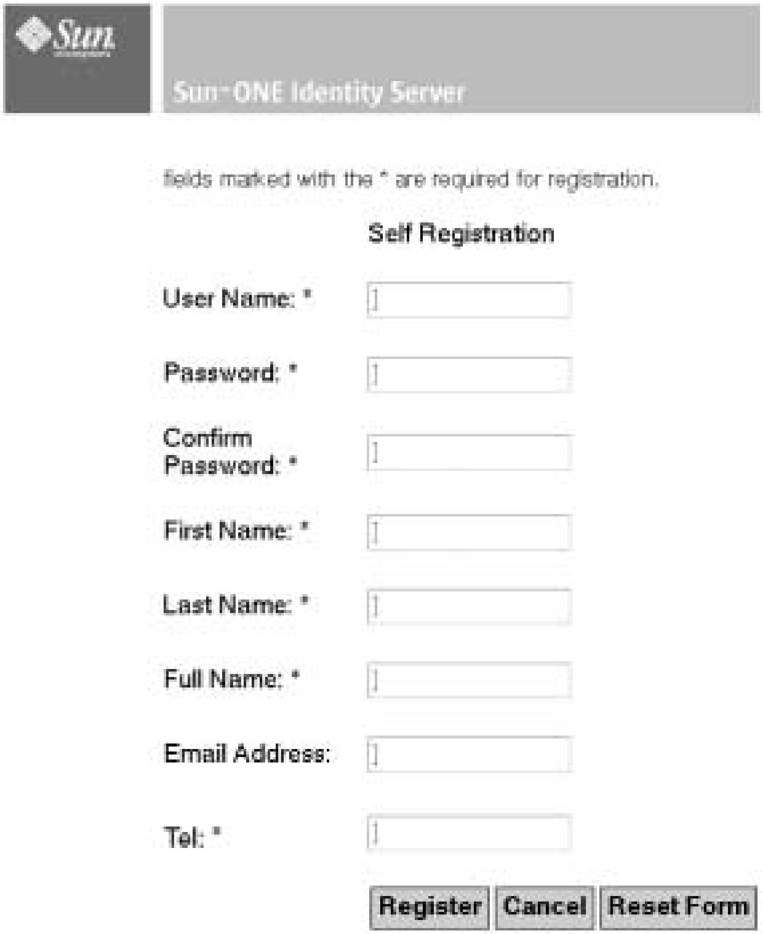
The first Prompts (all defined in Callbacks 16) are required fields in the Self-Registration module. They are User Name, Password, Confirm Password, First Name, Last Name, and Full Name. Each Callback contains the attribute / value pair, `isRequired="true"`. E-mail Address is, by default, not a field the user is required to fill in but that can be changed by adding `isRequired="true"` as an attribute of that particular Name Callback. To add a field for the user to fill in a telephone number, `Membership.xml` should be modifed with the Name Callback defined in Code Example 3-18.

Code Example 3-18 Telephone Number Name Callback

```
<NameCallback isRequired="true"attribute="telephonenumber">
               <Prompt> Tel:</Prompt>
               </NameCallback>
```

Add user data fields which are normally requested as part of a user profile. By default, attributes from the following LDAP objectClasses can be easily added: `top`, `person`, `organizationalPerson`, `inetOrgPerson`, `iplanet-am-user-service`, `inetuser`. Figure 3-12 is the Self-Registration Login Requirement Screen after the telephone number field has been added to `Membership.xml`. Identity Server and the deployment container should be restarted after modifying `Membership.xml`.

Figure 3-12 Self-Registration Login Requirement Screen



fields marked with the * are required for registration.

Self Registration

User Name: *

Password: *

Confirm Password: *

First Name: *

Last Name: *

Full Name: *

Email Address:

Tel: *

Configuring Authentication Localization Properties

A localization properties file specifies the screen text and messages that an administrator or user will see when directed to an authentication module's service attribute configuration page. Each authentication module has its own localization properties file that follows the naming format `amAuth`*modulename*`.properties`; for example, `amAuthLDAP.properties`. They are located in *IdentityServer_base*`/SUNWam/ locale/`. The default character set is ISO-8859-1 (English). Following are some concepts behind the configuration of this file.

- The data following the equal (=) sign in each key/value pair could be translated to a specific language as necessary and copied into the corresponding locale directory. In Code Example 3-19, 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.

For reference, Code Example 3-19 is a portion of the file `amAuthLDAP.properties`.

Code Example 3-19 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
a6=User Naming Attribute
a7=User Entry Search Attribute
...
```

Additional information on this file and how to modify its contents can be found in "Configuring Console Localization Properties" on page 251 of Chapter 6, "Service Management," in this manual.

Modifying The Core Authentication Service

Some attributes in the Core Authentication Service need to be extended in order for the Authentication Service to recognize any newly created authentication module.

NOTE `amAuth.xml`, located in `/etc/opt/SUNWam/config/xml`, defines the *Core Authentication Service*. When making these changes directly to the XML file, the old file has to be removed and the newly modified one reloaded using the `amadmin` command line tool. More information on the command line tool can be found in the *Sun ONE Identity Server Administration Guide*.

Pluggable Auth Module Classes Attribute

This global attribute specifies the Java classes of the authentication modules available within the Identity Server deployment. By default, this includes the modules listed in "Authentication Service Modules" on page 66. To define a new authentication module, this field takes a value equal to the full class name (including package) of the new module. This modification can also be made in the `iplanet-am-auth-authenticators` attribute of `amAuth.xml`. Code Example 3-20 illustrates the default values for this attribute.

Code Example 3-20 `iplanet-am-auth-authenticators` Attribute

```
...
<AttributeSchema name="iplanet-am-auth-authenticators"
                    type="list"
                    syntax="string"
                    i18nKey="a117">
                    <DefaultValues>
<Value>com.sun.identity.authentication.modules.radius.RADIUS</Value>
<Value>com.sun.identity.authentication.modules.ldap.LDAP</Value>
<Value>com.sun.identity.authentication.modules.membership.Membership</Value
>
<Value>com.sun.identity.authentication.modules.unix.Unix</Value>
<Value>com.sun.identity.authentication.modules.anonymous.Anonymous</Value>
<Value>com.sun.identity.authentication.modules.cert.Cert</Value>
<Value>com.sun.identity.authentication.modules.application.Application</Val
ue>
<Value>com.sun.identity.authentication.modules.safeword.SafeWord</Value>
<Value>com.sun.identity.authentication.modules.securid.SecurID</Value>
<Value>com.sun.identity.authentication.modules.httpbasic.HTTPBasic</Value>
 <Value>com.sun.identity.authentication.modules.nt.NT</Value>
                    </DefaultValues>
                </AttributeSchema>
...
```

Organization Authentication Modules Attribute

This organization-type attribute lists the authentication modules available to the organizations. An administrator can choose the authentication method for their organization from this list. The default authentication method is LDAP. This modification can also be made in the `iplanet-am-auth-allowed-modules` attribute of `amAuth.xml`. Code Example 3-21 illustrates the default values for this attribute.

Code Example 3-21 `iplanet-am-auth-allowed-modules` Attribute

```
<AttributeSchema name="iplanet-am-auth-allowed-modules"
                    type="multiple_choice"
                    syntax="string"
                    i18nKey="a101">
                    <ChoiceValues>
<ChoiceValue i18nKey="LDAP">LDAP</ChoiceValue>
<ChoiceValue i18nKey="RADIUS">RADIUS</ChoiceValue>
<ChoiceValue i18nKey="Membership">Membership</ChoiceValue>
<ChoiceValue i18nKey="Unix">Unix</ChoiceValue>
<ChoiceValue i18nKey="Anonymous">Anonymous</ChoiceValue>
<ChoiceValue i18nKey="Cert">Cert</ChoiceValue>
<ChoiceValue i18nKey="SafeWord">SafeWord</ChoiceValue>
<ChoiceValue i18nKey="SecurID">SecurID</ChoiceValue>
<ChoiceValue i18nKey="NT">NT</ChoiceValue>
<ChoiceValue i18nKey="HTTPBasic">HTTPBasic</ChoiceValue>
                    </ChoiceValues>
                    <DefaultValues>
                        <Value>LDAP</Value>
                    </DefaultValues>
                </AttributeSchema>
```

Authentication Programming Interfaces

Identity Server provides programming interfaces to extend the functionality of the Authentication Service in two ways. The Authentication Programming Interfaces provides interfaces that can be used remotely by either Java or C applications to utilize the authentication features of Identity Server. The Service Programming Interfaces can be used to plug new authentication modules, written in Java, into the Identity Server authentication framework.

Exception Handling When using either the Authentication SPI (`AMLoginModule` class) or Authentication API (`AuthContext` class), `AuthLoginException` should be used for exception handling instead of `LoginException`. `AuthLoginException` has the capability for handling localized message information and exception chaining. See the JavaDocs for more details.

Application Programming Interfaces

Identity Server provides both a Java Authentication API and a C Authentication API for writing authentication clients that remote applications can use to gain access to the Authenticate Service for authentication purposes. This communication between the API and the Authentication Service occurs by sending XML messages over `HTTP(S)`. The `remote-auth.dtd` is the template used in formatting the XML request messages sent to Identity Server and for parsing the XML return messages received by the external application. "The remote-auth.dtd Structure" on page 135 details the document data for informational purposes; it can also be accessed in from *IdentityServer_base*/`SUNWam/dtd`.

NOTE If contacting the Authentication Service directly through its URL (`http://`*identity_server_host.domain_name:port/service_deploy_uri*/`authservice`) without the API, a detailed understanding of `remote-auth.dtd` will be needed for generating and interpreting the messages passed between the client and server. Sample response and return XML messages can be found in "XML Messages" on page 167.

Authentication API For Java Applications

As briefly mentioned in "Authentication Via The Java API" on page 65, external Java applications can authenticate users with the Identity Server Authentication Service by using the Authentication API for Java. The API are organized in a package called `com.sun.identity.authentication` and can be executed locally or remotely. The classes and methods defined in this package may be incorporated into a Java application to allow communication with the Authentication Service. They are used to initiate the authentication process and communicate authentication credentials to the specific modules within the Authentication Service.

Using The Authentication API For Java

The first step necessary for an external Java application to authenticate to Identity Server is to create a new `AuthContext` object (`com.sun.identity.authentication.AuthContext`). The `AuthContext` class is defined for each authentication request as it initiates the authentication process. Since Identity Server can handle multiple organizations, `AuthContext` is initialized, at the least, with the name of the organization to which the requestor is authenticating. Once an `AuthContext` object has been created, the `login()` method is called indicating to the server what method of authentication is desired. One of the following two methods can be used. Code Example 3-22 can be used specifically for Organization-based Authentication.

Code Example 3-22 Method For Organization-based Authentication

```
          public void login()
                                    throws AuthLoginException
```

Code Example 3-23 can be used to define any type of authentication method.

Code Example 3-23 Method For Defining Authentication Method

```
          public void login(IndexType type, String indexName)
                                    throws AuthLoginException
```

`IndexName` is the value of the authentication method. `IndexType` can have any of the following values:

- `AuthContext.IndexType.ROLE` defines Role-based Authentication.
- `AuthContext.IndexType.SERVICE` defines Service-based Authentication.
- `AuthContext.IndexType.USER` defines User-based Authentication.
- `AuthContext.IndexType.LEVEL` defines Authentication Level-based Authentication.
- `AuthContext.IndexType.MODULE_INSTANCE` defines Module-based Authentication.

The `getRequirements()` method then calls the objects that will be populated by the user. Depending on the parameters passed with the instantiated `AuthContext` object and the two method calls, Identity Server responds to the client request with the correct login requirement screens. For example, if the requested user is authenticating to an organization configured for LDAP authentication only, the server will respond with the LDAP login requirement screen to supply a user name and a password. The client must then loop by calling the `hasMoreRequirements()` method until the required credentials have been entered. Once entered, the credentials are submitted back to the server with the method call `submitRequirements()`. The final step is for the client to make a `getStatus()` method call to determine if the authentication was successful. If successful, the caller obtains a session token for the user; if not, a `LoginException` is thrown.

NOTE Because the Authentication Service is built on the JAAS framework, the Authentication API can also invoke any authentication modules written purely with the JAAS API.

Java Authentication API Samples

Identity Server provides two sample programs to demonstrate how to use the Authentication API for Java. They can be found in *IdentityServer_base*/`SUNWam/samples/authentication/`.

Certificate Authentication Sample The *IdentityServer_base*/`SUNWam/samples/authentication/Cert` directory provides a sample source code file, `CertLogin.java`, and a `readme.html` file that illustrates a Java application which utilizes digital certificates for authentication.

LDAP Authentication Sample The *IdentityServer_base*/`SUNWam/samples/authentication/LDAP` directory provides a sample source code file, `LDAPLogin.java`, and a `readme.html` file that illustrates a Java application which authenticates to the LDAP module. This sample can be easily modified to authenticate to other existing or customized authentication modules.

Authentication API For C Applications

As briefly mentioned in "Authentication Via The C API" on page 65, C applications can authenticate users with the Identity Server Authentication Service by using the Authentication API for C. C applications authenticate to Identity Server using these interfaces in much the same way as the "Authentication API For Java Applications." The C application contacts the Authentication Service to initiate the authentication process, and the Authentication Service responds with a set of requirements. The client application submits authentication credentials back to the Authentication Service and receives further authentication requirements back until there are no more to fulfill. After all requirements have been sent, the client makes one final call to determine if authentication has been successful or has failed. The C API can be found in */IdentityServer_base*/`SUNWam/agents`. This directory also includes a C API `samples` directory.

CAUTION Previous releases of Identity Server contained C libraries in *IdentityServer_base*/`lib/capi`. The `capi` directory is being deprectated, and is curently available for backward compatability. It will be removed in the next release, and therefore it is highly recommended that existing application paths to this directory are changed and new applications do not access it. Paths include `RPATH`, `LD_LIBRARY_PATH`, `PATH`, compiler options, etc.)

Using The Authentication API For C

The sequence of calls necessary to authenticate to Identity Server begins with the function call `am_auth_create_auth_context`. This call will return an `AuthContext` structure used for the rest of the authentication calls. Once an `AuthContext` structure has been initialized, the `am_auth_login` function is called. This indicates to the Authentication Service that an authentication is desired. Depending on the parameters passed when creating the `AuthContext` structure and making the `am_auth_login` function call, the Authentication Service will determine the login requirements with which to respond. For example, if the requested authentication is to an organization configured for LDAP authentication (and no authentication module chaining is involved), the server will respond with the requirements to supply a user name (which corresponds to the NameCallback element in The remote-auth.dtd Structure) and a password (which corresponds to the PasswordCallback element in The remote-auth.dtd Structure). The client loops on function call `am_auth_has_more_requirements`, and in this specific case there will be two, and fills in the needed information and submits this back to the server with function call am_auth_submit_requirements. The final step is to make function call `am_auth_get_status` to determine if the authentication was successful or not.

C Authentication API Sample

Identity Server provides a sample program to demonstrate how an external C application can use the API to authenticate a user via Identity Server. The sample can be found in *IdentityServer_base*`/SUNWam/agents/samples/common/`. By default, the C Authentication sample looks in *IdentityServer_base*`/SUNWam/agents/config` for a properties file named `AMAgent.properties`.

C Authentication Sample Properties Code Example 3-24 lists the properties that are needed by the C Authentication API; some of these are defined in `AMAgent.properties` and some are not. Those that are not can be added to the file so they needn’t be identified for each function call. For example, `com.sun.am.auth.orgName`, which identifies the organization from which authentication is desired, can be added to `AMAgent.properties`.

Code Example 3-24 AMAgent.properties File

```
# SOME PROPERTIES LISTED ARE NOT PRE-EXISTING IN THE PROPERTIES FILE

# the identity server naming service url
com.sun.am.namingURL=http://serverexample.domain.com:58080/amserver/namings
ervice
# the directory to use for logging
com.sun.am.logFile=/home/uid/logs/auth-log
```

Code Example 3-24 AMAgent.properties File *(Continued)*

```
# the logging level, all:5 being the highest and all:3 being medium
com.sun.am.logLevels=all:5
# the directory containing the certificate and key databases
com.sun.am.sslCertDir=/home/level/certdir
# the prefix of the cert7.db and key3.db files, if any
com.sun.am.certDbPrefix=
# the password to the key3.db file
com.sun.am.certDBPassword=11111111
# true to trust SSL certificates not in the client cert7.db
com.sun.am.trustServerCerts=true
# the nick name of the client certificate in the cert7.db
com.sun.am.auth.certificateAlias=Cert-Nickname
# the identity server organization desired for authentication
com.sun.am.auth.orgName=dc=sun,dc=com
```

C Authentication API Header File The C Authentication API header file, `am_auth.h`, can be found in *IdentityServer_base*`/SUNWam/agents/include`. It contains the function prototypes for the function calls available in the C Authentication API. It has been reproduced in Code Example 3-25 for informational purposes.

Code Example 3-25 am_auth.h C Authentication API Header File

```
/* -*- Mode: C -*- */
/*
 ...
 */

#ifndef __AM_AUTH_H__
#define __AM_AUTH_H__

#include <stdlib.h>
#include <am.h>
#include <am_properties.h>
#include <am_string_set.h>

AM_BEGIN_EXTERN_C

typedef struct am_auth_context *am_auth_context_t;

/*
 * Different types of authentication parameters.
 */
typedef enum am_auth_idx {
    AM_AUTH_INDEX_AUTH_LEVEL = 0,
    AM_AUTH_INDEX_ROLE,
    AM_AUTH_INDEX_USER,
    AM_AUTH_INDEX_MODULE_INSTANCE,
```

Code Example 3-25 am_auth.h C Authentication API Header File *(Continued)*

```
    AM_AUTH_INDEX_SERVICE
} am_auth_index_t;

/*
 * Enumeration of authentication statuses.
 */
typedef enum am_auth_status {
    AM_AUTH_STATUS_SUCCESS = 0,
    AM_AUTH_STATUS_FAILED,
    AM_AUTH_STATUS_NOT_STARTED,
    AM_AUTH_STATUS_IN_PROGRESS,
    AM_AUTH_STATUS_COMPLETED
} am_auth_status_t;

/*
 * Language locale structure.
 */
typedef struct am_auth_locale {
    const char *language;
    const char *country;
    const char *variant;
} am_auth_locale_t;

/*
 * Enumeration of types of callbacks.
 */
typedef enum am_auth_callback_type {
    ChoiceCallback = 0,
    ConfirmationCallback,
    LanguageCallback,
    NameCallback,
    PasswordCallback,
    TextInputCallback,
    TextOutputCallback
} am_auth_callback_type_t;

/*
 * Choice callback structure.
 */
typedef struct am_auth_choice_callback {
    const char *prompt;
    boolean_t allow_multiple_selections;
    const char **choices;
    size_t choices_size;
    size_t default_choice;
    const char **response; /* selected indexes */
    size_t response_size;
} am_auth_choice_callback_t;

/*
 * Confirmation callback structure.
 */
typedef struct am_auth_confirmation_callback_info {
    const char *prompt;
    const char *message_type;
```

Code Example 3-25 am_auth.h C Authentication API Header File *(Continued)*

```
    const char *option_type;
    const char **options;
    size_t options_size;
    const char *default_option;
    const char *response; /* selected index */
} am_auth_confirmation_callback_t;

/*
 * Language callback structure.
 */
typedef struct am_auth_language_callback_info {
    am_auth_locale_t *locale;
    am_auth_locale_t *response; /* locale */
} am_auth_language_callback_t;

/*
 * Name callback structure.
 */
typedef struct am_auth_name_callback_info {
    const char *prompt;
    const char *default_name;
    const char *response; /* name */
} am_auth_name_callback_t;

/*
 * Password callback structure.
 */
typedef struct am_auth_password_callback_info {
    const char *prompt;
    boolean_t echo_on;
    const char *response; /* password */
} am_auth_password_callback_t;

/*
 * Text Input callback structure.
 */
typedef struct am_auth_text_input_callback_info {
    const char *prompt;
    const char *default_text;
    const char *response; /* text */
} am_auth_text_input_callback_t;

/*
 * Text Output callback structure.
 */
typedef struct am_auth_text_output_callback_info {
    const char *message;
    const char *message_type;
} am_auth_text_output_callback_t;

/*
 * Primary callback structure.  The callback_type field
 * represents which type of callback this instance of callback
 * is representing.  Based on the type, the user must use the
 * appropriate member of the union.
```

Code Example 3-25 am_auth.h C Authentication API Header File *(Continued)*

```
 */
typedef struct am_auth_callback {
    am_auth_callback_type_t callback_type;
    union am_auth_callback_info {
        am_auth_choice_callback_t choice_callback;
        am_auth_confirmation_callback_t confirmation_callback;
        am_auth_language_callback_t language_callback;
        am_auth_name_callback_t name_callback;
        am_auth_password_callback_t password_callback;
        am_auth_text_input_callback_t text_input_callback;
        am_auth_text_output_callback_t text_output_callback;
    } callback_info;
} am_auth_callback_t;

/*
 * Initialize the authentication modules.
 *
 * Parameters:
 * auth_init_params  The property handle to the property file
 * which contains the properties to initialize the
 * authentication library.
 *
 * Returns:
 *   AM_SUCCESS
 *             If the initialization of the library is successful.
 *
 *   AM_NO_MEMORY
 *             If unable to allocate memory during initialization.
 *
 *   AM_INVALID_ARGUMENT
 *             If auth_init_params is NULL.
 *
 *   Others (Please refer to am_types.h)
 *             If the error was due to other causes.
 */
AM_EXPORT am_status_t
am_auth_init(const am_properties_t auth_init_params);

/*
 * Create a new auth context and returns the handle.
 *
 * Parameters:
 *   auth_ctx   Pointer to the handle of the auth context.
 *
 *   org_name   Organization name to authenticate to.
 *              May be NULL to use value in property file.
 *
 *   cert_nick_name
 *              The alias of the certificate to be used if
 *              the client is connecting securely.  May be
 *              NULL in case of non-secure connection.
 *
 *   url        Service URL, for example:
 *              "http://pride.red.iplanet.com:58080/amserver".
 *              May be NULL, in which case the naming service
```

Code Example 3-25 am_auth.h C Authentication API Header File *(Continued)*

```
 *               URL property is used.
 *
 * Returns:
 *   AM_SUCCESS
 *               If auth context was successfully created.
 *
 *   AM_NO_MEMORY
 *               If unable to allocate memory for the handle.
 *
 *   AM_INVALID_ARGUMENT
 *               If the auth_ctx parameter is NULL.
 *
 *   AM_AUTH_CTX_INIT_FAILURE
 *               If the authentication initialization failed.
 */
AM_EXPORT am_status_t
am_auth_create_auth_context(am_auth_context_t *auth_ctx,
                            const char *org_name,
                            const char *cert_nick_name,
                            const char *url);

/*
 * Destroys the given auth context handle.
 *
 * Parameters:
 *   auth_ctx   Handle of the auth context to be destroyed.
 *
 * Returns:
 *   AM_SUCCESS
 *               If the auth context was successfully destroyed.
 *
 *   AM_INVALID_ARGUMENT
 *               If the auth_ctx parameter is NULL.
 *
 */
AM_EXPORT am_status_t
am_auth_destroy_auth_context(am_auth_context_t auth_ctx);

/*
 * Starts the login process given the index type and its value.
 *
 * Parameters:
 *   auth_ctx   Handle of the auth context.
 *
 *   auth_idx   Index type to be used to initiate the login process.
 *
 *   value      Value corresponding to the index type.
 *
 * Returns:
 *   AM_SUCCESS
 *               If the login process was successfully completed.
 *
 *   AM_INVALID_ARGUMENT
 *               If the auth_ctx or value parameter is NULL.
 *
```

Code Example 3-25 am_auth.h C Authentication API Header File *(Continued)*

```
 *   AM_FEATURE_UNSUPPORTED
 *              If the auth_idx parameter is invalid.
 *
 */
AM_EXPORT am_status_t
am_auth_login(am_auth_context_t auth_ctx, am_auth_index_t auth_idx,
              const char *value);

/*
 * Logout the user.
 *
 * Parameters:
 *   auth_ctx   Handle of the auth context.
 *
 * Returns:
 *   AM_SUCCESS
 *              If the logout process was successfully completed.
 *
 *   AM_INVALID_ARGUMENT
 *              If the auth_ctx parameter is NULL.
 *
 */
AM_EXPORT am_status_t
am_auth_logout(am_auth_context_t auth_ctx);

/*
 * Abort the authentication process.
 *
 * Parameters:
 *   auth_ctx   Handle of the auth context.
 *
 * Returns:
 *   AM_SUCCESS
 *              If the abort process was successfully completed.
 *
 *   AM_INVALID_ARGUMENT
 *              If the auth_ctx parameter is NULL.
 *
 */
AM_EXPORT am_status_t
am_auth_abort(am_auth_context_t auth_ctx);

/*
 * Checks to see if there are requirements to be supplied to
 * complete the login process. This call is invoked after
 * invoking the login() call. If there are requirements to
 * be supplied, then the caller can retrieve and submit the
 * requirements in the form of callbacks.
 *
 * The number of callbacks may be retrieved with a call to
 * am_auth_num_callbacks() and each callback may be retrieved
 * with a call to am_auth_get_callback(). Once the requirements
 * for each callback are set, am_auth_submit_requirements() is
 * called.
 *
```

Code Example 3-25 am_auth.h C Authentication API Header File *(Continued)*

```
 * Repeat until done.
 *
 * Parameters:
 *   auth_ctx   Handle of the auth context.
 *
 * Returns:
 *   B_TRUE
 *              If there are more requirements.
 *   B_FALSE
 *              If there are no more requirements.
 *
 */
AM_EXPORT boolean_t
am_auth_has_more_requirements(am_auth_context_t auth_ctx);

/*
 * Gets the number of callbacks.
 *
 * Parameters:
 *   auth_ctx   Handle of the auth context.
 *
 * Returns:
 *      Number of callbacks.
 *
 */
AM_EXPORT size_t
am_auth_num_callbacks(am_auth_context_t auth_ctx);

/*
 * Gets the n-th callback structure.
 *
 * Parameters:
 *   auth_ctx   Handle of the auth context.
 *
 *   index      The index into the callback array.
 *
 * Returns:
 *      Returns a pointer to the am_auth_callback_t structure
 *      which the caller needs to populate.
 *
 */
AM_EXPORT am_auth_callback_t *
am_auth_get_callback(am_auth_context_t auth_ctx, size_t index);

/*
 * Submits the responses populated in the callbacks to the server.
 *
 * Parameters:
 *   auth_ctx   Handle of the auth context.
 *
 * Returns:
 *   AM_SUCCESS
 *              If the submitted requirements were processed
 *              successfully.
 *
```

Code Example 3-25 am_auth.h C Authentication API Header File *(Continued)*

```
 *   AM_AUTH_FAILURE
 *              If the authentication process failed.
 *
 *   AM_INVALID_ARGUMENT
 *              If the auth_ctx parameter is NULL.
 *
 */
AM_EXPORT am_status_t
am_auth_submit_requirements(am_auth_context_t auth_ctx);

/*
 *
 * Get the status of the authentication process.
 *
 * Parameters:
 *   auth_ctx   Handle of the auth context.
 *
 * Returns:
 *
 *   AM_AUTH_STATUS_FAILED
 *              The login process has failed.
 *
 *   AM_AUTH_STATUS_NOT_STARTED,
 *              The login process has not started.
 *
 *   AM_AUTH_STATUS_IN_PROGRESS,
 *              The login is in progress.
 *
 *   AM_AUTH_STATUS_COMPLETED,
 *              The user has been logged out.
 *
 *   AM_AUTH_STATUS_SUCCESS
 *              The user has logged in.
 *
 */
AM_EXPORT am_auth_status_t
am_auth_get_status(am_auth_context_t auth_ctx);

/*
 * Get the sso token id of the authenticated user.
 *
 * Parameters:
 *   auth_ctx   Handle of the auth context.
 *
 * Returns:
 *      A zero terminated string representing the sso token,
 *      NULL if there was an error or the user has not
 *      successfully logged in.
 *
 */
AM_EXPORT const char *
am_auth_get_sso_token_id(am_auth_context_t auth_ctx);

/*
 * Gets the organization to which the user is authenticated.
```

Code Example 3-25 am_auth.h C Authentication API Header File *(Continued)*

```
 *
 * Parameters:
 *   auth_ctx   Handle of the auth context.
 *
 * Returns:
 *      A zero terminated string representing the organization,
 *      NULL if there was an error or the user has not
 *      successfully logged in.
 *
 */
AM_EXPORT const char *
am_auth_get_organization_name(am_auth_context_t auth_ctx);

/*
 * Gets the authentication module/s instances (or plugins)
 * configured for an organization, or sub-organization name that
 * was set during the creation of the auth context.
 * Supply the address of a pointer to a structure
 * of type am_string_set_t. Module instance names are
 * returned in am_string_set_t. Free the memory
 * allocated for this set by calling am_string_set_destroy().
 *
 * Returns NULL if the number of modules configured is zero.
 *
 * Parameters:
 *   auth_ctx               Handle of the auth context.
 *   module_inst_names_ptr  Address of a pointer to am_string_set_t.
 *
 * Returns:
 *   AM_SUCCESS
 *              If the submitted requirements were processed
 *              successfully.
 *
 *   AM_AUTH_FAILURE
 *              If the authentication process failed.
 *
 *   AM_INVALID_ARGUMENT
 *              If the auth_ctx parameter is NULL.
 *
 *   AM_SERVICE_NOT_INITIALIZED
 *              If the auth service is not initialized.
 *
 */
AM_EXPORT am_status_t
am_auth_get_module_instance_names(am_auth_context_t auth_ctx,
                                  am_string_set_t** module_inst_names_ptr);

AM_END_EXTERN_C
#endif
```

Authentication Option For Other Applications

Applications written in a language other than Java or C can exchange authentication information with Identity Server using the XML / HTTP(s) interface. Using the URL `http://`*server_name.domain_name*`:`*port*`/`*service_deploy_uri*`/authservice`, an application can open a connection using the HTTP POST method and exchange XML messages with the Authentication Service. The structure of the XML messages is defined by "The remote-auth.dtd Structure" on page 135. In order to access the Authentication Service in this manner, the client application must contain the following:

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

XML Messages

The following code examples illustrate how customers might configure the XML messages posted to the Authentication Service.

NOTE Although the client application need only write XML based on the `remote-auth.dtd`, when these messages are sent they include additional XML code produced by the Authentication API. This additional XML code is not illustrated in the following examples.

Code Example 3-26 illustrates the initial XML message sent to the Identity Server. It opens a connection and asks for authentication requirements regarding the `exampleorg` organization to which the user will login.

Code Example 3-26 Initial AuthContext XML Message

```
<?xml version="1.0" encoding="UTF-8"?>
<AuthContext version="1.0">
<Request authIdentifier="0">
<NewAuthContext orgName="dc=exampleorg,dc=com">
</NewAuthContext>
</Request>
</AuthContext>
```

Code Example 3-27 illustrates the successful response from Identity Server that contains the `authIdentifier`, the session identifier for the initial request.

Code Example 3-27 AuthIdentifier XML Message Response

```
<?xml version="1.0" encoding="UTF-8"?>
<AuthContext version="1.0">
<Response
authIdentifier="AQIC5wM2LY4SfcwmVdbgTX+9WzyWSPlWjb1oVb5esqDlkaY=">
<LoginStatus status="in_progress">
</LoginStatus>
</Response>
</AuthContext>
```

Code Example 3-28 illustrates the client response message back to Identity Server. It specifies the type of authentication module needed by the user to log in.

Code Example 3-28 Second Request Message With Authentication Module Specified

```
<?xml version="1.0" encoding="UTF-8"?>
<AuthContext version="1.0">
<Request authIdentifier="AQIC5wM2LY4SfcwmVdbgTX+9WzyWSPlWjb1oVb5esqDlkaY=">
<Login>
<IndexTypeNamePair indexType="moduleInstance">
<IndexName>LDAP</IndexName>
</IndexTypeNamePair>
</Login>
</Request>
</AuthContext>
```

Code Example 3-29 illustrates the return message from Identity Server which specifies the authentication module's login requirements. In this case, the LDAP requirements include a user name and password. Note the page time out value of 120 seconds.

Code Example 3-29 Return XML Message With Login Callbacks

```
<?xml version="1.0" encoding="UTF-8"?>
<AuthContext version="1.0">
<Response
authIdentifier="AQIC5wM2LY4SfcwmVdbgTX+9WzyWSPlWjb1oVb5esqDlkaY=">
```

Code Example 3-29 Return XML Message With Login Callbacks *(Continued)*

```
<GetRequirements>
<Callbacks length="3">
<PagePropertiesCallback isErrorState="false">
<ModuleName>LDAP</ModuleName>
<HeaderValue>This server uses LDAP Authentication</HeaderValue>
<ImageName></ImageName>
<PageTimeOut>120</PageTimeOut>
<TemplateName></TemplateName>
<PageState>1</PageState>
</PagePropertiesCallback>
<NameCallback>
<Prompt>User Name: </Prompt>
</NameCallback>

<PasswordCallback echoPassword="false">
<Prompt> Password: </Prompt>
</PasswordCallback>

</Callbacks>
</GetRequirements>
</Response>
</AuthContext>
```

Code Example 3-30 illustrates the client responses to the call for login requirements. They specify `amadmin` as the user and `11111111` for the password.

Code Example 3-30 Response Message With Callback Values

```
<?xml version="1.0" encoding="UTF-8"?>
<AuthContext version="1.0">
<Request authIdentifier="AQIC5wM2LY4SfcwmVdbgTX+9WzyWSPlWjb1oVb5esqDlkaY=">
<SubmitRequirements>
<Callbacks length="3">

<NameCallback>
<Prompt>User Name:</Prompt>
<Value>amadmin</Value>
</NameCallback>

<PasswordCallback echoPassword="false">
<Prompt>Password:</Prompt>
<Value>11111111</Value>
</PasswordCallback>
</Callbacks>
</SubmitRequirements>
</Request>
</AuthContext>
```

Code Example 3-31 illustrates that a successful authentication has occurred. As the value of `<Subject>` uses the Java serialization, it can not be used by non-Java client applications. It's value is retreived by all applications from the session token.

Code Example 3-31 Successful Authentication XML Message

```
<?xml version="1.0" encoding="UTF-8"?>
<AuthContext version="1.0">
<Response
authIdentifier="AQIC5wM2LY4SfcwmVdbgTX+9WzyWSPlWjb1oVb5esqDlkaY=">
<LoginStatus status="success"
ssoToken="AQIC5wM2LY4SfcwmVdbgTX+9WzyWSPlWjb1oVb5esqDlkaY="
successURL="http://torpedo.red.iplanet.com:/amconsole">
<Subject>AQICweczOhuelZ5TqD9kKOtiAepxqGP23q4oTnNMuJY//lI2S4KD1/gEN84uLwDGHl
llyFSthxoKLM7NDH
h2vwAvrDmpsomJvUnbqnJJ90DS+28njGiDv+lv8FqIVhhbxrctbiIUEOHYK0FzXnXjPYizdCmiW
XJ+9DJ8T2HbYIDxn9U6eVNAMPq3uVb/RFuErEm5MuPu7PnWeCic12SZre4ZEcw8TI45NKNjd/NZ
ZD97bcqL5gEV7SVHspFldZKmo9vA86aEkvMs9P53RiJtrusHN1FKt9+4JqSrdcVLKMzJVAr3z5E
ohwHh9/hzd7hguc0661gz7IqkT7WEpve/E8R4em0mg3HgHg7Bg7i3AkyX6YSkoAncdVXMdmWnb7
OV5cBgUjO8zs8Pp5/3dA1XlwACmOqjxshk6Y6Ld6TAQ90qRFwymC1RdLGGCRnrt33kmYVyB1lJy
JxT8utPKyDOEKFRHh57NlKTbFhBKc1IGcdQ2crHifpXawx6YouQgQSWGdsqW9IahY4+lqbBTPnG
DyZkKz9yy2ZKVjDR05Hwku8elvEwBE40XTJ3gF/mbwCGbh3cyprahLqRXboy8eoEQf3ubQmR2My
+bh+NrsRfzfFV5oCcpJE6DtvYE/4zO+uKk3FbG+/NUJzAAor920V/0prtYeS58ZPW8C7qwXINaW
0xdMQV+pgE3NZvMlp5GeZlSIMmSCtXD49n4tqopSlsoK+eiwPODKxp992+6/uJhhVHH5I0Ozuy6
CDM
dCJDGvnMENVCUZvki3+tb92fqQbVWixM4Ca6Nnz3jTIKk2uhm559jq9hra8gHHOfnnu4e5jZjzf
RdkO3GodiTMOHDnQATHtvT1PBXgorTfUwUa4ZjptvzFulHSi4eQaqs4Z8FAX2OAr8XGHRkhBwox
rhjYiCDBpkNmpEiFNhWnTT3bwkAUFhtoDg6836kwHfxeLXKAz3T6qyNQzT+larSXUxrt/TIjwDP
R3vg4GF4RzbHlWA1WQtUS/9Qe/N3aegEEEvxPvo9fWq</Subject>
</LoginStatus>
</Response>
</AuthContext>
```

Service Programming Interfaces

Identity Server provides the capability to plug new, Java-based authentication modules into its framework allowing proprietary authentication providers to be managed using the Identity Server console. Custom authentication modules must first be created using Java and, once created, can be added to the list of available authentication modules.

NOTE For instructions on other files needed to plug-in a custom authentication module, see "Integrating A Custom Authentication Module" on page 142.

New authentication modules are added by using the `com.iplanet.authentication.spi` package. The SPI implements the JAAS `LoginModule`, and provides additional methods to access the Authentication Service and module configuration properties files. Because of this architecture, any custom JAAS authentication module will work within the Authentication Service.

NOTE This guide does not document the JAAS. For more information on these APIs, see the *Java Authentication And Authorization Service Developer's Guide*. Additional information can be found at `http://java.sun.com/products/jaas/`.

Implementing A Custom Authentication Module

Custom authentication modules extend the `com.sun.identity.authentication.spi.AMLoginModule` class. The class must also implement the `init()`, `process()` and `getPrincipal()` methods in order to communicate with the authentication module configuration files. The callbacks are then dynamically generated based on this file. Other methods that can be defined include `setLoginFailureURL` and `setLoginSuccessURL` which defines URLs to send the user to based on a failed or successful authentication, respectively.

NOTE To make use of the account locking feature with custom authentication modules, the `InvalidPasswordException` exception should be thrown when the password is invalid. For more information on account lockout, see "Account Locking" on page 121.

Identity Server contains a sample exercise for integrating a custom authentication module with files that have already been created. This sample documents the procedure to integrate an authentication module into the Identity Server deployment. All the files needed to compile, deploy and run the sample authentication module can be found in the *IdentityServer_base*`/SUNWam/samples/authentication/providers` directory. The instruction file is the `Readme.html` file in the same directory. The following sections will use files from this sample as example code.

Implement The Principal Class

After following the procedures in "Configuring The Authentication Module" on page 144, the next step is to write a Principal class which implements `java.security.Principal`; in the sample, the constructor takes the username as an argument and represents the authenticated user. If authentication is successful, the module will return this principal to the Authentication Service which extracts information for inclusion into the session token. Code Example 3-32 illustrates this implementation with the `SamplePrincipal.java` code.

Code Example 3-32 SamplePrincipal.java Code

```
package com.iplanet.am.samples.authentication.providers;

import java.io.IOException;

import javax.security.auth.*;
import javax.security.auth.login.*;
import javax.security.auth.callback.*;

import java.security.Principal;


public class SamplePrincipal implements Principal, java.io.Serializable {

/**
* @serial
*/
    private String name;


    public SamplePrincipal(String name) {
        if (name == null)
            throw new NullPointerException("illegal null input");

        this.name = name;
    }

/**
* Return the LDAP username for this <code>SamplePrincipal</code>.
*
* <p>
*
* @return the LDAP username for this <code>SamplePrincipal</code>
*/
    public String getName() {
        return name;
    }

/**
* Return a string representation of this
*<code>SamplePrincipal</code>.
* <p>
*
```

Code Example 3-32 SamplePrincipal.java Code *(Continued)*

```
* @return a string representation of this
* <code>SamplePrincipal</code>.
*/
    public String toString() {
        return("SamplePrincipal:  " + name);
    }

/**
* Compares the specified Object with this
* <code>SamplePrincipal</code> for equality. Returns true if
* the given object is also a <code>SamplePrincipal</code>
* and the two SamplePrincipals have the same username.
*
* <p>
*
* @param o Object to be compared for equality with this
*          <code>SamplePrincipal</code>.
*
* @return true if the specified Object is equal equal to this
*          <code>SamplePrincipal</code>.
*/
    public boolean equals(Object o) {
        if (o == null)
            return false;

        if (this == o)
            return true;

        if (!(o instanceof SamplePrincipal))
            return false;
        SamplePrincipal that = (SamplePrincipal)o;

        if (this.getName().equals(that.getName()))
            return true;
        return false;
    }

/**
* Return a hash code for this <code>SamplePrincipal</code>.
*
* <p>
*
* @return a hash code for this <code>SamplePrincipal</code>.
*/
    public int hashCode() {
        return name.hashCode();
    }
}
```

Implement The LoginModule Interface

Following the implementation of the Principal Class, `com.sun.identity.authentication.spi.AMLoginModule` must be subclassed and the `init()`, `process()`, and `getPrincipal()` methods called. `AMLoginModule` is an abstract class that provides the methods to access Sun ONE Identity Server services and the authentication module configuration file. (`AMLoginModule` implements the JAAS `LoginModule`.)

public void init(Subject subject, Map sharedState, Map options); `init()` is an abstract method used to initialize the `LoginModule` with the relevant information. If the `LoginModule` does not understand some of the data stored in the `sharedState` or `options` parameters, it will be ignored. This method is called by `AMLoginModule` after the module has been instantiated, and prior to any calls to its other public methods. The method implementation should store away the provided arguments for future use. The `init()` method may peruse the provided `sharedState` to determine what additional authentication states were provided by other LoginModules; it may also traverse through the provided options to determine what configurations were defined to affect the LoginModule's behavior. It may save option values in variables for future use.

public int process(javax.security.auth.callback.Callback[] callbacks, int state) The `process()` method is called to authenticate a Subject. This method implementation performs the actual authentication. For example, it may prompt for a user name and password, and attempt to verify the password against a database. If the LoginModule requires some form of user interaction (retrieving a user name and password, for example), it should be accomplished here. (LoginModules should remain independent of the different types of user interaction.) The `process()` method should invoke the `handle` method of the javax.security.auth.callback to perform the user interaction and set the appropriate results (user name, password, etc.). The `AMLoginModule` will then internally pass an array of appropriate Callbacks (a `NameCallback` for the user name, a `PasswordCallback` for the password, et. al.) to the Authentication User Interface which will perform the requested user interaction and set the appropriate values in the Callbacks. Consider the following steps when writing the `process()` method.

1. Perform the authentication.
2. If the authentication is successful, save the principal data.
3. Return -1 if the authentication succeeds.
4. Throw a `LoginException` (such as AuthLoginException) or return the relevant state specified in the authentication module configuration file if the authentication fails.

5. If multiple states are available to the user, the Callback array from a previous state may be retrieved by using the `getCallback(int state)` method. The underlying login module keeps the `Callback[]` from the previous states until the login process is completed.

6. If a module writer needs to substitute dynamic text in the next state, the writer could use the `getCallback()` method to get the `Callback[]` for the next state and modify the output text or prompt. Calling `replaceCallback()` will update the Callback array. This allows a module writer to dynamically generate challenges, passwords or user IDs.

NOTE Each authentication session creates a new instance of the LoginModule class. The reference to the class will be released once the authentication session has either succeeded or failed. It is important to note that any static data or reference to any static data in LoginModule must be thread-safe.

public Principal getPrincipal(); This method should be called only once at the end of a successful authentication session. A login session is deemed successful when all pages in the authentication module's credentials file have been sent and the module has not thrown an exception. This method retrieves the authenticated token string that the user will be known by in the Identity Server environment.

Code Example 3-33 illustrates the LoginModuleSample.java code.

Code Example 3-33 LoginModuleSample.java Code

```
package com.iplanet.am.samples.authentication.providers;

import java.util.Map;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.login.LoginException;
import com.sun.identity.authentication.spi.AMLoginModule;
import com.sun.identity.authentication.spi.AuthLoginException;

               public class LoginModuleSample extends AMLoginModule {

                   private String userTokenId;
                   private String userName;
                   private String lastName;
                   private java.security.Principal userPrincipal = null;

                   public LoginModuleSample() throws LoginException{
                       System.out.println("LoginModuleSample()");
                   }
```

Code Example 3-33 LoginModuleSample.java Code *(Continued)*

```
                   public void init(Subject subject, Map sharedState, Map
options) {
                    System.out.println("LoginModuleSample initialization");
                   }

                  public int process(Callback[] callbacks, int state) throws
AuthLoginException {
                        int currentState = state;

                        if (currentState == 1) {
                         userName = ((NameCallback) callbacks[0]).getName();
                            lastName =  ((NameCallback)
callbacks[1]).getName();
                          if (userName.equals("") || lastName.equals("")) {
                               throw new AuthLoginException("names must not
be empty");
                            }
                            return 2;
                        } else if (currentState == 2) {
                            System.out.println("Replace TExt first: " +
userName +
                                " last: " + lastName);
                            // set #REPLACE# text in next state
                            Callback[] callbacks2 = getCallback(3);
                            String msg =
((NameCallback)callbacks2[0]).getPrompt();
                            int i = msg.indexOf("#REPLACE#");
                           String newMsg = msg.substring(0, i) + userName +
                                msg.substring(i+9);
                           replaceCallback(3, 0, new NameCallback(newMsg));
                            // set #REPLACE# in password callback
                            msg =
((PasswordCallback)callbacks2[1]).getPrompt();
                            i = msg.indexOf("#REPLACE#");
                            newMsg = msg.substring(0, i) + lastName +
msg.substring(i+9);
                          replaceCallback(3, 1, new PasswordCallback(newMsg,
false));
                            return 3;
                        } else if (currentState == 3) {
                            int len = callbacks.length;
                            for (int i=0; i<len; i++) {
                                if (callbacks[i] instanceof NameCallback) {
                                   System.out.println("Callback Prompt-> " +
                                        ((NameCallback)
callbacks[i]).getPrompt());
                                } else if (callbacks[i] instanceof
PasswordCallback) {
                                   System.out.println("Callback Prompt-> " +
                                        ((PasswordCallback)
callbacks[i]).getPrompt());
                                }
```

Code Example 3-33 LoginModuleSample.java Code *(Continued)*

```
                           }
                           return 4;
                       } else if (currentState == 4) {
                           int len = callbacks.length;
                           for (int i=0; i<len; i++) {
                               if (callbacks[i] instanceof NameCallback) {
                                   System.out.println("Callback Value-> " +
                                       ((NameCallback)
callbacks[i]).getName());
                               } else if (callbacks[i] instanceof
PasswordCallback) {
                                   System.out.println("Callback Value-> " +
                                       ((PasswordCallback)
callbacks[i]).getPassword());
                               }
                           }
                           userTokenId = userName;
                           // return -1 for login successful
                           return -1;
                       }
                       throw new AuthLoginException("Invalid state : " +
currentState);
                   }

                   public java.security.Principal getPrincipal() {
                       if (userPrincipal != null) {
                           return userPrincipal;
                       } else if (userTokenId != null) {
                          userPrincipal = new SamplePrincipal(userTokenId);
                           return userPrincipal;
                       } else {
                           return null;
                       }
                   }
               }
```

Implementing A Pure JAAS Module

Identity Server supports pure JAAS pluggable authentication modules. Pure JAAS modules extend the JAAS `LoginModule` rather than `AMLoginModule`. Details on how to write a JAAS module can be found in the LoginModule Developer's Guide. A pure JAAS module is plugged in to the Authentication framework using the Authentication API as detailed in "Implementing A Custom Authentication Module" on page 171. Code Example 3-34 is a sample of how the code can be written.

Code Example 3-34 JAAS LoginModule Sample Code

```
package com.sun.identity.authentication.modules.sample;
               import java.util.*;
               import java.io.IOException;
               import javax.security.auth.*;
               import javax.security.auth.callback.*;
               import javax.security.auth.login.*;
               import javax.security.auth.spi.*;
               import com.iplanet.am.util.*;
/**
* <p> This sample LoginModule authenticates users with password.
* <p> This LoginModule only recognizes one user: testUser
* <p> testUser's password is: testPassword
* <p> If testUser successfully authenticates itself,
* a <code>SamplePrincipal</code> with the testUser's user name
* is added to the Subject.
* <p> This LoginModule recognizes the debug option.
* If set to true in the login Configuration,
* debug messages will be output to the output stream, System.out.
*/
               public class SampleLoginModule implements LoginModule {

                   // initial state
                   private Subject subject;
                   private CallbackHandler callbackHandler;
                   private Map sharedState;
                   private Map options;
                   private com.iplanet.am.util.Debug debug;
                   // configurable option

                   // the authentication status
                   private boolean succeeded = false;
                   private boolean commitSucceeded = false;

                   // username and password
                   private String username;
                   private char[] password;

                   // testUser's SamplePrincipal
                   private SamplePrincipal userPrincipal;

/**
* Initialize this <code>LoginModule</code>.
* <p>
* @param subject the <code>Subject</code> to be authenticated.
* @param callbackHandler a <code>CallbackHandler</code> for
* communicating with the end user (prompting for user names and
* passwords, for example). <p>
*
* @param sharedState shared <code>LoginModule</code> state. <p>
*
* @param options options specified in the login
* <code>Configuration</code> for this particular
* <code>LoginModule</code>.
*/
```

Code Example 3-34 JAAS LoginModule Sample Code *(Continued)*

```
                   public void initialize(Subject subject, CallbackHandler
callbackHandler,
                       Map sharedState, Map options) {

                       debug =
com.iplanet.am.util.Debug.getInstance("SampleLoginModule");
                       this.subject = subject;
                       this.callbackHandler = callbackHandler;
                       this.sharedState = sharedState;
                       this.options = options;

                       // initialize any configured options
                       debug.message("SampleLoginModule - initialize");
                   }

 /**
* Authenticate the user by prompting for a user name and password.
* <p>
* @return true in all cases since this <code>LoginModule</code>
* should not be ignored.
* @exception FailedLoginException if the authentication fails. <p>
* @exception LoginException if this <code>LoginModule</code>
* is unable to perform the authentication.
*/
                   public boolean login() throws LoginException {
                       // prompt for a user name and password
                       if (callbackHandler == null) {
                        throw new LoginException("Error: no CallbackHandler
available " +
                             "to garner authentication information from the
user");
                       }

                       Callback[] callbacks = new Callback[5];
                       callbacks[0] = new NameCallback("user name: ");
                       callbacks[1] = new PasswordCallback("password: ",
false);
                       LanguageCallback ll = new LanguageCallback();
                       TextOutputCallback toc = new TextOutputCallback
                           (TextOutputCallback.INFORMATION,"This is Sample
Login Module");
                       java.util.Locale locale = new
java.util.Locale("zh","TW");
                       TextInputCallback tic = new TextInputCallback("Enter
your name","abc");
                       tic.setText("fdfdfdfd");

                       ll.setLocale(locale);
                       callbacks[2] = ll;
                       callbacks[3] = toc;
                       callbacks[4] = tic;

                       try {
```

Code Example 3-34 JAAS LoginModule Sample Code *(Continued)*

```
                           callbackHandler.handle(callbacks);
                          username = ((NameCallback)callbacks[0]).getName();
                           char[] tmpPassword =
((PasswordCallback)callbacks[1]).getPassword();
                           java.util.Locale l = ((LanguageCallback)
callbacks[2]).getLocale();
                           String input = ((TextInputCallback)
callbacks[4]).getPrompt();
                           debug.message("username is .. :" + username);

                           if (l != null) {
                             debug.message("locale is .. :" + l.toString());
                           }

                           if (input != null) {
                               debug.message("User has entered" + input);
                           }
                           return true;
                       } catch (java.io.IOException ioe) {
                           throw new LoginException(ioe.toString());
                       } catch (UnsupportedCallbackException uce) {
                           throw new LoginException("Error: " +
uce.getCallback().toString() +
                               " not available to garner authentication
information " +
                               "from the user");
                       } catch (Exception ex) {
                           ex.printStackTrace();
                           throw new LoginException(ex.toString());
                       }
                   }

/**
* <p> This method is called if the LoginContext's
* overall authentication succeeded
* (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL
* LoginModules succeeded).
*
* <p> If this LoginModule's own authentication attempt
* succeeded (checked by retrieving the private state saved by the
* <code>login</code> method), then this method associates a
* <code>SamplePrincipal</code>
* with the <code>Subject</code> located in the
* <code>LoginModule</code>. If this LoginModule's own
* authentication attempted failed, then this method removes
* any state that was originally saved.
*
* <p>
*
* @exception LoginException if the commit fails.
*
* @return true if this LoginModule's own login and commit
* attempts succeeded, or false otherwise.
*/
```

Code Example 3-34 JAAS LoginModule Sample Code *(Continued)*

```
                   public boolean commit() throws LoginException {
                     SamplePrincipal principal = new SamplePrincipal("abc");

                       if (debug.messageEnabled()) {
                           debug.message("commit.... SUCCESSFULE..." +
principal.toString());
                       }

                       if (principal != null &&
                           !subject.getPrincipals().contains(principal))
                       {
                           subject.getPrincipals().add(principal);
                           debug.message ("Done added user to principal");
                       }
                           return true;
                   }

/**
* <p> This method is called if the LoginContext's
* overall authentication failed.
* (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL
* LoginModules did not succeed).
*
* <p> If this LoginModule's own authentication attempt
* succeeded (checked by retrieving the private state saved by the
* <code>login</code> and <code>commit</code> methods),
* then this method cleans up any state that was originally saved.
*
* @exception LoginException if the abort fails.
*
* @return false if this LoginModule's own login and/or commit
* attempts failed, and true otherwise.
*/
                   public boolean abort() throws LoginException {
                       return true;
                   }

/**
* Logout the user.
*
* <p> This method removes the <code>SamplePrincipal</code>
* that was added by the <code>commit</code> method.
 *
* @exception LoginException if the logout fails.
*
* @return true in all cases since this <code>LoginModule</code>
* should not be ignored.
*/
                   public boolean logout() throws LoginException {
                       return true;
                   }
               }
```

Implementing Authentication Post Processing

The Authention SPI includes the `AMPostAuthProcessInterface` which can be implemented for post-processing tasks. (`AMPostLoginProcessInterface` from previous versions of Identity Server has been deprecated.) `AMPostAuthProcessInterface` can be executed on either successful or failed authentication or on logout. A post-processing task might include adding attributes to a user's session after successful authentication, sending notification to an administrator after failed authentication, or general clean-up after logout (i.e.: clearing cookies or logging out other system components). The Core Authentication Service contains the Authentication PostProcessing Class attribute which contains the authentication post-processing class name as its value. Custom post processing interfaces can also be implemented. Code Example 3-35 is sample code that uses the post-processing interface to adding the property TEST1 and its value TESTVALUE1 to the user's session token.

Code Example 3-35 Sample Code For Authentication Post Processing

```
//package com.sun.identity.authentication.spi;

import com.sun.identity.authentication.spi.*;
import com.iplanet.sso.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.iplanet.services.util.Debug;

/**
 * The <code>AMPostAuthProcessInterface</code> interface needs to
 * be implemented by services and applications to do post
 * authentication processing.
 * The post processing can be per ORGANIZATION or SERVICE or ROLE
 */

public class SampleSetupSession implements AMPostAuthProcessInterface
{

   private static Debug debug = Debug.getInstance("sampleSessionSetup");

  /** Post processing on successful authentication.
    * @param requestParamsMap - map contains HttpServletRequest parameters
    * @param ssoToken - user's session
    * @exception Authentication Exception when there is an error
    */

  public void onLoginSuccess(Map requestParamsMap,
                             HttpServletRequest request,
                             HttpServletResponse response,
                             SSOToken ssoToken)
        throws AuthenticationException
  {
      debug.message("SampleSetupSession:onLoginSuccess called");
```

Code Example 3-35 Sample Code For Authentication Post Processing *(Continued)*

```
      try {
          ssoToken.setProperty("TEST1", "TESTVALUE1");
      } catch (Exception ex) {
          debug.message("SampleSetupSession:onLoginSuccess exception while 
setting 
property :"+ex);
      }
  }

  /** Post processing on failed authentication.
    * @param requestParamsMap - map contains HttpServletRequest parameters
    * @exception AuthenticationException when there is an error
    */
  public void onLoginFailure(Map requestParamsMap,
                             HttpServletRequest req,
                             HttpServletResponse res)
        throws AuthenticationException
  {
      debug.message("SampleSetupSession:onLoginFailure called");
  }


 /** Post processing on Logout.
   * @param requestParamsMap - map contains HttpServletRequest parameters
   * @param HttpServletRequest - Servlet request
   * @param HttpServletResponse - Servlet response
   * @param ssoToken - user's session
   */

 public void onLogout(HttpServletRequest req,
                      HttpServletResponse res,
                      SSOToken ssoToken)
       throws AuthenticationException
  {
      debug.message("SampleSetupSession:onLogout called");
  }
}
```

SPI Sample

The *IdentityServer_base*/`SUNWam/samples/authentication/providers` directory provides two sample source code files, `LoginModuleSample.java` and `SamplePrincipal.java`, that are used to illustrate how to integrate a sample login module into the Identity Server deployment. The instruction file is the `Readme.html` file in the same directory.

Authentication Samples

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

- Certificate Authentication Sample
- LDAP Authentication Sample
- MSISDN (Wireless) Module
- SPI Sample

Certificate Authentication Sample

The *IdentityServer_base*/`SUNWam/samples/authentication/Cert` directory provides a sample source code file, `CertLogin.java` file that illustrates a Java application which utilizes digital certificates for authentication. The instruction file is the `readme.html` file in the same directory.

LDAP Authentication Sample

The *IdentityServer_base*/`SUNWam/samples/authentication/LDAP` directory provides a sample source code file, `LDAPLogin.java` file that illustrates a Java application which authenticates to the LDAP module. This sample can be easily modified to authenticate to other existing or customized authentication modules. The instruction file is the `readme.html` file in the same directory.

MSISDN (Wireless) Module

This sample details the steps to integrate a custom login module based on the Mobile Station Integrated Services Digital Network (MSISDN). This module will make it possible to perform non-interactive authentication based upon the unique ID of a particular handset. All the files needed to compile, deploy and run the module can be found in the *IdentityServer_base*/`SUNWam/samples/authentication/msisdn` directory. The instruction file is the `Readme.html` file in the same directory.

SPI Sample

This sample details the steps to integrate a sample login module into the Identity Server deployment. All the files needed to compile, deploy and run the sample authentication module can be found in the *IdentityServer_base*/`SUNWam/samples/authentication/providers` directory. The instruction file is the `Readme.html` file in the same directory.

Chapter 4

Single Sign-On And Sessions

The Session Service is a key component of the Sun™ ONE Identity Server single sign-on (SSO) solution that enables users to authenticate once yet 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 Session Service, the SSO solution, and the SSO APIs. It contains the following sections:

- Overview
- Cookies and Sessions
- Session Structure
- Cross-Domain Support For SSO
- SSO API
- SSO Samples

Overview

A user wanting to access resources protected by Identity Server 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 access privileges or *policy*. If a user wants to access *several* resources protected by Identity Server, the Session Service provides proof of authorization so there is no need to re-authenticate; this is *single sign-on*. As different DNS domains generally have common users who need to gain access to their services in a single session, Identity Server supports a *cross-domain single sign-on* functionality.

NOTE In an Identity Server deployment, all Identity Server instances must be located in one primary cookie domain. The deployment may have multiple instances for high-availability but they may not be located in multiple DNS domains.

The Session Service provides the functionality to maintain information about an authenticated user's session across all applications participating in a single sign-on. It is responsible for:

- Generating session identifiers.
- Maintaining a master copy of the session's state information.
- Implementing the time-dependent behavior of sessions.
- Implementing the session's life cycle events (i.e.: logout, session destruction).
- Generating the session's life cycle event notifications.
- Implementing session failover facilities.

NOTE The *Sun ONE Identity Server Deployment Guide* contains a detailed section explaining the complete life cycle of a user session.

Session Service Concepts

The following concepts are closely tied together when discussing the Session Service and SSO. To understand the differences between them, consider the following definitions and how they will be used in this chapter.

Session

A *session* is a data structure held in the Identity Server memory that contains session information about an authenticated user.

Session ID

A *session identifier* (ID) is an opaque, globally unique string that programmatically identifies a specific session instance. With the session ID, a resource is able to retreive session information.

SSOToken

An SSOToken is a data structure, defined by the SSO API, that represents a snapshot of the session local to the particular application's memory.

Single Sign-On Process

The next sections describe the process that occurs when a user attempts to gain access to a resource protected by Identity Server.

Contacting A Protected Resource

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 and, inspects it to see if it contains a Session ID. If none exists, the request is redirected to Identity Server where the Session Service creates a Session for the requesting user. Initially, the session is in an *invalid* state and does not contain user identity information. It does though contain the aforementioned randomly-generated session ID to represent the user's session. Once the session / session ID is created, the Authentication Service sets a cookie with the session ID only and sends it to the client browser. Simultaneously, a login page is generated by the Authentication Service and returned to the user based upon their configured method of authentication (LDAP, RADIUS, etc.).

NOTE For more information on the different methods of authentication, see "Authentication Methods" on page 103 in Chapter 3, "Authentication Service," of this manual.

Providing User Credentials

The user, having received the login page (as well as the session ID) fills in the appropriate credentials based on the type of authentication. After entering 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 (using the session ID) and the session state is set to *valid*. The user can now be redirected to the resource they were attempting to access.

NOTE In reality, the user can only be redirected to the resource if their assigned policy permits it. More information on the Policy Service can be found in Chapter 7, "Policy Service," of this manual.

Cookies and Sessions

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 by which it has been generated. It does not imply that the user is authenticated. Cookies are domain-specific; for example, a cookie generated by DomainA cannot be used in DomainB. Cookies will only be passed to a server in the domain for which the cookie is set. Conversely, servers may only set a cookie in their own domain.

In an Identity Server deployment, the cookie contains the Session ID, an encrypted string generated by the Session Service. With the session ID, a protected resource can get access to the Session where the user's session information is stored. This information is then used for session validation.

NOTE Details on the attributes stored in the session token can also be found in "Authentication Methods" on page 103 in Chapter 3, "Authentication Service," of this manual.

Session Structure

When a user is successfully authenticated they are assigned a valid session. This session contains a number of attributes and properties that define the user's identity and some time-dependent behaviours (for example, the maximum time before the session expires). The following sections detail these attributes.

NOTE The values of most of these attibutes and properties are set by services other than the Session Service (primarily, the Authentication Service). The Session Service only provides storage for session information and enforces some of the time-dependent behaviour.

Fixed Attributes

The session token contains the following fixed attributes concerning the authenticated user:

- `ID`—This is the Session ID, a randomly-generated session identifier.
- `ClientDomain`—This is the DNS domain in which the client is located.
- `ClientID`—This is the user DN or the application's principal name.

- `Type`—This is the user or application type.
- `State`—This is the state of the session: valid, invalid, destroyed or inactive.
- `maxIdleTime`—This is the maximum time in minutes without activity before the session will expire and the user must reauthenticate.
- `maxSessionTime`—This is the maximum time in minutes before the session expires and the user must reauthenticate.
- `maxCachingTime`—This is the maximum time in minutes before the client contacts Identity Server to refresh cached session information.
- `latestAccessTime`—This is the last time the user has accessed the resource.
- `creationTime`—This is the time at which the session token was set to a valid state.

Protected And Custom Properties

The session token also contains an extensible set of properties that are divided into two subsets: protected (or core) properties and custom properties. Protected properties are set by Identity Server. Custom properties are set remotely by any application that knows the Session ID.

Protected Properties

The current protected properties are:

- `Organization`—This is the DN of the organization to which the user belongs.
- `Principal`—This is the DN of the user.
- `Principals`—This is a list of names to which the user has authenticated. (This property may have more then one value defined as a pipe separated list.)
- `UserId`—This is the user's DN as returned by the module, or in the case of modules other than LDAP or Membership, the user name. (All `Principals` must map to the same user. The UserID is the user DN to which they map.)
- `UserToken`—This is a user name. (All `Principals` must map to the same user. The UserToken is the user name to which they map.)
- `Host`—This is the host name or IP address for the client.
- `authLevel`—This is the highest level to which the user has authenticated.

- AuthType—This is a pipe separated list of authentication modules to which the user has authenticated (i.e. module1|module2|module3).
- Role—Applicable for role-based authentication only, this is the role to which the user belongs.
- Service—Applicable for service-based authentication only, this is the service to which the user belongs.
- loginURL—This is the client's login URL.
- Hostname—This is the host name of the client.
- cookieSupport—This attribute contains a value of true if the client browser supports cookies.
- authInstant—This is a string that specifies the time at which the authentication took place.
- SessionTimedOut—This attribute contains a value of true if the session has timed out.

Custom Properties

The custom properties currently used are:

- clientType—This is the device type of the client browser.
- Locale—This is the locale of the client.
- CharSet—This is the determined character set for the client.

Cross-Domain Support For SSO

Identity Server supports cross-domain SSO. A user authenticated to Identity Server in one DNS domain can access resources in another, integrated DNS domain. This cross-domain functionality is achieved using the Cross-Domain Controller servlet in Identity Server and Policy Agents installed in web containers. The Controller communicates with the policy agent that resides on servers where the protected resources are kept.

NOTE The Authentication Service handles SSO requests while the Cross-Domain Controller servlet handles cross-domain SSO requests.

Policy Agents

A *policy agent* polices the web container on which a protected resource lives by enforcing a user's assigned policies. They are an integral part of the cross-domain SSO functionality. Two types of policy agents are supported by Identity Server: the web agent and the J2EE/Java agent. The web agent enforces URL-based policy while the J2EE/Java agent enforces J2EE-based security and policy. Both types are available for installation separately from Identity Server and can be downloaded. Additional information can be found in the *Sun ONE Identity Server Web Policy Agents Guide* and *J2EE Policy Agents Guide* . General information on the Policy Service can be found in Chapter 7, "Policy Service," of this manual.

Cross-Domain Controller

The Cross-Domain Controller is a servlet responsible for redirecting user requests. The default URL for it is `http(s)://identity_server_host.domain_name:port/amserver/cdcservlet` . There are three scenarios where the Controller comes into play:

1. If a request for a protected resource contains no session ID, the agent redirects the user to the Controller which, in turn, redirects the user to the appropriate Authentication Service module. Assuming the user is authenticated, this scenario would then follow the path outlined in either Step 2 or Step 3.

NOTE	The authentication process itself is discussed in Chapter 3, "Authentication Service," of this manual.

2. If a request for a protected resource already contains a session ID set in a cookie for the same DNS domain in which the resource is deployed, the agent retrieves it and sends an XML/HTTP request to the Naming, Session and Policy Services to retrieve the identity, session and policy information for the requesting user. The user is allowed or denied access to the resource based on this information.

3. If a request for a protected resource does not contain a session ID set in a cookie for the same DNS domain in which the resource is deployed (i.e.: it carries a session ID set in a different DNS domain from the one in which the Identity Server is deployed), the agent redirects the request to the Controller with a Liberty AuthnRequest in the query string. The Controller then finds the session ID, extracts it from the cookie, places it in a Liberty AuthnResponse and sends it back to the agent. The agent finds the session ID, extracts it from the

AuthnResponse, sets it in a cookie for the new domain, and sends an XML/HTTP request to the Naming, Session and Policy Services to retrieve the identity, session and policy information for the requesting user. The user is allowed or denied access to the resource based on this information.

NOTE The Liberty AuthnRequest and AuthnResponse are part of the "Federation Management Protocols." More information can be found in Chapter 9, "Federation Management," of this manual.

A Cross-Domain SSO Scenario

In one scenario, the Identity Server instance for DomainA is its authentication provider. A user authenticates to Identity Server in DomainA and, after authentication, the session is set for DomainA. ServerB, on the other hand, is protected by a policy agent talking to an Identity Server in DomainB.

NOTE This is just one scenario; it is not obligatory to have an installed instance of Identity Server in both domains to use the cross-domain feature.

The Identity Server instance in DomainB recognizes the DomainA instance as an authentication provider. If UserA, after authenticating to DomainA, requests a resource on ServerB, the policy agent for DomainB checks for a session ID and will find that there is none (authorizing access to DomainB, that is). The agent then redirects the request to the Cross-Domain Controller running with the Identity Server instance in DomainB. The servlet, following the path outlined in Step 3 on page 193, finds the session ID from DomainA, extracts it from the cookie, places it in a Liberty AuthnResponse and sends it back to the agent. The agent finds the session ID and sets a cookie for DomainB using the session ID. The agent then sends an XML/HTTP request to the Naming, Session and Policy Services deployed in DomainB. Since the instance of Identity Server in DomianB recognizes the instance of Identity Server in DomainA as an authentication provider, DomainB retrieves identity, session and policy information for the requesting user from DomainA. The user is then allowed or denied access to the resource based on this information.

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 the URL parameters, and a new cookie can be set for DomainB with the same cookie name and value.

Enabling Cross-Domain Single Sign-On

As described, in order to exchange session information across two different domains, Policy Agents and the Cross-Domain Controller communicate with each other. By default, Identity Server is installed with the servlet. Policy agents, on the other hand, are installed separately. When installing the agent, the option to configure it for CDSSO must be selected. The cookie domain for the agent must be configured after installation. This is done by editing the `AMAgent.properties` file. The `com.sun.am.policy.agents.cookieDomainList` property must be set with the domain in which the agent is installed. If the field is left blank, the cookie domain will be set to the FQDN of the web server on which the agent is installed. Additional information on enabling cross-domain single sign-on can be found in the *Web Policy Agents Guide* and the *J2EE Policy Agents Guide*.

SSO API

The Session Service provides Java and C API to allow external applications to participate in the SSO functionality. All Identity Server services (except for Authentication) require a valid session (programmatically referred to as `SSOToken`) to process a HTTP request. External applications wishing to use the SSO functionality must also use the `SSOToken` to authenticate the user's identity. With the SSO API, an external application can retrieve it and, in turn, the user's identity, session and policy information. The application then uses this information to determine whether to provide user access to a protected resource.

After successfully authenticating to Identity Server, a user carries their Session ID with them using browser cookies or URL query parameters. Now, each time a user requests access to a protected application, the application needs to verify their identity. Assume a user authenticates to `http://www.orgA.com/Store` successfully and later tries to access `http://www.orgA.com/UpdateInfo`, a service that is *SSO-enabled*. Rather than having the second application authenticate the user again, it can use the API and the user's session to determine if the user is already authenticated. If the methods determine that the user has already been authenticated (and the session is still valid), access to this page can be achieved. Otherwise, the user would be prompted to authenticate again. The SSO API can also be used to create or destroy a `SSOToken`, or to listen for `SSOToken` events. (An *event* might be a `SSOToken` timing out because the user has reached the their maximum time limit.) Following are both the Java API Overview and C API Overview.

Java API Overview

In Java, the main classes of the SSO API are SSOTokenManager, SSOToken and SSOTokenListener. The SSOTokenManager class is used to get, destroy, validate, and refresh a session token which is represented by the SSOToken class. The SSOTokenListener class allows the application to be notified when a SSOToken has become invalid, for example when a session has timed out.

SSOTokenManager Class

The SSOTokenManager class contains the methods needed to get, validate, destroy and refresh session tokens. SSOTokenManager is implemented using the singleton design pattern. In order to obtain an instance of SSOTokenManager, the SSOTokenManager.getInstance() method must be called. An instance of SSOTokenManager can then be used to instantiate an SSOToken object using one of the overloaded forms of the createSSOToken() method.

The destroyToken() method would be called to invalidate and delete a token when its session has ended. The isValidToken() and validateToken() methods can be called to verify whether a token is valid, or authenticated. isValidToken() returns true or false depending on whether the token is valid or invalid, respectively. validateToken() throws an exception only when the token is invalid; nothing happens if the token is valid. The refreshSession() method resets the idle time of the session. Code Example 4-1 illustrates one way in which the SSOTokenManager class can be used.

Code Example 4-1 Sample Uses Of SSOTokenManager Code

```
try {
        /* get an instance of the SSOTokenManager */
        SSOTokenManager ssoManager = SSOTokenManager.getInstance();

        /* The request here is the HttpServletRequest. Get
        /* SSOToken for session associated with this request. */
        SSOToken ssoToken = ssoManager.createSSOToken(request);

        /* use isValid method to check if token is valid or not.
         * This method returns true for valid token, false otherwise. */
        if (ssoManager.isValidToken(ssoToken)) {
          /* If token is valid, this information may be enough for
           * some applications to grant access to the requested
           * resource. A valid user represents a user who is
           * already authenticated. An application can further
           * utilize user identity information to apply
           * personalization logic.
           */
        } else {
           /* Token is not valid, redirect the user login page. */
        }
```

Code Example 4-1 Sample Uses Of SSOTokenManager Code *(Continued)*

```
          /* Alternative: use of validateToken method to check
             * if token is valid */
          try {
            ssoManager.validateToken(ssoToken);
            /* handle token is valid */
          } catch (SSOException e) {
            /* handle token is invalid */
          }

          /*refresh session. idle time should be 0 after refresh. */
          ssoManager.refreshSession(ssoToken);

        } catch (SSOException e) {
          /* An error has occurred. Do error handling here. */
        }
```

SSOTokenID Interface

The `SSOTokenID` interface is used to identify the `SSOToken` object.

CAUTION The string value of `SSOTokenID` is globally unique and must only be known to the client browser, Identity Server and the application code. Exposing it to unauthorized users or applications can lead to a security breach by allowing a malicious attacker to impersonate a user.

SSOToken Interface

The `SSOToken` interface represents a *single sign-on* token returned from the `SSOTokenManager.createSSOToken()` method, and contains information such as the authenticated principal name, authentication method, and session information (session idle time, maximum session time, etc.). The `SSOToken` interface has methods to get predefined session information, such as `getAuthType()` for the authentication type, as well as a method `getProperty()` to get any information about the session, predefined or otherwise (for example, information set by the application). The method `setProperty()` can be used by the application to set application-specific information in the session. The `addSSOTokenListener()` method can be used to set a listener to be invoked when the session state has become invalid.

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

Code Example 4-2 shows an example of SSOToken code.

Code Example 4-2 Sample Use Of SSOToken

```
/* get http request output stream for output */
      ServletOutputStream out = response.getOutputStream();

      /* get the sso token from http request */
      SSOTokenManager ssoManager = SSOTokenManager.getInstance();
      SSOToken ssoToken = ssoManager.createSSOToken(request);

      /* get the sso token ID from the sso token */
      SSOTokenID ssoTokenID = ssoToken.getTokenID();
      out.println("The SSO Token ID is "+ssoTokenID.toString());

      /* use validate method to check if the token is valid */
      try {
        ssoManager.validateToken(ssoToken);
        out.println("The SSO Token validated.");
      } catch (SSOException e) {
        out.println("The SSO Token failed to validate.");
      }

      /* use isValid method to check if the token is valid */
      if (!ssoManager.isValidToken(token)) {
        out.println("The SSO Token is not valid.");
      } else {
        /* get some values from the SSO Token */
        java.security.Principal principal = ssoToken.getPrincipal();
          out.println("Principal name is "+principal.getName());
        String authType = ssoToken.getAuthType();
          out.println("Authentication type is "+authType);
          int authLevel = ssoToken.getAuthLevel();
          out.println("Authentication level is "+authLevel);
        long idleTime = ssoToken.getIdleTime();
          out.println("Idle time is "+idleTime);
          long maxIdleTime = ssoToken.getMaxIdleTime();
          out.println("Max idle time is "+maxIdleTime);
          long maxTime = token.getMaxSessionTime();
          out.println("Max session time is "+maxTime);
          String host = ssoToken.getHostName();
          out.println("Host name is "+host);
          /* host name is a predefined information of the session,
          /* and can also be obtained the following way */
          String hostProperty = ssoToken.getProperty("HOST");
          out.println("Host property is "+hostProperty);
          /* set application specific information in session */
          String appPropertyName = "appProperty";
          String appPropertyValue = "appValue";
          ssoToken.setProperty(appPropertyName, appPropertyValue);
          /* now get the app specific information back */
          String appValue = ssoToken.getProperty(appPropertyName);
          if (appValue.equals(appPropertyValue)) {
              out.println("Property "+appPropertyName+",  value
"+appPropertyValue+" verified to be set.");
```

Code Example 4-2 Sample Use Of SSOToken *(Continued)*

```
          } else {
              out.println("ALERT: Setting property "+appPropertyName+"
failed!");
          }
      }
```

A code sample using the `getTokenID` method is illustrated in Code Example 4-3. With this code, a cookie is created from an `SSOToken` in order to make SSO work for protected resources not residing on the same server as Identity Server.

Code Example 4-3 Sample Code To Create A Cookie From Session Token

```
// Get SSOToken string
String strToken = null;
strToken = getSSOToken().getTokenID().toString();
// Set it to response as cookies
String s = strToken;
String ssotokencookiename = "iPlanetDirectoryPro";
String ssotokencookiedomain = ".mydomain.com.tw";
String ssotokencookiepath = "/";
String gt = "/welcomepage.jsp";
Cookie cookie = new Cookie(ssotokencookiename,s);
cookie.setDomain(ssotokencookiedomain);
cookie.setPath(ssotokencookiepath);
response.addCookie(cookie);
response.sendRedirect(gt);
```

SSOTokenEvent

The `SSOTokenEvent` interface represents a token event. An event is, for example, when a session has been idle for over a maximum idle time limit, or when a session has reached its maximum allowed time.

SSOTokenListener

The `SSOTokenListener` interface represents a token notification object. An implementation of the `SSOTokenListener` interface must be written, then registered with the `SSOTokenManager` to be invoked when a token event occurs.

The SSOTokenListener interface provides a mechanism to notify applications when a session token has become invalid due to, for instance, the session reaching maximum idle time or the maximum session time. Applications wishing to be notified must write an implementation of the SSOTokenListener interface, then register the implementation through the SSOToken.addSSOTokenListener method. When the SSOToken state has become invalid, the SSOTokenListener implementation's ssoTokenChanged method will be invoked with a SSOTokenEvent object containing the event type, time, and SSOToken object with the new SSOToken state and other properties of the SSOToken.

Code Example 4-4 Sample Code For SSOToken Event And SSOToken Listener

```
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: " + id.toString() + "is not valid
anymore");
          /* redirect user to login */
          ......
        } catch (Exception e) {
              System.out.println(e.getMessage());
        }
      }
      }

      public SampleTestRoutine {
        ...
        SSOTokenManager ssoManager = SSOTokenManager.getInstance();
        SSOToken ssoToken = SSOManager.createSSOToken(request);
        SSOTokenListener sampleListener = new SampleTokenListener();
        ssoToken.addSSOTokenListener(sampleListener);
      ...
      }
```

Sample SSO Java Files

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

1. With the SSO Servlet Sample, a session token can be created for an application that runs on the Identity Server server.
2. With the Remote SSO Sample, a session token can be created for an application that runs on a server other than the Identity Server server.

3. With the Command Line SSO Sample, a session token can be created by a session ID string and passed through the command line.

The sample files are located in the *IdentityServer_base*/`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.

C API Overview

The C API are provided in the `SUNWamcom` package which comes with Identity Server or any of its downloadable agents. The package includes header files, libraries and samples.

CAUTION Previous releases of Identity Server contained C libraries in *IdentityServer_base*/lib/capi. The capi directory is being deprectated, and is curently available for backward compatability. It will be removed in the next release, and therefore it is highly recommended that existing application paths to this directory are changed and new applications do not access it. Paths include RPATH, LD_LIBRARY_PATH, PATH, compiler options, etc.)

C SSO Include Files

Include files for the C SSO API are am_sso.h and am_notify.h. am_sso.h must be included for any SSO routines. am_notify.h must be included for parsing notification messages from the server and calling SSO listeners.

C SSO Properties

Certain properties must be read in and passed to am_sso_init(), the routine which initializes C API. Because of this, am_sso_init() must be called before any other SSO interface. The default properties file used is AMAgent.properties, located in *IdentityServer_base*/SUNWam/config/. The following properties must be set:

- The com.sun.am.namingURL property specifies the URL for the Naming Service. This service is used to find the URL of the Session Service for the given SSOToken ID. This property must be set as:

  ```
  com.sun.am.namingURL =
  https://myhost.mydomain.com:58080/amserver/namingservice
  ```

- The com.sun.am.notificationEnabled and com.sun.am.notificationURL properties specify whether notification is enabled, and if enabled, a URL where the application can listen for messages from Identity Server. These properties must be set as:

  ```
  com.sun.am.notificationEnabled=true
  ```

NOTE If com.sun.am.notificationEnabled is not found in the properties file, the default is false.

```
com.sun.am.notificationURL=https://myhost.mydomain.com:8000/myU
RL
```

- The `com.sun.am.sso.cacheEntryLifeTime` property specifies how long, in minutes, a session token can live in cache before it should be removed. This property must be set as:

  ```
  com.sun.am.sso.cacheEntryLifeTime=5
  ```

  If not set, the default is 3 minutes.

- The `com.sun.am.sso.checkCacheInterval` property specifies how often, in minutes, the cache should be checked for entries that have reached the cache entry life time. This property must be set as:

  ```
  com.sun.am.sso.checkCacheInterval=5
  ```

- The `com.sun.am.sso.maxThreads` specify the maximum number threads the SSO API should invoke for handling notifications. The API maintains a thread pool and invokes a thread for each notification. If the maximum number of threads has been reached, the notification will wait until a thread is available. If not specified the default maximum number of threads is 10. This property must be set as:

  ```
  com.sun.am.sso.maxThreads = 5
  ```

- The `com.sun.am.cookieEnabled` property specifies whether the session ID found in the cookie is URL encoded. If true, it will be URL decoded before sent to Identity Server for any session operation. This property must be set as:

  ```
  com.sun.am.cookieEncoded = true|false
  ```

More information on properties in the `AMAgent.properties` file can be found in the *Web Policy Agents Guide* and the *J2EE Policy Agents Guide*.

C SSO interfaces

The C SSO interfaces consist of the following routines. A detailed description of the input and output parameters for each interface is in the header files.

- Initialization and Cleanup
- Get, Validate, Refresh And Destroy SSO Token
- Get Session Information Interfaces
- Get And Set Property Interfaces
- Listener And Notify Interfaces

Initialization and Cleanup

To use the C SSO API, the `am_sso_init()` routine needs to be called before any other routines. This interface initializes the internal SSO module. At the end of all SSO routines, `am_cleanup()` should be called to cleanup the internal SSO module. Code Example 4-5 on page 204 is a code sample for these interfaces.

am_sso_init() initializes internal data structures for talking to the Session Service. It takes a properties input parameter that contains name /value pairs from a configuration or properties file, and returns a status on the success or failure of the initialization. The properties used by the C SSO API are covered in "C SSO Properties" on page 202.

am_cleanup() cleans up all internal data structures created by `am_sso_init`, `am_auth_init`, or `am_policy_init`. `am_cleanup()` needs to be called only once when using any of the Identity Server C API interfaces (authentication, SSO or policy).

Code Example 4-5 Code Sample For am_sso_init and am_cleanup

```
#include <am_sso.h>

int main() {
   am_properties_t *properties;
   am_status_t status;

   /* create a properties handle */
   status = am_properties_create(&properties);
   if (status != AM_SUCCESS) {
        printf("am_properties_create failed.\n");
        exit(1);
   }

   /* load properties from a properties file */
   status = am_properties_load(properties, "./myPropertiesFile");
   if (status != AM_SUCCESS) {
       printf("am_properties_load failed.\n");
       exit(1);
   }

   /* initialize SSO module */
   status = am_sso_init(properties);
   if (status != AM_SUCCESS) {
       printf("am_sso_init failed.\n");
       return 1;
   }

    /* login through auth module, and do auth functions.
     * ...
     */

   /* do sso functions
```

Code Example 4-5 Code Sample For am_sso_init and am_cleanup *(Continued)*

```
        * ...
        */

        /* done - cleanup. */
        status = am_cleanup();
        if (status != AM_SUCCESS) {
            printf("am_cleanup failed!\n");
            return 1;
        }
        /* free memory for properties */
        status = am_properties_destroy(properties);
        if (status != AM_SUCCESS) {
            printf("Failed to free properties.\n");
            return 1;
        }

        /* exit program successfully. */
        return 0;
    }
```

Get, Validate, Refresh And Destroy SSO Token

A user needs to be authenticated to get the token ID for their login session. A token can be obtained with the token ID and the `am_sso_create_sso_token_handle` interface. This interface checks to see if the token is in its local cache and, if not, goes to the server to get the session information associated with the token ID and caches it. If the reset flag is set to `true`, this interface will refresh the idle time of the token on the server. Here is the interface of `am_sso_create_sso_token_handle`:

- `am_status_t am_sso_create_sso_token_handle(am_sso_token_handle_t * sso_token_handle_ptr, const char *sso_token_id, boolean_t refresh_token);`

Once a token handle is obtained, the caller can check if the session is valid with the `am_sso_is_valid_token` interface. The `am_sso_token_validate` interface will flush the token handle in the local cache (if any) and go to the server to fetch the latest session information. The `am_sso_refresh_token` will also flush the token handle in the local cache (if any) and go to the server to fetch the session information. In addition, it will reset the idle time of the session on the server. Here are the token-related interfaces:

- `boolean_t     am_sso_is_valid_token(am_sso_token_handle_t sso_token_handle);`
- `am_status_t am_sso_validate_token(am_sso_token_handle_t sso_token_handle);`

- `am_status_t am_sso_refresh_token(am_sso_token_handle_t sso_token_handle);`

When caller is done with a token handle, it must be freed by calling `am_sso_destroy_sso_token_handle` to prevent memory leak. Here is that interface:

- am_status_t am_sso_destroy_sso_token_handle(am_sso_token_handle_t sso_token_handle);

The session associated with the token can be invalidated or ended with `am_sso_invalidate_token`. Although this ends the session for the user, the proper way to log out is through `am_auth_logout`. Using the former interface to end a session will result in authentication resources associated with the session to remain on the server unnecessarily until the session has timed out. Here is the interface for `am_sso_invalidate_token`:

- `am_status_t am_sso_invalidate_token(am_sso_token_handle_t sso_token_handle);`

Get Session Information Interfaces

The following interfaces make it convenient to get server-defined information (or properties) about the session associated with a token. This can include the session idle time, max session time, etc.

- `const char * am_sso_get_sso_token_id(const am_sso_token_handle_t sso_token_handle);`
- `const char * am_sso_get_auth_type(const am_sso_token_handle_t sso_token_handle);`
- `unsigned long am_sso_get_auth_level(const am_sso_token_handle_t sso_token_handle);`
- `time_t          am_sso_get_idle_time(const am_sso_token_handle_t sso_token_handle);`
- `time_t          am_sso_get_max_idle_time(const am_sso_token_handle_t sso_token_handle);`
- `time_t          am_sso_get_time_left(const am_sso_token_handle_t sso_token_handle);`
- `time_t          am_sso_get_max_session_time(const am_sso_token_handle_t sso_token_handle);`
- `const char * am_sso_get_principal(const am_sso_token_handle_t sso_token_handle);`

- `am_string_set_t am_sso_get_principal_set(const am_sso_token_handle_t sso_token_handle);`
- `const char * am_sso_get_host(const am_sso_token_handle_t sso_token_handle);`

Get And Set Property Interfaces

The get and set property interfaces allows an application to get any property (server or application defined) and to set any property in a session. Note that `am_sso_set_property` will update the `sso_token_handle` with the latest session properties from Identity Server, including the new property that was set. In addition, if the property that is given in `prop_name` is a protected property, `am_sso_set_property` will return success, however the value given will not be set as it is a property protected by Identity Server. These interfaces are:

- `const char * am_sso_get_property(const am_sso_token_handle_t sso_token_handle, const char *prop_name);`
- `am_status_t am_sso_set_property(am_sso_token_handle_t sso_token_handle, const char *prop_name, const char *prop_value);`

Code Example 4-6 is a sample of the SSO get, set, create, refresh, validate, invalidate, and destroy interfaces.

Code Example 4-6 Sample Code For Get, Set, Create, Refresh, Validate, Invalidate, and Destroy Interfaces

```
     /* initialize sso as in previous sample */

     am_status_t status = NULL;
     am_sso_token_handle_t sso_handle = NULL;
     char *session_status = NULL;
     am_string_set_t principal_set = NULL;

     /* create sso token handle */
     status = am_sso_create_sso_token_handle(&sso_handle, sso_token_id,
false);
     if (status != AM_SUCCESS) {
         printf("Failed getting sso token handle for sso token id %s.\n",
sso_token_id);
         return 1;
     }

     /* check if session is valid */
     session_status = am_sso_is_valid_token(sso_handle) ? "Valid" :
"Invalid";
     printf("Session state is %s\n", session_status);
```

Code Example 4-6 Sample Code For Get, Set, Create, Refresh, Validate, Invalidate, and Destroy Interfaces *(Continued)*

```
     /* check if session is valid using validate. This also updates the
handle with info from the server */
     status = am_sso_validate_token(sso_handle);
     if (status == AM_SUCCESS) {
         printf("Session state is valid.\n");
     } else if (status == AM_INVALID_SESSION) {
         printf("Session status is invalid.\n");
     } else {
         printf("Error validating sso token.\n");
         return 1;
     }

     /* get info on the session */
     printf("SSO Token ID is %s.\n", am_sso_get_sso_token_id(sso_handle));
     printf("Auth type is %s.\n", am_sso_get_auth_type(sso_handle));
     printf("Auth level is %d.\n", am_sso_get_auth_level(sso_handle));
     printf("Idle time is %d.\n", am_sso_get_idle_time(sso_handle));
     printf("Max Idle time is %d.\n", am_sso_get_max_idle_time(sso_handle));
     printf("Time left is %d.\n", am_sso_get_time_left(sso_handle));
     printf("Max session time is %d.\n",
am_sso_get_max_session_time(sso_handle));
     printf("Principal is %s.\n", am_sso_get_principal(sso_handle));
     printf("Host is %s.\n", am_sso_get_host(sso_handle));
     principal_set = am_sso_get_principal_set(sso_handle);
     if (principal_set == NULL) {
            printf("ERROR: Principal set is NULL!\n");
     }else {
            printf("Principal set size %d.\n", principal_set->size);
            for (i = 0; i < principal_set->size; i++) {
                printf("Principal[%d] = %s.\n", i,
principal_set->strings[i]);
            }
            am_string_set_destroy(principal_set);
     }

     /* get "HOST" property on the session. Same as am_sso_get_host(). */
     printf("Host is %s.\n", am_sso_get_property(sso_handle, "HOST"));

     /* set a application defined property and get it back */
     status = am_sso_set_property(sso_handle, "AppPropName",
"AppPropValue");
     if (status != AM_SUCCESS) {
         printf("Error setting property.\n");
         return 1;
     }
     printf("AppPropName value is %s.\n", am_sso_get_property(sso_handle,
"AppPropName");

     /* refresh token, idle time should be 0 after refresh */
     status = am_sso_refresh_token(sso_handle);
     if (status != AM_SUCCESS) {
         printf("Error refreshing token !\n");
         return 1;
```

Code Example 4-6 Sample Code For Get, Set, Create, Refresh, Validate, Invalidate, and Destroy Interfaces *(Continued)*

```
     }
     printf("After refresh, idle time is %d.\n",
am_sso_get_idle_time(sso_handle));

     /* end this session abruptly. am_auth_logout() is the right way to end
session */
     status = am_sso_invalidate_token(sso_handle);
     if (status != AM_SUCCESS) {
         printf("Error invalidating token.\n");
         return 1;
     }

     /* we're done with sso token handle. free memory for sso handle. */
     status = am_sso_destroy_sso_token_handle(sso_handle);
     if (status != AM_SUCCESS) {
         printf("Failed to free sso token handle.\n");
         return 1;
     }

     /* call am_cleanup, and other cleanup routines as in previous sample */
```

Listener And Notify Interfaces

Applications can be notified when a session has become invalid, possibly because it has been idle over a time limit, or it has reached the maximum session time. This is done by implementing a listener function of type `am_sso_token_listener_func_t`, which takes a SSO token handle, event type, event time, application-defined arguments handle, and a boolean argument to indicate whether the listener function should be called in the calling thread or dispatched to a thread from the internal thread pool managed by the C SDK. This listener function must be registered to be invoked when the session has ended and notification must be enabled for an application to receive notifications. Notification is enabled by setting the property `com.sun.am.notificationEnabled` to true, and by providing a URL where the application is receiving HTTP messages from Identity Server. The URL where the application is receiving messages from the Identity Server is expected to take any message from the server (as an XML string) and pass it to `am_notify()`. `am_notify()` will parse the message and invoke session listeners or policy listeners depending on whether the message is a session or policy notification. Code Example 4-7 is a sample implementation of SSOToken listener and how to register it.

Code Example 4-7 Sample Implementation Of SSOToken Listener

```
     void sample_listener_func(
                   am_sso_token_handle_t sso_token_handle,
                   const am_sso_token_event_type_t event_type,
                   const time_t event_time,
                   void *opaque)
     {
         if (sso_token_handle != NULL) {
             const char *sso_token_id =
am_sso_get_sso_token_id(sso_token_handle);
             boolean_t is_valid = am_sso_is_valid_token(sso_token_handle);
             printf("sso token id is %s.\n",
                    sso_token_id==NULL?"NULL":sso_token_id);
             printf("session state is %s.\n",
                      is_valid == B_TRUE ? "valid":"invalid");
             printf("event type %d.\n", event_type);
             printf("event time %d.\n", event_time);
         }
         else {
             printf("Error: sso token handle is null!");
         }
         if (opaque)
             *(int *)opaque = 1;
         return;
     }

     int main(int argc, char *argv[]) {

     am_status_t status;
     char *sso_token_id = argv[1];
     int listener_func_done = 0;

     /* initialize sso as in previous samples */

     /* get sso token handle */
     status = am_sso_create_sso_token_handle(&sso_handle, sso_token_id,
false);

     /* register listener function. notification must be enabled, if not,
status AM_NOTIF_NOT_ENABLED will be returned. */
 status = am_sso_add_sso_token_listener(sso_handle, sample_listener_func,
&listener_func_done, B_TRUE);
     if (status != AM_SUCCESS) {
         printf("Failed to register sample listener function.\n");
         return 1;
     }
```

C SSO Sample

A sample for the C SSO API is provided in the SUNWamcom package. The README file in the samples directory contains information on each sample including compile instructions and how to run the samples for testing. The sample for C SSO is am_sso_test.c. The usage is am_sso_test -u [user] -p [password] [-f properties file] [-l logfile]. Identity Server must be available with LDAP authentication to test the sample. See the README file and the sample itself for more information.

Java versus C API

The following table provides a side by side comparison of the Java and C SSO API.

Table 4-1 Comparison Between Java And C SSO API

Java Interface	C Interface
SSOTokenManager	am_status_t
SSOTokenManager.getInstance()	am_sso_init(am_properties_t properties)
SSOToken	am_status_t
SSOTokenManager.createSSOToken(String tokenId)	am_sso_create_sso_token_handle(am_sso_token_handle_t *sso_token_handle_ptr, const char *sso_token_id, am_bool_t reset_idle_timer)
boolean	boolean_t
SSOTokenManager.isValidToken(SSOToken token)	am_sso_is_valid_token(const am_sso_token_handle_t sso_token_handle)
void	am_status_t
SSOTokenManager.validateToken(SSOToken token)	am_sso_validate_token(const am_sso_token_handle_t sso_token_handle)
void	am_status_t
SSOTokenManager.destroyToken(SSOToken token)	am_sso_invalidate_token(const am_sso_token_handle_t sso_token_handle)
void	am_status_t
SSOTokenManager.refreshSession(SSOToken token)	am_sso_refresh_session(am_sso_token_handle_t sso_token_handle)
Principal	char *
SSOToken.getPrincipal()	am_sso_get_principal(const am_sso_token_handle_t sso_token_handle)

Table 4-1 Comparison Between Java And C SSO API *(Continued)*

Java Interface	C Interface
int	unsigned long
SSOToken.getAuthLevel()	am_sso_get_auth_level(const am_sso_token_handle_t sso_token_handle)
String	char *
SSOToken.getAuthType()	am_sso_get_auth_type(const am_sso_token_handle_t sso_token_handle)
String	char *
SSOToken.getHostName()	am_sso_get_host(const am_sso_token_handle_t sso_token_handle)
long	time_t
SSOToken.getIdleTime()	am_sso_get_max_idle_time(const am_sso_token_handle_t sso_token_handle)
long	time_t
SSOToken.getMaxIdleTime()	am_sso_get_max_idle_time(const am_sso_token_handle_t sso_token_handle)
SSOTokenID	char *
SSOToken.getTokenID()	am_sso_get_sso_token_id(const am_sso_token_handle_t sso_token_handle)
String	char *
SSOToken. getProperty(java.lang.String name)	am_sso_get_property(const am_sso_token_handle_t sso_token_handle, const char *property_name)
void	am_status_t
SSOToken.setProperty(String name, String value)	am_sso_set_property(am_sso_token_handle_t sso_token_handle, const char *name, const char *value)
void	am_status_t
SSOToken.addSSOTokenListener(SSOTokenListener listener)	am_sso_add_sso_token_listener(am_sso_token_handle_t sso_token_handle, const am_sso_token_listener_func_t listener, void *args, boolean_t dispatch_in_sep_thread)
String	am_status_t
SSOToken.getProperty("principals");	am_sso_get_principal_set(am_sso_token_handle_t sso_handle)
N/A	am_status_t
	am_sso_destroy_sso_token_handle(am_sso_token_handle_t sso_handle)

Table 4-1 Comparison Between Java And C SSO API *(Continued)*

Java Interface	C Interface
N/A	void am_cleanup()

Non-Web-Based Applications

Identity Server provides the SSO API primarily for web-based applications, although it can be extended to any non-web-based applications with limitations. With non-web-based applications, their 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 session token. The method used for this process is application-specific.
2. Command line applications, such as `amadmin`, can be used. In this case, session 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.

SSO Samples

Identity Server provides the files necessary to compile and run a sample SSO application. There are three ways in which this can be done:

- Compiling and running a SSO application local to Identity Server.
- Installing and running the SSO SDK from a remote client.
- Running the SSO application from the command line.

More specific information on these samples can be found in “Sample SSO Java Files” on page 200.

Chapter 5

Identity Management

The Identity Management module of Identity Server contains an XML template file and application programming interfaces (APIs) that provide the 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 these identity-related features. It contains the following sections:

- Overview
- Identity-related Objects
- Object Templates And ums.xml
- amEntrySpecific.xml
- Identity Management SDK
- Identity Management Samples

Overview

The Identity Management module allows for the management of Identity-related Objects using the Identity Server console or command line tools. These objects, that are created and managed via Identity Server, are actually stored as LDAP entries in Directory Server. To bridge the gap between the two products, Identity Server provides interfaces that are used to create and delete identity-related objects as well as get, add, modify, or remove their attributes.

Identity Server Console

All aspects of the Identity Server console are covered in Chapter 2, “The Identity Server Console,” of this manual and the *Sun ONE Identity Server Administration Guide*.

ums.xml

This file defines a set of *templates* that contain the configuration information needed to set up each identity-related object created with Identity Server as an LDAP entry in the Directory Server data store. More information on `ums.xml` can be found in “Object Templates And ums.xml” on page 220.

Identity Management Software Development Kit (SDK)

The SDK is used to integrate the management functions of Identity Server into external applications or services. More information on the SDK can be found in “Identity Management SDK” on page 225.

Identity-related Objects

Identity Server defines and manages the following identity-related objects:

- Organizations
- Containers
  - Organizational Units (referred to as *containers* in the console)
  - People Containers
  - Group Containers
- Users
- Groups
  - Static Groups
  - Assignable Groups (Dynamic)

  - Filtered Groups
- Roles
  - Static Roles
  - Filtered Roles

These identity-related objects are not LDAP objects as defined in the Directory Server schema. These objects are configured using an Identity Server schema, managed using the Identity Server application and only stored in Directory Server. In other words, an identity-related object in Identity Server does not necessarily correspond to its LDAP counterpart in Directory Server. But, because they are stored in Directory Server, these Identity Server objects must be mapped to the existing Directory Server schema. Thus, Identity Server object entries are appended with *marker object classes*.

Marker Object Classes

An identity-related object stored in Directory Server is identified as such through the use of special marker object classes appended to its LDAP entry. These object classes are defined in the Identity Server schema, `ds_remote_schema.ldif`, located in *IdentityServer_base*/`SUNWam/ldif`. When a marker object class is added to a Directory Server entry, Identity Server is able to access and manage that entry using its console or command line tools. For example, an enterprise's existing directory schema may use *organizational unit* as its root rather than the default *organization*; by adding the Identity Server *organization* marker object class, `sunManagedOrganization`, to the LDAP entries of the organizational unit, Identity Server can manage it as the organization's root. It is through the use of marker object classes that Identity Server can manage most existing directory structures. The marker object classes are:

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

- `iplanet-am-managed-filtered-group`
- `iplanet-am-managed-role`
- `iplanet-am-managed-filtered-role`

For information on how to configure an existing directory tree within Identity Server, see the *Sun ONE Identity Server Migration Guide*.

Identity-related Objects As LDAP Entries

Following is a discussion of the Identity Server objects and how they map to LDAP entries in Directory Server.

Organizations

Represented by the marker object class `sunManagedOrganization`, organization is the root entry of an Identity Server tree. It generally maps to an LDAP `organization` or `organizationalUnit` object class.

Containers

Functionally, there are three types of containers in Identity Server.

Organizational Units

Represented by the marker object class `iplanet-am-managed-org-unit`, an organizational unit is referred to as a *container* in the Identity Server console. It generally maps to the LDAP `organizationalUnit` object class and can contain sub-organizations, other containers, roles, groups, and users.

People Containers

Represented by the marker object class `iplanet-am-managed-people-container`, a people container is an organizational unit which is a parent for user entries. It generally maps to the LDAP `organizationalUnit` object class and can contain sub-people containers and users.

Group Containers

Represented by the marker object class `iplanet-am-managed-group-container`, a group container is an organizational unit which is a parent for any number of group entries. It generally maps to the LDAP `organizationalUnit` object class and can only contain groups and other group containers.

Users

Represented by the markerobject class `iplanet-am-managed-person`, a user is the representation of a person. It maps to an LDAP `inetOrgPerson`. It is a leaf node that may not contain other entries.

Groups

Functionally, there are three types of groups in Identity Server. Assignable Groups (Dynamic) (by default) and Static Groups are configured using the Membership By Subscription option in the console. Filtered Groups are configured by choosing the Membership By Filter option in the console.

Assignable Groups (Dynamic)

Represented by the marker object class `iplanet-am-managed-assignable-group`, an assignable group is one in which an administrator wants to explicitly add the user to a group. For example, Larry wants to give Ramona permission to look at his employees' telephone numbers so he adds her to the ReadPhoneNumbers group. In Directory Server, member entries contain the `memberof` LDAP attribute (`inetAdmin` object class) and the group membership is dynamically established.

NOTE Assignable groups are referred to as Dynamic when seen in the console as, technically, they are created with an LDAP filter albeit a static one.

Static Groups

Represented by the marker object class `iplanet-am-managed-static-group`, a static group is one in which members are added by appending the `groupOfUniqueNames` object class to the LDAP group entry itself. It can contain users, filtered groups or other static sub-groups. This type of group can be enabled using the Administration Service in the console. By default, it is disabled and all groups created are of the type "Assignable Groups (Dynamic)."

Filtered Groups

Represented by the marker object class `iplanet-am-managed-filtered-group`, a filtered group is created through the use of an LDAP filter. All user entries are funneled through the filter and dynamically assigned to the group. The filter would look for a specified attribute in an entry and return those entries that contain the attribute as a member of the group.

Roles

Functionally, there are two types of roles in Identity Server. Roles can only be created in organizations, suborganizations and generic containers; they can not be configured in people containers.

Static Roles

Represented by the marker object class `iplanet-am-managed-role`, a static role is a role entry in which the members are added by appending the `groupOfUniqueNames` object class to the role entry itself. It can contain users.

Filtered Roles

Represented by the marker object class `iplanet-am-managed-filtered-role`, a filtered role is created through the use of an LDAP filter. All user entries are funneled through the filter and dynamically assigned to the role. The filter would look for a specified attribute in an entry and return those entries that contain the attribute as a member of the role.

Object Templates And ums.xml

The `ums.xml` provides a set of parameters, or *templates*, that contain the LDAP configuration information for all Identity-related Objects managed using Identity Server. The templates are used to create LDAP entries for the identity-related objects so they can be stored in Directory Server. In addition, the templates are used for the dynamic generation of roles and the construction of object searches. The file can be found in the *IdentityServer_base*`/SUNWam/config/ums` directory; it is based on the `sms.dtd` which is defined in Chapter 6, “Service Management,” of this manual.

NOTE	These templates can be modified by administrators to alter the behavior of the Java interfaces. But, if `ums.xml` is modified and reloaded, there will be inconsistencies between the entries created prior to the modifications and the newer ones.

Structure Of ums.xml

The `ums.xml` defines three types of templates: Structure, Creation and Search. Structure templates define the Directory Server information tree 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 within the Directory Server information tree. In other words, these templates define the child nodes (roles, groups, containers) that are created IN ADDITION to the creation of the object itself. There are six attributes that need to be defined for each object's structure.

- `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 Relative Distinguished Name (RDN) for the object. RDN is "ou=People" or "cn=ContainerDefaultTemplateRole". For the core structure templates such as Organization or OrganizationaUnit, the value defined at run time (when you create Org's or containers from console or CLI). Thats why you don't see the RDN value for the core ones. Where as for others such as PeopleContainer & DefaultOrgRole, you see the RDN's. You can specify the RDN values for the PeopleContainers, Groups that can be created. A note of caution that the naming attribute specified in the RDN, for example ou from ou=Groups should match the naming attribute defined in the Group Creation template. For example, an organization has `o=org` as its naming attribute while a people container uses `ou=People`.
- `childNode`—This attribute specifies the child nodes (roles, groups, containers) that will be created in tandem with the object. The value is the name of the structure template for the respective object.
- `template`—This attribute specifies the name of the Creation template used to create this object.
- `filter`—This attribute is not currently used.
- `priority`—For internal use only, the value of this attribute should always remain 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 object's template.

- `name`—This attribute defines the type of object that the template will create. It is also the name of the template itself. This attribute should not be modified.

- `javaclass`—This attribute defines the name of the Java class used to instantiate the object. This attribute should not be modified.
- `required`—This attribute defines the required LDAP object classes and attributes for the object.
- `optional`—This attribute defines the optional LDAP object classes and attributes for the object.
- `validated`—This attribute is reserved for future use.
- `namingattribute`—This attribute specifies the LDAP attribute used to name the object. For instance, the Basic User creation template has as its `namingattribute` the value of the LDAP attribute, `uid`.

Search Templates

Search templates are used to define how searches for identity-related objects are performed in Directory Server. This template defines a default search filter and the attributes returned in the 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 value the search will look for.
- `attrs`—This attribute specifies the LDAP attributes that need to be returned.

NOTE For a listing of interfaces applicable to each identity-related objects, see "amEntrySpecific.xml" on page 224.

Modifying ums.xml

Any LDAP attributes or object classes not already present in the Directory Server LDAP schema 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 `employeeNumber` attribute, the `ums.xml` file does not need to be modified because this attribute already exists in the `inetorgperson` object class. Generally, the `ums.xml` file does not need to be modified. Some circumstances where this file would need to be modified are:

- if Identity Server 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 Migration Guide.*

CAUTION It is recommended that the `ums.xml` configuration file be backed up before any modifications are made.

Adding Custom Object Classes

If a service developer wants 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. The DAI Service would then need to be deleted from Directory Server and the modified ums.xml reloaded using the `amadmin` command line tool.

Once `ums.xml` has been modified, the new object classes and attributes must be defined in an XML service file which would then be imported into Identity Server using the procedures described in Chapter 6, "Service Management," of this manual. This configures Identity Server to manage the new object classes from the console.

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.

DAI Service

When Identity Server is installed, the `ums.xml` file is stored in Directory Server as the Directory Access Instructions (DAI) service. The DAI service is only available for modification through the Directory Server; it is not available through the Identity Server console or command line interface. The Identity Server SDK gets the configuration information from this directory tree node, when needed, to create an identity-related object or perform a search. Any attribute specified in the `ums.xml` can be set for a created object. If `ums.xml` is modified, the DAI Service would need to be deleted from Directory Server and reloaded using the `amadmin` command line tool. To delete the DAI Service from Directory Server, delete the DAI branch (`ou=DAI,ou=services,`*root-suffix*) or use the `amadmin` command line tool with the *-r* option. To reload `ums.xml`, use `amadmin` and the *-s* option. (The administrator user and password options will also be used for both.) For more detailed information on the command line tools, see the *Sun ONE Identity Server Administration Guide* .

NOTE When using the `amadmin` command line tool to reload `ums.xml`, the full DN of the `amadmin` user must be used as a parameter. If not, the LDAP Authentication Service will not be able to find the adminsitrator in its search for the user DN. For example, instead of using `amadmin -u amadmin -w 11111111` -s *ums.xml file path,* the input command would be:

`amadmin -u "uid=amadmin,ou=people,dc=example_org,dc=com" -w 11111111 -s` *ums.xml file path*

amEntrySpecific.xml

The purpose of the `amEntrySpecific.xml` service file is to define attributes from an existing directory to display on the Identity Server console's functional pages for all Identity-related Objects. These functional pages are as follows:

- 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 object is clicked.
- Search—The Search link is in the top left frame of the Identity Server console.

Each 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," of this manual.

NOTE Dynamic attributes are not supported in `amEntrySpecific.xml`.

If a service developer wants to customize the console's functional pages with attributes that are not default to the Identity Server tree, they would modify the `amEntrySpecific.xml` file. For example, to display an attribute on the group page, the new attribute needs to be added to the `amEntrySpecific.xml` file. Any 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. Note that based on the information in“any Attribute” on page 264, this Organization Status attribute will be displayed on the Search page and is not an attribute requiring a value for creation.

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.

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

Identity Management SDK

The Identity Server SDK contains an API for identity management. These interfaces can be used by developers to integrate management functions into external applications or services that will be managed by Identity Server. The API functions 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 Directory Server.

Interfaces

Below are brief explanations of the Identity Management API.

NOTE All operations performed using the API open and close LDAP connections via a connection pool. The connection pool size can be set in the serverconfig.xml file. For more information, see Appendix B, "serverconfig.xml File," in this manual.

AMAssignableDynamicGroup

The `AMAssignableDynamicGroup` interface provides the methods used to manage "Assignable Groups (Dynamic)." 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 the *BasicAssignableDynamicGroup*; and the search template used is the *BasicAssignableDynamicGroupSearch*. It does not have a pre-defined structural template.

AMCallback

`AMCallBack` is a plug-in class that needs to be extended by external applications in order to do special pre / post-processing for the creation, deletion and modification operations for users, organizations, roles and groups.

AMConstants

`AMConstants` is the base interface for all identity-related objects. It is used to define constants for use with the SDK (constants associated with searches, etc.).

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.

AMEventListener

The `AMEventListener` interface that can be used to monitor and react to events. This *listener* can be called when an identity-related object is removed, renamed or modified. It must be implemented using the following procedure:

1. Implement the `AMEventListener` interface.
2. Get an instance of the object to which `AMEventListener` will listen.

   For example, get an `AMUser` object and add the listener: `AMUser.addEventListener()`.
3. When an event changes something in this object, the listener will be called.

CAUTION Identity Server does not currently support attaching an event listener to template creation code.

AMFilteredRole

The `AMFilteredRole` interface provides the methods used to manage “Filtered Roles.” Associated with this object are the following `ums.xml` templates that define its behavior at runtime. The creation template used is *BasicFilteredRole*; and the search template used is *BasicFilteredRoleSearch*. It does not have a pre-defined structural template.

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.

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 structural template used by this class is *GroupContainer*; the creation template used is *BasicGroupContainer*, and the search template is *BasicGroupContainerSearch*.

AMObject

`AMObject` provides basic methods to manage identity-related objects. Since this is a generic class, it does not have any templates (as defined in “Object Templates And ums.xml” on page 220) 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 structural template used by this class is *Organization*; the creation template used is *BasicOrganization*, and the search template is *BasicOrganizationSearch*.

NOTE The `AMOrganization` interface contains methods that can be used to search through identity-related objects in Directory Server. More information can be found in "Search Methods In The SDK" on page 231.

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 structural template used by this class is *OrganizationalUnit*; the creation template used is *BasicOrganizationalUnit*, and 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 structural template used by this class is *PeopleContainer*; the creation template used is *BasicPeopleContainer*, and the search template is *BasicPeopleContainerSearch*.

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 *BasicManagedRole*; and the search template used is *BasicManagedRoleSearch*. It does not have a pre-defined structural template.

AMSearchControl

The `AMSearchControl` class provides a way to customize search behavior. Common behaviors are time limit, result limit and virtual list view.

Code Example 5-2 Sample Code Using AMSearchControl

```
SSOTokenManager manager = SSOTokenManager.getInstance();
SSOToken token = manager.createSSOToken(new
AuthPrincipal("uid=amadmin,ou=People,dc=example,dc=com"), "11111111");
suo = getSampleUserOperations(token);
amsc = new AMStoreConnection(token);
//System.out.println(suo.createUser(amsc));
AMSearchControl amc = new AMSearchControl();
amc.setTimeOut (2000);
amc.setSearchScope (AMConstants.SCOPE_ONE);
AMPeopleContainer amp =
amsc.getPeopleContainer("ou=people,dc=example,dc=com");
Set userset = (amp.searchUsers(amc, "(uid=u*)")).getSearchResults();
Object users[] = userset.toArray();
System.out.println((String)users[0]);
System.exit(0);
```

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.

AMStoreConnection

The `AMStoreConnection` class provides the means to establish a connection to the data store Directory Server and provides methods to create, remove and get different types of identity-related objects. A `SSOToken` is required in order to instantiate a `AMStoreConnection` object.

AMTemplate

The `AMTemplate` interface represents a service template associated with `AMObject`. Identity Server distinguishes between virtual and entry attributes. As defined for Sun ONE Directory Server, 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. An *entry attribute* is a non-inherited attributes.

NOTE More information on virtual attributes can be found in “Virtual Attribute” on page 481 of Appendix E, “Directory Server Concepts,” in this manual.

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. The templates are: `DYNAMIC_TEMPLATE` and `ORGANIZATION_TEMPLATE`. `DYNAMIC_TEMPLATE` are implemented using CoS; `ORGANIZATION_TEMPLATE` does not have virtual attributes or LDAP attributes.

Template Priority

When an 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. (This conflict will only occur with services that contain “Dynamic Attributes.”) The priority is defined by the value of the “cosQualifier Attribute” as discussed in Chapter 6, “Service Management,” of this manual. (The comparison values are `default`, `override`, and `merge-schemes`.) The priority level for a service template is set when then template is created using the Identity Server console. The levels are Highest, Higher, High, Medium, Low, Lower, and Lowest. Templates with higher priorities will be favored over templates with lower priorities when `default` is the value of `cosQualifier`. 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 merged. If the value is `override`, the priority level of the template takes precedence over any priority specified in the user profile. Merge-schemes signifies that the priority values will not be used, but a merged list of attribute values from all templates will be assigned. Templates which do not have an explicitly assigned priority are considered to have the lowest priority possible, or no priority.

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 *BasicUser*; and the search template used is *BasicUserSearch*. It does not have a pre-defined structural template.

Default Implementation Of AMUser

There is a default implementation of `AMUser`. Assuming an `SSOToken` and a user DN, the code to find the user status is illustrated in Code Example 5-3.

Code Example 5-3 Sample Code To Find User Status

```
AMStoreConnection conn = new AMStoreConnection (ssoToken) ;
AMUser user = conn.getUser (userDN) ;
if (user.isActivated()) {
....
```

Code Example 5-3 Sample Code To Find User Status *(Continued)*

```
} else {
....
}
```

AMUserPasswordValidation

`AMUserPasswordValidation` is an interface to plugin external modules to validate user names and passwords. The methods of this class must be overridden by the implementation plugin modules. The modules will be invoked whenever a userID or password value is being added or modified using Identity Server console, the `amadmin` CLI or the SDK directly.

Search Methods In The SDK

The SDK provides a variety of methods to conduct searches throughout the organizational tree. They are provided within the AMOrganization interface. Criteria is needed by the API to perform a search. Typically, the criteria is a LDAP search filter string, the scope of the search (one level or sub-tree), and where the search will begin (the base DN). The SDK provides the APIs to conduct searches and obtain results for all identity objects.

NOTE The SDK always includes the objectclass used to search so it is not required to explicitly include the filter. For example if searching for users, the SDK will include the default user search filter provided in the BasicUserSearch search template in the ums.xml.

This section specifically discusses one of the search methods: `searchUsers`. (For information on all of the search methods, refer to the Identity Server Javadocs.) Code Example 5-4 is the set of different search methods available for `searchUsers`.

Code Example 5-4 Available Search Methods For searchUsers

```
     public Set searchUsers(String wildcard, int level)
         throws AMException, SSOException;

     public Set searchUsers(String wildcard, Map avPairs, int level)
         throws AMException, SSOException;
```

Code Example 5-4 Available Search Methods For searchUsers

```
    public AMSearchResults searchUsers(String wildcard, Map avPairs,
AMSearchControl searchControl)
        throws AMException, SSOException;

    public AMSearchResults searchUsers(String wildcard,
 AMSearchControl searchControl)
        throws AMException, SSOException

    public AMSearchResults searchUsers(String wildcard,
 AMSearchControl searchControl, String avfilter)
        throws AMException, SSOException;
```

Search Method Parameters

Here are brief descriptions of some of the search method parameters.

AMSearchControl

This class provides a way to specify detailed search criteria such as the scope of the search, the maximum results, time out value, etc. It must be implemented for all searches to set these criteria.

wildCard

This parameter can be used to specify the wild card used for naming attributes. For example, if searching for all users whose naming attributes (uid or cn) start with "Ma", then the wild card could be Ma*.

avPair

This parameter is a map of attribute/value pairs that need to be added to a search filter. The key of the map is the attribute name and the value is a set of values. The SDK will construct a filter from this `avPair` map. Each of the pairs in the map will be OR ("|") and not AND (&) to construct the filter.

avFilter

In most cases it will be sufficient to OR the attributes, but this parameter provides flexibility for applications to pass their own search filter to meet search criteria. Such filters could be a complex LDAP search fiter as in the following example:

```
(&(objectclass=iplanet-am-managed-person)((customEmployeeNumer=12*)
&(customDepartment=3459932)))
```

This example illustrates when two conditions (the employee number and department number) need to be met. For this purpose, AND (&) is used.

NOTE The methods that return a `java.util.Set` will throw an exception if the search fails as a result of exceeding the search limit or the time limit. In such cases, even partial results of the failed search will not be returned. To obtain the partial results in such cases, the methods that return an `AMSearchResults` object must be used. The error code can be verified by using the class methods to check if the search was successful.

searchUsers Sample Code

Code Example 5-5 demonstrates how to search for all users in an organization (DN: `dc=example,dc=com`) who belong to department *3459932* and whose user names end with *smith*.

Code Example 5-5 Sample Code For Search Methods

```
// Note obtain a valid token of a principal who has privileges to
// perform this operation.
SSOToken token = getSSOToken();

// Create an AMStoreConnection and obtain an AMOrganization
// instance for dc=example, dc=com
AMStoreConnection amc = new AMStoreConnection(token);
AMOrganization amOrg = amc.getOrganization("dc=example,dc=com");

// Construct the search filter
// Need to retrieve all usernames ending with smith
String wildCard = "*smith"
Map avPair = new HashMap();
Set departmentValue = new HashSet();
departmentValue.add("3459932");
avPair.put("customDepartment", departmentValue);

// Set the search control
AMSearchControl = new AMSearchControl();
// Sub tree search
searchControl.setSearchScope(AMConstants.SCOPE_SUB);
// Time out 3000 milliseconds.
searchControl.setTimeOut(3000);
// Would like to get only first 100 results
searchControl.setMaxResults(100);

// Perform the search
AMSearchResults results = amOrg.searchUsers(wildcard, avPair,
                                             searchControl);
// Check if any time out or size limit errors occured.
if (results.getErrorCode == AMSearchResults.SUCCESS) {
     // Process the results
} else {
     // Verify the error condition and take appropriate action
}
```

Here the filter to conduct the search will look like:

```
(&(uid=*smith)(objectclass=inetorgperson)((customerDepartment="3459
932")))
```

To add an additional department, one more value can be added to the search as in:

```
(&(uid=*smith)(objectclass=inetorgperson)((customerDepartment="3459
932")|(customerDepartment="3459933")))
```

Search Groups Sample Code

Code Example 5-6 uses interfaces from the `com.iplanet.am.sdk` package to search groups.

Code Example 5-6 Search Groups Code Sample

```
try {
    Set orgSet1 = new HashSet();
    Set orgSet2 = new HashSet();
    Set orgSet3 = new HashSet();
    Set orgSet4 = new HashSet();
    AMSearchResults results = null;
    AMSearchControl ctl = new AMSearchControl(); //use default values
    String DN = "ou=Groups,dc=idp1,dc=com";
    AMOrganizationalUnit org = conn.getOrganizationalUnit(DN);
    if (org.isExists()) {
    //get all groups in this OU:
        orgSet1 = org.getAssignableDynamicGroups(AMConstants.SCOPE_SUB);
//get Assignable Dynamic Groups
       orgSet2 = org.getDynamicGroups(AMConstants.SCOPE_SUB); //get Dynamic
Groups
        orgSet3 = org.getStaticGroups(AMConstants.SCOPE_SUB); //get Static
Groups

    //set up the avPairs for the search on attribute within group
        Map avPairs = new HashMap();
        Set set = new HashSet(1);
        set.add("true");
        avPairs.put("iplanet-am-group-subscribable", set);
        results = org.searchAssignableDynamicGroups( "*", avPairs, ctl );
//returns all subscribable groups
        orgSet4 = results.getSearchResults();
    }
    //Print the results
    return "Assignable Dynamic Groups: " + orgSet1.toString() +
            "Dynamic Groups: " + orgSet2.toString() +
            "Static Groups: " + orgSet3.toString() +
            "Group with subscribable=true:" + orgSet4.toString();
    } catch (Exception ex) {
        ex.printStackTrace();
```

Code Example 5-6 Search Groups Code Sample

```
    return "got errors";
    }
```

Email Notification And The SDK

`amProfile.properties` is the localization file for the SDK. All strings that may be visible via an error message or a feature are stored in this file as `key=value` pairs. The file itself is located in *IdentityServer_base*/`SUNWam/locale`. Although all of the properties are not discussed in this section, there are some worth noting that pertain to email notification. The Administration Service has a number of notification attributes: User Creation, User Deletion and User Modification notification lists. When a user profile is created, deleted or modified, a notification email will be sent to the addresses listed as values of these attributes. To modify the message that is sent, the following `key=value` pairs in `amProfile.properties` need to be modified.

- 490=The user creation email subject can be defined with this key. The default is `WARNING: user creation notice`.
- 491=The user deletion email subject can be defined with this key. The default is `WARNING: user deletion notice`.
- 492=The user modification email subject can be defined with this key. The default is `WARNING: user modification notice`.
- 493=The user creation email body text can be defined with this key. The default is `user is created:` followed by the DN of the user.
- 494=The user deletion email body text can be defined with this key. The default is `user is deleted:` followed by the DN of the user.
- 495=The user modification email body text can be defined with this key. The default is `user is modified:` *user_DN*. attribute is changed: attribute old_value: *original_value* new value: *modified_value*
- 497=The entity from which the email comes is defined with this key. The default is Identity-Server.

More information on the Administration Service and the notification attributes themselves can be found in the *Sun ONE Identity Server Administration Guide*.

Caching And The SDK

Caching in the Identity Management SDK is used for storing all AMObject attributes (i.e., attributes of identity-related objects) that are retrieved from Directory Server. The cache does not hold AMObject directly, only its attributes. All attributes retrieved from Directory Server using the methods AMObject.getAttributes(), AMObject.getAttribute(String name) or AMObject.getAttributes(setAttributeNames) will be cached. Table 5-1 contains a listing of the recorded cache properties.

Table 5-1 Recorded Cache Properties

Information Name	What is recorded
Number of requests during this interval	Number of get requests during the specified interval
Number of cache hits during this interval	Number of hits during the specified interval
Hit ratio for this interval	Hit ratio for the specified interval
Total number of requests since server start	Overall number of get requests since a server re-start
Total number of cache hits since server start	Overall number of hits since a server re-start
Overall Hit ratio	Overall hit ratio since a server re-start
Total Cache Size	The total size of the cached information

Cache properties can be configured by modifying attributes in the AMConfig.properties file. For more information see “SDK Caching” on page 446 of Appendix A, “AMConfig.properties File,” in this manual.

Installing The SDK Remotely

It is possible for an external application to perform management functions on the Directory Server without installing the full Identity Server application at the external location. By installing the SUNWamsdk package using the pkgadd utility (or the installer), the Identity Management SDK can be installed on a non-Identity Server machine. For more details on the Identity Management SDK only installation option, refer to the *Java Enterprise System Installation Guide.*

NOTE If the SUNWamsdk package is installed remotely and Identity Server is running in SSL mode, a certificate database needs to be created. Create the database using the Sun ONE Web Server command line tool certutil or the Web Server console and then copy the database to the remote machine. For more information, see the Sun ONE Web Server documentation set.

Management Function Samples

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

NOTE Identity Server can authenticate and authorize against directories other than Sun ONE Directory Server (for example, Microsoft™ Active Directory), but Identity Server can not perform management functions against these directories such as creating users or deleting organizations.

Creating Objects

Typically, three steps are involved in creating an object with the SDK. The following three steps are specific to creating users but can be modified for any object.

To Create A User

1. Get AMStoreConnection object to connect to the data store.

2. From the AMStoreConnection, get AMPeopleContainer object where the users will be created.

3. In AMPeopleContainer object, create users.

Code Example 5-7 Sample Code To Create A User

```
/**
   * This method will describe the SDK usage for creating a user.
   * It uses AMStoreConnection to get the organization object
   * It also uses the Set Parameters to store the different
   * attributes of the user. It throws
   * an AMException if it's unble to create it and we throw
   * message "unable to create" to the GUI by catching the same
   */

    public String createUser(HttpServletRequest req, Set parameters,
AMStoreConnection conn) {
        try {
```

Code Example 5-7 Sample Code To Create A User *(Continued)*

```
            Map userAttributeMap = new HashMap();
            if (parameters.contains("uid")) {
                uid = req.getParameter("uid");
                storeUserAttributes("uid", uid, userAttributeMap);
            }
            if(parameters.contains("firstname")) {
                firstName = req.getParameter("firstname");
                storeUserAttributes("givenname", firstName,
userAttributeMap);
            }
            if(parameters.contains("lastname")) {
                lastName = req.getParameter("lastname");
                storeUserAttributes("sn", lastName, userAttributeMap);
            }
            if(parameters.contains("password")) {
                passWord = req.getParameter("userPassword");
                storeUserAttributes("userPassword", passWord,
userAttributeMap);
            }

            Map userMap1 = new HashMap();
            userMap1.put(uid, userAttributeMap);
            String orgDN = req.getParameter("orgName");
            String dn = "ou=People" + "," + orgDN;
            AMPeopleContainer ampc = conn.getPeopleContainer(dn);
            ampc.createUsers(userMap1);
            userDN = "uid=" + uid + "," + dn;
            /*
             * This is to keep the context of the user
             */
            contextUser = conn.getUser(userDN);
            return showCreateUserSuccess();
        } catch (Exception ex) {
            ex.printStackTrace();
          return "Unable to create";
    }
```

To Create An Organization

1. Get `AMStoreConnection` object to connect to the data store.

2. From the `AMStoreConnection`, get `AMOrganization` object for the top level organization.

3. In `AMOrganization` object, create sub-organization.

NOTE `org.createUsers` creates users directly under the organization. In order to create users in a people container, use the `AMPeopleContainer` object.

Retrieve Templates

Code Example 5-8 retrieves a service's dynamic templates by opening a connection to 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 5-8 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("dc=toporg,dc=com");
     // retrieve Dynamic  type AMTemplate for iPlanetAMSessionService
     AMTemplate template = org.getTemplate("iPlanetAMSessionService",
AMTemplate.DYNAMIC_TEMPLATE);
     // retrieve attributes
     String maxSessionTime =
template.getStringAttribute("iplanet-am-session-max-session-time");
     ...
```

TIP As an alternative to creating a new XML service file, `amUser.xml` can be modified. In this case, unregister the old `amUser` service file, modify it and re-register the modified file. Attribute/value pairs need to be integrated into the `amUser.properties` file for newly-defined internationalization keys. `ums.xml` does not need to be modified for this option.

Identity Management Samples

Identity Server contains samples that illustrate user management functions. These include a sample to add an attribute to the user profile and one to illustrate how to create organizations, users, roles, and services using the SDK. They can be found in *IdentityServer_base*`/SUNWam/samples/um`.

Adding User Attributes

This sample explains how to add new attributes to the User profile so that those new attributes can be managed via the user page in the Identity Server console. There are 2 ways this can be achieved: modify the existing `amUser.xml`, or create a new XML service file and import it into Identity Server.

Creating Objects With The SDK

This sample contains sample Java code that can be generated and run to create some identity-related objects including an organization, roles and users. The defined `SampleOrgOperations.java` creates an organization, gets the registered services, and adds them. `SampleUserOperations.java` and `SampleRoleOperations.java` can also be used for their respective purposes.

Chapter 6

Service Management

Sun™ ONE Identity Server provides a mechanism for the definition and management of services and their configuration data. Both eXtensible Markup Language (XML) files and Java™ interfaces are used for this purpose. This chapter provides information on how to define a service, the structure of the XML files and the service management application programming interfaces (API). It contains the following sections:

- Overview
- Defining A Custom Service
- DTD Files
- XML Service Files
- Service Management SDK

Overview

A *service* is a group of attributes 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 application. As an example of the first scenario, after creating a payroll module, a developer can create an XML service file that might include attributes to define an employee name, an hourly pay rate and an income tax rate. This XML file is then integrated into the Identity Server deployment so that these three attributes and their corresponding values can be stored in, and managed from, the Sun ONE Directory Server data store and Identity Server console, respectively.

Identity Server provides the necessary tools for administrators to define, integrate and manage groups of attributes as a service. Creating a service for management using the Identity Server console involves preparing 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, customize and manage the service using the Identity Server console. More specific information on this process can be found in "Defining A Custom Service" on page 243.

NOTE Throughout this chapter, the term *attribute* is used to illustrate 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 an XML service file.

XML Service Files

XML service files enable Identity Server to manage attributes that are stored in Directory Server. It is important to remember that 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 though, Identity Server loads a number of services it uses to manage the attributes of its own features; it manages and uses these values. 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 `/etc/opt/SUNWam/config/xml`. For more specific information on the XML files used in service management, see "XML Service Files" on page 286.

NOTE Any application with LDAP attributes can have its 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 "Defining A Custom Service" on page 243.

Document Type Definition Structure Files

The format of an XML file 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 the different types of XML files it uses. The

DTDs are located in *IdentityServer_base*/SUNWam/dtd. This chapter primarily concerns itself with sms.dtd, the file that defines the structure for all XML service files. Additional information on Identity Server DTDs can be found in "DTD Files" on page 253.

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 gives application developers the interfaces necessary to register and un-register services as well as manage 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 294.

Defining A Custom Service

To define a service for management using Identity Server, the developer must create an XML service file as well as configure an LDIF file for any object classes not already defined in Directory Server. Both, the XML service file and the new LDIF schema, must then be imported into Directory Server. Once imported, the service can be registered to an organization using Identity Server and its attributes managed and customized by the Identity Server administrator. The following steps detail the procedure used to define a service. The sections following the procedure explain each step in more detail.

1. Create an XML service file containing a group of attributes.

   This XML service 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 245. An explanation of the DTD syntax can be found in "The sms.dtd Structure" on page 255.

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

   Loading an LDIF file into Directory Server will add any newly defined or modified LDAP object classes and attributes to the directory tree. This step is only necessary when defining dynamic, policy and user attributes. (Using Identity Server-specific object classes and attributes do not require that changes be made to the LDAP schema.) Instructions on extending the LDAP schema can be found in “Extending The Directory Server Schema” on page 249. Additional information on identity-related objects and the Identity Server schema can be found in Chapter 5, “Identity Management,” of this manual and the *Sun ONE Identity Server Deployment Guide*, respectively. The Sun ONE Directory Server documentation contains information on the LDAP schema.

3. Import the XML service file into Directory Server using `amadmin`.

   Information on importing an XML service file and the `amadmin` command line utility can be found in “Importing The XML Service File” on page 251 and the *Sun ONE Identity Server Administration Guide,* respectively.

4. Configure a localization properties file and copy it into the *IdentityServer_base*`/SUNWam/locale` directory.

   The localization properties file must be created with accurate `i18nKey` fields. These console names map to fields 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 Console Localization Properties” on page 251 and “Localization Properties Files” on page 87 of Chapter 3, “Authentication Service,” in this manual.

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 abstract objects and updating `amEntrySpecific.xml` can be found in Chapter 5, “Identity Management,” of this manual. Information on modifying `amUser.xml` can be found in “Modifying A Default XML Service File” on page 288.

6. Register the service using Identity Server console.

   After importing the service into Directory Server, it can be registered to an organization and the attributes managed through the Identity Server console. Information on how this can be done is in the Service Configuration chapter in the *Sun ONE Identity Server Administration Guide*. Information on how to register the service using the command line can be found in "Registering The Service" on page 253.

Creating A Service File

The information in this section corresponds to Step 1 on page 243, 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. The DTD syntax can be found in "The sms.dtd Structure" on page 255.

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 `iPlanetAMAuth`*module_name*Service.)
- 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 as a whole, a specific service or a single 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 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 console. Global attributes are stored in Directory Server using specially-defined LDAP attributes so 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. An example of this would be the Platform Service.

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 is dependent on the address of an organization's LDAP server and the structure of their directory tree, respectively. Organization attributes are stored in Directory Server using specially-defined LDAP attributes so the LDAP schema does not need to be extended to add a new organization attribute.

NOTE Organization attributes are not inherited by sub-organizations. Only dynamic attributes can be inherited. For additional information, see “Attribute Inheritance” on page 248.

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 and roles which, in general, have access to any service assigned to its parent organization. Dynamic attributes are inherited by users that possess a role or belong to the organization. Because dynamic 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*. For information on these features, see Appendix E, "Directory Server Concepts," in this manual 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 specify the access control actions (or *privileges*) associated with a service. They become a part of the rules when rules are added to a policy. Examples include `canForwardEmailAddress` and `canChangeSalaryInformation`. The actions specified by these attributes can be associated with a resource if the `IsResourceNameAllowed` element is specified in the attribute definition. For example, in the web agent XML service file, `amWebAgent.xml`, `GET` and `POST` are defined as policy attributes with an associated URL resource as `IsResourceNameAllowed` is specified.

NOTE Out of the box, only the Policy Configuration Service uses policy attributes although they can be defined for any number of services.

User Attributes

User attributes are defined for 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. 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. For example, if 70 attributes are user-defined and an organization has two million users, each attribute is stored two million times. This, of course, only occurs if the service is assigned to the user and a value is set for them. 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 assigned to the user for the user attributes to be displayed on their User 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 attributes by registering it and creating a service template. Then, when a user possesses a role or belongs to an organization to which the service is registered, they inherit the dynamic attributes of the role or the service, respectively. Inheritance only occurs, though, when the service possessed is explicitly assigned to the user. A user can inherit attributes from multiple roles or parent organizations.

TIP Service templates created for a parent organization contain attributes that *trickle down* to sub-organizations. Therefore it is not necessary to create templates for sub-organizations unless the attribute values are being customized. Creating a large number of service templates will have a performance impact.

ContainerDefaultTemplateRole Attribute

Dynamic attributes are used in an XML service file if an administrator wants to define a particular attribute as one which is inherited by all identity objects to which the service is registered. After uploading the XML service file and registering the service to an organization or role, all users in the sub-trees of the organization or role will inherit the dynamic attributes. To accomplish this, Identity Server uses *classic CoS* and role templates (as described in Appendix E, "Directory Server Concepts"). ContainerDefaultTemplateRole is a default *filtered* role configured for each organization in which the LDAP object class iplanet-am-managed-person is the default filter. Every user in Identity Server is a member of iplanet-am-managed-person so every user in the organization possesses ContainerDefaultTemplateRole. Identity Server creates a separate CoS template for each registered service which points to the service's dynamic attributes. Because of this, any user who has ContainerDefaultTemplateRole (all of them, by default) will inherit the dynamic attributes of the service. The LDIF entry for ContainerDefaultTemplateRole is illustrated in Code Example 6-1.

Code Example 6-1 `ContainerDefaultTemplateRole` LDIF Entry

```
dn: cn=ContainerDefaultTemplateRole,o=example
objectClass: top
objectClass: nscomplexroledefinition
objectClass: nsfilteredroledefinition
objectClass: nsroledefinition
objectClass: ldapsubentry
nsRoleFilter: (objectclass=iplanet-am-managed-person)
```

Modifying Inheritance

The `nsRoleFilter` attribute (as displayed in Code Example 6-1 may be modified to allow objects other than users to inherit from `ContainerDefaultTemplateRole`. Formatting its value as, for example, `(|(objectclass=iplanet-am-managed-person)(objectclass=organization))` allows users and organizations to inherit the dynamic attributes. Any valid filter syntax can be used although typically it would be limited to attributes or objectclasses in the user entries. In addition, the relevant objectclass from the LDAP attributes must also be added to the entry.

Extending The Directory Server Schema

The information in this section corresponds to Step 2 on page 244, extending the LDAP schema in 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 the `ldapmodify` command line tool and an LDIF file as input.

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

To Extend The Directory Server LDAP Schema

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` *IdentityServer_base*`/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, 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 6-2.

Code Example 6-2 Sample LDIF Listing For Mail Service

```
objectClasses: ( 1.2.NEW
 NAME 'am-sample-mail-service'
 DESC 'SampleMail Service' SUP top AUXILIARY
 MAY ( am-sample-mail-service-status $
 am-sample-mail-root-folder $
 am-sample-mail-sentmessages-folder $
 am-sample-mail-indent-prefix $
 am-sample-mail-initial-headers $
 am-sample-mail-inactivity-interval $
 am-sample-mail-auto-load $
 am-sample-mail-headers-perpage $
 am-sample-mail-quota $
 am-sample-mail-max-attach-len $
 am-sample-mail-can-save-address-book-on-server )
 X-ORIGIN ’user defined’ )
attributeTypes: ( 11.24.1.996.1
 NAME ’am-sample-mail-service-status’
 DESC ’SampleMailService Attribute’
 SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
 X-ORIGIN ’user defined’ )
```

Adding Identity Server Object Classes To Existing Users

If a new service is created and the service's users already exist, the service's object classes need to be added to the user's LDAP entries. To do this, Identity Server provides migration scripts for performing batch updates to already-existing user entries. No LDIF file need be created. These scripts and the procedures are described in the *Sun ONE Identity Server Migration Guide*. Alternatively, registered services can be added to each user by selecting the service on their Properties page although, for an organization with many users, this would be time-consuming.

CAUTION It is not recommended to use `ldapmodify` to extend the schema.

Importing The XML Service File

The information in this section corresponds to Step 3 on page 244, importing an XML service file into Identity Server. This step is important as it serves to populate Directory Server and Identity Server with the newly defined service attributes.

1. Change to the Identity Server install directory:

   `cd` *IdentityServer_base*`/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 must be deleted before importing the newly modified XML service file. Information on this function can be found in the Sun ONE Directory Server documentation.

Configuring Console Localization Properties

The information in this section corresponds to Step 4 on page 244, configuring a localization properties file. A localization properties file specifies the locale-specific screen text that an administrator or user will see when directed to a service's attribute configuration page.

NOTE For certain services, this file also localizes error messages, Java exceptions and email notification specifics. This section though concerns itself only with service-related values. Additional information can be found in “Localization Properties Files” on page 87 of Chapter 3, “Authentication Service,” in this manual.

The localization properties files are located in the *IdentityServer_base*`/SUNWam/locale` directory. They are generally named using the format `am`*service_name*`.properties`. Code Example 6-3 is the localization properties file for the Client Detection service named `amClientDetection.properties`.

Code Example 6-3 amClientDetection.Properties File

```
...
# attr descriptions msgs
#
iplanet-am-client-detection-service-description=Client Detection
a100=Client Types
a101=Default Client Type
a102=Client Detection Class
a103=Client Detection Enabled
a100.link=Edit
unknown_key=requested key is not available in the property
null_key=null key passed to getProperty
null_clientType=client type is null
unknown_clientType=requested clientType doesn't exist
update_error=notification received between setproperty and store. Need to do
setproperty again.
```

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` fields defined for each attribute in a service in the XML service file. The keys also determine the order in which the fields are displayed on screen as the keys are displayed 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.

TIP If a localization properties file is modified, Identity Server needs to be restarted to see the changes. If importing a new localization properties file, Identity Server does not need to be restarted.

Localizing With Two Languages

When one instance of Identity Server is localized with two languages, the localization properties files still go into the same directory. Each file name would be appended with a suffix to match the locale. For example, if French localization packages are added, the file name would be `am`*service_name*`_fr.properties`. If Spanish localization packages are added, that properties file name would be `am`*service_name*`_es.properties`.

NOTE	Information on downloading and installing localized versions of Identity Server can be found at `http://wwws.sun.com/software/download/inter_ecom.html`.

Updating Files For Abstract Objects

For information corresponding to Step 5 on page 244, updating the `amEntrySpecific.xml`, see Chapter 5, “Identity Management,” of this manual. For information corresponding to Step 5, updating the `amUser.xml`, see “XML Service Files” on page 286.

Registering The Service

The information in this section corresponds to Step 6 on page 245, 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*. An alternate process to register a service is to use the `amAdmin.dtd`, batch processing templates and the command line. Information can be found in “The amAdmin.dtd Structure” on page 265 and “Batch Processing With XML Templates” on page 290.

NOTE	To register a service, ensure that Identity Server is properly binding to the Directory Server.

DTD Files

Identity Server contains numerous DTD files to define the structures for the XML files used in Identity Server . The DTDs are located in *IdentityServer_base*`/SUNWam/dtd` and include:

- 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 "Authentication Programming Interfaces" on page 153 in Chapter 3, "Authentication Service," of this manual.
- 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 265.
- amWebAgent.dtd—defines the structure for XML files used to handle requests from, and send responses to, web agents. This file is deprecated and remains for purposes of backward compatibility.
- policy.dtd—defines the structure for XML files used to store policies in Directory Server. Information on this document can be found in "Policy Definition Type Document" on page 312 of Chapter 7, "Policy Service," of this manual.
- remote-auth.dtd—defines the structure for XML files used by the Authentication Service's remote Authentication API. Information on this document can be found in "The remote-auth.dtd Structure" on page 135 of Chapter 3, "Authentication Service," of this manual.
- 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 Appendix B, "serverconfig.xml File," in this manual.
- sms.dtd—which defines the structure for XML service files. Information on this document can be found in "The sms.dtd Structure" on page 255.
- web-app_2_2.dtd—defines the structure for XML files used by the Identity Server deployment container to deploy J2EE applications. The corresponding XML file is called a *deployment descriptor* which specifies container options and describes specific configuration requirements to be resolved by the deployer.

CAUTION None of the DTD files should be modified. The APIs and their internal parsing functions are based on the installed definitions. Any alterations to the DTD files will hinder the operation of Identity Server.

The sms.dtd Structure

The sms.dtd defines the data structure for all XML service files. It is located in the *IdentityServer_base*/SUNWam/dtd directory. The sms.dtd enforces the developer to define each service attribute as one of five 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, Policy, and User. More information on these types can be found in "Service Attributes" on page 245.

An explanation of the main elements defined by the sms.dtd follows. Each element includes a number of XML attributes which are also explained. Explanations of the remaining elements can be found in the sms.dtd file itself. Identity Server currently supports only about some 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 allows for the definition of multiple services per one XML file. Its immediate sub-element is the Service Element. Code Example 6-4 on page 255 illustrates the *ServicesConfiguration* element as defined in the amClientDetection.xml service file located in /etc/opt/SUNWam/config/xml.

Code Example 6-4 ServicesConfiguration and Service Element

```
...
<ServicesConfiguration>
<Service name="iPlanetAMClientDetection" version="1.0">
<Schema...>
...
```

Service Element

The *Service* element defines the schema for one given service. A number of different services can be defined in one XML file using this element, although this is not recommended. Currently, Identity Server supports the following sub-elements: Schema Element (which defines the service's attributes as either *Global*, *Organization*, *Dynamic*, *User* or *Policy*) and *Configuration*. The required XML attributes for the *Service* element are the name of the service, such as *iPlanetAMLogging*, and the version number of the XML service file itself. Code Example 6-4 on page 255 also illustrates the *Service* element, its attributes and the opening Schema tag.

Schema Element

The *Schema* element is the parent of the family of elements that define the service's attributes and their default values. The sub-elements can be the Global Element, Organization Element, Dynamic Element, User Element or Policy Element. The required XML attributes of the *Schema* element include the serviceHierarchy Attribute, the i18nFileName Attribute, the i18nKey Attribute, and the propertiesViewBeanURL Attribute.

serviceHierarchy Attribute

When a new service is configured, its name will be dynamically displayed in the Navigation frame of the console based on the value of this attribute. The value is a "/" separated string. Each "/" portion of the string represents a level in the hierarchy. Code Example 6-5 on page 256 illustrates the `serviceHierarchy` attribute as defined in `amClientDetection.xml`. `iPlanetAMClientDetection` is the name of the service. The name used for display in the console, though, is defined by the `i18nKey` (or i18nKey Attribute), and retrieved from the service's localization file defined by the i18nFileName Attribute. In this example, the value of `iplanet-am-client-detection-service-description` will be found in `amClientDetection.properties` and its value displayed. The service name will be displayed below the Identity Server Configuration header in the left frame of the Service Configuration module. To prevent a service from displaying in the console, either remove the `serviceHierarchy` attribute or set it's value to `""`, as in `serviceHierarchy=""`.

NOTE `DSAMEConfig` as used in Code Example 6-5 and all XML service files refers to the Identity Server Configuration header. The use of DSAME is a holdover from the previous name of Identity Server. This is defined in the `amAdminModuleMsgs.properties` file located in *IdentityServer_base/*`SUNWam/locale`.

Code Example 6-5 `i18nFileName`, `i18nKey` and `serviceHierarchy` Attributes

```
...
<Schema
 serviceHierarchy="/DSAMEConfig/iPlanetAMClientDetection"
```

Code Example 6-5 `i18nFileName`, `i18nKey` and `serviceHierarchy` Attributes

```
            i18nFileName="amClientDetection"
 i18nKey="iplanet-am-client-detection-service-description">
...
```

i18nFileName Attribute

The `i18nFileName` attribute refers to the localization properties files. It takes a value equal to the name of the localization properties file for the defined service (minus the `.properties` file extension). For example, Code Example 6-5 defines the name of the properties file as `amClientDetection`.

i18nKey Attribute

The value of the `%i18nIndex` attribute maps to the final, localized name of the service to be displayed in the Identity Server console as it is defined in the localization properties file.

NOTE The `%i18nIndex` attribute is defined as an *entity* at the top of the `sms.dtd`. In the configured XML service files, `%i18nIndex` is replaced by `i18nKey` and its corresponding value.

For example, Code Example 6-5 refers to the value of the `iplanet-am-client-detection-service-description` attribute as defined in `amClientDetection.properties`. This value is the name of the service as it will be displayed in the Identity Server console; in this case, **Client Detection** is the name defined in `amClientDetection.properties`. (Remember, the value of the defined attribute might not be in English.) More information on the localization properties file can be found in Chapter 3, “Authentication Service,” of this manual.

NOTE If the `i18nKey` value is blank `(i18nKey="")`, the Identity Server console will not display the attribute.

propertiesViewBeanURL Attribute

The default display for a service is a simple table showing the attribute name and its value. The `propertiesViewBeanURL` attribute provides the URL to the Java bean used by the console to generate this display. It is possible to override the default display by creating a new class and defining the URL to this class as a value of this attribute. If no value is specified, the display is created by the console.

Service Attribute Elements

The next five elements are sub-elements of the “Schema Element” on page 256; they are the declarations of the service’s Identity Server attributes. When defining a service, each attribute must be defined as either a Global Element, an Organization Element, a Dynamic Element, a User Element, or a Policy Element. Any configuration of these elements (all of them or none of them) can be used depending on the service. Each attribute defined within these elements is itself defined by an AttributeSchema Element.

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 “Global Attributes” on page 245.

serviceObjectClasses Attribute. The `serviceObjectClasses` attribute is a global attribute defined in an XML service file that contains either dynamic or user elements (attributes). The value of this attribute is an object class set in the LDAP entries (stored in Directory Server) for users whom are registered to the service. It allows any user with this object class to be dynamically assigned the service’s dynamic or user attributes, if any exist.

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

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

Code Example 6-6 serviceObjectClass Defined As Global Element

```
...
<Global>
            <AttributeSchema name="serviceObjectClasses"
                type="list"
                syntax="string"
                i18nKey="">
                <DefaultValues>
 <Value>iplanet-am-client-detection-service</Value>
```

Code Example 6-6 serviceObjectClass Defined As Global Element

```
...
<Global>
                </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 246.

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 use the Directory Server features, CoS and roles. For additional information, see “Dynamic Attributes” on page 246.

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 247.

Policy Element

The Policy element defines Identity Server attributes intended to provide actions (or *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 of privileges might include `canChangeSalaryInformation` and `canForwardEmailAddress`. For additional information, see “Policy Attributes” on page 247.

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 directory tree, maintainPriority which defines whether priority is to be honored among its peer elements, and "i18nKey Attribute" on page 257.

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 "Service Attribute Elements" on page 258 as well as the SubSchema element described in "SubSchema Element" on page 260. It defines the structure for each configurable parameter (or attribute) of a service. 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 the "name Attribute", the "type Attribute", the "uitype Attribute", the "syntax Attribute", the "cosQualifier Attribute", rangeStart, rangeEnd, minValue, maxValue, validator, the "any Attribute", the "propertiesViewBeanURL Attribute" on page 257 and, the "i18nKey Attribute" on page 257. Code Example 6-7 on page 260 illustrates an *AttributeSchema* element taken from amUser.xml, its attributes and their corresponding values. Note that this example attribute is a Dynamic attribute.

Code Example 6-7 AttributeSchema Element With Attributes

```
...
<Dynamic>
  <AttributeSchema name="iplanet-am-user-login-status"
                type="single_choice"
                syntax="string"
                any="display"
                i18nKey="d105">
                <ChoiceValues>
 <ChoiceValue i18nKey="u200">Active</ChoiceValue>
 <ChoiceValue i18nKey="u200">Inactive</ChoiceValue>
                </ChoiceValues>
                <DefaultValues>
```

Code Example 6-7 `AttributeSchema` Element With Attributes *(Continued)*

```
                    <Value>Active</Value>
                </DefaultValues>
 </AttributeSchema>
...
```

name Attribute

This required XML attribute defines the a name for the attribute. Any string format can be used but attribute names must be in lower-case. Code Example 6-7 on page 260 defines it with a value of `iplanet-am-user-login-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 any 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 choose a single value from a list of options. A default value must be defined from the list.
- `multiple_choice` specifies that the user can choose multiple values from a list of options. A default value must be defined from the list.

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 ChoiceValues sub-element, ChoiceValue. Code Example 6-7 on page 260 defines the `type` as `single_choice`; the `ChoiceValues` attribute defines the list of options as `Active` and `Inactive` with the DefaultValue as `Active`.

syntax Attribute

The `syntax` attribute defines the format of the value. The default value for syntax is `string` but, it can be defined as any 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 percentage.
- `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.

NOTE If creating policy, note that the policy schema only supports boolean, string, password, dn, email, numeric, percent, number, decimal_number, and number_range. It does not support paragraph, encrypted_password, decimal_range, xml, and date (some of which are not defined above).

uitype Attribute

This attribute specifies the HTML element that will be displayed in the Identity Server console. Possible values include `radio`, `link`, `button`, or `name_value_list`. No value defined for this attribute displays a default element based on the information in Table 2-1 on page 55 of Chapter 2, “The Identity Server Console.”

NOTE The “type Attribute”specifies the kind of value an attribute will take. The “syntax Attribute” defines the format of that value. The “uitype Attribute” specifies the HTML element. The values of these attributes can be mixed and matched to alter the console display. See “To Change The Default Attribute Display Elements” on page 54 of Chapter 2, “The Identity Server Console,” in this manual for information on how these attributes work together.

DefaultValues Sub-Element. Defining a syntax might also necessitate 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.

CAUTION Default values of User attributes are not inherited by users when the service is assigned using the Identity Server console.

For example, all instances of the LDAP Authentication Service use the `port` attribute so a default value of `389` is 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.) Code Example 6-8 on page 263 from `amAuthLDAP.xml` illustrates this scenario.

Code Example 6-8 DefaultValues In `amAuthLDAP.xml`

```
...
<Organization>
                <AttributeSchema name="iplanet-am-auth-ldap-server"
                    type="list"
                    syntax="string"
                    i18nKey="a101">
                    <DefaultValues>
                        <Value>identity_server_host.com:389</Value>
                    </DefaultValues>
                </AttributeSchema>
...
```

cosQualifier Attribute

This attribute defines how Identity Server resolves conflicting dynamic attribute values assigned to the same user object. The value of `cosQualifier` will appear as a qualifier to the `cosAttribute` in the LDAP entry of the CoS definition.

NOTE The priority level is defined by the Conflict Resolution Level attribute. More information on this attribute can be found in the *Sun ONE Identity Server Administration Guide*.

The value of `cosQualifier` can be defined as:

- `default` indicates that if there are two conflicting attributes assigned to the same user object, the one with the highest priority takes precedence. For more information on CoS conflicts, see Appendix E, “Directory Server Concepts,” in this manual.
- `override` indicates that the CoS template value defined at the service itself overrides any priority value defined in the user entry; that is, CoS takes precedence over a defined 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 template values. For example , if there are multiple templates for a particular service that contains dynamic attributes and they are applied to a user (based on the user's roles), a merged list of attributes will be returned. merge-schemes works only for dynamic (or COS) type attributes.

If the cosQualifier 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 both the administrator and end user profile pages. The attribute is read / write for administrators and end users.
- adminDisplay specifies that the attribute will display on the administrator profile page only. It will not appear on the 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 end users. It is displayed on the end user profile pages as a non-editable label.
- 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 signifying it as a required field.
- optional specifies that a value for the attribute is not required in order for the object to be created. The attribute will display on the Create page without an asterisk signifying it as an optional field.
- filter specifies that the attribute will display on the Advanced 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.

NOTE Setting the *any* attribute to "" (any="") will prevent the attribute to which it refers from being displayed in the console.

The amAdmin.dtd Structure

The amAdmin.dtd defines the data structure for an XML file which can be used to perform batch operations on the directory tree using the amAdmin command line tool. The file reflects the structure of the Identity Server SDK and is located in the *IdentityServer_base*/SUNWam/dtd directory. Possible command line operations include reads and gets on the attributes as well as creations and deletions of Identity Server objects (roles, organizations, users, people containers, and groups).

NOTE XML files that are created based on the amAdmin.dtd are used as input with the amAdmin command line tool. More information on this tool can be found in the *Sun ONE Identity Server Administration Guide.*

The following sections explain the elements and attributes of the amAdmin.dtd using the sample XML templates installed with Identity Server for illustration. These samples can be found in *IdentityServer_base*/SUNWam/samples/admin/cli/bulk-ops.

Requests Element

The *Requests* element is the root element of the XML file. It must contain at least one sub-element to define the object(s) (Organization, Container, People Container, Role and/or Group, et. al.) upon which the configured actions can be performed. The *Requests* element must contain at least one of the following sub-elements:

- OrganizationRequests
- SchemaRequests
- ServiceConfigurationRequests
- ContainerRequests
- PeopleContainerRequests

- `RoleRequests`
- `GroupRequests`
- `UserRequests`
- `ListAccts`

To enable batch processing, the root element can take more than one of these sub-element requests.

CAUTION If multiple sub-elements are specified, they must occur in the order in which they appear in the `amAdmin.dtd`. For example, a `CreateUser` cannot come before a `CreateRole` in the same OrganizationRequests element.

Code Example 6-9 illustrates the *Requests* element tag and its corresponding *OrganizationRequests* sub-element which details the creation of two roles, two groups, a suborganization, a container, and a people container in the organization with the LDAP Distinguished Name (DN), `dc=example,dc=com`.

Code Example 6-9 Portion Of `createRequests.xml`

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

    <CreateSubOrganization createDN="SubOrg1"/>
    <CreateContainer createDN="Container1"/>
    <CreatePeopleContainer createDN="People2"/>
    <CreateRole createDN="ManagerRole"/>
    <CreateRole createDN="EmployeeRole"/>
    <CreateGroup createDN="ContractorsGroup"/>
    <CreateGroup createDN="EmployeesGroup"/>

</OrganizationRequests>
...
```

OrganizationRequests Element

The *OrganizationRequests* element defines actions that can be performed on Organization objects. The required XML attribute for this element is the LDAP DN of the organization on which the configured requests will be performed. This element can have one or more sub-elements. (Different *OrganizationRequests* elements can be defined in one file to modify more than one organization.) Code Example 6-9 defines a myriad of objects to be created under the top level organization, dc=example,dc=com. The sub-elements of *OrganizationRequests* include:

- CreateSubOrganization
- CreateContainer
- CreatePeopleContainer
- CreateGroupContainer
- CreateRole
- CreateUser
- CreateGroup
- CreatePolicy
- RegisterServices
- ModifySubOrganization
- ModifyServiceTemplate
- AddServiceTemplateAttributeValues
- RemoveServiceTemplateAttributeValues
- GetServiceTemplate
- DeleteServiceTemplate
- ModifyPeopleContainer
- ModifyRole
- GetSubOrganizations
- GetPeopleContainers
- GetRoles
- GetGroups
- GetUsers
- CreateServiceTemplate
- UnregisterServices
- GetRegisteredServiceNames
- GetNumberOfServices
- DeleteRoles

- `DeleteGroups`
- `DeletePolicy`
- `DeletePeopleContainers`
- `DeleteSubOrganizations`
- `AddSubConfiguration`
- `DeleteSubConfiguration`
- `CreateAuthenticationDomain`
- `CreateHostedProvider`
- `CreateRemoteProvider`
- `DeleteAuthenticationDomain`
- `DeleteProvider`
- `GetProvider`
- `GetAuthenticationDomain`
- `ModifyHostedProvider`
- `ModifyRemoteProvider`
- `ModifyAuthenticationDomain`

ContainerRequests Element

The *ContainerRequests* element defines actions that can be performed on Container objects. The required XML attribute for this element is the LDAP DN of the container on which the configured requests will be performed. This element can have one or more sub-elements. (Different *ContainerRequests* elements can be defined in one file to modify more than one container.) The syntax for this element is basically the same as that of the *OrganizationRequests* element illustrated in Code Example 6-9 on page 266. The sub-elements of *ContainerRequests* can include:

- `CreateSubContainer`
- `CreatePeopleContainer`
- `CreateGroupContainer`
- `CreateRole`
- `CreateGroup`
- `CreateServiceTemplate`
- `ModifyServiceTemplate`

- `AddServiceTemplateAttributeValues`
- `RemoveServiceTemplateAttributeValues`
- `GetServiceTemplate`
- `ModifySubContainer`
- `ModifyPeopleContainer`
- `ModifyRole`
- `GetSubContainers`
- `GetPeopleContainers`
- `GetRoles`
- `GetGroups`
- `GetUsers`
- `CreateUser`
- `RegisterServices`
- `UnregisterServices`
- `DeleteServiceTemplate`
- `GetRegisteredServiceNames`
- `GetNumberOfServices`
- `DeleteRoles`
- `DeleteGroups`
- `DeletePeopleContainers`
- `DeleteSubContainers`

PeopleContainerRequests Element

The *PeopleContainerRequests* element defines actions that can be performed on People Container objects. The required XML attribute for this element is the LDAP `DN` of the people container on which the configured requests will be performed. This element can have one or more sub-elements. (Different *PeopleContainerRequests* elements can be defined in one document to modify more than one people container.) The syntax for this element is basically the same as that of the *OrganizationRequests* element illustrated in Code Example 6-9 on page 266. The sub-elements of *PeopleContainerRequests* can include:

- `CreateSubPeopleContainer`

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

RoleRequests Element

The *RoleRequests* element defines actions that can be performed on roles. The required XML attribute for this element is the LDAP `DN` of the role on which the configured requests will be performed. This element can have one or more sub-elements. (Different *RoleRequests* elements can be defined in one document to modify more than one role.) The syntax for this element is the same as that of the *OrganizationRequests* element illustrated in Code Example 6-9 on page 266. The sub-elements of *RoleRequests* can include:

- `CreateServiceTemplate`
- `ModifyServiceTemplate`
- `GetServiceTemplate`
- `GetNumberOfUsers`
- `GetUsers`
- `RemoveUsers`
- `AddUsers`

GroupRequests Element

The *GroupRequests* element defines actions that can be performed on Group objects. The required XML attribute for this element is the LDAP `DN` of the group on which the configured requests will be performed. This element can have one or more sub-elements. (Different *GroupRequests* elements can be defined in one document to modify more than one group.) The syntax for this element is the same as that of the *OrganizationRequests* element illustrated in Code Example 6-9 on page 266. The sub-elements of *GroupRequests* can include:

- `CreateSubGroup`

- `GetSubGroups`
- `GetNumberOfUsers`
- `GetUsers`
- `ModifySubGroups`
- `AddUsers`
- `RemoveUsers`
- `DeleteSubGroups`

UserRequests Element

The *UserRequests* element defines actions that can be performed on User objects. The required XML attribute for this element is the LDAP `DN` of the user on which the configured requests will be performed. This element can have one or more sub-elements. (Different *UserRequests* elements can be defined in one document to modify more than one user.) The syntax for this element is the same as that of the *OrganizationRequests* element illustrated in Code Example 6-9 on page 266. The sub-elements of *UserRequests* can include:

- `RegisterServices`
- `UnregisterServices`

ServiceConfigurationRequests Element

The *ServiceConfigurationRequests* element defines actions that can be performed on a specific service. The required XML attribute for this element is *serviceName*; it specifies the service on which the configured requests will be performed. This element can have one or more sub-elements. The syntax for this element is the same as that of the *OrganizationRequests* element illustrated in Code Example 6-9 on page 266. The sub-elements of *ServiceConfigurationRequests* can include:

- `AddSubConfiguration`
- `DeleteSubConfiguration`
- `DeleteAllServiceConfiguration`

AddSubConfiguration Element

The *AddSubConfiguration* element adds a secondary schema to an existing service. The AttributeValuePair Element must be defined for each attribute configured in the subconfiguration. The required XML attributes are `subConfigName`, `subConfigID`, `priority` and `serviceName`.

NOTE Attributes defined for a subconfiguration are validated against attributes defined in a subschema based on sms.dtd. A subconfiguration is defined for an organization, choosing from attributes globally defined in the subschema. For more information, see “SubSchema Element.”

DeleteSubConfiguration Element

The *DeleteSubConfiguration* element deletes an existing secondary schema from a service. The required XML attributes are `subConfigName` and `serviceName` which takes a string value.

DeleteAllServiceConfiguration Element

The *DeleteAllServiceConfiguration* element deletes all configurations relating to a service and removes them from the data store. The required XML attribute is `userAtt` which specifies whether to delete the user attributes related to the service.

AttributeValuePair Element

The *AttributeValuePair* element can be a sub-element of many of the batch processing requests. It can have two sub-elements, *Attribute* and *Value*, neither of which may have sub-elements. Code Example 6-10 illustrates that a sub-people container, `ou=SubPeople2,ou=People2,dc=example,dc=com`, and a user, `dpUser`, will be created with the attributes of the two objects defined as per the attribute/value pairs.

Attribute Element

The *Attribute* element must be paired with a *Value* element. The *Attribute* element itself contains no other elements. The required XML service attribute for the *Attribute* element is `name` which is equal to the name of the attribute that is being processed. Any string format can be used without spaces.

Value Element

The *Value* element defines the value of the Attribute element. More than one *Value* element can be specified for an Attribute. The *Value* element contains no other elements and it contains no XML service attributes.

Code Example 6-10 Another Portion Of `createRequests.xml`

```
...
<PeopleContainerRequests DN="ou=People2,dc=example,dc=com">

        <CreateSubPeopleContainer createDN="SubPeople2">
                <AttributeValuePair>
```

Code Example 6-10 Another Portion Of `createRequests.xml` *(Continued)*

```
                        <Attribute name="description"/>
                        <Value>SubPeople description</Value>
                </AttributeValuePair>
        </CreateSubPeopleContainer>

        <CreateUser createDN="dpUser">
           <AttributeValuePair>
              <Attribute name="cn"/>
              <Value>dpUser</Value>
           </AttributeValuePair>
           <AttributeValuePair>
              <Attribute name="sn"/>
              <Value>dpUser </Value>
           </AttributeValuePair>
           <AttributeValuePair>
              <Attribute name="userPassword"/>
              <Value>12345678</Value>
           </AttributeValuePair>
        </CreateUser>
...
```

Create*Object* Elements

The *CreateSubOrganization*, *CreateContainer*, *CreatePeopleContainer*, *CreateRole*, *CreateGroup*, *CreateServiceTemplate*, *CreateUser*, *CreateSubContainer*, *CreateSubGroup*, *CreateSubPeopleContainer* elements create a sub-organization, container, people container, role, group, service template, user, sub-container, sub-group, and sub-people container, respectively. The object is created in the DN that is defined in the *<Object>Requests* element under which the particular Create*<Object>* element is being defined. AttributeValuePair Elements may be defined (or not). The required XML attribute for each element is `createDN`; it takes the `DN` of the object to be created. Code Example 6-10 on page 272 illustrates an example of *CreateSubPeopleContainer* and *CreateUser*. The DN is defined in the *PeopleContainerRequests* element as `ou=People2,dc=example,dc=com`.

NOTE *CreateGroup*/*CreateSubGroup* and *CreateRole* each have an additional attribute: `groupType` and `roleType`, respectively. `groupType` defines whether it is a static group, a filtered group or an assignable (dynamic) group. `roleType` defines whether it is a static role or a filtered role.

CreatePolicy Element

The *CreatePolicy* element creates one or more policy attributes. It takes the *Policy* element as a sub-element; `createDN` is the required XML attribute which takes the `DN` of the organization where the policy will be created. This and the following nested elements are illustrated in Code Example 6-11 on page 276. This file is `SamplePolicy.xml`, part of the policy sample application located in *IdentityServer_base*`/SUNWam/samples/policy.`

NOTE The following policy elements are the elements extracted from `amAdmin.dtd` for inclusion into the `policy.dtd`. More information can be found in “Policy Definition Type Document” on page 312 in Chapter 7, “Policy Service,” of this manual.

Policy Element. The *Policy* sub-element defines the permissions or *rules* of the policy and to whom / what the rule applies or the *subject*. It also defines whether or not the policy is a *referral* (delegated) policy and whether there are any restrictions (or *conditions*) to the policy. It may contain one or more of the following sub-elements: *Rule*, *Conditions*, *Subjects*, or *Referrals*. The required XML attributes are `name` which specifies the name of the policy and `referralPolicy` which identifies whether or not the policy is a delegated one.

Rule Element. The *Rule* sub-element defines the specific permission of the policy and 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* element defines the name of the service to which the policy applies. This element represents the service type. It contains no other elements. The value is exactly as that defined in the service’s XML file (based on the `sms.dtd`). The XML service attribute for the *ServiceName* element is the name of the service (which takes a string value).

- 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.

- AttributeValuePair Element

  The *AttributeValuePair* sub-element defines the action names and corresponding action values of the rule. For additional information, see "AttributeValuePair Element" on page 272.

Subjects Element. The *Subjects* sub-element identifies a collection of objects to which the policy applies; this overview collection is chosen based on membership in a group, ownership of a role or individual users. It takes the *Subject* sub-element. The XML attributes it can be defined with are `name` which defines a name for the collection, `description` which takes a description and `includeType` which defines whether the collection is as defined or its inverse (i.e.: the policy applies to users who are NOT members of the subject).

Subject Element. The *Subject* sub-element identifies a collection of objects to which the policy applies; this collection pinpoints more specific objects from the collection defined by the Subjects element. Membership can be based on roles, group membership or simply a listing of individual users. It takes as a sub-element the AttributeValuePair Element. Its required XML attribute is `type` which identifies a generic collection of objects from which the specifically defined subjects are taken. Other XML attributes include `name` which defines a name for the collection and `includeType` which defines whether the collection is as defined or its inverse (i.e.: the policy applies to users who are NOT members of the subject).

Referrals Element. The *Referrals* sub-element identifies a collection of policy assignments. It takes the *Referral* sub-element. The XML attributes it can be defined with are `name` which defines a name for the collection and `description` which takes a description. (Code Example 6-11 is not an example of a referral policy so there is not a Referrals element definition.)

Referral Element. The *Referral* sub-element identifies a specific policy assignment. It takes as a sub-element the AttributeValuePair Element. Its required XML attribute is `type` which identifies a generic collection of assignments from which the specifically defined referrals are taken. It can also include the `name` attribute which defines a name for the collection. (Code Example 6-11 is not an example of a referral policy so there are no Referral elements definition.)

Conditions Element. The *Conditions* sub-element identifies a collection of policy restrictions (time range, authentication level, et.al.). It must contain one or more of the *Condition* sub-element. The XML attributes it can be defined with are `name` which defines a name for the collection and `description` which takes a description.

Condition Element. The *Condition* sub-element identifies a specific policy restriction (time range, authentication level, et.al.). It takes as a sub-element the AttributeValuePair Element. Its required XML attribute is `type` which identifies a generic collection of restrictions from which the specifically defined conditions are taken. It can also include the `name` attribute which defines a name for the collection.

NOTE The Condition element might be used to configure policy for different URIs on the same domain. For example, `http://org.example.com/hr` can only be accessed by `org.example.net` from 9 am to 5 pm yet `http://org.example.com/finance` can be accessed by `org.example2.net` from 5 am to 11 pm. By defining an IP Condition attribute/value pair together with a SimpleTime Condition attribute/value pair and specifying `http://org.example.com/hr/*.jsp` as the resource, the policy would apply to all the JSPs under `http://org.example.com/hr`.

Code Example 6-11 SamplePolicy.xml

```
<Requests>
<OrganizationRequests DN="dc=iplanet,dc=com">

<CreatePolicy createDN="dc=iplanet,dc=com">
    <Policy name="PolicyOne" referralPolicy="false" >
    <Rule name="dsdasd">
    <ServiceName name="SampleWebService" />
    <ResourceName name="http://www.sun.com/public" />
        <AttributeValuePair>
            <Attribute name="GET" />
            <Value>allow</Value>
        </AttributeValuePair>
        <AttributeValuePair>
            <Attribute name="DELETE" />
            <Value>allow</Value>
        </AttributeValuePair>
        <AttributeValuePair>
            <Attribute name="PUT" />
            <Value>allow</Value>
        </AttributeValuePair>
        <AttributeValuePair>
            <Attribute name="POST" />
            <Value>allow</Value>
        </AttributeValuePair>
    </Rule>
    <Subjects name="Subjects1" description="">
    <Subject name="subject1" type="Organization">
        <AttributeValuePair>
            <Attribute name="Values"/>
            <Value>dc=iplanet,dc=com</Value>
            <Value>o=nicp,dc=iplanet,dc=com</Value>
        </AttributeValuePair>
    </Subject>
```

Code Example 6-11 SamplePolicy.xml *(Continued)*

```
    </Subjects>
    <Conditions name="Conditions1" description="">
    <Condition name="condition1" type="SampleCondition">
        <AttributeValuePair>
            <Attribute name="userNameLength"/><Value>5</Value>
        </AttributeValuePair>
    </Condition>
    </Conditions>
    </Policy>
</CreatePolicy>

</OrganizationRequests>
</Requests>
```

CreateServiceTemplate Element

The *CreateServiceTemplate* element creates a service template for the organization defined under 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 6-12 illustrates a User service template being registered to `ou=Container1,dc=example,dc=com`.

Code Example 6-12 cont`CreateServiceTemplateRequests.xml` File

```
...
<Requests>
<ContainerRequests DN="ou=Container1,dc=example,dc=com">

   <CreateServiceTemplate>
       <Service_Name>iPlanetAMUserService</Service_Name>
   </CreateServiceTemplate>

</ContainerRequests>
</Requests>
```

Delete*Object* Elements

The *DeleteSubOrganizations*, *DeletePeopleContainers*, *DeleteGroups*, *DeleteRoles*, *DeleteSubContainers*, *DeleteSubGroups*, *DeleteSubPeopleContainers*, and *DeleteUsers* elements delete a sub-organization, people container, group, role, sub-container, sub-group, sub-people container and user, respectively. The object is deleted from the DN that is defined in the *<Object>Requests* element under which the particular *Delete<Object>* element is being 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; it's default value is *false*. The `DN` sub-element takes a character value equal to the DN of the object to be deleted. Code Example 6-13 illustrates an example of some of these concepts. The DN is defined in the *OrganizationRequests* element as `dc=example,dc=com`.

Code Example 6-13 orgDeleteRequests.xml

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

    <DeleteRoles>
       <DN>cn=ManagerRole,dc=example,dc=com</DN>
       <DN>cn=EmployeeRole,dc=example,dc=com</DN>
    </DeleteRoles>

    <DeleteGroups deleteRecursively="true">
       <DN>cn=EmployeesGroup,dc=example,dc=com</DN>
       <DN>cn=ContractorsGroup,dc=example,dc=com</DN>
    </DeleteGroups>

    <DeletePeopleContainers deleteRecursively="true">
       <DN>ou=People1,dc=example,dc=com</DN>
    </DeletePeopleContainers>

    <DeleteSubOrganizations deleteRecursively="true">
       <DN>o=sun.com,dc=example,dc=com</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 6-14 illustrates the deletion of the Membership Authentication Service from `dc=example,dc=com`.

Code Example 6-14 orgDeleteServiceTemplateRequests.xml

```
<Requests>
<OrganizationRequests DN="dc=example,dc=com">
      <DeleteServiceTemplate serviceName="iPlanetAMAuthMembershipService"
schemaType="organization"/>

</OrganizationRequests>
</Requests>
```

Modify*Object* Elements

The *ModifySubOrganization*, *ModifyPeopleContainer*, *ModifySubContainer*, *ModifyRole*, and *ModifySubGroups* elements change the specified object. AttributeValuePair Elements can be defined for the listed elements. The required XML attribute is `modifyDN` which takes the `DN` of the object to be modified. Code Example 6-15 illustrates how the people container's description can be modified.

Code Example 6-15 contModifyPeoplecontainerRequests.xml

```
<Requests>
<ContainerRequests DN="dc=sun,dc=com">

    <ModifyPeopleContainer
modifyDN="ou=Test,ou=Test1,ou=People1,dc=sun,dc=com">
      <AttributeValuePair>
         <Attribute name="Description"/>
         <Value>Sun ONE Identity Server Modify</Value>
      </AttributeValuePair>
    </ModifyPeopleContainer>

</ContainerRequests>
</Requests>
```

ModifyServiceTemplate Element

The *ModifyServiceTemplate* element changes a specified service template. AttributeValuePair Element must be defined for *ModifyServiceTemplate* to change the values. The required XML attributes are `serviceName` which takes a string value, `schemaType` which defines the attribute group (Global, Organization, Dynamic, User or Policy) and `roleTemplate`. A search level attribute can also be defined. It takes a value of either `SCOPE_ONE` or `SCOPE_SUB`. `SCOPE_ONE` will retrieve just the groups at that node level; `SCOPE_SUB` gets groups at the node level and all those underneath it.

Get*Object* Elements

The *GetSubOrganizations, GetUsers, GetSubGroups, GetGroups, GetSubContainers, GetRoles, GetPeopleContainers* and *GetSubPeopleContainers* 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 will be returned within the organization that is defined in the *<Object>Requests* element under which the particular *Get<Object>* element is being defined. The required XML attribute for all but *GetGroups* and *GetRoles* is `DNsOnly` and takes a `true` or `false` value. (This attribute is explained in more detail in DNs Only Attribute.) The required XML attribute of *GetGroups* and *GetRoles* is `level` which takes a value of either `SCOPE_ONE` or `SCOPE_SUB`. `SCOPE_ONE` will retrieve just the groups at that node level; `SCOPE_SUB` gets groups at the node level and all those underneath it. Code Example 6-16 illustrates how these elements can be modeled. The top-level DN is defined in the *OrganizationRequests* element as `o=isp`.

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 6-16 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>
...
```

Get*Service* 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 6-17 illustrates the *GetNumberOfServices* element.

Code Example 6-17 orgGetNumberOfServiceRequests.xml

```
<Requests>

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

    <GetNumberOfServices/>

</OrganizationRequests>

</Requests>
```

*Action*ServiceTemplate Element

The *GetServiceTemplate* and *DeleteServiceTemplate* elements get or delete a service template for the organization defined under the *OrganizationRequests* element, respectively. There are no sub-elements; the elements themselves must be empty. The required XML attributes are `serviceName` which takes a string value and `schemaType`.

*Action*ServiceTemplateAttributeValues Element

The *AddServiceTemplateAttributeValues* and *RemoveServiceTemplateAttributeValues* elements get or delete attribute values defined in a service template for the organization defined under the *OrganizationRequests* element, respectively. AttributeValuePair Element must be defined for each attribute to be added or removed. The required XML attributes are `serviceName` which takes a string value, `roleTemplate` and `schemaType` which defines the attribute group (Global, Organization, Dynamic, User or Policy). A search level attribute can also be defined. It takes a value of either `SCOPE_ONE` or `SCOPE_SUB`. `SCOPE_ONE` will retrieve just the groups at that node level; `SCOPE_SUB` gets groups at the node level and all those underneath it.

*Action*Services 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 6-18 illustrates how the *RegisterServices* element is modeled.

Code Example 6-18 orgRegisterServiceRequests.xml

```
<Requests>
<OrganizationRequests DN="dc=sun,dc=com">

    <RegisterServices>
```

Code Example 6-18 orgRegisterServiceRequests.xml *(Continued)*

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

</OrganizationRequests>

</Requests>
```

SchemaRequests Element

The *SchemaRequests* element consists of all requests to be performed on the XML file that defines a particular service. It has two required XML attributes: *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). The "i18nFileName Attribute" on page 257 or a SubSchema (which specifies the complete hierarchy of the subschema separated by a "/") can also be defined.

NOTE See "Service File Naming Conventions" on page 245 for information on how the name is defined.

This element can have one or more sub-elements. (Different *SchemaRequests* elements can be defined in one document to modify more than one service.) The sub-elements of *SchemaRequests* can include:

- `RemoveDefaultValues`
- `RemovePartialDefaultValues`
- `AddDefaultValues`
- `ModifyDefaultValues`
- `GetServiceDefaultValues`
- `AddChoiceValues`
- `RemoveChoiceValues`
- `ModifyType`
- `ModifyUIType`
- `Modifyi18nKey`
- `ModifySyntax`

- `AddPropertiesViewBean`
- `AddStartRange`
- `AddEndRange`
- `AddSubSchema`
- `AddAttributeSchema`
- `RemoveSubSchema`
- `RemoveAttributeSchema`

Code Example 6-19 illustrates the opening of the *Requests* element tag and its corresponding *SchemaRequests* sub-element. The file is adding the choice *Deleted* to the Default User Status drop-down menu in the User Service.

Code Example 6-19 `schemaAddChoiceValuesRequests.xml`

```
...
<Requests>
<SchemaRequests serviceName="iPlanetAMUserService"
        SchemaType="dynamic"
        i18nKey="">
<AddChoiceValues>
     <AttributeValuePair>
          <Attribute name="iplanet-am-user-login-status"/>
                        <Value>Active</Value>
                        <Value>Inactive</Value>
                        <Value>Deleted</Value>

      </AttributeValuePair>

</AddChoiceValues>
</SchemaRequests>
</Requests>
```

RemoveDefaultValues Element

The *RemoveDefaultValues* element removes the default values from the service specified in the parent *SchemaRequests* element. It takes a sub-element of *Attribute* that specifies the service attribute which contains the values to be removed. The *Attribute* sub-element itself must be empty; it takes no sub-element. There is no required XML attribute. The syntax for this element is the same as that illustrated in Code Example 6-20.

Code Example 6-20 RemoveDefaultValues Element Code

```
...
<Requests>
<SchemaRequests serviceName="iPlanetAMUserService"
            SchemaType="dynamic">
<RemoveDefaultValues>
     <Attribute name="preferredlanguage"/>
</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 an AttributeValuePair Element which specifies the name of the attribute and the new default value; one or more attribute/value pairs can be defined. Code Example 6-21 illustrates how the *AddDefaultValues* element can be modeled.

Code Example 6-21 AddDefaultValues Element Code

```
...
<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.

Federation Management Elements

The following elements consist of requests that can be performed on Identity Server configured federations. They are:

- CreateAuthenticationDomain
- DeleteAuthenticationDomain
- GetAuthenticationDomain
- ModifyAuthenticationDomain
- CreateRemoteProvider
- CreateHostedProvider
- DeleteProvider
- GetProvider
- IDPAuthContextInfo
- SPAuthContextInfo
- AuthMethodQueryString
- ModifyRemoteProvider
- ModifyHostedProvider
- ListAccts

For more information on these elements, see the DTD file itself located in the *IdentityServer_base*/`SUNWam/dtd` directory.

XML Service Files

Identity Server uses XML files to define service attributes as well as perform batch processing operations. This section contains information on the XML files included with Identity Server and how they are used.

Default XML Service Files

Identity Server installs services to manage the configurations of its components. The attributes for these services are managed using the Identity Server console; 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 `etc/opt/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.
- `amAuthConfig.xml`—Defines configuration attributes for the Authentication service.
- `amAuthHTTPBasic.xml`—Defines attributes for the HTTP Basic Authentication service.
- `amAuthLDAP.xml`—Defines attributes for the LDAP Authentication service.
- `amAuthMembership.xml`—Defines attributes for the Membership-based Authentication service.
- `amAuthNT.xml`—Defines attributes for the Windows-based NT 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.
- `amAuthenticationDomainConfig.xml`—Defines attributes for the Authentication Configuration service.
- `amClientData.xml`—Defines client types for the Client Detection service.
- `amClientDetection.xml`—Defines attributes for the Client Detection service.

- amEntrySpecific.xml—Defines attributes for the displaying attributes on the Create, Properties and Search pages for a custom service.
- amDSS.xml—Defines attributes for the Certificate Security service.
- amG11NSettings.xml—Defines attributes for the Globalization Settings service.
- amLogging.xml—Defines attributes for the Logging service.
- amNaming.xml—Defines attributes for the Naming service.
- amPasswordReset.xml—Defines attributes for the Password Reset service.
- amPlatform.xml—Defines attributes for the Platform service.
- amPolicy.xml—Defines attributes for the Policy service.
- amPolicyConfig.xml—Defines configuration attributes for the Policy service.
- amProviderConfig.xml—Defines attributes for Federation Management service.
- amSAML.xml—Defines attributes for the SAML service.
- amSession.xml—Defines session attributes for single sign-on.
- amUser.xml—Defines attributes for the User service.
- amWebAgent.xml—Defines attributes for the policy 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 one. 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 defined 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 code illustrated in Code Example 6-22 to the SubSchema name=User element in *IdentityServer_base*/SUNWam/config/xml/amUser.xml.

Code Example 6-22 `nsaccountlock` Example Attribute

```
...
<AttributeSchema name="nsaccountlock"
type="single_choice"
syntax="string"
any="filter"
isChangeableByUser="yes"
i18nKey="u13">
<ChoiceValues>
        <Value>true</Value>
        <Value>false</Value>
</ChoiceValues>
<DefaultValues>
         <Value>false</Value>
</DefaultValues>
</AttributeSchema>
...
```

2. Update the *IdentityServer_base*/`SUNWam/locale/en_US/amUser.properties` file with the new `i18nKey` tag `u13` as illustrated in Code Example 6-23 (including the text to be used for display).

Code Example 6-23 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 modified XML service file, `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 "Defining A Custom Service" on page 243.

Batch Processing With XML Templates

The `--data` or `-t` option of `amadmin` is used to perform batch processing via 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 unregister 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 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 in the Directory Server.

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

XML Templates

All of the batch processing XML templates perform operations on the DIT; they create, delete, or get attribute information on user objects. These XML templates follow the structure defined by the `amAdmin.dtd` and are located in *IdentityServer_base*`/SUNWam/samples/admin/cli/bulk-ops`. The batch processing XML templates provided with Identity Server include:

- `contCreateRoleRequests.xml`—Creates a role for a container object.

- contCreateServiceTemplateRequests.xml—Creates a service template for a container object.
- contModifyPeoplecontainerRequests.xml—Modifies a people container object.
- contModifyRoleRequests.xml—Modifies a role assigned to a container object.
- contModifySubcontainerRequests.xml—Modifies a sub-container object.
- createRequests.xml—Creates a multitude of objects.
- deleteGroupRequests.xml—Deletes the sub-group of a group container.
- getRequests.xml—Passes information about a multitude of objects in a specific organization.
- orgCreateServiceTemplateRequests.xml—Creates service templates for anorganization.
- orgDeleteRequests.xml—Deletes a multitude of objects under a specific organization.
- orgDeleteServiceTemplateRequests.xml—Deletes a service template under a specific organization.
- orgGetNumberOfServiceRequests.xml—Passes a listing of an organization's total number of registered services.
- orgGetRegisteredServiceRequests.xml—Passes a listing the names of an organization's registered services.
- orgModifyRequests.xml—Changes values for identity-related objects in an organization.
- orgModifyServiceTemplateRequests.xml—Changes values for the registered service template of an organization.
- orgRegisterServiceRequests.xml—Registers services for an organization.
- orgUnRegisterServiceRequests.xml—Unregisters services for an organization.
- pcDeleteRequests.xml—Deletes attributes for a people container object.
- pcModifyUserRequests.xml—Modifies user attributes in a people container object.
- roleCreateServiceTemplateRequests.xml—Creates a service template for a role.

- roleModifyServiceTemplateRequests.xml—Changes values for the registered service template of a role.
- schemaAddChoiceValuesRequests.xml—Adds a selection of values to an existing service's attribute from which the user can choose.
- schemaAddDefaultValuesRequest.xml—Adds a default value to an existing service's attribute.
- schemaDeleteChoiceValueRequest.xml—Deletes a value from an existing service's attribute choices.
- schemaDeleteDefaultValueRequest.xml—Deletes a default value from an existing service's attribute.
- schemaGetServiceDefaultValueRequest.xml—Retrieves a default value from an existing service's attribute.
- schemaModifyDefaultValueRequest.xml—Changes the default value of an existing service's attribute.

NOTE The final XML templates (serviceConfigurationRequests.xml, serviceAddSubConfigurationRequests.xml, and serviceDeleteSubConfigurationRequests.xml) follow the sms.dtd format and are used for service sub-configurations. One use for these can be found in "Multi-LDAP Authentication Module Configuration" on page 126 of Chapter 3, "Authentication Service," in this manual.

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 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 i18nKey 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 288.

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.

Creating Users Using A Modified Directory Server Schema

There might be a need to modify the Directory Server LDAP schema in order to create users with new object classes. The procedure follows:

1. Modify the Directory Server LDAP schema with the new object classes and attributes.

   For more information on how to do this, 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 authentication module credentials file and put it in the *IdentityServer_base*`/SUNWam/lib` directory.

   This file contains the attribute-value pairs for the internationalization keys used in the file created in Step 2. More information can be found in "Configuring The Authentication Module" on page 144 of Chapter 3, "Authentication Service," in this manual.

NOTE Alternately, the path to the module configuration properties file can be put in the classpath of the web container's JVM.

4. Load the XML service file using the `amadmin` command line interface.

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

5. Register the new service to the desired organization using the Identity Server console.

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

6. 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.

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. The interfaces and methods can be found in `com.sun.identity.sm`.

ServiceSchemaManager Class

The `ServiceSchemaManager` class in the `com.sun.identity.sm` package provides interfaces to manage a service's schema. It must implement `ServiceSchema` which represents a single schema element in the service.

Retrieve Logging Location

Code Example 6-24 uses the `ServiceSchemaManager` class to retrieve the `iplanet-am-logging-location` attribute value from the Logging Service at the following DN: `ou=iPlanetAMLoggingService,ou=services,o=isp`.

Code Example 6-24 Retrieve Logging Location Sample

```
*********
SSOTokenManager manager = SSOTokenManager.getInstance();
SSOToken token = manager.createSSOToken(new
AuthPrincipal("uid=amadmin,ou=People,dc=org,dc=com"), "11111111");
ServiceSchemaManager ssm = new ServiceSchemaManager(token,
"iPlanetAMLoggingService", "1.0");
ServiceSchema ss = ssm.getGlobalSchema();
```

Code Example 6-24 Retrieve Logging Location Sample

```
Map p = ss.getAttributeDefaults();
************
```

Retrieve User Or Dynamic Attributes

Code Example 6-25 uses the `ServiceSchemaManager` to define the `ServiceSchema` user attributes. `AMUser.getAttributes(..)` is then called to obtain the attribute / value pairs.

Code Example 6-25 Retrieve User Or Dynamic Attributes

```
ServiceSchemaManager ssm = new ServiceSchemaManager(serviceName, token);
ServiceSchema sm = ssm.getSchema(SchemaType.USER);
if (sm != null) {
     Set userAttributes = ss.getAttributeSchemaNames();
     // Since USER or DYNAMIC attributes are stored as ldap attributes you
can call..
     amUser.getAttributes(userAttributes);
}
```

Retrieve Attribute Values

Code Example 6-26 illustrates one way to retrieve attribute values from a service.

Code Example 6-26 Sample Code To Retrieve Attribute Values

```
package com.iplanet.am.samples.sdk;

import java.io.*;
import java.net.*;
import java.util.*;
import com.iplanet.sso.*;
import com.iplanet.am.sdk.*;
import com.sun.identity.authentication.internal.*;
import com.sun.identity.sm.*;
import javax.servlet.*;
import javax.servlet.http.*;


public class SampleUserOperations {

    SSOToken token = null;
```

Code Example 6-26 Sample Code To Retrieve Attribute Values *(Continued)*

```
/**
* This user will be used for further sample operations on the
* same object
*/
    private static AMUser contextUser = null;
    private static String passWord = null;
    private static String uid = null;
    private static String lastName = null;
    private static String firstName = null;
    String userDN = null;

    private static Map scuObjMap = new HashMap();

    public static AMStoreConnection amsc = null;
    public static SampleUserOperations suo;

//Here we will try to get the value of the organization type
//attribute "iplanet-am-auth-ldap-bind-dn" of the service
//"iPlanetAMAuthLDAPService" for the organization
//DN "dc=iplanet,dc=com".
    public static void main(String args[]) {
        try {
            SSOTokenManager manager = SSOTokenManager.getInstance();
//If possible create the token using the tokneid or httprequest.
            SSOToken token = manager.createSSOToken(new
AuthPrincipal("uid=amadmin,ou=People,dc=iplanet,dc=com"), "11111111");
            suo = getSampleUserOperations(token);
            amsc = new AMStoreConnection(token);
            ServiceConfigManager scm = new ServiceConfigManager(token,
"iPlanetAMAuthLDAPService", "1.0");
            String orgName = "dc=iplanet,dc=com";
            ServiceConfig sc = scm.getOrganizationConfig(orgName, null);
            Map mp = sc.getAttributes();
            Iterator itr =
((HashSet)mp.get("iplanet-am-auth-ldap-bind-dn")).iterator();
            System.out.println("bind dn for the org -" + orgName + "-is-" +
(String)itr.next());
            System.exit(0);
        } catch (Exception e) {
            System.out.println("Exception Message: " + e.getMessage());
            e.printStackTrace();
        }
    }

/* Basic Constructor */

    public SampleUserOperations(SSOToken token) {
        this.token = token;
        scuObjMap.put(token, this);
    }

/* Use the same  object for multiple operations */
```

Code Example 6-26 Sample Code To Retrieve Attribute Values *(Continued)*

```
    public static SampleUserOperations getSampleUserOperations(SSOToken
token) {
        SampleUserOperations scuObj =
(SampleUserOperations)scuObjMap.get(token);
        if (scuObj == null ) {
            scuObj = new SampleUserOperations(token);
        }
        return scuObj;
    }

/**
* This method will describe the SDK usage for creating a user.
* It uses AMStoreConnection to get the organization object
* It uses the Set Parameters to store the different attributes of
* the user. This method is used for command line.
* It throws an AMException if unable to create it and we throw
* message "unable to create" to the GUI by catching the same
*/
    public String createUser(AMStoreConnection conn) {
        try {
            Map userAttributeMap = new HashMap();
                uid = "user";
                storeUserAttributes("uid", uid, userAttributeMap);
                firstName = "user";
                storeUserAttributes("givenname", firstName,
userAttributeMap);
                lastName = "one";
                storeUserAttributes("sn", lastName, userAttributeMap);
                passWord = "userone";
                storeUserAttributes("userPassword", passWord,
userAttributeMap);

            Map userMap1 = new HashMap();
            userMap1.put(uid, userAttributeMap);
/**
* Provide the DN according to the DIT
*/
            String dn = "ou=People,o=iplanet.com,o=isp";
            AMPeopleContainer ampc = conn.getPeopleContainer(dn);
            ampc.createUsers(userMap1);
            userDN = "uid=" + uid + "," + dn;
/*
* This is to keep the context of the user
*/
            contextUser = conn.getUser(userDN);
            return "Successfully added the user: " + uid;
        } catch (Exception ex) {
            ex.printStackTrace();
        }
          return "Unable to create";
    }

/**
* This method will describe the SDK usage for creating a user.
```

Code Example 6-26 Sample Code To Retrieve Attribute Values *(Continued)*

```
* It uses AMStoreConnection to get the organization object
* It uses the Set Parameters to store the different attributes of
* the user.
* It throws an AMException if unable to create it and we throw
* message "unable to create" to the GUI by catching the same
*/

    public String createUser(HttpServletRequest req, Set parameters,
AMStoreConnection
conn) {
        try {
            Map userAttributeMap = new HashMap();
            if (parameters.contains("uid")) {
                uid = req.getParameter("uid");
                storeUserAttributes("uid", uid, userAttributeMap);
            }
            if(parameters.contains("firstname")) {
                firstName = req.getParameter("firstname");
                storeUserAttributes("givenname", firstName,
userAttributeMap);
            }
            if(parameters.contains("lastname")) {
                lastName = req.getParameter("lastname");
                storeUserAttributes("sn", lastName, userAttributeMap);
            }
            if(parameters.contains("password")) {
                passWord = req.getParameter("userPassword");
                storeUserAttributes("userPassword", passWord,
userAttributeMap);
            }

            Map userMap1 = new HashMap();
            userMap1.put(uid, userAttributeMap);
            String orgDN = req.getParameter("orgName");
            String dn = "ou=People" + "," + orgDN;
            AMPeopleContainer ampc = conn.getPeopleContainer(dn);
            ampc.createUsers(userMap1);
            userDN = "uid=" + uid + "," + dn;
            /*
             * This is to keep the context of the user
             */
            contextUser = conn.getUser(userDN);
            return showCreateUserSuccess();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
          return "Unable to create";
    }

/**
* This method describes the SDK usage for modifying the user.
*/
    public String modifyUser(HttpServletRequest req) {
        HashMap modifyMap = new HashMap();
```

Code Example 6-26 Sample Code To Retrieve Attribute Values *(Continued)*

```
        lastName = req.getParameter("lastname");
        storeUserAttributes("sn", lastName, modifyMap);
        firstName = req.getParameter("firstname");
        storeUserAttributes("givenname", firstName, modifyMap);
        passWord = req.getParameter("userpassword");
        storeUserAttributes("userPassword", passWord, modifyMap);

        try {
            contextUser.setAttributes(modifyMap);
            contextUser.store();
            return showModifyUserSuccess();
        } catch (Exception ex) {
            System.out.println("Exception occured");
        }
        return "Unable to modify";
    }

/**
* This method describes the SDK usage for deleting the user.
*/

    public String deleteUser() {
        try {
            contextUser.delete(false);
            return "Deleted successfully";
        } catch (Exception ex) {
            System.out.println("Exception occured");
        }
        return "Unable to delete";
    }

/* This method is for the GUI purposes */

    public String showCreateUser() {
        StringBuffer sb = new StringBuffer();
        sb.append("<HTML>");
        sb.append("<HEAD>");
        sb.append("</HEAD>");
        sb.append("<BODY>");
        sb.append("<FORM name=\"allattributes\" METHOD=POST
ACTION=\"/amserver/sdksample\">");
        sb.append("<TABLE>");
        sb.append("<TR>");
        sb.append("<TD ALIGN=LEFT VALIGN=MIDDLE><B>Login ID</B></TD>");
        sb.append("<TD VALIGN=MIDDLE><INPUT TYPE=\"text\" NAME=\"uid\"
VALUE=\"\"
SIZE=32 MAXLENGTH=64></TD><TD><B>Under Organization</B></TD>");
        sb.append("<TD VALIGN=MIDDLE><INPUT TYPE=\"text\" NAME=\"orgName\"
VALUE=\"\"
SIZE=32 MAXLENGTH=64></TD>");
        sb.append("</TR>");
        sb.append("<TR>");
        sb.append("<TD ALIGN=LEFT VALIGN=MIDDLE><B>First Name</B></TD>");
        sb.append("<TD VALIGN=MIDDLE><INPUT TYPE=\"text\" NAME=\"firstname\"
```

Code Example 6-26 Sample Code To Retrieve Attribute Values *(Continued)*

```
VALUE=\"\" SIZE=32 MAXLENGTH=64></TD>");
        sb.append("</TR>");
        sb.append("<TR>");
        sb.append("<TD ALIGN=LEFT VALIGN=MIDDLE><B>Last Name</B></TD>");
        sb.append("<TD VALIGN=MIDDLE><INPUT TYPE=\"text\" NAME=\"lastname\"
VALUE=\"\"
SIZE=32 MAXLENGTH=64></TD>");
        sb.append("</TR>");
        sb.append("<TR>");
        sb.append("<TD ALIGN=LEFT VALIGN=MIDDLE><B>Password</B></TD>");
        sb.append("<TD VALIGN=MIDDLE><INPUT TYPE=\"password\"
NAME=\"userpassword\"
VALUE=\"\" SIZE=12></TD>");
        sb.append("</TR>");
        sb.append("<TR>");
        sb.append("<TD ALIGN=LEFT VALIGN=MIDDLE><B>Confirm
Password</B></TD>");
        sb.append("<TD VALIGN=MIDDLE><INPUT TYPE=\"password\"
NAME=\"passwordagain\"
VALUE=\"\" SIZE=12></TD>");
        sb.append("</TR>");
        sb.append("<TR>");
        sb.append("<TD><input type=SUBMIT NAME=\"usersubmit\">");
        sb.append("</TD></TR>");
        sb.append("</TABLE>");
        sb.append("</FORM>");
        sb.append("</BODY>");
        sb.append("</HTML>");
        return sb.toString();
    }

    private void storeUserAttributes(String attribute, String value, Map
userMap) {
        Set userSet = new HashSet();
        userSet.add(value);
        userMap.put(attribute, userSet);
    }

/* This method is for the GUI purposes */

    private String showCreateUserSuccess() {
        StringBuffer sb = new StringBuffer();
        sb.append("<HTML>");
        sb.append("<HEAD>");
        sb.append("</HEAD>");
        sb.append("<BODY>");
        sb.append("Created Successfully");
        sb.append("<FORM name=\"usersuccessful\" METHOD=POST
ACTION=\"/amserver/sdksample\">");
        sb.append("<TABLE>");
        sb.append("<TR>");
        sb.append("<TD><input type=SUBMIT NAME=\"modifyuser\"
VALUE=\"Modify\">");
        sb.append("</TD></TR>");
```

Code Example 6-26 Sample Code To Retrieve Attribute Values *(Continued)*

```
        sb.append("</TABLE>");
        sb.append("</FORM>");
        sb.append("</BODY>");
        sb.append("</HTML>");
        return sb.toString();
    }

    /* This method is for the GUI purposes */

    public String showModifyUser() {
        StringBuffer sb = new StringBuffer();
        sb.append("<HTML>");
        sb.append("<HEAD>");
        sb.append("</HEAD>");
        sb.append("<BODY>");
        sb.append("uid:" + uid);
        sb.append("<FORM name=\"showmodify\" METHOD=POST
ACTION=\"/amserver/sdksample\">");
        sb.append("<TABLE>");
        sb.append("<TR>");
        sb.append("<TD ALIGN=LEFT VALIGN=MIDDLE><B>First Name</B></TD>");
        sb.append("<TD VALIGN=MIDDLE><INPUT TYPE=\"text\" NAME=\"firstname\"
VALUE=\"");
        sb.append(firstName + "\" SIZE=32 MAXLENGTH=64></TD>");
        sb.append("</TR>");
        sb.append("<TR>");
        sb.append("<TD ALIGN=LEFT VALIGN=MIDDLE><B>Last Name</B></TD>");
        sb.append("<TD VALIGN=MIDDLE><INPUT TYPE=\"text\" NAME=\"lastname\"
VALUE=\"");
        sb.append(lastName + "\" SIZE=32 MAXLENGTH=64></TD>");
        sb.append("</TR>");
        sb.append("<TR>");
        sb.append("<TD ALIGN=LEFT VALIGN=MIDDLE><B>Password</B></TD>");
        sb.append("<TD VALIGN=MIDDLE><INPUT TYPE=\"password\"
NAME=\"userpassword\"
VALUE=\"");
        sb.append(passWord + "\" SIZE=12></TD>");
        sb.append("</TR>");
        sb.append("<TR>");
        sb.append("<TD><input type=SUBMIT NAME=\"modifyusersubmit\">");
        sb.append("</TD></TR>");
        sb.append("</TABLE>");
        sb.append("</FORM>");
        sb.append("</BODY>");
        sb.append("</HTML>");
        return sb.toString();
    }

/* This method is for the GUI purposes */

    private String showModifyUserSuccess() {
        StringBuffer sb = new StringBuffer();
        sb.append("<HTML>");
        sb.append("<HEAD>");
        sb.append("</HEAD>");
```

Code Example 6-26 Sample Code To Retrieve Attribute Values *(Continued)*

```
        sb.append("<BODY>");
        sb.append("Modified Successfully");
        sb.append("<FORM name=\"modifyusersuccessful\" METHOD=POST
ACTION=\"/amserver/sdksample\">");
        sb.append("<TABLE>");
        sb.append("<TR>");
        sb.append("<TD><input type=SUBMIT NAME=\"deleteusersubmit\"
VALUE=\"Delete\">");
        sb.append("</TD></TR>");
        sb.append("</TABLE>");
        sb.append("</FORM>");
        sb.append("</BODY>");
        sb.append("</HTML>");
        return sb.toString();
    }

/* This method is for the GUI purposes */

    public String showDeleteUser() {
        StringBuffer sb = new StringBuffer();
        sb.append("<HTML>");
        sb.append("<HEAD>");
        sb.append("</HEAD>");
        sb.append("<BODY>");
        sb.append("<FORM name=\"showdelete\" METHOD=POST
ACTION=\"/amserver/sdksample\">");
        sb.append("<TABLE>");
        sb.append("<TR>");
        sb.append("<TD><input type=SUBMIT NAME=\"deleteusersubmit\">");
        sb.append("</TD></TR>");
        sb.append("</TABLE>");
        sb.append("</FORM>");
        sb.append("</BODY>");
        sb.append("</HTML>");
        return sb.toString();
    }

}
```

Chapter 7

Policy Service

Sun™ ONE Identity Server includes a Policy Service that allows for the defintion, management and enforcement of policies that control access to protected resources. It allows administrators to configure and administer these conditions 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:

- Overview
- Policy Agent Process
- Policy Types
- Policy-Based Resource Management
- Policy Definition Type Document
- Policy SDK
- Policy Samples

Overview

A *policy* defines rules that specify authorization privileges to an organization's protected resources. To elaborate, every business has resources, applications and services to protect, manage and monitor. Policies control the usage of, and access to, these resources by defining who can do what to which resource, when and how. A policy, when applied to an object, defines the resources said object can access.

NOTE	Programmatically, an object is a principal. A *principal* can be an individual, a corporation, a role, or a group; anything that can have an identity. for more information, see the Java™ 2 Platform Standard Edition Javadocs.

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 Mark execute the `changeSalary` method in the `PayCheck` class?” or “Can 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

Identity Server provides a Policy Service for the creation and administration of policies. It allows administrators to define, modify, grant, revoke and delete permissions to protect resources within the Identity Server deployment. Typically, a policy service includes a data store, a library of interfaces that allows for the creation, administration and evaluation of policies, and a policy enforcer or *policy agent*. Identity Server uses Sun ONE Directory Server for data storage, and provides Java and C APIs for policy evaluation and the Identity Server console for policy administration. A selection of downloadable policy agents are used to enforce the policies.

Policy Configuration Service

A Policy Configuration Service is provided to allow the configuration of policy-related attributes for each organization via the Identity Server console. Resource name implementations and Directory Server data stores for use with the Authentication Service can be defined in this service. For additional information, see the Policy Configuration Service Attributes chapter of the *Sun ONE Identity Server Administration Guide*.

Caching Subject Evaluations

To improve policy evaluation performance, subject evaluations are cached for a period of minutes as defined by the Subjects Result Time To Live attribute in the Policy Configuration service. These cached policy decisions are referred to until the time defined in the Subjects Result Time To Live attribute has elapsed. Once this time has been reached, the next time the policy is evaluated its decision would reflect the user’s changed state, if applicable. (For example, if the user has logged out.)

amldapuser Definition

`amldapuser` is a user created during installation that is used to bind to, and search, Directory Server during LDAP and Membership authentication. It is also used in the Policy Configuration service. Once the LDAP, Membership or Policy Configuration services are registered to an organization, the password for this user (configured during installation) must be entered. For more information, see the *Sun ONE Identity Server Migration Guide* .

Policy Agents

The Policy Agent is the Policy Enforcement Point (PEP) for a server on which an enterprise's resources are stored. Policy agents are provided under separate cover from Identity Server. The most current Sun ONE Identity Server Policy Agents can be downloaded from the the Sun Microsystems Download Center.

In one scenario, a Human Resources web server protected by a remotely-installed Identity Server might have an agent installed on it. This agent would prevent personnel without the proper policy from viewing confidential salary information and other sensitive data. The policies are defined by the Identity Server administrator, stored within the Identity Server deployment and used by the policy agent to allow or deny users access to the remote web server's content. More information on installing and administrating the policy agents can be found in the *Sun ONE Identity Server J2EE Policy Agents Guide* or *Web Policy Agents Guide*.

NOTE Policy is evaluated in no particular order although as they are evaluated, if one action value evaluates to *deny*, subsequent policies are not evaluated. *Deny* decisions override *allow* decisions.

Policy agents enforce decisions only on web URLs (`http://...`). However, agents can be written using the Java and C Policy Evaluation APIs to enforce policy on other resources.

NOTE A portion of the "Policy Samples" on page 367 illustrates how to use the Policy Evaluation class.

In addition, the Resource Comparator attribute in the Policy Configuration Service would also need to be changed from its default configuration to:

```
serviceType=Name_of_LDAPService|class=com.sun.identity.policy.plugin
s.SuffixResourceName|wildcard=*|delimiter=,|caseSensitive=false
```

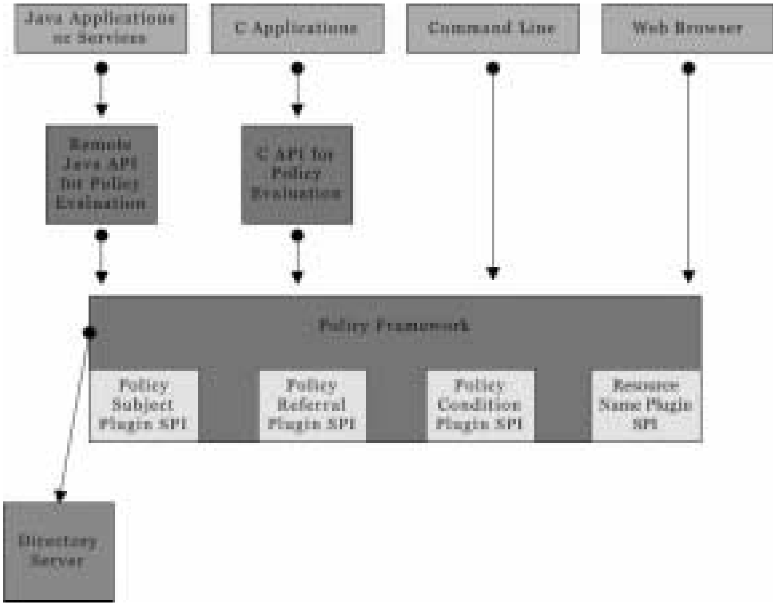
Alternately, providing an implementation such as `LDAPResourceName` to implement `com.sun.identity.policy.interfaces.ResourceName` and configuring the Resource Comparator appropriately would also work.

NOTE The fields for the Resource Comparator attribute are explained in detail in “Chapter 34, Policy Configuration Attributes” of the *Sun ONE Identity Server Administration Guide*.

Architecture

The Policy Service allows for the protection of all types of applications and resources. 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 Policy Service Architecture


Java Applications or Services
C Applications
Command Line
Web Browser
Remote Java API for Policy Evaluation
C API for Policy Evaluation
Policy Framework
Policy Subject Plugin SPI
Policy Referral Plugin SPI
Policy Condition Plugin SPI
Resource Name Plugin SPI
Directory Server

Policy Agent Process

The process for protected web resources begins when a web browser requests a URL that resides on a server protected by the policy agent. The server's installed policy agent intercepts the request and checks for existing authentication credentials (a session token).

NOTE This complete authentication process is detailed in Chapter 3, "Authentication Service," of this manual.

Assuming the agent has intercepted a request and validated the existing session token, the following process is followed.

1. Although the session token is valid, the user's existing authentication level might still be insufficient. If so, the Identity Server Authentication Service will present the appropriate login page, prompting the user for additional credentials.
2. The Authentication Service verifies that the new credentials are also valid. For example, if the resource is protected by the LDAP authentication module, the username and password are verified with information stored in Directory Server. In cases where other authentication modules are being used, credentials are verified by the appropriate data store.
3. Once the user's credentials are properly authenticated, the agent issues a request to the Naming Service which defines the URLs used to access Identity Server's internal services.
4. The Naming Service returns locators for the Policy Service, and the agent sends a request to the Policy Service to get policy decisions applicable to the user.
5. Based on the policy decisions for the resource being accessed, the user is either allowed or denied access.

Assuming the agent has intercepted a request for which there is no existing session token, the agent redirects the user to their default login page even if the resource is protected using a different authentication method. The user must pass both their default authentication method and the configured resource authentication method before access can be granted.

NOTE Policy-based resource authentication and user authentication are separate types of authentication. More information on this can be found in "Policy-Based Resource Management" on page 311.

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 introduced in this section. "Policy Definition Type Document" on page 312 expands on these definitions.

Normal Policy

A policy that defines access permissions is referred to as a *normal* policy. A normal policy consists of Rules, Subjects and Conditions.

Rules

A *rule* contains a resource, one or more sets of an action, and a value. It defines the policy.

- A *resource* defines the specific object that is being protected; for instance, an HTML page or a user's salary information accessed using a human resources service.
- An *action* is the name of an operation that can be performed on the resource; examples of web server actions are POST or GET. An allowable action for a human resources service might be `canChangeHomeTelephone`.
- A *value* defines the permission for the action, i.e: allow or deny.

NOTE It is acceptable to define an action without resources.

Subjects

A *subject* defines the user or collection of users (for instance, a group or those who possess a specific role) that the policy affects. Subjects are assigned to policies. The general rule for subjects is that the policy would apply only if the user is a member of at least one subject in the policy. The default subjects are:

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

- Organization

Identity Server Roles Versus LDAP Roles

An Identity Server role is created using Identity Server. These roles have object classes mandated by Identity Server. An LDAP Role is any role definition that uses the Directory Server role capability. These roles have object classes mandated by Directory Server role definition. All Identity Server roles can be used as Directory Server roles. All Directory Server roles are not necessarily Identity Server roles. LDAP roles can be leveraged from an existing directory by configuring the Policy Configuration Service. Identity Server roles can only be accessed via the hosting Identity Server Policy Service. Evaluating membership in Identity Server roles will be faster as it accesses the Identity Server SDK and cache. The LDAP Role Search filter can be modified in the Policy Configuration Service to narrow the scope and improve performance.

Nested Roles

Nested roles can be evaluated correctly as LDAP Roles in the subject of a policy definition. However, the LDAP entry representing the nested role must be created using the Directory Server console. Identity Server console or SDK does not support creating nested roles.

Conditions

A *condition* defines the situations in which a policy is applicable; for instance, a 7 am to 10 am condition defined in a policy restrains the subject(s) to access between 7 am to 10 am. Multiple conditions can be configured. The rule for conditions is that the policy would apply to the subject only if at least one condition for each condition type defined in the policy is satisfied.

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` and are explained further in “Policy Definition Type Document” on page 312.

Referral Policy

A policy used for delegation is a *referral* policy. Envision a top-level organization with a sub-organization; in order for the sub-organization administrator to create normal policies, a referral policy must be configured at the top-level organization pointing to the sub-organization. This referral allows the sub-organization to create normal policies that will only affect those in the sub-organization. A referral policy controls delegation for both policy creation and policy 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.
- A *referral* defines the identity object to which the policy creation or evaluation is being referred.

NOTE In most cases, a referral is an identity object although it can be any object that implements the Referral interface.

For example, consider a deployment whose root level organization is `dc=example,dc=com with` sub-organizations `dc=sunone,dc=example,dc=com` and `dc=suntwo,dc=example,dc=com`. In order to define or evaluate policies at `dc=sunone,dc=example,dc=com` or `dc=suntwo,dc=example,dc=com`, two referral policies must be created that point from `dc=example,dc=com` to `dc=sunone,dc=example,dc=com` and `dc=suntwo,dc=example,dc=com`, respectively. Each referral policy contains only the resource (or resource prefix) being managed. In this example, if `dc=sunone,dc=example,dc=com` manages `http://www.sunone.com/`, the referral policy at `dc=example,dc=com` contains `http://www.sunone.com/` in its rule and refers policy creation to the `dc=sunone,dc=example,dc=com` sub-organization. (Policies for any resource besides `http://www.sunone.com/` can not be created or evaluated until the resource itself is defined in a referral policy.) Only after creating root level referral policies can policies at the sub-organization level be created. There are two types of referral policies: *peer organization* and *sub-organization*. They delegate to an organization on the same tree level and an organization on a sub-level, respectively.

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 does not apply to the root organization. Therefore, overall management policies must be defined at the root level organization.

Policy-Based Resource Management

Some organizations require an advanced authentication scenario where a user authenticates against a particular module based on the resource that they are attempting to access. Policy-based resource management is a function of Identity Server in which the user does not need to pass their default authentication module in order to access a web resource.

Limitations

Policy-based resource management contains the following limitations:

1. All policies applicable to the resource require the same authentication scheme or level of authentication. For example, if `abc.html` is defined in a policy for the LDAP authentication module, it can not be defined in a policy for the Certificate-based authentication module.
2. Level and scheme are the only conditions that can be defined for this policy.
3. This feature does not work across different DNS domains.

To Configure Policy-based Resource Management

Once both Identity Server and a policy agent have been installed, policy-based resource manangement can be configured. To do this, it is necessary to point Identity Server to the Gateway servlet.

1. Open `AMAgent.properties`.

   `AMAgent.properties` can be found (in a Solaris environment) in `/export/SUNWam/agents/config/`.

2. Comment out the following line:

   ```
   #com.sun.am.policy.am.loginURL =
   http://identity_server_host.domain_name:port/amserver/UI/Login
   ```
   .

3. Add the following line to the file:

   ```
   com.sun.am.policy.am.loginURL =
   http://identity_server_host.domain_name:port/amserver/gateway
   ```

4. Restart the agent.

Policy Definition Type Document

Configured policy is stored in Directory Server in XML. In Directory Server, the XML-encoded data is not broken up but stored in one place. Although policy is defined and configured using the `amadmin.dtd` (or the console), it is actually stored in Directory Server as XML that is based on the `policy.dtd`. The `policy.dtd` contains the policy element tags extracted from the `amadmin.dtd` (without the policy creation tags). So, when the Policy Service loads policies from Directory Server, it parses the XML based on the `policy.dtd`. The `amadmin.dtd` is only used when creating policy with the command line. This section describes the structure of policy.dtd. The `policy.dtd` itself can be found in *IdentityServer_base*`/SUNWam/dtd`.

NOTE Information on how to use the console and the `amadmin` interface can be found in the *Sun ONE Identity Server Administration Guide*.

Policy Element

Policy is the root element that defines the permissions or *rules* of a policy and to whom / what the rule applies or the *subject*. It also defines whether or not the policy is a *referral* (delegated) policy and whether there are any restrictions (or *conditions*) to the policy. It may contain one or more of the following sub-elements: *Rule, Conditions*, *Subjects*, or *Referrals*. The required XML attribute is `name` which specifies the name of the policy. The `referralPolicy` attribute identifies whether or not the policy is a referral policy; it defaults to a normal policy if not defined. Optional XML attributes include *name* and *description*.

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 and can take three sub-elements: *ServiceName*, *ResourceName*, or *AttributeValuePair*. It defines the type of service or application for which the policy has been created as well as the resource name and the actions which are performed on it. A rule can be defined without any actions; for example, a referral policy rule doesn't have any actions.

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

ServiceName Element

The *ServiceName* element defines the name of the service to which the policy applies. This element represents the service type. It contains no other elements. The value is exactly as that defined in the service's XML file (based on the `sms.dtd`). The XML service attribute for the *ServiceName* element is the name of the service (which takes a string value).

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.

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. 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 values. Allow / deny or yes / no are examples of action values. Other action values can be either boolean, numeric, or strings. The values are 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.

CAUTION Deny rules always take precedence over allow rules. For example, if one policy denies access and another allows it, the result is a deny (provided all other conditions for both policies are met). It is recommended that deny policies be used with extreme caution as they can lead to potential conflicts. If explicit deny rules are used, policies assigned to a user through different subjects (such as role and/or group membership) may result in denied access. Typically, the policy definition process should only use allow rules. The default deny may be used when no other policies apply.

Subjects Element

The *Subjects* sub-element identifies a collection of objects to which the policy applies; this overview collection is chosen based on membership in a group, ownership of a role or individual users. It takes the *Subject* sub-element. The XML attributes it can be defined with are `name` which defines a name for the collection, `description` which takes a description and `includeType` which is not currently used.

Subject Element

The *Subject* sub-element identifies a collection of objects to which the policy applies; this collection pinpoints more specific objects from the collection defined by the Subjects element. Membership can be based on roles, group membership or simply a listing of individual users. It takes as a sub-element the AttributeValuePair Element. It's required XML attribute is `type` which identifies a generic collection of objects from which the specifically defined subjects are taken. Other XML attributes include `name` which defines a name for the collection and `includeType` which defines whether the collection is as defined or its inverse (i.e.: the policy applies to users who are NOT members of the subject).

NOTE When multiple subjects are defined, at least one of the subjects should apply to the user for the policy to apply. When a subject is defined using `includeType` set to false, the user should not be a member of that subject or a member of one of the `includeType` subjects for the policy to apply to that user.

Referrals Element

The *Referrals* sub-element identifies a collection of policy referrals. It takes the *Referral* sub-element. The XML attributes it can be defined with are `name` which defines a name for the collection and `description` which takes a description.

Referral Element

The *Referral* sub-element identifies a specific policy referral. It takes as a sub-element the AttributeValuePair Element. It's required XML attribute is `type` which identifies a generic collection of assignments from which the specifically defined referrals are taken. It can also include the `name` attribute which defines a name for the collection.

Conditions Element

The *Conditions* sub-element identifies a collection of policy restrictions (time range, authentication level, et.al.). It must contain one or more of the *Condition* sub-element. The XML attributes it can be defined with are `name` which defines a name for the collection and `description` which takes a description.

NOTE The conditions element is an optional element in a policy.

Condition Element

The *Condition* sub-element identifies a specific policy restriction (time range, authentication level, et.al.). It takes as a sub-element the AttributeValuePair Element. Its required XML attribute is `type` which identifies a generic collection of restrictions from which the specifically defined conditions are taken. It can also include the `name` attribute which defines a name for the collection.

NOTE The Condition element might be used to configure policy for different URIs on the same domain. For example, `http://org.example.com/hr` can only be accessed by `org.example.net` from 9 am to 5 pm yet `http://org.example.com/finance` can be accessed by `org.example2.net` from 5 am to 11 pm. By defining an IP Condition attribute/value pair together with a SimpleTime Condition attribute/value pair and specifying `http://org.example.com/hr/*.jsp` as the resource, the policy would apply to all the JSPs under `http://org.example.com/hr`.

Policy SDK

The Policy Service provides Java and C API to allow external applications to participate in its functionality. With the SDK, applications can determine privileges and manage policies.

Java SDK For Policy

The crux of the Policy Service is the Java SDK. It defines the following packages:

- `com.sun.identity.policy` provides the APIs for administering (creating, deleting, modifying) and evaluating policies. It is used by the Identity Server console and/or the command line interface.
- `com.sun.identity.policy.interfaces` provides source interfaces used to implement custom subjects, conditions, referrals and resource comparators.
- `com.sun.identity.policy.client` are APIs used by remote Java applications that need to evaluate policies and get policy decisions.

TIP `AMConfig.properties` must be copied from Identity Server to a client machine as well as the respective jars to run test code in a remote environment. Some properties (like the notification url for remote client) need to be modified for their functionality to work.

Policy API For Java

The `com.sun.identity.policy` package provides the classes and methods to manage, administer and evaluate policies. They can be used by the Identity Server console or the `amadmin` command line interface tool. Select classes and methods are discussed in this section.

Policy Evaluation Classes

The following information introduces some of the classes that can be used to evaulate configured policies for access to a protected resource.

PolicyEvaluator Class `com.sun.identity.policy.PolicyEvaluator` can be integrated into Java applications to evaluate policy privileges and provide policy decisions. This class provides support for both boolean and non-boolean type policies. A `PolicyEvaluator` is 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.

NOTE A boolean false value overrides a boolean true value. Once an action is determined to have a false value, other values are not evaluated.

-
    - Arguments:
        - `com.iplanet.sso.SSOToken`: The SSOToken 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 session token is not valid or has expired.
        - Throws `com.sun.identity.policy.PolicyException` if the result could not be computed for any reason other than a token problem.
- `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 session token is not valid or expired.
        - Throws `com.sun.identity.policy.PolicyException` if the result could not be computed for any reason other than a token problem.
- `getResourceResult`—obtains the policy and ascertains privileges for non-boolean decisions. Possible values for the scope of this method are `self` and `subtree`. `self` gets the policy decision for the specified resource only. `subtree` includes the policy decisions for all resources (defined in the policies) which are sub-resources of the specified resource.

To illustrate, the `PolicyEvaluator` class can be used to display the links for a list of resources to which an authenticated user has access. The `getResourceResult` method would be used to get the list of resources. The `resourceName` parameter would be `http://host.domain:port` which would return all the resources to which the user has access on that server. These resources are returned as a `PolicyDecision` based on the user's defined policies. If the user is allowed to access resources on different servers, this method needs to be called for each server.

NOTE Not all resources that have policy decisions are accessible to the user. The `ActionDecision`(s) contained in policy decisions carry this information.

ProxyPolicyEvaluator Class

`com.sun.identity.policy.ProxyPolicyEvaluator` allows a privileged user (top level administrator, organization administrator, policy administrator, or organization policy administrator) to get policy privileges and evaluate policy decisions for any user in their respective scope of administration. `com.sun.identity.policy.ProxyPolicyEvaluatorFactory` is the singleton class used to get `ProxyPolicyEvaluator` instances.

Code Example 7-1 Public Methods For ProxyPolicyEvaluator

```
/**
 * Evaluates a simple privilege of boolean type. The privilege
 * indicates if the user identified by the principalName
 * can perform specified action on the specified resource.
 *
 * @param principalName principal name for whom to
 * compute the privilege.
 * @param resourceName name of the resource
 * for which to compute policy result.
 * @param actionName name of the action the user is trying to
 * perform on the resource
 * @param env run time environment parameters
 *
 * @return the result of the evaluation as a boolean value
 *
 * @throws PolicyException exception form policy framework
 * @throws SSOException if sso token is invalid
 *
 */
public boolean isAllowed(String principalName, String resourceName,
    String actionName, Map env) throws PolicyException, SSOException;

/**
 * Gets policy decision for the user identified by the
 * principalName for the given resource
 *
 * @param principalName principal name for whom to compute the
 * policy decision
 * @param resourceName name of the resource for which to
 * compute policy decision
 * @param env run time environment parameters
 *
 * @return the policy decision for the principal for the given
 * resource
 * @throws PolicyException exception form policy framework
 * @throws SSOException if sso token is invalid
 *
 */
public PolicyDecision getPolicyDecision(String principalName,
    String resourceName, Map env)
    throws PolicyException, SSOException;

/**
 * Gets protected resources for a user identified by the
 * principalName. Conditions defined in the policies
 * are ignored while computing protected resources.
 * Only resources that are  subresources of the  given
 * rootResource or equal to the  given rootResource would
 * be returned.
 * If all policies applicable to a resource are
 * only referral policies, no ProtectedResource would be
 * returned for such a resource.
 * @param principalName principal name for whom
 * to compute the privilege.
```

Code Example 7-1 Public Methods For ProxyPolicyEvaluator *(Continued)*

```
     * @param rootResource  only resources that are subresources
     * of the given rootResource or equal to the  given
     * rootResource would be returned. If
     * <code>PolicyEvaluator.ALL_RESOURCES</code>
     * is passed as rootResource, resources under
     * all root  resources of the service
     * type are considered while computing protected
     * resources.
     *
     * @return set of protected resources. The set contains
     * ProtectedResource objects.
     *
     * @throws PolicyException exception form policy framework
     * @throws SSOException if sso token is invalid
     * @see ProtectedResource
     *
     */
    public Set getProtectedResourcesIgnoreConditions(String principalName,
        String rootResource)  throws PolicyException, SSOException
```

PolicyEvaluator Class `com.sun.identity.policy.client.PolicyEvaluator` evaluates policies and provides policy decisions for remote applications which do not have a direct access to Directory Server (for example, if there is a firewall). The `com.sun.identity.policy.client.PolicyEvaluator` defined in "PolicyEvaluator Class" on page 316 requires direct LDAP access to policies stored in Directory Server. This class `com.sun.identity.policy.client.PolicyEvaluator` is implemented using XML over HTTP(s). It stores a cache of policy decisions for faster responses and maintains the cache in sync with the Policy Service on the instance of Identity Server using the notification and polling mechanism.

NOTE The PolicyEvaluator class can be used in a deployment container running Identity Server, or in a stand alone Java Virtual Machine (JVM) running the Identity Server SDK. Respective to the JVM, a property must be defined to point to `serverconfig.xml` which, in turn, points to Directory Server. This is done by launching the JVM with the following argument:

```
-D
"com.iplanet.coreservices.configpath=/etc/opt/SUNWam/
config/ums"
```

Policy Management Classes

The following classes can be used by system administrators to manage policies in Identity Server. The interfaces for this functionality are also 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 by passing a privileged user's session token or by passing a privileged user's session token with an organization name. Some of this class's 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 and 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, a policy event would be created and passed to the registered policy listeners whenever there is a change in a policy rule. This class works with the `PolicyListener` class in the `com.sun.identity.policy.interface` package.

Policy Plugin API For Java

The following classes 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

`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

`Subject` defines methods that can determine if an authenticated user (possessing an SSOToken) is a member of the given subject.

Referral

`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

`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

`PolicyListener` defines an interface to register for policy events when a policy is added, removed or changed. It is used by the Policy Service to send notifications and by listeners to review policy change events.

C Library For Policy

Identity Server also provides a library of policy evaluation APIs to enable integration of the policy functionality into for C applications. The C library provides a comprehensive set of interfaces that query policy results of 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. 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.

CAUTION Previous releases of Identity Server contained C libraries in *IdentityServer_base*/lib/capi. The capi directory is being deprecated, and is curently available for backward compatability. It will be removed in the next release, and therefore it is highly recommended that existing application paths to this directory are changed and new applications do not access it. Paths include RPATH, LD_LIBRARY_PATH, PATH, compiler options, etc.)

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://*identity_server_host.domain_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://*identity_server_host.domain_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.

Policy Evaluation API for C

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:

`map_ptr`	After successful execution of this function, this pointer will be assigned a new instance of `am_map_t` structure and all the entries in `source_map` 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:

`map_ptr`	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:

map	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:

map	The map to which the key-value pair must be added.
key	The key for the entry.
value	The value for the entry.
replace	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:

`map`	The map to which the key-value pair must be added.
`key`	The key for the entry.
`value_iter_ptr`	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:

`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:

- 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:

`entry_iter`	The iterator that must be destroyed.
`value_iter_t`	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:

- Returns non-zero 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:

- Returns non-zero 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:

- Returns non-zero 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:

- Returns non-zero 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:

`source_ptr`	The `am_properties_t` instance whose data must be copied.
`properties_ptr`	The pointer to the `am_properties_t` 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:

`properties`	The `am_properties_t` instance to which the key-value pairs needs to be loaded into.
`file_name`	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:

properties	The am_properties_t instance to which the key-value pairs needs to be saved.
file_name	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_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 `am_properties_t` 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 is 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:

properties	The am_properties_t instance from which the keys value needs to be extracted.
key	The key whose value will be returned.
value_ptr	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 is 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:

properties	The am_properties_t instance from which the keys value needs to be extracted.
key	The key whose value will be returned.

`default_value`	The value to be returned in case of any error condition.
`value_ptr`	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 is 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 For C

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:

`remoteUser`	After policy evaluation, this variable is assigned the name of the remote user.
`advice_map`	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: 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 buflength);
} am_resource_traits_t;
```

Parameters

This structure has the following components:

`cmpFuncPtr`	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.

`hasPatterns`	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.
`getResourceRoot`	This function takes a given resource and finds a root resource name for that resource. The length is the size of the rootResourceName buffer. The minimum required length of the rootResourceName 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 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:

service_name	The name of the service about to be created.
instance_name	The instance name of the new service. Currently this parameter is unused.
rsrcTraits	The resource traits structure that contains the pointers to the actual implementations of the resource name operations.
service_config_properties	The am_properties_t 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:

`policy_handle`	The policy service which needs to be destroyed.
`sso_token`	The single sign-on token of the user who must be evaluated for accessing the resource and perform a given action.
`resource_name`	The string form of the resource that the user wants to perform the action on.
`action_name`	The action the user wants to perform on the action.
`env_parameter_map`	The environment parameters like IP address the user is accessing the resource from.
`policy_response_map_ptr`	The response structure that will be populated after evaluation of the policies.
`policy_result`	Pointer to am_policy_result_t 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_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 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:

`policy_handle`	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:

`policy_handle`	The policy service instance whose configuration needs to be examined.
`notification_data`	The notification data that was received by the caller.
`notification_data_len`	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:

`policyResourceName`	The resource name as defined in the policy. This is the only parameter that can have patterns it them.
`resourceName`	The name of the resource accessed by the user.

usePatterns	The patterns in the policyResourceName if present must be used if this parameter is set to AM_TRUE 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:

`resourceName`	The resource name for which the root resource must be extracted.
`resourceRoot`	The pointer where the resource root will be copied to.
`length`	The length of the `resourceRoot` 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

Initialization variables are properties defined in AMAgent.properties. For a listing of these variables see the *Sun ONE Identity Server Web Policy Agents Guide* or *J2EE Policy Agents Guide*.

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:

config_file	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:

`sso_token`	The `sso_token` from the Identity Server cookie. This parameter may be NULL if there is no cookie present.
`url`	The URL whose accessibility is being determined. This parameter may not be NULL.
`action_name`	The action (GET, POST, etc.) being performed on the specified URL. This parameter may not be NULL.
`client_ip`	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.
`env_parameter_map`	A map containing additional information about the user attempting to access the specified URL. This parameter may not be NULL.
`result`	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:

request_url	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:

`data`	A buffer containing data as obtained from the request.
data_length	The length of the data contained in the buffer.

Returns:

None

Description:

This function processes the notification data as sent by 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 `am_web_is_access_allowed()`.
advice_map	The advice map contained in the policy result as obtained from the call to function `am_web_is_access_allowed()`.

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 the 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:

`result`	The result obtained from the call to function am_web_is_access_allowed().

`setFunc`	A function pointer for the call back function to be used when iterating over various values of the result.
args	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:

`memory`	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:

`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 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

Initialization variables are properties defined in `AMAgent.properties`. For a listing of these variables see the *Sun ONE Identity Server J2EE Policy Agents Guide* or *Web Policy Agents Guide*.

Policy Samples

Sample files have been included to help understand how the Policy Service can be used. These samples are located in *IdentityServer_base*`/SUNWam/samples/ policy` and `Readme.html` offers detailed instructions on how to:

- Add a new policy service.
- Develop and add custom Subjects, Referrals, and Conditions.
- Create policies using the `amadmin` command line tool.
- Develop and run a programmatic policy evaluation (i.e.: without the use of a policy agent).
- Construct policies programmatically and add them to the Directory Server data store.

Chapter 8

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 SAML assertions. This chapter explains SAML and defines how it is used within Identity Server. It contains the following sections:

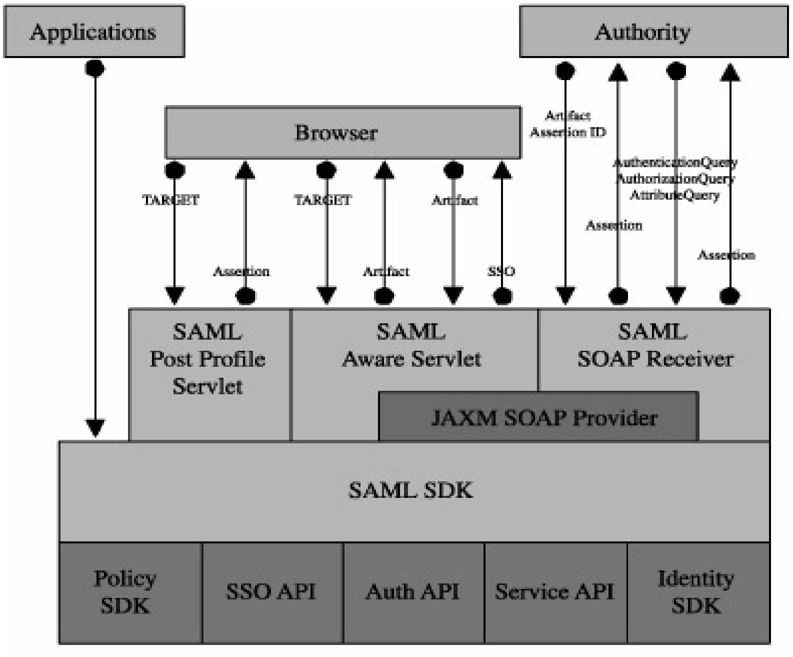
- Overview
- SAML Component Details
- amSAML.xml
- SAML SDK
- SAML Samples

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. The latest SAML information and specifications can be found at the Oasis Security Services Technical Committee home page.

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 machine, 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, access authorization decisions or attributes. 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.

Figure 8-1 SAML Interaction Within Identity Server



The lighter colored boxes are components of the SAML service.

The SAML Service allows Identity Server to work in the following ways:

- Users can authenticate against Identity Server and access trusted partner sites without having to reauthenticate. (This is a single sign-on process independent of the proprietary Identity Server process discussed in Chapter 4, “Single Sign-On And Sessions,” of this manual.)
- 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.
- The SAML SDK can be used to build Authentication, Authorization Decision and Attribute Assertions.
- The SAML Service provides pluggable XML-based digital signature signing and verifying.

NOTE Although the Federation Management module, as described in Chapter 9, "Federation Management," of this manual integrates aspects of the SAML specifications, it is independent of the Identity Server SAML Service as described in this chapter.

Accessing The SAML Service

The SAML Service can be accessed using a web browser or the SAML SDK. An end user would authenticate to Identity Server using a web browser and, when authorized to do so, access URLs from trusted partner sites. Developers, on the other hand, would integrate the API into their applications to enable them to exchange security information with Identity Server. For example, a Java application can use the SAML API to accomplish single sign-on. After obtaining a `SSOToken` from Identity Server, the application can call the `doWebArtifact()` method of the `SAMLClient` class which will send a SOAP request for authorization information to Identity Server and, if applicable, redirect the application to the destination site.

SAML Component Details

The following sections explain specific details of the components of the SAML Service. They include:

- Profile Types
- Assertion Types
- SAML SOAP Receiver

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 and, subsequently, processed at the trusted partner site. Identity Server supports two profiles that use HTTP: the Web Browser Artifact Profile and the Web Browser POST profile. Either of these profiles can be used in the case of single sign-on between two SAML-enabled entities, allowing an already authenticated user to access resources from a trusted partner site. Each profile has its benefits that include:

- Because Web Browser POST profile does not require the SOAP, it is more firewall-friendly and involves less steps and server side processing.
- Web Browser Artifact Profile requires less processing overhead because there is no assertion signing as there is in Web Browser POST profile.
- Web Browser Artifact Profile works without Javascript-enabled browsers.

NOTE The profile methods can be initiated through a web browser or the SAML API. More information on the API method can be found in "SAML SDK" on page 379.

Web Browser Artifact Profile

The Web Browser Artifact Profile defines interaction between three parties: a user equipped with a web browser, an authority site, and a trusted partner site. When an authenticated user attempts to access a trusted partner site (generally by clicking a link), they are directed to a transfer service at the authority site. In Identity Server, the transfer service is the SAML Aware Servlet. The base of the transfer URL is `http(s)://`*identity_server_host.domain_name*`:`*port*`/`*server_deploy_uri*`/SAMLAwareServlet`; it is appended with the URL of the location to which the user is requesting access (`?TARGET=URL_of_destination`). The SAML Aware Servlet then provides the following functions as part of the Web Browser Artifact Profile:

1. It compares the SAML Service's configured 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. Configured targets specify a domain and/or a port number. More information on this attribute can be found in the *Sun ONE Identity Server Administration Guide*.

2. Assuming the TARGET location was found in the list of Trusted Partner Sites, the SAML Aware Servlet looks for and validates the session token from the inbound request.

   Without a valid session token, Identity Server will not create an assertion.

3. The SAML Aware Servlet then 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 *assertion* contains the security information and is built from the user's session information and optional attribute information from the `siteAttributeMapper` class. (More information on the `siteAttributeMapper` can be found in "com.sun.identity.saml.plugins" on page 381.) The assertion can be signed.

NOTE 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 redirects the user's browser to the Artifact Receiver URL with a query string containing the artifact and the original TARGET location.

   The Artifact Receiver URL is based on mapping configurations defined in the SAML Service. More information on this can be found in the SAML Service Attributes chapter of the *Sun ONE Identity Server Administration Guide.*

NOTE In Identity Server, the Artifact Receiver URL and the SAML Aware Servlet are one and the same. Other SAML implementations might not integrate the two servlets.

5. At the Artifact Receiver URL, the artifact is extracted from the query string to find the SOAP Receiver URL.

   The SAML SDK extracts the source ID from the artifact and uses it to find the SOAP Receiver URL in the SAML Service configuration. "SAML SOAP Receiver" on page 376 has more information on the use of SOAP, a communications specification integrating XML and HTTPS.

6. A SAML request containing the artifact is then sent to the SOAP Receiver URL at the trusted partner site requesting the assertion to which the artifact points.

   The Artifact Receiver URL uses SOAP binding to request the assertion.

7. The SOAP Receiver URL accepts the returned artifact query from the trusted partner site and responds by sending the correct assertion in a SOAP response.

8. The assertion is processed, mapping the user account information from the trusted partner site to the target site's user account.

   The user is either granted or denied access to the trusted partner site. If access is granted a `SSOToken` is generated, a cookie is set to the browser and the user is redirected to the TARGET location.

NOTE A sample has been provided to test the Web Browser Artifact Profile function. "SAML Samples" on page 385 has more information.

Web Browser POST Profile

The Web Browser POST Profile allows security information to be supplied to a trusted partner site using the HTTP POST method (and 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)://`*identity_server_host.domain_name*`:`*port*`/`*server_deploy_uri*`/SAMLPOSTProfileServlet`; it is appended with the URL of the location to which the user is requesting access (`?TARGET=URL_of_destination`). The SAML POST Profile Servlet provides functions for the two POST Profile interactions. In the first interaction between the user and Identity Server:

1. Identity Server 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 can be found in the SAML Service Attributes chapter of the *Sun ONE Identity Server Administration Guide*.

2. It generates an assertion using the `AssertionManager` class of the SAML SDK.

   "com.sun.identity.saml" on page 379 contains information on the `AssertionManager` class.

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.

5. The user's browser is then directed to the location based on this information.

In the second interaction between the user and the trusted partner site:

1. The trusted partner site 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.

The POST profile function is provided by either of two means: an HTTP request using the SAMLPOSTProfileServlet, or an SAMLClient API call [doWebPost( )] to a Java application.

NOTE A sample has been provided to test the Web Browser POST Profile function. "SAML Samples" on page 385 has more information.

Single Use Policy With POST Profile

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, the 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.

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 a sample 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="2001-11-05T17:23:00GMT-02:00">
        <saml:AuthenticationStatement
AuthenticationMethod="urn:oasis:names:tc:SAML:1.0:am:password"
AuthenticationInstant="2001-11-05T17:22:00GMT-02:00">
            <saml:Subject>
                <saml:NameIdentifier NameQualifier="example.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.

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.

NOTE Identity Server uses SOAP, a message communications specification integrating XML and HTTPS, to transport requests and responses in its “Web Browser Artifact Profile” on page 372.

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 assertion. It responds to queries for authentication, attributes or authorization decisions as well as those that include an assertion identifier reference or artifact by returning assertions.

NOTE The access URL for the SAML SOAP Receiver is `http(s)://`*identity_server_host.domain_name*`:`*port*`/`*server_deploy_uri*`/SAMLSOAPReceiver`. The SAML SOAP Receiver only supports the POST method.

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 identifier references 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. The available methods are:

- NOAUTH
- BASICAUTH

- SSL
- SSLWITHBASICAUTH

This option is configured in the Trusted Partner Sites attribute of the SAML Service in the form:

`SourceID=`*source_id_of_site*`|SOAPUrl=`*url_of_site*`|AuthType=`*chosen_auth_option*`|User=`*user_id*

NOTE The value `user=`*user_id* is used only with the Basic Authentication and SSL With Basic Authentication options.

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 SAML Attributes chapter of the *Sun ONE Identity Server Administration Guide*.

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, the old `amSAML.xml` service file needs to be deleted, and the newly modified file reloaded using `amadmin`. Information on how to use `amadmin` can be found in The amadmin Command Line Tool chapter of the *Sun ONE Identity Server Administration Guide*. Information on the other SAML Service attributes can also be found in the *Sun ONE Identity Server Administration Guide*.

SAML SDK

Identity Server contains a SAML SDK made up of several Java 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`

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` provides methods to execute either the Artifact or POST profile from within an application as opposed to a web browser. Its methods include:

- `getAssertionByArtifact`—returns an assertion for a corresponding artifact.
- `doWebPOST`—is designed to do the SAML web-browser POST profile.
- `doWebArtifact`—is designed to do the SAML web-browser profile with artifact.

com.sun.identity.saml.assertion

This package contains the classes needed to create, manage, and integrate, an XML assertion into an 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 package defines classes common to all SAML elements including site_ID, issuer name and server host. It also contains all SAML-related exceptions.

CAUTION The date format, `yyyy-MM-dd'T'HH:mm:ss'+/-'HH:mm` , which was used in JDK 1.3.1 with IS 6.0 is no longer supported in IS 6.1. The correct format in JDK 1.4.1 for use in Identity Server 6.1 is:

`yyyy-MM-dd'T'HH:mm:ss'+/-'HHmm`

or

`yyyy-MM-dd'T'HH:mm:ss'GMT''+/-'HH:mm`

For example, the following are correct:

2003-04-22T01:20:02 -0001 (with a space before the zone sign)

2003-04-22T01:20:02GMT-00:01

2003-04-22T01:20:02-0001

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 external partner site user accounts to Identity Server user accounts for purposes of single sign-on. A default account mapper implementation is provided. If a site-specific account mapper is not configured, this default mapper is used.

NOTE The default account mapper class is `com.sun.identity.saml.plugin.DefaultAccountMapper`.

For example, assume the single sign-on is configured from site A to site B, then a site-specific account mapper can be developed and added to site B's Trusted Partner Sites listing in this format:

`sourceid=`*site_A_source_id* `| accountmapper=`*class_name_of_site specific_account_mapper* `| ...`

When site B processes the assertion received through either SAML profile, it finds out the source ID of the originating site and locates the account mapper corresponding to that site.

NOTE Turning on the Debug Service in `AMConfig.properties` file, would log additional information concerning the account mapper. For example, was it loaded or what is the user name and organization to which it has been mapped. Information on this can be found in Appendix A, "AMConfig.properties File," in this manual.

- `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 Identity Server understands. A default attribute mapper is provided. A site-specific attribute mapper can be developed in this format:

`sourceid=`*site_source_id* `|`
`attributemapper=`*class_name_of_site_specific_attribute_mapper* `|...`

- `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. A site-specific action mapper can be developed in this format:

  `sourceid=`*site_source_id* `|`
  `actionmapper=`*class_name_of_site_specific_action_mapper*`|...`

- `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 the web browser Artifact or POST profiles, 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. A site attribute mapper can be developed in this format:

  `sourceid=`*site's source ID* `|`
  `siteattributemapper=`*class_name_of_site_specific_siteattribute_mapper*`|...`

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`.

AuthorizationDecisionQuery Sample

There are many ways to form an authorization decision query and have the decision assertion returned. Code Example 8-3 illustrates one way to do it.

Code Example 8-3 AuthorizationDecisionQuery Code Sample

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

Code Example 8-3 AuthorizationDecisionQuery Code Sample *(Continued)*

```
            authzQuery = new AuthorizationDecisionQuery(eviSubject1, actions,
                                 evidence, resource);
             try {
                 assertion = am.getAssertion(authzQuery, destID);
             } catch (SAMLException e) {
                 out.println("--failed. Exception:" + e);
             }
```

com.sun.identity.saml.xmlsig

All SAML assertions, requests and responses may be signed using this signature API. This is an SPI in which the interfaces can be implemented and proprietary XML / signature implementations can be plugged in. This package contains the classes needed to sign and verify. By default, the keystore provided with the JDK is used and the key type is DSA. The configuration properties for this functionality are in AMConfig.properties. Information on these properties can be found in “SAML” on page 450 of Appendix A, “AMConfig.properties File.” See “SAML Samples” for information on the signature functionality.

SAML Samples

There are several samples that can be accessed from the Identity Server installation. They are located in *IdentityServer_base*/`SUNWam/samples/saml`. 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 *IdentityServer_base*/`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 *IdentityServer_base*/`SUNWam/samples/saml/query`.
- A sample application for achieving SSO using either the Web Browser Artifact or the Web Browser POST profiles. This sample is located in *IdentityServer_base*/`SUNWam/samples/saml/sso`.
- A sample that illustrates how to use the XMLSIG API. It details how to configure for XML signing and is located in *IdentityServer_base*/`SUNWam/samples/saml/xmlsig`.

A README file is included with each sample with information and instructions on how to use it.

Chapter 9

Federation Management

Sun™ ONE Identity Server 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 Concepts
- Federation Process
- The Federation Management Interface
- Federation Management Protocols
- Federation Management API
- 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 of these service providers. These attributes uniquely identify the individual with that provider and can include a name, phone number, passwords, 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 link 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 the local identities configured with multiple service providers. A federated identity allows users to login at one service provider's site and move to an affiliated service provider site without having to re-authenticate or re-establish their identity. The **Liberty Alliance Project** was implemented for this purpose.

NOTE The Liberty Alliance Project's specifications, v. 1.0 is no longer supported by Identity Server. As there are virtually no 1.0 deployments, this does not have a serious impact.

The Liberty Alliance Project

The goal of the Liberty Alliance Project is to enable individuals and organizations to easily conduct network 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 service provider accounts.
- Single sign-on where a user can log in, authenticate to one service provider and gain access to other service providers with which they have federated without having to log in again.
- Authentication context where service providers with federated 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 an identity or service provider site and is automatically logged out of all sites that maintain a live session.
- Account linking termination where users can choose to stop their account federation.

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 includes service providers (who offer web-based services to users), identity providers (service providers that also maintian and manage identity information), and the users themselves. Once a circle of trust is established between providers, users can choose to federate any or all identities they might have with the service providers that have joined this circle, enabling them to make use of the federated authentication capabilities.

Liberty Specifications 1.1 Changes

Identity Server 6.0 used the Liberty Specifications, v.1.0 as the basis for its Federation Management Service. This current version of Identity Server is based on the Liberty Specifications, v.1.1. Some of the changes include:

1. Name registration can now be initiated both from a service provider and from an identity provider rather than just a service provider.
2. The XML Digital Signature requirements have been enhanced for clairty.
3. The sample applications are compliant with Liberty 1.1.
4. A response message has been added to Logout.

NOTE This chapter does not describe in detail the Liberty Alliance Project's specifications, v. 1.1. For more information on the specifications and the authentication process referred to above, see the Liberty Alliance Project's website at http://www.projectliberty.org.

Federation Concepts

The Federation Management module built into the Identity Server is designed to be compatible with the Liberty Alliance Project's version 1.1 specifications. A number of concepts are derived from these specifications. For clarification, definitions of these concepts are provided in this section.

Account Federation (Identity Federation)

Account Federation occurs when a user chooses to unite distinct service accounts and identity provider accounts. Users retain the individual account information with each provider in the while, simultaneously, establishing a link that allows the exchange of authentication information between them.

Authentication Domain

See Circle Of Trust.

Circle Of Trust

A Circle Of Trust is a group of service providers who agree to join together in order to exchange user authentication information using Liberty-enabled technologies. This Circle Of Trust must contain at least one Identity Provider. Once a Circle Of Trust is established, single sign-on is enabled between all the providers.

NOTE A Circle Of Trust is, at times, referred to as an *Authentication Domain* although it is not a domain in the Internet sense of the word.

Common Domain

A common domain is a unique, Liberty-specifiedInternet domain between a Service Provider and an Identity Provider in a Circle Of Trust. The CommonDomain(Introduction) protocol helps service providers to find the preferred identity provider in a circle of trust. When a user authenticates to a service provider, a common domain cookie is used to store the user's identity provider. (`_liberty_idp` is the cookie name.) When the user attempts to access a new service provider in the circle of trust, the service provider reads the common domain cookie and the request can be forwarded to the correct identity provider.

Federated Identity

A federated identity refers to the amalgamation of the account information in all service providers accessed by one user (personal data, authentication information, buying habits and history, shopping preferences, etc.). The information is administered by the user yet, with the user's consent, their privilege to access information is securely shared with their providers of choice.

Federation Termination

Users have the ability to terminate federations. Federation termination (or *defederation*) results in the cancellation of affiliations established between the user's identity provider and federated service provider accounts.

Identity Provider

Identity providers are service providers that specialize in providing authentication services. As the administrating service for authentication, they also maintain and manage identity information. Authentication accomplished by an Identity Provider is honored by all service providers with whom it is affiliated.

Name Identifier

To help preserve anonymity, 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 using a pseudonym or *name identifier*. Neither the identity provider nor the service provider should have knowledge of the user's actual identity.

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, libraries, universities, and governmental agencies.

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.

Single Sign-on

Single sign-on is established when a user with a Federated Identity authenticates to an Identity Provider. Because they have previously opted-in for federation, they are now able to access affiliated service providers without having to re-authenticate.

Trusted Provider

A Trusted Provider is a generic term for one of a group of service and identity providers in an Circle Of Trust. Users can transact and communicate with Trusted Providers in a secure environment.

Federation Process

Out of the box, Identity Server has two options for user authentication. The first is the proprietary Authentication Service; the second is the Liberty-enabled Federation Management. With the first option when a user tries to access a resource protected by Identity Server, they are redirected to the Authentication Service for authorization using an Identity Server login page. When the user provides credentials, the Authentication Service verifies them and either allows or denies access.

NOTE More information on the process flow of the proprietary Authentication Service can be found in Chapter 3, “Authentication Service,” of this manual.

With the second option, Liberty-enabled Federation Management, when a user attempts to access an identity or service provider resource protected by Identity Server, the authentication process begins with a search for a valid Identity Server session token.

NOTE As discussed in “Cookies and Sessions” on page 190 of Chapter 4, “Single Sign-On And Sessions,” an Identity Server session token is inserted in a web server cookie.

If a session token is found, the user is granted access to the requested page. This page, assuming it belongs to a member of the Circle Of Trust, contains a link which, if clicked, directs the user through the Federation Single Sign-On Process process. If, on the other hand, no session token is found, the user moves through the Pre-Login Process.

NOTE This sample process occurs when there is one identity provider. The flow changes a bit if there are mutiple identity providers.

Pre-Login Process

The purpose of the Pre-Login process is to establish a valid user session at the service provider site. When no Identity Server session token is found, the pre-login process begins with the search for another type of cookie, a Federation cookie.

NOTE The Federation cookie is not the Identity Server cookie discussed in "Cookies and Sessions" on page 190 of Chapter 4, "Single Sign-On And Sessions." The Federation cookie is a persistent cookie; it lives on the computer's hard drive and is read into cache each time the browser is opened.

If, after the search for an Identity Server session token proves null, no Federation cookie is found either, an Identity Server login page is displayed and the user submits credentials to the Authentication Service. When authenticated by the Identity Server, the user is redirected to the requested page which contains a link to allow the user to federate their identity. If the user clicks this link, they are directed through the Federation Single Sign-On Process.

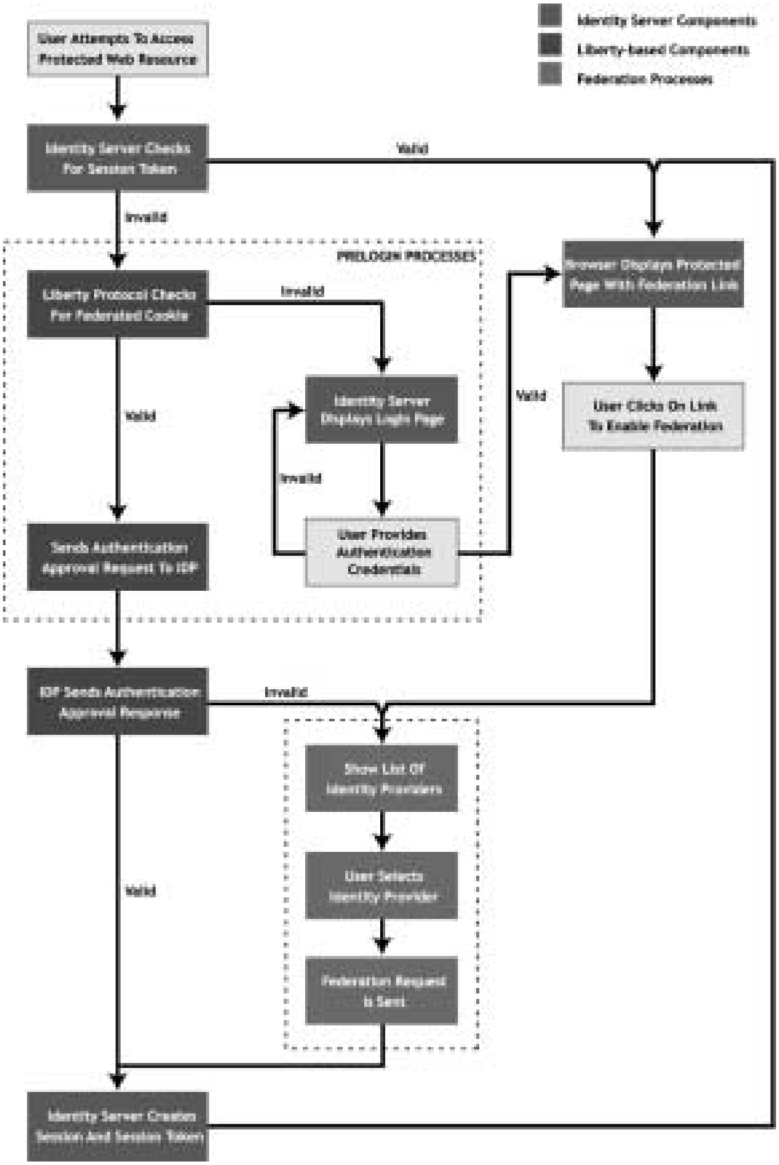
If though, after the search for an Identity Server session token proves null, a valid Federation cookie is found, it means the user has already been federated but not yet authenticated by an identity provider within the Circle Of Trust. This is confirmed by sending a request for authentication to the user's identity provider. When an affirmative response is received back, the user is redirected to the Identity Server Authentication Service where they are automatically granted a session token and redirected to the requested page. If the response from the identity provider is negative (for instance, if the session has timed out), the user is sent back through the Federation Single Sign-On Process.

Federation Single Sign-On Process

The Federation Single Sign-On process gives the user the option to authenticate to, and federate their identity with, an identity provider. This federation affords access to all provider sites within the Circle Of Trust (including the service provider which houses the resource originally requested) without having to reauthenticate. Therefore, if an identity provider has responded to a request for authentication confirmation in the negative, the user is directed to a list of identity providers from which they can choose for authentication using the Federation Single Sign-On process. The user selects an identity provider and sends their credentials for authentication. Once authentication is granted, the user is redirected to the Identity Server Authentication Service where they are automatically granted a session token and redirected to the requested page which contains a link to allow the user to federate their identity for future single sign-on opportunities.

NOTE A user who has an account defined with a service provider can only login to that service provider without federating. This is a l*ocal login*. A *federated login* is when that same user logs into a different service provider via federation with an identity provider. The service provider has the option to turn off local login if it wants to honor federated login only. Both types of logins are demonstrated in the samples, located in *IdentityServer_base*`/SUNWam/samples/liberty/`.

Figure 9-1 Liberty-enabled Identity Server Authentication Process Flow


Identity Server Components
Liberty-based Components
Federation Processes
User Attempts To Access Protected Web Resource
Identity Server Checks For Session Token
Valid
Invalid
PRELOGIN PROCESSES
Liberty Protocol Checks For Federated Cookie
Invalid
Browser Displays Protected Page With Federation Link
Identity Server Displays Login Page
Valid
Valid
User Clicks On Link To Enable Federation
Invalid
User Provides Authentication Credentials
Sends Authentication Approval Request To IDP
IDP Sends Authentication Approval Response
Invalid
Show List Of Identity Providers
User Selects Identity Provider
Federation Request Is Sent
Valid
Identity Server Creates Session And Session Token

The Federation Management Interface

The Federation Management module uses JavaServer Pages (JSP) to define the look and feel of its pages. JSP are HTML files that contain additional code to generate dynamic content. More specifically, JSP contain HTML code to display static text and graphics, as well as application code to generate information. When the page is displayed in a web browser, it will contain both the static HTML content and, in the case of the Federation Management module, dynamic content retrieved via calls to the Federation Management API. An administrator can customize the look and feel of the interface by changing the HTML tags in the JSP, but the APIs invoked must not be changed. By default, the JSP are located in *IdentityServer_base*/`SUNWam/web-apps/services/config/federation/default`. The files in this directory provide a default interface. 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 would be modified as needed. Table 9-1 a list of the JSP with details on what each page is used for and the invoked APIs that cannot be modified.

Table 9-1 Federation Management Module JSP

File Name and its Purpose	Invoked APIs
CommonLogin.jsp displays the links to the login pages of the trusted Identity Providers as well as the local login page. It is displayed when the user is not logged in locally or at an Identity Provider site. The list of identity providers is obtained by the `getIDPList(hostedProviderID)` method.	• LibertyManager.getLoginURL(request) • LibertyManager.getInterSiteURL(request) • LibertyManager.getIDPList(providerID) • LibertyManager.getNewRequest(request) • LibertyManager.getSuccintID(idpID) • LibertyManager.cleanQueryString(request)
Error.jsp displays an error page when an error has occurred.	No APIs are invoked.
Federate.jsp is displayed when the user clicks a Federate link. It displays a drop-down list of all providers with which the user is not yet federated. This list is constructed from the `getProvidersToFederate(userName, providerID)` method.	• LibertyManager.isLECPProfile(request) • LibertyManager.getAuthnRequestEnvelope(request) • LibertyManager.getUser(request) • LibertyManager.getProvidersToFederate(providerID,userDN)
FederationDone.jsp displays the status of federation (success or cancelled). It checks this status using the `isFederationCancelled(request)` method.	• LibertyManager.isFederationCancelled(request)

Table 9-1 Federation Management Module JSP *(Continued)*

File Name and its Purpose	Invoked APIs
Footer.jsp displays a branded footer included on all the pages.	No APIs are invoked.
Header.jsp displays a branded header included on all the pages.	No APIs are invoked.
ListOfCOTs.jsp displays a list of Circles Of Trust. When a user is authenticated by an identity provider and the service provider belongs to more than one Circle Of Trust, they will be shown this JSP to select an authentication domain as their preferred domain. In the case that the provider belongs to only one domain, this page will not be displayed. The list is obtained by using the `getListOfCOTs(providerID)` method.	• LibertyManager.getListOfCOTs(providerID)
LogoutDone.jsp displays the status of the local logout operation.	• LibertyManager.isLogoutSuccess(request)
NameRegistration.jsp is displayed when a federated user chooses to register a new Name Identifier from a service provider to an identity provider. When the Name Registration link is clicked, this JSP is displayed.	• LibertyManager.getUser(request) • LibertyManager.getRegisteredProviders(userDN)
NameRegistrationDone.jsp displays the status of `NameRegistration.jsp`. When finished, this page is displayed.	• LibertyManager.isNameRegistrationSuccess(request) • LibertyManager.isNameRegistrationCanceled(request
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 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.	• LibertyManager.getUser(request) • LibertyManager.getFederatedProviders(userDN)
TerminationDone.jsp displays the status of federation termination (success or cancelled). It checks status using the `isTerminationCancelled(request)` method.	• LibertyManager.isTerminationSuccess(request) • LibertyManager.isTerminationCanceled(request)

Federation Management Protocols

The Federation Process is based on one of the protocols defined by the Liberty Alliance Project's specifications. They are:

- Single Sign-on and Federation Protocol
- Federation Termination Notification Protocol
- Name Registration Protocol
- Single Log-Out Protocol
- IDP Introduction Protocol

Identity Server provides the implementation of these Liberty protocols. A brief explanation of each protocol follows.

Single Sign-on and Federation Protocol

This is the protocol on which Identity Server bases the Federation Process. It defines the process that a user at a service provider goes through to authenticate their identity with an identity provider. It also specifies the means by which a service provider obtains an Authentication Assertion from an identity provider to allow 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 provider to provider. This is primarily the Browser Artifact SSO profile.
- Form POST-based Single Sign On and Federation Protocol relies on a browser POST to communicate between providers.

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. Said provider will notify all other providers in the Circle Of Trust when a user defederates. 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 provider to provider.
- HTTP Redirect-based Federation Termination Notification Protocol.

Name Registration Protocol

At the time of federating a user account, the identity provider generates a *name identifier*, a pseudonym that all providers use when referring to the user during communications. This is the *IDP Provided NameIdentifier*. The identity provider may also register the name identifier at any time subsequent to federation. In addition, a service provider may register an *SP Provided NameIdentifier* which it expects the identity provider to use when communicating with it about the user account. If neither of the previous two scenarios occurs, the *IDP Provided NameIdentifier* generated at the time of federation is used by both providers.

Single Log-Out Protocol

This is the protocol used to synchronize the session log-out functionality across all sessions that were authenticated and created 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 calls between providers.
- HTTP Redirect-based Single Log-Out Protocol.
- IDP Initiated HTTP GET Protocol.

IDP Introduction Protocol

In a Circle Of Trust that has more than one identity provider, the service providers need a way to determine which provider is preferred by the user. 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 can implement this protocol.

NOTE It is recommended that the system clocks of all identity provider and service provider servers are synchronized. A time sync server can be used for this purpose.

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 specific 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 a specific 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.

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 *IdentityServer_base*`/SUNWam/samples/liberty/` directory. Instructions on how to implement the samples can be found in the `README.html` 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. It also demonstrates the configuration of different authentication levels for different resources. At least two server machines are required for this sample.
- 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.

Chapter 10

Auditing Features

Sun™ ONE Identity Server provides a Logging Service to record information such as user activity, traffic patterns, and authorization violations. The Logging API allow external applications to take advantage of the Logging Service. In addition, the debug files allow administrators to troubleshoot their installation. This chapter explains these auditing features. It contains the following sections:

- Logging Service Overview
- Log Files
- Logging Features
- Logging API
- Logging SPI
- Debug Files

Logging Service Overview

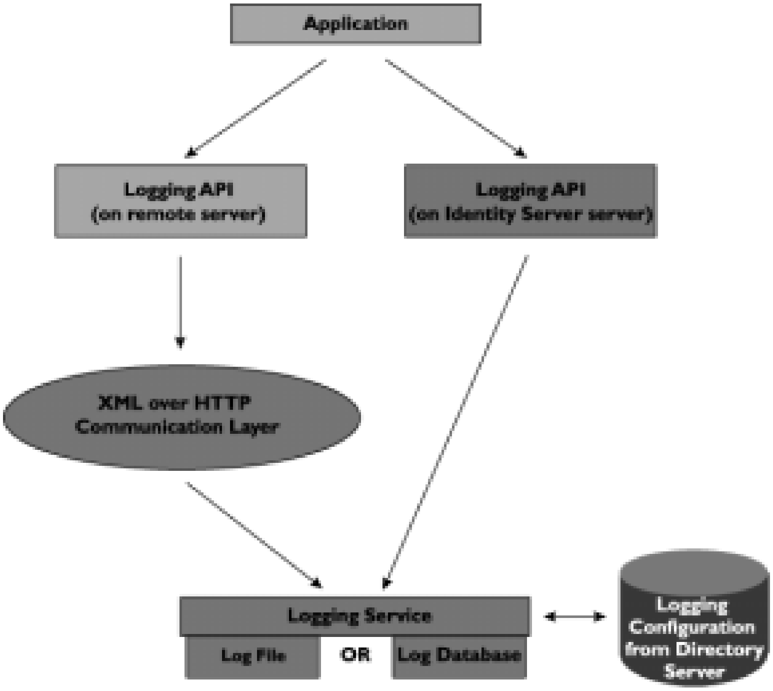
The Logging Service enables all Identity Server services to record information that might be useful to the administrator in one centralized location. The information may include access denials and approvals, authorization violations and code exceptions. Logging allows administrators to analyze user activity, Identity Server traffic patterns and authorization violations. As with all Identity Server services, the Logging Service uses a global service 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). The default location for all log files is `/var/opt/SUNWam/logs`.

NOTE This default log directory can be reconfigured after installation by modifying the Log Location attribute in the Logging Service. More information can be found in the Logging Service Attributes chapter in the *Sun ONE Identity Server Administration Guide.*.

Logging Architecture

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 Identity Server. An application accesses the Logging Service by calling the Logging API. (If remote, the API uses a XML over HTTP layer to send the logging request to the Logging Service.) The Identity Server SDK loads the configuration data (stored in Directory Server) into the Logging Service when Identity Server starts up or when any logging configuration data is changed via the console. This data includes the log message format, log file name, maximum log size, and the number of history files. Any exception message will be logged, based on the configuration values. Figure 10-1 illustrates the architecture of the Logging Service.

Figure 10-1 Logging Service Architecture


Application
Logging API (on remote server)
Logging API (on Identity Server server)
XML over HTTP Communication Layer
Logging Service
Log File
OR
Log Database
Logging Configuration from Directory Server

amLogging.xml

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 `/etc/opt/SUNWam/config/xml`. These values are applied across the Identity Server deployment and inherited by every configured organization. The structure of `amLogging.xml` is defined by the `sms.dtd`. Information on this document can be found in "The sms.dtd Structure" on page 255 of Chapter 6, "Service Management." Specific information on the Logging Service attributes can be found in the Logging Service Attributes chapter in the *Sun ONE Identity Server Administration Guide*.

Log Files

The log files record a number of events for each of the services it monitors. These files should be checked by the administrator on a regular basis. The default directory for the log files is `/var/opt/SUNWam/logs`.

NOTE The log file directory can be configured in the Logging Service via the Identity Server console.

Recorded Events

The Logging Service logs information passed to the `LogRecord` class by the client. Out-of-the-box, the contents of the `LogRecord` that will be logged are:

Time

This record is the date (`YYYY-MM-DD`) and time (`HH:MM:SS`) at which the log message was recorded.

Data

This record details the description of the user activity, errors or other useful information which the application wants to log.

ModuleName

This record is the name of the Identity Server service or application being logged. Additional information on the value of this field can be found in "Adding Log Data" on page 416.

Domain

This field records the Identity Server domain to which the user belongs.

Log Level

This record corresponds to the Java 2 Platform, Standard Edition (J2SE) version 1.4 log level of the log record.

Login ID

This field is the ID of the user attempting to access the application. The information (the user to whom the log information belongs) is taken from the session token.

IP Address

This field records the IP address from which the operation was performed.

Logged By

This field is the user who writes the log record. The information is taken from the session token passed during `logger.log(logRecord, ssoToken)`.

Host Name

This field is the host name from which the operation was performed.

Additional fields can also be logged. The new field names must first be added to the `amLogging.xml` service file and the modified service file then reloaded into the Directory Server. The new values for these fields would then be included in the LogRecord Class passed to the Logging Service. More information on how to modify and load an XML service file can be found in "Defining A Custom Service" on page 243 of Chapter 6, "Service Management."

NOTE Only the flat file format can accomodate new logging fields. Other formats might contain steps not documented here. An example would be the database table where a new column must also be added to the table.

Log File Formats

Identity Server can record events in flat text files or a relational database. (The JDK SPI allows extending existing handlers or adding new ones.)

Flat File Format

The default flat file format is the W3C Extended Log Format (ELF). In leveraging this format, the Logging Service records the default logging fields in each log record. Code Example 10-1 illustrates an authentication log record formatted for a flat file. In order, the fields for these values are TIME, DATA, MODULENAME, DOMAIN, LOGLEVEL, LOGINID, IPADDR, LOGGEDBY, and HOSTNAME.

Code Example 10-1 Flat File Record From amAuthentication.access

```
"08-07-2003 07:58:26"   "Login Success service->adminconsoleservice"    LDAP
dc=example,dc=com       INFO    uid=amAdmin,ou=People,dc=example,dc=com
129.149.247.58 "cn=dsameuser,ou=DSAME Users,dc=example,dc=com"
cache1nwk.SFBay.Sun.COM
```

Relational Database Format

For Java 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 the data. Oracle® and MySQL databases are currently supported.

NOTE JDBC technology is an API for accessing tabular data source using Java. It provides connectivity to a wide range of SQL databases, and access to other tabular data sources, such as spreadsheets or flat files.

Table 10-1 contains the schema for a relational database.

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.
MODULENAME	VARCHAR2(255)	The name of the Identity Server service invoking the log record.

Table 10-1 Relational Database Log Format *(Continued)*

Column Name	Data Type	Description
`DOMAIN`	VARCHAR2(255)	Identity Server domain of the user.
`LOGLEVEL`	VARCHAR2(255)	JDK 1.4 log level of the log record.
`LOGINID`	VARCHAR2(255)	Login ID of the user who performed the logged operation.
`IPADDR`	VARCHAR2(255)	IP Address of the machine from which the logged operation was performed.
`LOGGEDBY`	VARCHAR2(255)	Login ID of the user who writes the log record.
`HOSTNAME`	VARCHAR2(255)	Host name of machine from which the logged operation was performed.

Oracle Database

In order to log to an Oracle database, the Log Location attribute in the Identity Server Logging Service and the driver variable in the database itself need to be modified. Using the Identity Server console, change the value of the Log Location attribute to:

`jdbc:oracle:thin:`*@hostname*`:1521:`*database_name*

In the database itself, change the value for the driver to:

```
oracle.jdbc.driver.OracleDriver
```

MySQL Database

In order to log to an MySQL database, the Log Location attribute in the Identity Server Logging Service and the driver variable in the database itself need to be modified.

NOTE There is a limitation in the data length for MySQL JDBC logging as MySQL does not support data of more than 255 characters.

Using the Identity Server console, change the value of the Log Location attribute to:

`jdbc:mysql:`*//hostname:port/database_name*

In the database itself, change the value for the driver to:

```
com.mysql.jdbc.Driver
```

CAUTION When MySQL is installed on Solaris or other Unix platforms and modifications are made to the Logging Service, logging into the MySQL database shows the warning message *Syntax error or access violation*.

Java Enterprise System Installation Logs

Events recorded during installation are stored in `/var/sadm/install/logs`. As Identity Server is installed via Java Enterprise System (JES), the events are recorded by the JES installer. The four installation logs are:

- Java_Enterprise_System_Config_Log
- Java_Enterprise_System_Summary_Report_install
- Java_Enterprise_System_install
- Java_Enterprise_System_shared_component_install

Identity Server Service Logs

There are two different types of service log files: access and error. Access log files record general auditing information concerning the deployment (successful or failed authentications, new federations, etc.). Error log files record errors that occur within the application. Flat log files are appended with the `.error` or `.access` extension; database column names end with `_ERROR` or `_ACCESS`. For example, a flat file logging console events would be named `amConsole.access` while a database column logging the same events would be called `AMCONSOLE_ACCESS`. The following sections describe the log files recorded by the Logging Sevice.

Session Logs

The Logging Service records the following events for the Session Service:

- `Login`
- `Logout`
- `Session Idle TimeOut`
- `Session Max TimeOut`
- `Failed To Login`
- `Session Reactivation`

- `Session Destroy`

The session logs are prefixed with `amSSO`.

Console 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 console logs are prefixed with `amConsole`.

Authentication Logs

The Authentication component logs user logins and logouts. The authentication logs are prefixed with `amAuthentication`.

Federation 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 federation logs are prefixed with `Federation`.

Policy Logs

The Policy component records policy-related events including, but not limited to, policy administration (policy creation, deletion and modification) and policy evaluation. The policy logs are prefixed with `amPolicy`. Code Example 10-2 on page 410 is a collection of sample records that might appear in the policy logs.

Code Example 10-2 Sample Policy Log Records

```
#Fields: time   Data    ModuleName      Domain  LogLevel        LoginID
IPAddr LoggedBy        HostName

"08-07-2003 11:08:19"   "Created policy test successfully in
Organization dc=iplanet,dc=com"    amPolicy.access "Not Available"
INFO    uid=amAdmin,ou=People,dc=iplanet,dc=com /192.18.120.236
uid=amAdmin,ou=People,dc=iplanet,dc=com 192.18.120.236

"08-07-2003 11:08:55"   "Modified policy test successfully in
Organization dc=iplanet,dc=com"   amPolicy.access "Not Available"
INFO    uid=amAdmin,ou=People,dc=iplanet,dc=com /192.18.120.236
uid=amAdmin,ou=People,dc=iplanet,dc=com 192.18.120.236

"08-07-2003 11:09:05"   "Removed policy test successfully in
Organization dc=iplanet,dc=com"    amPolicy.access "Not Available"
INFO    uid=amAdmin,ou=People,dc=iplanet,dc=com /192.18.120.236
uid=amAdmin,ou=People,dc=iplanet,dc=com 192.18.120.236
```

Code Example 10-2 Sample Policy Log Records *(Continued)*

```
"08-07-2003 11:15:43"   "Policy Evaluation result of Policy test in
Organization dc=iplanet,dc=com for service iPlanetAMWebAgentService,
resource http://moonshadow.red.iplanet.com:80/*.html and action names
[GET, POST] is GET=[allow]\\n." amPolicy.access "Not Available" INFO
uid=amAdmin,ou=People,dc=iplanet,dc=com /192.18.120.236
uid=amAdmin,ou=People,dc=iplanet,dc=com 192.18.120.236
```

Agent Logs

The policy agent logs are responsible for logging exceptions regarding log resources that were either allowed or denied to a user. The agent logs are prefixed with `amAgent`. `amAgent` logs reside on the agent server only. Agent events are logged on the Identity Server machine in the Authentication Logs. For more information on this function, see the correct documentation for the policy agent inquestion.

SAML Logs

The SAML component records SAML-related events including, but not limited to, assertion and artifact creation or removal, response and request details, and SOAP errors. The session logs are prefixed with `amSAML`.

amAdmin Logs

The command line logs record event errors that occur during operations using the command line tools. These include, but are not limited to, loading a service schema, creating policy and deleting users. The command line logs are prefixed with `amAdmin`. More information can be found in “Command Line Logging” on page 412.

Logging Features

The Logging Service has a number of special features which can be enabled for additional functionality. They include To Enable Secure Logging, Command Line Logging and Remote Logging.

To Enable Secure Logging

This optional feature adds additional security to the logging function. Secure Logging enables detection of unauthorized changes to, or tampering of, the security logs. No special coding is required to leverage this feature. Secure Logging is accomplished by using a pre-registered certificate configured by the system administrator. This Manifest Analysis and Certification (MAC) is generarated and stored for every log record. A special "signature" log record is periodically inserted that represents the signature for the contents of the log written to that point. The combination of the two records ensures that the logs have not been tampered with. Secure Logging can be enabled by performing the following steps:

1. Create a certificate with the name *Logger* and install it in the deployment container running Identity Server.

   Refer to the documentation that comes with the deployment container for details.

2. Turn on Secure Logging in the Logging Service configuration using the Identity Server console and save the change.

   The administrator can also modify the default values for the other attributes in the Logging Service.

3. Create a file in the *IdentityServer_base*/`SUNWam/config` directory that contains the certificate database password and name it `.wtpass`.

NOTE The file name and the path to it is configurable in the AMConfig.properties file. For more information see the "Certificate Database" on page 447 of Appendix A, "AMConfig.properties File."

   Ensure that the deployment container user is the only administrator with read permissions to this file for security reasons.

4. Restart the server after making these changes.

Command Line Logging

The `amadmin` command line tool has the ability to create, modify and delete identity objects (organizations, users, and roles, for example) in Directory Server. This tool can also load, create, and register service templates. The Logging Service can record these command line actions by invoking the `-t` option. If the `com.iplanet.am.logstatus` property in `AMConfig.properties` is enabled

(ACTIVE) then a log record will be created. (This property is enabled by default.) The command line logs are prefixed with `amAdmin`. More information can be found in Chapter 8, "The amadmin Command Line Tool" in the *Sun ONE Identity Server Administration Guide*.

Remote Logging

Identity Server supports remote logging. This allows a client using the Identity Server SDK to create log records on an instance of Identity Server deployed on a remote machine.

Using Remote Logging

Remote logging can be initiated in any of the following scenarios:

- When the logging URL in the Naming Service of one Identity Server instance points to a remote instance and there is a trust relationship configured between the two, logs will be written to the remote Identity Server instance.
- When the Identity Server SDK is installed against a remote Identity Server instance and a client (or a simple Java class) running on the SDK server uses the logging APIs, the logs will be written to the remote Identity Server machine.
- When logging APIs are used by Identity Server agents.

Enabling Remote Logging

To enable remote logging, ensure that the following information is regarded.

- If using Sun ONE Web Server, the following environment variables need to be set in the `server.xml` configuration file.

  a. `java.util.logging.manager=com.sun.identity.log.LogManager`

  b. `java.util.logging.config.file=/`*IdentityServer_base*`/SUNwam/lib/LogConfig.properties`

  ❍ If the Java™ 2 Platform, Standard Edition being used is 1.4 or later, this is accomplished by invoking the following at the command line:

  ```
  java -cp
  /IdentityServer_base/SUNWam/lib/am_logging.jar:/IdentityServer_base/S
  UNWam/lib/xercesImpl.jar:/IdentityServer_base/SUNWam/lib/xmlPars
  erAPIs.jar:/IdentityServer_base/SUNWam/lib/jaas.jar:/IdentityServer_
  base/SUNWam/lib/xmlParserAPIs.jar:/IdentityServer_base/SUNWam/lib
  ```

```
/servlet.jar:/IdentityServer_base/SUNWam/locale:/IdentityServer_base/
SUNWam/lib/am_services.jar:/IdentityServer_base/SUNWam/lib/am_sd
k.jar:/IdentityServer_base/SUNWam/lib/jss311.jar:/IdentityServer_base
/SUNWam/lib:.
```

```
-Djava.util.logging.manager=com.sun.identity.log.LogManager
```

```
-Djava.util.logging.config.file=/IdentityServer_base/SUNwam/lib/
LogConfig.properties <logTestClass>
```

- ❍ If the Java 2 Platform, Standard Edition being used is earlier than 1.4, this is accomplished by invoking the following at the command line:

```
java
-Xbootclasspath/a:/IdentityServer_base/SUNWam/lib/jdk_logging.ja
r -cp
/IdentityServer_base/SUNWam/lib/am_logging.jar:/IdentityServer_base/S
UNWam/lib/xercesImpl.jar:/IdentityServer_base/SUNWam/lib/xmlPars
erAPIs.jar:/IdentityServer_base/SUNWam/lib/jaas.jar:/IdentityServer_
base/SUNWam/lib/xmlParserAPIs.jar:/IdentityServer_base/SUNWam/lib
/servlet.jar:/IdentityServer_base/SUNWam/locale:/IdentityServer_base/
SUNWam/lib/am_services.jar:/IdentityServer_base/SUNWam/lib/am_sd
k.jar:/IdentityServer_base/SUNWam/lib/jss311.jar:/IdentityServer_base
/SUNWam/lib:.
```

```
-Djava.util.logging.manager=com.sun.identity.log.LogManager
```

```
-Djava.util.logging.config.file=/IdentityServer_base/SUNwam/lib/
LogConfig.properties <logTestClass>
```

- Ensure that the following parameters are configured in `LogConfig.properties` located in *IdentityServer_base*/`SUNWam/lib`.

a. `iplanet-am-logging-remote-handler=com.sun.identity.log.handlers.RemoteHandler`

b. `iplanet-am-logging-remote-formatter=com.sun.identity.log.handlers.RemoteFormatter`

c. `iplanet-am-logging-remote-buffer-size=1`

Remote logging supports buffering on the basis of the number of log records. This value defines the log buffer size by the number of records. Once the buffer is full, all buffered records will be flushed to the server.

d. `iplanet-am-logging-buffer-time-in-seconds=3600`

This value defines the time-out period in which to invoke the log buffer-cleaner thread.

e. `iplanet-am-logging-time-buffering-status=OFF`

This value defines whether log buffering (and the buffer-cleaner thread) is enabled or not. By default this feature is turned off.

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. The Identity Server Logging API extend the core logging API in the Java™ 2 Standard Edition Development Kit (JDK) 1.4. Only the `Logger` and `LogRecord` classes are enhanced. They are contained in the package `com.sun.identity.log`.

TIP An overview of the JDK 1.4 logging function can be found at `http://java.sun.com/j2se/1.4.1/docs/guide/util/logging/overview.html` The Javadocs for the JDK 1.4 logging API themselves can be found at `http://java.sun.com/j2se/1.4.1/docs/api/java/util/logging/package-summary.html.`

Setting Environment Variables

The following shared library environment variables need to be set in the executable for an application that is using the Logging Service.

- `-D"java.util.logging.manager=com.sun.identity.log.LogManager"`
- `-D"java.util.logging.config.class=com.sun.identity.log.s1is.LogConfigReader"`

NOTE See "Enabling Remote Logging" on page 413 for instructions.

If SSL is enabled for Identity Server, the following parameter also needs to be added:

- `-D"java.protocol.handler.pkgs=com.iplanet.services.comm"`

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 record (LogRecord) 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.
    - Logger.log(logRecord, String credential) had been added to call the authorization hook. The credential is accepted as a ssoToken string. The default authorization hook checks validitity of the ssoToken. Data is not logged at all if this check fails.
    - Logger.log(logRecord) simply calls Logger(logRecord, String cred) with credential value of null. And thus the default authorization check does not allow logging when an application uses this interface.

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. The ssoToken is passed to the logRecord constructor and used to populate the log fields discussed in "Recorded Events" on page 405. The session token passed during the logger.log(logRecord, ssoToken) log request is used to authorize the user. The user can only log with a valid ssoToken.

Adding Log Data

The following sections illustrate ways to use the Logging API for adding log file information.

Adding ModuleName Data

The ModuleName value can be added to a log file using the logRecord.addLogInfo(key, value) API. If a module name is not added, the name of the log will be used to populate this field. For example, authentication information is logged in the amAuthentication.access file using an internal session token ("dsameuser" ssoToken). If user Joe123 attempts to authenticate, the LoginID will be Joe123, and the LoggedBy user will be dsameuser.

NOTE The LoggedBy entry is populated from the SSOToken passed during `logger.log(logRecord, ssoToken)` call.

If the authentication module information (such as LDAP, Membership, etc.) is not added by the APIs, `amAuthentication.access` will be the value of the `ModuleName` field.

Adding Log Level Data

A LogLevel is passed in the `LogRecord` constructor using the following code:

```
LogRecord(Level level, String msg)
```

While using the logging APIs, any JDK 1.4 defined log levels can be passed.

Caching Log Records

Identity Server supports log record caching both locally and remotely based on the configurable buffering properties discussed in “Remote Logging” on page 413. Caching is supported for either type of log file although not when secure logging is enabled.

Flushing Log Records

Identity Server provides `Logger.flush()` to expunge all the cached log records.

Sample Logging Code

Code Example 10-3 provides sample code to illustrate one way in which the logging API can be used to write Identity Server records.

Code Example 10-3 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 SPI are Java packages that can be used to develop plug-ins for customized features. The SPI are organized in the `com.sun.identity.log.spi` package. More information on the SPI can be found in the Javadocs located at *IdentityServer_base*/`SUNWam/docs`.

Log Verifier Plugin

If secure logging is enabled, the log files are verified periodically to detect any attempt of tampering. If tampering is detected, the action taken can be customized by following the steps below.

1. Implement the `com.sun.identity.log.spi.IVerifierOutput` interface with 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 `/etc/opt/SUNWam/config/xml/amLogging.xml` file with the name of the new class.

Log Authorization Plugin

The Logging Service allows a class to be plugged in that will determine whether a `LogRecord` is logged or discarded based on the authorization of the owner of the session token performing the event.

NOTE The `IAuthorizer` interface accepts a SSOToken and the logrecord being written.

There are several ways to accomplish this. For example:

1. Get the applicable role or DN of the user from the `SSOToken` and check it against a pre-configured (or hardcoded) list of roles/users that are allowed access. The administrator must configure a role and assign all policy agents and entities (for example, applications) that can possibly log to Identity Server to this role.

2. Instantiate a `PolicyEvaluator` and call `PolicyEvaluator.isAllowed(ssotoken, logname);`. This entails defining a policy XML to model log access and registering it with Identity Server.

In general:

1. Implement the `com.sun.identity.log.spi.IAuthorizer` interface with 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 `/etc/opt/SUNWam/config/xml/amLogging.xml` file with the name of the new class.

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

Debug Files

The debug files are not a feature of, nor generated by, the Logging Service. They are written using different APIs which are independent of the logging APIs. 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 *IdentityServer_base*`/SUNWam/lib/` directory. For more information on the debug properties, see Appendix A, “AMConfig.properties File.”

Debug Levels

There are several levels of information that can be recorded to the debug files. The debug level is set using the `com.iplanet.services.debug.level` property in `AMConfig.properties`.

1. Off—No debug information is recorded.
2. Error—This level is used for production. During production, there should be no errors in the debug files.
3. Warning—Currently, using this level is not recommended.

4. Message—This level alerts to possible issues using code tracing. Most Identity Server modules use this level to send debug messages.

CAUTION Warning and Message levels should not be used in production. They cause severe performance degradation and an abundance of debug messages.

Debug Output Files

A debug file does not get created until a module writes to it. Therefore, in the default `error` mode no debug files may be generated. The debug files that get created on a basic login with the debug level set to `message` include:

- amAuth
- amAuthConfig
- amAuthContextLocal
- amAuthLDAP
- amCallback
- amClientDetection
- amConsole
- amFileLookup
- amJSS
- amLog
- amLoginModule
- amLoginViewBean
- amNaming
- amProfile
- amSDK
- amSSOProvider
- amSessionEncodeURL
- amThreadManager

The most often used files are the amSDK, amProfile and all files pertaining to authentication. The information captured includes the date, time and message type (Error, Warning, Message).

Using Debug Files

The debug level, by default, is set to `error`. The debug files might be useful to an administrator when they are:

- Writing a custom authentication module.
- Writing a custom application using the Identity Server SDKs. The `amProfile` and `amSDK` debug files capture this information.
- Troubleshooting access permissions while using the console or SDK. The `amProfile` and `amSDK` debug files also capture this information.
- Troubleshooting SSL.
- Troubleshooting the LDAP authentication module. The `amAuthLDAP` debug file captures this information.

The debug files should go hand in hand with any troubleshooting guide we might have in the future. For example when SSL fails, someone might turn on debug to message and look in the amJSS debug file for any specific cert errors.

Multiple Identity Server Instances And Debug Files

Identity Server contains the `ammultiserverinstall` script that can be used to configure numerous instances of the server. If the multiple server instances are configured to use different debug directories, each individual instance has to have both read and write permissions to the debug directories. More information on the `ammultiserverinstall` script can be found in the *Sun ONE Identity Server Administration Guide.*

Chapter 11

Client Detection Service

The Sun™ ONE Identity Server Authentication Service has the capability of being accessed from many client 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 Service is used for this purpose. This chapter offers information on the service, and how it can be used to recognize the client type. It contains the following sections:

- Overview
- Client Data
- Client Detection API

Overview

The Identity Server Authentication Service has the capability to process requests from multiple browser type clients. Thus, the service can be used to authenticate users attempting to access applications based in HTML, WML or other protocols.

CAUTION The Identity Server console though can not be accessed from any client type except HTML.

The client detection API can be used to determine the protocol of the requesting client browser and retrieve the correctly formatted pages for the particular client type.

NOTE Out of the box, Identity Server only defines client data for supported HTML client browsers. A list of supported browsers can be found in Chapter 1, “Introduction” under the section “Client Browser Support” on page 42.

Client Detection Process

Since any user requesting access to Identity Server must first be successfully authenticated, browser type client detection is accomplished within the Authentication Service. When a client's request is passed to Identity Server, it is directed to the Authentication Service. Within this service, the first step in user validation is to identify the browser type using the `User-Agent` field stored in the HTTP request.

NOTE The `User-Agent` field contains *product tokens* which contains information about the browser type client originating the HTTP request. The tokens are a standard used to allow communicating applications to identify themselves. The format is `software/version library/version`.

The `User-Agent` information is then matched to browser type data defined and stored in the `amClientData.xml` file.

CAUTION User-Agent information is defined in amClientData.xml but this information is stored in Directory Server under Client Detection Service.

Based on this Client Data, correctly formatted browser pages are sent back to the client for authentication (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. (If there is no matching client data, the default type is returned.)

NOTE The userAgent must be a part of the client data configured for all browser type clients. It can be a partial string or the exact product token.

Enabling Client Detection

By default, the client detection capability is disabled; this then assumes the client to be of the `genericHTML` type (i.e. Identity Server will be accessed from a HTML browser). The preferred way to enable the Client Detection Service is to use the Identity Server console and select the option in the Client Detection Service itself. For more information, see the *Sun ONE Identity Server Administration Guide*. To enable client detection using the amClientDetection.xml, the iplanet-am-client-detection-enabled attribute must be set to true. amClientDetection.xml must then be deleted from Directory Server and reloaded using amAdmin. The following procedure illustrates the complete enabling process.

1. Import client data XML file using the amadmin command
   */IdentityServer_base/*SUNWam/bin/amadmin -u *amadmin_DN* -w *amadmin_password* -t *name_of_XML_file*

   This step is only necessary if the client data is not already defined in amClientData.xml. The XML file is based on the "The sms.dtd Structure" on page 255 of Chapter 6, "Service Management."

2. Restart Identity Server.

3. Login to Identity Server console.

4. Go to Service Configuration and click the ClientDetectionproperties.

5. Enable Client Detection.

6. Make sure the imported data can be viewed with Identity Server console.

   Click on the Edit button next to the Client Data attribute.

7. Create a directory for new client type and add customized JSPs.

   Create a new directory in */IdentityServer_base/*SUNWam/web-apps/services/config/auth/default/ and add JSPs for the new client type. Code Example 11-1 on page 425 is a login page written for a WML browser.

Code Example 11-1 Login.jsp Written In WML

```
<?xml version="1.0"?>

<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">

<!-- Copyright Sun Microsystems, Inc.  All Rights Reserved -->

<wml>
<head>
<meta http-equiv="Cache-Control" content="max-age=0"/>
</head>

 <card id="authmenu" title="Username">
<do type="accept" label="Enter">

<go method="get" href="/wireless">
<postfield name="TOKEN0" value="$username"/>
```

Code Example 11-1 Login.jsp Written In WML

```
<postfield name="TOKEN1" value="$password"/>
</go>
</do>
<p>
Enter username:
<input type="text" name="password"/>
</p>
<p>
Enter password:
<input type="text" name="username"/>
</p>
</card>
</wml>
```

Client Data

In order to detect client types, Identity Server needs to recognize their identifying characteristics. These characteristics identify the features of all supported types and are defined in the `amClientData.xml` service file. The full scope of client data available is defined as a schema in `amClientData.xml`. The configured Identity Server client data available for HTML-based browsers is defined as sub-configurations of the overall schema: genericHTML and its parent HTML.

NOTE Parent profiles (or *styles*, as they are referred to in the Identity Server console) are defined with properties that are common to its configured child devices. This allows for the dynamic inheritance of the parent properties to the child devices making the device profiles easier to mange.

HTML

`HTML` is a base style containing properties common to HTML-based browsers. It might have several branches including web-based HTML (or genericHTML), cHTML (Compact HTML) and others. All configured devices for this style could inherit these properties which include:

- `parentId`—identifies the base profile. The default value is `HTML`.
- `clientType`—an arbitrary string which uniquely identifies the client. The default value is `HTML`.
- `filePath`—is used to locate the client type files (templates and JSP files). The default value is `html`.

- contentType—defines the content type of the HTTP request. The default value is text/html.
- genericHTML—defines a client that will be treated as HTML. The default value is true.

NOTE This attribute does not refer to the similarly named genericHTML style.

- cookieSupport—defines whether cookies are supported by the client browser. The default value is true which sets a cookie in the response header. The other two values could be False which sets the cookie in the URL and Null which allows for dynamic cookie detection. In the first request, the cookie is set in both the response header and the URL; the actual mode is then detected and set from the subsequent request.

NOTE Although the Client Detection Service supports a cookieless mode, Identity Server console does not. Therefore, enabling this function will not allow login to the console. This feature is provided for wireless applications and others that will support it.

- CcppAccept-Charset—defines the character encoding used by Identity Server to send a response to the browser. The default value is UTF-8.

genericHTML

genericHTML is a configured device that inherits properties from the HTML style as well as defining its own properties. It refers to a HTML browser (Netscape Navigator™, Microsoft® Internet Explorer, or Mozilla™). Its properties include:

- parentId—identifies the base profile for the configured device. The default value is HTML.
- clientType—an arbitrary string which uniquely identifies the client. The default value is genericHTML.
- userAgent—a search filter used to compare/match the user agent defined in the HTTP header. The default value is Mozilla/4.0.

- CcppAccept-Charset—defines the character encoding set supported by the browser. The default values are UTF-8;ISO-8859-1;ISO-8859-2;ISO-8859-3;ISO-8859-4;ISO-8859-5;ISO-8859-6;ISO-8859-7;ISO-8859-8;ISO-8859-9;ISO-8859-10;ISO-8859-14;ISO-8859-15;Shift_JIS;EUC-JP;ISO-2022-JP;GB18030;GB2312;BIG5;EUC-KR;ISO-2022-KR;TIS-620;KOI8-R.

NOTE 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.

Client Detection API

Identity Server is packaged with a Java API which can implement the client detection functionality. The client detection API are in a package called com.iplanet.services.cdm. This package provides the interfaces and classes needed to retrieve client properties. The client detection procedure would include defining the client type characteristics (as stated in “Client Data” on page 426) as well as implementing the client detection API within the external application.

The client detection capability is provided by ClientDetectionInterface, a pluggable interface (not an API invoked by a regular application). It provides a getClientType method. The getClientType method extracts the client data from the browser’s incoming HttpRequest, matches the user agent information and returns the ClientType as a string. Upon successful authentication, the client type is added to the user’s session token. The ClientDetectionException handles any error conditions.

Chapter 12

Identity Server Utilities

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
- Password API Plug-Ins

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 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 Identity Server data stored in Directory Server. This includes identity data as well as Identity Server customization data.

Backup Script

Following is the script used for backing up data. The utility is named `am2bak` and can be found in the *IdentityServer_base*/`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 *IdentityServer_base*/`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:
     - `/etc/opt/SUNWam/config`
     - *IdentityServer_base*`/SUNWam/locale/`
     - *IdentityServer_base*`/SUNWam/servers/httpacl`
     - *IdentityServer_base*`/SUNWam/lib/*.properties` (Java property files)
     - *IdentityServer_base*`/SUNWam/bin/amserver.`*instance-name*
     - *IdentityServer_base*`/SUNWam/web-apps/services/WEB-INF/config`
     - *IdentityServer_base*`/SUNWam/web-apps/services/config`
     - *IdentityServer_base*`/SUNWam/web-apps/applications/WEB-INF/classes`
     - *IdentityServer_base*`/SUNWam/web-apps/applications/console`
     - `/etc/rc3.d/K03amserver.`*all_instances*
     - `/etc/rc3.d/S86amserver.`*all_instances*
     - *DirectoryServer_install_directory*`/slapd-`*host*`/config/schema/`
     - *DirectoryServer_install_directory*`/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:
     - *DirectoryServer_install_directory*`/alias`

   The script will backup the following Microsoft® Windows 2000 operating system files:

   - Configuration and Customization Files:

  - *IdentityServer_base*/`web-apps/services/WEB-INF/config/*`
  - *IdentityServer_base*/`locale/*`
  - *IdentityServer_base*/`web-apps/applications/WEB-INF/classes/*.properties` (Java property files)
  - *IdentityServer_base*/`servers/https-`*host*/`config/jvm12.conf`
  - *IdentityServer_base*/`servers/https-`*host*/`config/magnus.conf`
  - *IdentityServer_base*/`servers/https-`*host*/`config/obj.conf`
  - *DirectoryServer_install_directory*/`slapd-`*host*/`config/schema/*.ldif`
  - *DirectoryServer_install_directory*/`slapd-`*host*/`config/dse.ldif`
- Log And Debug Files:
  - `/var/opt/logs` (Identity Server log files)
  - `/var/opt/debug` (Identity Server debug files)
- Certificates:
  - *DirectoryServer_install_directory*/`alias`

The script will backup the following files for Sun ONE Application Server:

- *Application Server IdentityServer_base*/`config/server.xml`
- *Application Server IdentityServer_base*/`config/server.policy`
- *Application Server IdentityServer_base*/`applications/j2ee-modules/`
- *Application Server IdentityServer_base*/`applications/j2ee-modules/`

The script will backup the following files for BEA WebLogic:

- *weblogic-install-dir* / config / *weblogic_domain*> / startWebLogic.sh
- *weblogic-install-dir* / config / *weblogic_domain*/`startManagedWebLogic.sh`
- *weblogic-install-dir* / config / *weblogic_domain*/`.java.login.config`
- *weblogic-install-dir* / config / *weblogic_domain*/`config.xml`
- *weblogic-install-dir* / config / *weblogic_domain*/`applications/`

The script will backup the following files for IBM WebSphere:

- *Websphere-install-dir*/`Websphere/Appserver/bin/admin.config`
- *Websphere-install-dir*/`Websphere/Appserver/bin/setupCmdLine.sh`

- *Websphere-install-dir*`/Websphere/Appserver/installedApps/`

The script will backup the following files for Sun ONE Web Server:

- *IdentityServer_base*`/SUNWam/servers/httpacl`
- *IdentityServer_base*`/SUNWam/lib/*.properties`
- *IdentityServer_base*`/SUNWam/bin/amserver.`*instance_name*
- *IdentityServer_base*`/SUNWam/web-apps/services/WEB-INF/config`
- *IdentityServer_base*`/SUNWam/web-apps/services/config`
- *IdentityServer_base*`/SUNWam/web-apps/applications/WEB-INF/classes`
- *IdentityServer_base*`/SUNWam/web-apps/applications/console`

The script will backup the following common files for all containers:

- `/etc/opt/SUNWam/config`
- *IdentityServer_base*`/SUNWam/locale`
- *IdentityServer_base*`/SUNWam/lib/.properties`
- *IdentityServer_base*`/SUNWam/bin/amserver`.*instance-name*
- `/etc/rc3.d/K03amserver.`*all_instances*
- `/etc/rc3.d/S86amserver.`*all_instances*
- *DirectoryServer_install_directory* / slapd-*host*`/config/schema`
- *DirectoryServer_install_directory* / slapd-*host*`/config/dse.ldif`

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 *IdentityServer_base*`/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 *IdentityServer_base*/`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 *IdentityServer_base*/`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. Following is a summary of the utility API and their functions.

AdminUtils

This class contains the methods used to retrieve TopLevelAdmin DN and password. The information comes from the server configuration file, `serverconfig.xml`, located in */IdentityServer_base/*`SUNWam/config/ums`.

AMClientDetector

The `AMClientDetector` interface executes the Client Detection Class configured in the Client Detection Service to get the client type.

AMPasswordUtil

The `AMPasswordUtil` interface has two purposes:

1. Encrypting and decrypting any string.
2. Encrypting and decrypting special user passwords such as the password for dsameuser or proxy user.

NOTE Any remote application using this utility should have the value of the AMConfig property `am.encryption.pwd` copied to a properties file on the client side. This value is generated at installation time and stored in /*IdentityServer_base*/`SUNWam/lib/AMConfig.properties`. More information on this property can be found in the Encryption section of the Appendix A, “AMConfig.properties File.”

Debug

Debug allows an interface to file debug and exception information in a uniform format. It supports different levels of 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 can hurt performance. 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 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 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.

Password API Plug-Ins

The Password API plug-ins can be used to integrate password functions into applications. They can be used to generate new passwords as well as notify users when their password has been changed. These interfaces are PasswordGenerator and NotifyPassword, respectively. They can be found in the com.sun.identity.password.plugins package.

NOTE The Identity Server Javadocs can be accessed from any browser by copying the complete *IdentityServer_base*/SUNWam/docs/ directory into the *IdentityServer_base*/SUNWam/public_html directory and pointing the browser to http://*identity_server_host.domain_name*:*port*/docs/index.html.

There are samples (which include sample code) for these API that can be accessed from the Identity Server installation. They are located in *IdentityServer_base*/SUNWam/samples/console. They include:

Notify Password Sample

This sample details how to build a plug-in in which an administrator can define their own method of notification when a user has reset a password. Instructions for this sample are in the Readme.txt or Readme.html file located in *IdentityServer_base*/SUNWam/samples/console/NotifyPassword.

Password Generator Sample

This sample details how to build a plug-in in which an administrator can define their own method of randon password generation when a user’s password is reset using the Password Reset Service. Instructions for this sample are in the Readme.txt or Readme.html file located in *IdentityServer_base*/SUNWam/samples/console/PasswordGenerator.

Appendix A

AMConfig.properties File

AMConfig.properties is the resource configuration file for the Sun™ One Identity Server. It provides instructions for the Identity Server deployment. This chapter explains the attributes of AMConfig.properties. It contains the following sections:

- Overview
- Deployment Properties
- Configuration Properties
- Read-Only Properties

Overview

Identity Server is configured by placing application properties in plain text configuration files. These configuration files contain one proeprty per line and each has a corresponding value. Properties and their values are case-sensitive. Indentation of the properties is consistent throughout the file. Lines which begin with the characters "/*" are comments, and ignored by the application. Comments are completed with a last line that contains the closing characters "*/". The main configuration file for Identity Server is AMConfig.properties located in *IdentityServer_base*/SUNWam/lib. The following sections describe the properties and default values of AMConfig.properties.

NOTE	The Identity Server must be restarted for any modification in AMConfig.properties to take effect.

Deployment Properties

Following are the deployment-specific attributes configured in `AMConfig.properties`.

Identity Server

This section describe properties that define the Identity Server application.

Installation

These properties are defined during installation.

- `com.iplanet.am.server.host=`*identity_server_host.domain_name*

  The value of this property is the DNS domain name of the machine on which the Identity Server is located.

- `com.iplanet.am.server.port=58080`

  The value of this property is the port number used by the Identity Server. The default is 58080.

- `com.iplanet.am.jdk.path=/`*IdentityServer_base*`/SUNWam/java`

  The value of this property is the path to the JDK used by the Identity Server.

- `com.sun.identity.authentication.super.user=uid=amAdmin,ou=People, dc=`*top_level_org*`,dc=com`

  This property identifies the full LDAP DN of the super user configured during installation of Identity Server; it is `amadmin` by default. This user must always log in using LDAP authentication as they will always be authenticated against the Directory Server. The UID alone is generally used to login but the full DN as defined in this property can also be used.

Console

These properties are specific to the Identity Server console.

- `com.iplanet.am.console.host=`*identity_server_host.domain_name*

  The value of this property is the DNS domain name of the machine on which the Identity Server console is located.

- `com.iplanet.am.console.protocol=http`

The value of this property is the protocol used to communicate with the Identity Server. The default is `http`.

- `com.iplanet.am.console.port=58080`

  The value of this property is the port number of the machine on which the Identity Server console is located. The default is 58080.

The following directives can be added to the `AMConfig.properties` file to add their respective functionality to the Identity Server console.

- `com.iplanet.am.console.display.off=orgs,users,groups`

  If specified, Identity Server will not perform the initial search for a specified identity object that is done in the Navigation frame when the view menu is changed. For example, after a successful login, the default console view is the organization view. When the view is changed to Users, the Navigation frame is redrawn to display all users; a search is performed to obtain this information. With a large number of users, disabling this search can drastically reduce the time it takes to load the Identity Server console. A filter can then be used to find the desired users. This option is available for any of the view menu types. To disable the search, add any of the following values: `orgs`, `orgUnits`, `users`, `policies`, `groups`, `roles`, `groupContainers`, and `peopleContainers`. If more than one value, they are comma-separated.

CAUTION The service attribute in the Identity Server console that corresponds to this property is Display Options, an organization attribute in the Administration Service. This console option takes precedence over any value defined in `com.iplanet.am.console.display.off`. If configuring this property in `AMConfig.properties`, do not configure it using the console (or vica versa).

- `com.iplanet.am.console.set.cn=true`

  If specified, the user common name (`cn`) will not be displayed in the Create User screen but it will be generated based on information entered in the First Name (`givenname`), Initial and Last Name (`sn`) fields of the User profile page and displayed as a read-only value on screen.

Cookies

These properties are specific to Identity Server cookies.

- `com.iplanet.am.cookie.name=iPlanetDirectoryPro`

The value of this property is the name of the cookie. In an Identity Server deployment with more than one instance, it is recommended that the value of this property for one of the instances is changed.

NOTE The cookie name defined as `com.iplanet.am.cookie.name` is the Identity Server cookie and needs to be defined in a sticky load balancing situation. Do not use the HTTP session cookie as in some cases it is not retained.

- `com.iplanet.am.pcookie.name=DProPCookie`

  The value of this property is the name of the persistent cookie if that function is enabled.

- com.iplanet.am.cookie.secure=false

  This property allows the Identity Server cookie to be set in a secure mode in which the browser will only return the cookie when a secure protocol like HTTP(s) is used.

- com.iplanet.am.cookie.encode=COOKIE_ENCODE

  This property allows Identity Server to *URLencode* the cookie value which converts characters to ones that are understandable by HTTP.

Miscellaneous

This section is a catch-all for some miscellaneous and self-explanatory values.

- `com.iplanet.am.daemons=unix`
- `com.iplanet.am.locale=en_US`
- `com.iplanet.am.logstatus=ACTIVE`
- `com.iplanet.am.version=6.1`
- `com.iplanet.services.configpath=/etc/opt/SUNWam/config/ums`

  The value of this property is the path to the `serverconfig.xml` file. This file is discussed in Appendix B, “serverconfig.xml File.”

Directory Server

This section describe the properties for the Directory Server data store.

Installation

These properties define the Directory Server to which the Identity Server points.

- `com.iplanet.am.directory.host=`*identity_server_host.domain_name*

  The value of this property is the DNS domain name of the machine on which the Directory Server is located.

- `com.iplanet.am.directory.port=389`

  The value of this property is the port number of the machine on which the Directory Server is located. The default is `389`.

- `com.iplanet.am.server.protocol=http`

  The value of this property is the protocol used to communicate with the machine on which the Directory Server is located.

Directory Server Tree

The values of these properties are the top-level organization of the Directory Server tree defined during the installation process.

- `com.iplanet.am.defaultOrg=dc=`*top_level_org*`,dc=com`
- `com.iplanet.am.rootsuffix=dc=`*top_level_org*`,dc=com`
- `com.iplanet.am.domaincomponent=dc=`*top_level_org*`,dc=com`

Configuration Properties

There are a number of services configured in `AMConfig.properties` that can not be configured using the Identity Server console. These back-end services, and several attributes for other services, are defined in this section.

Debug Service

The Debug Service logs developer information in the case of application errors. (The Logging Service writes logs to be monitored by the application administrator.) More information on the Debug Service can be found in “Debug Files” on page 419 of Chapter 10, “Auditing Features.”

- `com.iplanet.services.debug.level=error`

The possible values for this property are: `off` | `error` | `warning` | `message`. They indicate the level of information recorded in the debug files.

- `com.iplanet.services.debug.directory=/var/opt/SUNWam/debug`

  The value of this property is the output directory for the debug information. This directory should be writable by the server process.

NOTE In defining values for the Debug Service, remember that trailing spaces are significant. Also, on a Microsoft® Windows® system, use forward slashes “/” to separate directories.Finally, spaces in the file name are allowed only on a Windows system.

Stats Service

The following properties are used to configure the Stats Service for recording service statistics. This service is used by the Identity Server SDK and the Session Service. Code Example A-1 is a portion of the stats file to illustrate the information that is recorded. The file is named `amSDKStats` by default.

Code Example A-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.iplanet.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.iplanet.services.stats.directory=/var/opt/SUNWam/debug`

  This property specifies the output directory for the statistics files. By default, it is the same as the debug directory.

- `com.iplanet.services.stats.state=off`

Possible values for this directive are: `off | file | console`. `file` will write to a file named `amSDKStats` under the directory specified in the `com.iplanet.services.stats.directory` property and `console` will write into the deployment container log files.

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.

Notification Service

The Notification Service allows Identity Server to send notifications to registered applications when an event has occurred (session destroyed, session timeout, etc.). This service also allows the single sign-on cache to stay up to date. The notification is basically a HTTP post message containing the *component notification* in its body.

- `com.iplanet.am.notification.url=` `http://`*identity_server_host.domain_name*`:`*port*`/amserver/notificationservice`

  The value of this property is the URI of the Notification Service.

When a notification task comes in, it is processed in the task queue. If it reaches the maximum length, further incoming requests will be rejected along with a ThreadPoolException, until the queue has vacancy

- `com.iplanet.am.notification.threadpool.size=10`

  This parameter is used to define the session thread pool for notification handling. It specifies the size of the pool as the total number of threads allowed.

- `com.iplanet.am.notification.threadpool.threshold= 100`

  This parameter specifies the maximum size of the task queue in the thread pool. A task is queued when no thread is available. If the number of unprocessed tasks reaches the value specified, no additional notification tasks will be accepted until there are vacancies. This value is dependent on the system memory resource; each task takes about 3k.

SDK Caching

The caching function in Identity Server is memory-based therefore when an identity-related object is created, deleted or modified, the cache is cleaned up. Each SDK cache entry stores a set of attributes and values of `AMObject` for a user. Because the size of each object is dependent upon the number of attributes it has, modifying these properties will affect the performance of Identity Server.

- `com.iplanet.am.sdk.cache.maxSize=10000`

  This property configures the size of the cache when caching is enabled. The value 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 property specifies the maximum number of concurrent sessions. Logging in when the maximum sessions has been met would send a Maximum Sessions error.

Online Certificate Status Protocol (OCSP)

OCSP is a protocol that specifies the syntax for communication between a server which holds certificate status and a client which is informed of said status.When a user attempts to access a server, OCSP sends a request for certificate status information and receives back a response of *current*, *expired* or *unknown*. If these properties are set, the certificate in question must be in the deployment container's certificate database. If the OCSP URL is set, the OCSP responder nickname must also be set or both will be ignored. If neither is set, the OCSP responder URL presented in the user's certificate will be used. If there is none in the user's certificate, no OCSP validation will be performed.

- `com.sun.identity.authentication.ocsp.responder.url`

  The value of this directive is the global OCSP responder URL for this instance of Identity Server, i.e. `http://ocsp.example.com/ocsp`.

- `com.sun.identity.authentication.ocsp.responder.nickname`

  The OCSP responder nickname refers to the Certificate Authority for the responder. This nickname is used to reference the Certificate Authority in the certificate itself.

Identity Object Processing

This property has a value equal to the implementation class of the module used for processing user creates, deletes, and modifies.

- `com.iplanet.am.sdk.userEntryProcessingImpl=`

Security

This property is used to enable Java security permissions. This permission is used to protect the Identity Server resources which should only be accessed by trusted resources. This permission is used to protect the admin DN and password as well as access to the encryption and decryption methods used to encrypt passwords. The default value is false. If enabled, modifications must be made to the deployed web container's Java policy file. This should be done as detailed in Code Example A-2.

Code Example A-2 Changes To Java Policy File

```
grant codeBase "file:{directory where jars are located}/-" {
      com.sun.identity.security.ISSecurityPermission "access",
                    "adminpassword,crypt";
 };
```

- `com.sun.identity.security.checkcaller=false`

SSL

This property is used to enable Secure Socket Layers (SSL). The default is `false`.

- `com.iplanet.am.directory.ssl.enabled=false`

Certificate Database

These properties are used by the command line utilities and SDK as well as the LDAP and Certificate-based authentication modules when initiating SSL connections to the Directory Server. It is also used when opening HTTP(S) connections from within the servlet container in the deployment container.

- `com.iplanet.am.admin.cli.certdb.dir=/`*IdentityServer_base*`/SUNWam/servers/alias`

  The value of this property is the name of the path to the certificate database.

- `com.iplanet.am.admin.cli.certdb.prefix=https-`*identity_server_host.domain_name*`-`*identity_server_host*`-`

  The value of this property is the certificate database prefix.

- `com.iplanet.am.admin.cli.certdb.passfile=/`*IdentityServer_base*`/SUNWam/config/.wtpass`

  The value of this property is the name of the file that contains the password for the certificate database.

NOTE When installing Identity Server, these values do not point to a configured certificate database. After creating the certificate database, these values should be reset to point to the Application Server as follows:

- `com.iplanet.am.admin.cli.certdb.dir=`/install_directory/SUNWappserver7/*domain*/*server_instance*/config
- `com.iplanet.am.admin.cli.certdb.prefix=`
- `com.iplanet.am.admin.cli.certdb.passfile=`/install_directory/SUNWappserver7/*domain*/*server_instance*/config`/.wtpass`

Identity Server should be restarted after the modifications.

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, 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.iplanet.am.replica.num.retries=0`

  This 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 property.

- `com.iplanet.am.replica.delay.between.retries=1000`

  This property specifies the delay time (in milliseconds) between the number of retries defined in the key above.

Event And LDAP Connection

These sets of 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 it is accomplished via a load balancer as not all load balancers return the same error codes.

Event Connection

- `com.iplanet.am.event.connection.num.retries=3`

  This value specifies the number of time to retry an event connection.

- `com.iplanet.am.event.connection.delay.between.retries=3000`

  This value specifies the delay time (in milliseconds) between the number of retries defined in the key above.

- `com.iplanet.am.event.connection.ldap.error.codes.retries=80,81,91`

  This key 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 an LDAP connection for the add, delete modify, read and search methods.

- `com.iplanet.am.ldap.connection.num.retries=3`

  This value specifies the number of time to retry an LDAP connection.

- `com.iplanet.am.ldap.connection.delay.between.retries=1000`

  This value specifies the delay time (in milliseconds) between the number of retries defined in the key above.

- `com.iplanet.am.ldap.connection.ldap.error.codes.retries=80,81,91`

  This key specifies the `LDAPException` errors for which the retries will occur. The value is any valid LDAP error code.

SAML

These properties identify SAML-related configurations including properties relating to the Identity Server keystore file.

- `com.sun.identity.saml.removeassertion=false`

  This property indicates if assertions associated with artifacts and now de-referenced should be removed from the cache. If set to `true`, assertions will be removed. Otherwise, the assertion will be kept in memory and removed only when it is expired itself.

Keystore Properties

Each Identity Server has a keystore file used to store the certificates used for XML signing and verification. A stored certificate might include a partner site's certificate and the public key used by Identity Server to verify SAML responses and assertions from the partner. The keystore also holds the Identity Server certificate and the private key it uses to sign assertions. For more information on generating the keystore, certificate aliases and other functions, read about the `keytool`, a key and certificate management utility, in the `Readme.html` and `keystore.html` files located in the *IdentityServer_base*`/SUNWam/samples/saml/xmlsig` directory.

- `com.sun.identity.saml.xmlsig.keystore=/`*IdentityServer_base*`/SUNWam/lib/keystore.jks`

  The value of this property is the name and location of the keystore file. Although, upon installing Identity Server, this property has a default value, the file itself is not initally generated and the name and location of the file can be changed.

- `com.sun.identity.saml.xmlsig.storepass=/`*IdentityServer_base*`/SUNWam/config/.storepass`

  The value of this property is the location of the password to the keystore.

- `com.sun.identity.saml.xmlsig.keypass=/`*IdentityServer_base*`/SUNWam/config/.keypass`

The value of this property is the location of the password to the private key which is used to sign the XML document.

- `com.sun.identity.saml.xmlsig.certalias=test`

  All entries (keys and trusted certificate entries) in the keystore file are accessed using unique aliases. The value of this property is the certificate alias of the Identity Server certificate which links to the private key used for signing assertions.

Miscellaneous Services

The following directives define the URIs for miscellaneous services.

- `com.iplanet.am.profile.host=`*identity_server_host.domain_name*

  The value of this property is the DNS domain name of the machine on which the Identity Server (and thus the Profile Service) is located.

- `com.iplanet.am.profile.port=58080`

  The value of this property is the port number used by the Identity Server (and thus the Profile Service). The default is 58080.

- `com.iplanet.am.naming.url=http://`*identity_server_host.domain_name*`:`*port*`/amserver/namingservice`

  The value of this property represents the URL where a request by the Identity Server or a remote single sign-on client will be sent to retreive the URLs of Identity Server internal services. This is the URI for the Naming Service.

Read-Only Properties

The following properties are read-only and should not be modified. Any changes to these directives may render the Identity Server unusable.

Installation

These properties identify values defined during the installation process.

- `com.iplanet.am.installdir=/`*IdentityServer_base*`/SUNWam`

  This value is the base directory for the application.

- `com.iplanet.am.install.basedir=/`*IdentityServer_base*`/SUNWam/web-apps/services/WEB-INF`

  This value is the base directory for the services.

- `com.iplanet.am.iASConfig=false`

  This property defines whether the Sun ONE Identity Server is running on the Sun ONE Application Server. The value is set during installation and must not be changed.

- `com.iplanet.am.console.remote=false`

  This property defines whether the console is installed on a remote or local machine. It is used by the Authentication Service and the console.

Deployment

These properties are used to identify the URIs for specific services and agents.

- `com.iplanet.am.services.deploymentDescriptor=/amserver`
- `com.iplanet.am.console.deploymentDescriptor=/amconsole`
- `com.iplanet.am.policy.agents.url.deploymentDescriptor=AGENT_DEPLOY_URI`

  This last property contains the name of the deployment container. Possible values here are BEA6.1, IBM 4.0.5, S1AS7.0, or WS.

- `com.sun.identity.webcontainer=WEB_CONTAINER`

NOTE Although the servlet and JSPs are deployment container independent, servlet 2.3 API `request.setCharacterEncoding()` (used to correctly decode incoming non-English characters) will not work if Identity Server is deployed on Sun ONE Web Server 6.0 or Sun ONE Application Server 7.0.

Shared Secret

This property is the shared secret for the Authentication Service.

- `com.iplanet.am.service.secret=AQIC5wM2LY4SfczLlj6134qMTx0nkE5XiFMg`

Session Properties

These properties are configurations for the Session Service.

- `com.iplanet.am.session.failover.enabled=false`

  This property is used to enable or disable the session failover feature. The following properties are used when this property is set to `true`.

  - `com.iplanet.am.localserver.protocol=http`
  - `com.iplanet.am.localserver.host=`*identity_server_host.domain_name*
  - `com.iplanet.am.localserver.port=58080`

- `com.iplanet.am.session.httpSession.enabled=true`

  When this property is set to true, an HttpSession will be created for the authenticated user in addition to an Identity Server session. This property is also related to session failover.

- `com.iplanet.am.session.invalidsessionmaxtime=3`

  This property disables a session if it is created and the user does not login before the time defined. The value is in minutes (for example, 3 minutes is the default value).

NOTE This value should always be greater than the time-out value in your authentication module properties file.

- `com.iplanet.am.session.client.polling.enable=false`

  If set to `true`, the client cache will invalidate itself after the amount of time defined in the next property, forcing data to reload.

- `com.iplanet.am.session.client.polling.period=180`

  This property defines the default polling period as 180 seconds.

- `com.sun.am.session.logging.enableHostLookUp=false`

  This property allows the session server (i.e. Identity Server) to look for the IP address from the host property and log it. If set to `true`, a reverse DNS lookup will be used to obtain the Domain Name from the IP address for logging purposes. If `false`, the IP address will be used thus, increasing performance.

- `com.iplanet.am.session.purgedelay=60`

This property defines the purge delay period in minutes. After a session times out, this is the extended time period for which the token will reside in the Session Service. This can be used by the client application to check if the session has timed out or not (using the SSO APIs). After this time period, the session is destroyed. The session token is in the INVALID state during this extended period.

NOTE The session does not remain for this extended life if the user logs out or the session is explicitly destroyed by another Identity Server component.

- `com.iplanet.am.naming.failover.url=`

  This property can be used by any remote SDK application that wants failover in, for example, session validation or getting the service URLs.

Simple Mail Transfer Protocol (SMTP)

The following directives can be set to any valid SMTP server and port.

- `com.iplanet.am.smtphost=localhost`

NOTE Because of how Microsoft® Windows 2000 processes this information, the default value of this directive, `localhost`, should be replaced by the actual mail server host name and the Identity Server should be restarted.

- `com.sun.identity.sm.smptpport=25`

Authentication

The following sections define properties used by the Authentication Service.

LDAP

- `com.iplanet.am.auth.ldap.createUserAttrList=`<*attr1,attr2,attr3...*>

  This property specifies a list of user attributes whose values will be retrieved from an external Directory Server during LDAP Authentication if the Authentication Service is configured for dynamically creating users. The new user created in the local Directory Server will have the values for these attributes retrieved from the external Directory Server.

SecurID

- `securidHelper.ports=58943`

  The value of this property is a space-separated list used by the SecurID Authentication module and helper(s).

Unix

- `unixHelper.port=58946`

  The value of this property is used in the Unix Authentication Service.

- `unixHelper.ipaddrs=`

  The value of this property can contain a list of trusted IP addresses. The IP addresses specified in this list are space-separated and will be read by the `amserver` script and passed to the Unix helper when starting it.

Security

Following are properties that define parameters for security purposes.

SecureRandom

This property specifies the factory class name for `SecureRandomFactory`.

- `com.iplanet.security.SecureRandomFactoryImpl=com.iplanet.am.util.JSSSecureRandomFactoryImpl`

  The available implementation classes are:

  a. `com.iplanet.am.util.JSSSecureRandomFactoryImpl` (uses JSS)

  b. `com.iplanet.am.util.SecureRandomFactoryImpl` (pure Java)

SocketFactory

This property specifies the factory class name for `LDAPSocketFactory`.

- `com.iplanet.security.SSLSocketFactoryImpl=com.iplanet.services.ldap.JSSSocketFactory`

  Available classes are:

  a. `com.iplanet.services.ldap.JSSSocketFactory` (uses JSS)

  b. `netscape.ldap.factory.JSSESocketFactory` (pure Java)

Encryption

These properties specify encryption information.

- `com.iplanet.security.encryptor=com.iplanet.services.util.JSSEncryption`

  The value specifies the encrypting class implementation. Available classes are:

  a. `com.iplanet.services.util.JCEEncryption`

  b. `com.iplanet.services.util.JSSEncryption.`

- `am.encryption.pwd=BcN2Vaek2TUcs3tvO7uW9bRIrcy/Koeo`

  This is the Data Encryption Standard (DES) encryption key password. The client needs this to decrypt the session ID for token creation. If decryption fails, the client will not be able to retrieve the protocol, server host and the server port information to construct the URL needed to search for a service. Do not change the value of this property without also re-encrypting the passwords in `serverconfig.xml`. For information, see Appendix B, "serverconfig.xml File."

NOTE When installing the Identity Server SDK remotely, the value of this property should be copied into the installation field labeled `The key used for encryption of passwords`. For information on how to install the Identity Server SDK remotely, see the Sun ONE Identity Server *Migration Guide*.

IP Address Checking

This property specifies whether the IP address of the client will be checked in `SSOToken` creations and validations.

- `com.iplanet.am.clientIPCheckEnabled=false`

Remote Policy API

These properties are defined for the Remote Policy API to use with policy agents.

- `com.sun.identity.agents.app.username=UrlAccessAgent`

  This property specifies the username for the Application authentication module.

- `com.sun.identity.agents.server.log.file.name=amRemotePolicyLog`

This property specifies the name of the file to use for logging remote policy messages. The directory where this file is located is defined in Logging Service settings.

- `com.sun.identity.agents.cache.size=1000`

  This property specifies the size of the cache created on the server where the policy agent resides.

- `com.sun.identity.agents.polling.interval=3`

  The polling interval is the duration of time for refreshing the cache

- `com.sun.identity.agents.notification.enabled=false`

  This property enable or disables notifications for remote policy API.

- `com.sun.identity.agents.notification.url=`

  This property defines the notification URL for remote policy API.

- `com.sun.identity.agents.logging.level=NONE`

  This property controls the granularity of logging for the remote policy API. The valid values are `ALLOW`, `DENY`, `BOTH` and `NONE`. The default value is `NONE`.

- `com.sun.identity.agents.use.wildcard=true`

  This property indicates whether to use wildcard for resource name comparison.

- `com.sun.identity.agents.header.attributes=cn,ou,o,mail,employeenumber,c`

  This property defines the 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.resource.comparator.class=com.sun.identity.policy.plugins.PrefixResourceName`
- `com.sun.identity.agents.resource.wildcard=*`
- `com.sun.identity.agents.resource.delimiter=/`
- `com.sun.identity.agents.resource.caseSensitive=false`

  This is to indicate whether case sensitivity is turned on or off during policy evaluation. The default value is `false` or off.

- `com.sun.identity.agents.true.value=allow`

  This value is ignored if the application does not access the method `PolicyEvaluator.isAllowed`.

Policy

This property defines weights for policy subjects, rules and conditions. These weights influence the order in which these components are evaluated. The value is three integers delimited by ":". These integers indicate the proportional CPU cost for evaluating the three components, respectively.

- `com.sun.identity.policy.Policy.policy_evaluation_weights=10:10:10`

Federation

These properties configure information for the Federation Management module.

- `com.sun.identity.federation.fedCookieName=fedCookie`

  This property defines the name of the federation cookie.

- `com.sun.identity.federation.services.signingOn=false`

  This property defines whether federation requests and responses will be signed before sending. It also defines whether federation requests and responses that are received will be verified for signature validity. The default is `false`.

FQDN Map

The Fully Qulified Domain Name (FQDN) Map is a simple map that enables the Authentication Service to take corrective action in the case where a user may have typed in an incorrect URL either by specifying partial hostname or IP address to access a protected resource.

Valid values must comply with the syntax of this property which represent invalid FQDN values mapped to correct counterparts. The valid format for specifying these maps is:

`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 *valid_name* is the FQDN host name to which the filter will redirect the user.

CAUTION Ensure that there are no invalid or overlapping values for the same invalid FQDN name.

This property can also be used for creating a mapping for more than one host name. This may be the case when applications hosted on a server are accessible by more than one host name. 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_address*`]=`*IP_address*

Any number of values 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[isserver]=isserver.mydomain.com`
- `com.sun.identity.server.fqdnMap[isserver.mydomain]=isserver.mydomain.com`
- `com.sun.identity.server.fqdnMap[`*IP_address*`]=isserver.mydomain.com`
- `com.sun.identity.server.fqdnMap[`*invalid_name*`]=`*valid_name*

Encryption Key

The value of this property is the password used to generate a symmetric key to encrypt and decrypt other sensitive data including the shared secret.

```
am.encryption.pwd=ro/LiN3pOxMXxtvbwf+owRFyzDYwxRTw
```

Appendix B

serverconfig.xml File

serverconfig.xml provides configuration information for the Sun™ One Identity Server regarding the Sun ONE Directory Server that is used as its data store. This chapter explains the elements of the file and how to configure it for failover, how can you have multiple instances, how can you undeploy the console and remove console files from a server. It contains the following sections:

- Overview
- server-config Definition Type Document
- Failover Or Multimaster Configuration

Overview

serverconfig.xml is located in /*IdentityServer_base*/SUNWam/config/ums. It contains the parameters used by the Identity SDK to establish the LDAP connection pool to Directory Server. No other function of the product uses this file. Two users are defined in this file: user1 is a Directory Server proxy user and user2 is the Directory Server administrator.

Proxy User

The *Proxy User* can take on any user's privileges (for example, the organization administrator or an end user). The connection pool is created with connections bound to the proxy user. Identity Server creates a proxy user with the DN of cn=puser,ou=DSAME Users,dc=example,dc=com. This user is used for all queries made to Directory Server by IIdentity Server. It benefits from a proxy user ACI already configured in the Directory Server and, therefore, can perform actions on behalf of a user when necessary. It maintains an open connection through which all queries are passed (retrieval of service configurations, organization information, etc.). The proxy user password is always encrypted. Code Example B-1 illustrates where the encrypted password is located in `serverconfig.xml`.

Code Example B-1 Proxy User In serverconfig.xml

```
<User name="User1" type="proxy">
<DirDN>
cn=puser,ou=DSAME Users,dc=example,dc=com
</DirDN>
<DirPassword>
AQICkc3qIrCeZrpexyeoL4cdeXih4vv9aCZZ
</DirPassword>
</User>
```

Admin User

`dsameuser` is used for binding purposes when the Identity Server SDK performs operations on Directory Server that are not linked to a particular user (for example, retrieving service configuration information). Proxy User performs these operations on behalf of `dsameuser`, but a bind must first validate the `dsameuser` credentials. During installation, Identity Server creates `cn=dsameuser,ou=DSAME Users,dc=example,dc=com`. Code Example B-1 illustrates where the encrypted `dsameuser` password is found in `serverconfig.xml`.

Code Example B-2 Admin User In serverconfig.xml

```
 <User name="User2" type="admin">
 <DirDN>
 cn=dsameuser,ou=DSAME Users,dc=example,dc=com
 </DirDN>
 <DirPassword>
 AQICkc3qIrCeZrpexyeoL4cdeXih4vv9aCZZ
 </DirPassword>
 </User>
```

server-config Definition Type Document

`server-config.dtd` defines the structure for `serverconfig.xml`. It is located in *IdentityServer_base*`/SUNWam/dtd`. This section defines the main elements of the DTD. Code Example B-3 on page 465 is an example of the `serverconfig.xml` file.

iPlanetDataAccessLayer Element

iPlanetDataAccessLayer is the root element. It allows for the definition of multiple server groups per XML file. Its immediate sub-element is the ServerGroup Element. It contains no attributes.

ServerGroup Element

ServerGroup defines a pointer to one or more directory servers. They can be master servers or replica servers. The sub-elements that qualify the *ServerGroup* include Server Element, User Element, BaseDN Element and MiscConfig Element. The XML attributes of *ServerGroup* are the name of the server group, and *minConnPool* and *maxConnPool* which define the minimum (1) and maximum (10) connections that can be opened for the LDAP connection pool. More than one defined ServerGroup element is not supported.

NOTE Identity Server uses a connection pool to access Directory Server. All connections are opened when Identity Server starts and are not closed. They are reused.

Server Element

Server defines a specific Directory Server instance. It contains no sub-elements. The required XML attributes of *Server* are a user-friendly name for the server, the host name, the port number on which the Directory Server runs, and the type of LDAP connection that must be opened (either simple or SSL).

NOTE For an example of automatic failover using the Server element, see “Failover Or Multimaster Configuration” on page 466.

User Element

User contains sub-elements that define the user configured for the Directory Server instance. The sub-elements that qualify *User* include *DirDN* and *DirPassword*. It's required XML attributes are the name of the user, and the type of user. The values for *type* identify the user's privileges and the type of connection that will be opened to the Directory Server instance. Options include:

- auth—defines a user authenticated to Directory Server.
- proxy—defines a Directory Server proxy user. See "Proxy User" on page 461 for more information.
- rebind—defines a user with credentials that can be used to rebind.
- admin—defines a user with Directory Server administrative privileges. See "Admin User" on page 462 for more information.

DirDN Element

DirDN contains the LDAP Distinguished Name of the defined user.

DirPassword Element

DirPassword contains the defined user's encrypted password.

CAUTION It is important that passwords and encryption keys are kept consistent throughout the deployment. For example, the passwords defined in this element are also stored in Directory Server. If the password is to be changed in one place, it must be updated in both places. Additionally, this password is encrypted using the key defined in Appendix A, "AMConfig.properties File." If the encryption key defined in the `am.encryption.pwd` property is changed, all passwords in `serverconfig.xml` must be re-encrypted using `ampassword --encrypt` *password*. More information on this encryption utility can be found in the *Sun ONE Identity Server Administration Guide*.

BaseDN Element

BaseDN defines the base Distinguished Name for the server group. It contains no sub-elements and no XML attributes.

MiscConfig Element

MiscConfig is a placeholder for defining any LDAP JDK features like cache size. It contains no sub-elements. It's required XML attributes are the name of the feature and its defined value.

Code Example B-3 serverconfig.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!--
 Copyright (c) 2002 Sun Microsystems, Inc. All rights reserved.

 Use is subject to license terms.

-->
<iPlanetDataAccessLayer>
        <ServerGroup name="default" minConnPool="1" maxConnPool="10">
                <Server name="Server1" host="identity_server_host.domain_name"
port="389"
type="SIMPLE" />
                <User name="User1" type="proxy">
                        <DirDN>
                                cn=puser,ou=DSAME Users,dc=example,dc=com
                        </DirDN>
                        <DirPassword>
                                AQICkc3qIrCeZrpexyeoL4cdeXih4vv9aCZZ
                        </DirPassword>
                </User>
                <User name="User2" type="admin">
                        <DirDN>
                               cn=dsameuser,ou=DSAME Users,dc=example,dc=com
                        </DirDN>
                        <DirPassword>
                                AQICkc3qIrCeZrpexyeoL4cdeXih4vv9aCZZ
                        </DirPassword>
                </User>
                <BaseDN>
                        dc=example,dc=com
                </BaseDN>
        </ServerGroup>
</iPlanetDataAccessLayer>
```

Failover Or Multimaster Configuration

Identity Server allows automatic failover to any Directory Server defined as a Server Element in `serverconfig.xml`. More than one server can be configured for failover purposes or multimasters. If the first configured server goes down, the second configured server will takeover. Code Example B-4 illustrates `serverconfig.xml` with automatic failover configuration.

Code Example B-4 Configured Failover in serverconfig.xml

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<!--
PROPRIETARY/CONFIDENTIAL. Use of this product is subject to license terms.
Copyright 2002 Sun Microsystems, Inc. All rights reserved.
-->
<iPlanetDataAccessLayer>
     <ServerGroup name="default" minConnPool="1" maxConnPool="10">
          <Server name="Server1" host="identity_server_host1.domain_name" port="389"
type="SIMPLE" />
          <Server name="Server2" host="identity_server_host2.domain_name" port="389"
type="SIMPLE" />
          <Server name="Server3" host="identity_server_host3.domain_name" port="390"
type="SIMPLE" />
          <User name="User1" type="proxy">
               <DirDN>
                    cn=puser,ou=DSAME Users,dc=example,dc=com
               </DirDN>
               <DirPassword>
                    AQIC5wM2LY4Sfcy+AQBQxghVwhBE92i78cqf
               </DirPassword>
          </User>
          <User name="User2" type="admin">
               <DirDN>
                    cn=dsameuser,ou=DSAME Users,dc=example,dc=com
               </DirDN>
               <DirPassword>
                    AQIC5wM2LY4Sfcy+AQBQxghVwhBE92i78cqf
               </DirPassword>
          </User>
          <BaseDN>
               o=isp
          </BaseDN>
     </ServerGroup>
</iPlanetDataAccessLayer>
```

Appendix C

WAR Files

Sun™ One Identity Server contains a number of web application archive (WAR) files. These packages contain Java™ servlets and JavaServer Pages™ (JSP) pages that add functionality to the application. This chapter explains WAR files in general, their contents in an Identity Server deployment and which files can be modified. It contains the following sections:

- Overview
- WARs And Their Contents
- Updating Modified WARs
- Redeploying Modified WARs

Overview

The Java 2 Platform, Enterprise Edition (J2EE) platform (on which Identity Server is built) uses a component model to create full-scale applications. A *component* is self-contained functional software code assembled with other components into a J2EE application. The J2EE application components (which can be deployed separately on different servers) include:

1. Client components (including dynamic web pages, applets, and a Web browser) that run on the client machine.
2. Web components (including servlets and JSP) that run within a web container.
3. Business components (code that meets the needs of a particular enterprise domain such as banking, retail, or finance) that also run within the web container.
4. Enterprise infrastructure software that runs on legacy machines.

The *web components* tier in the Identity Server model can be customized based on each organization's needs. This appendix concerns itself with this tier.

Web Components

When a web browser executes a J2EE application, it deploys server-side objects called *web components*. There are two types of web components: *Servlets* and *JavaServer Pages* (JSP).

- Servlets are small Java programs that dynamically process requests and construct responses from a web browser; they run within web containers.
- JSP are text-based documents that contain static template data [HTML, Scalable Vector Graphics (SVG), Wireless Markup Language (WML), or eXtensible Markup Language (XML)], and elements that construct dynamic content (in the case of Identity Server, servlets).

When a J2EE application is called, the JSP and corresponding servlets are constructed by the web browser.

Packaging Web Components

In general, all J2EE components are packaged separately and bundled together into an Enterprise Archive (EAR) file for application deployment. The Web Components, in particular, are packaged in *web application archives* (WAR). Each WAR contains the servlets and/or JSP, a deployment descriptor, and related resource files.

NOTE The WAR is the same format as a JavaARchive (JAR). However, an eXtensisible Markup Language (XML) deployment descriptor file must also be created.

Static HTML files and JSP are stored at the top level of the WAR directory. The top-level directory contains the WEB-INF sub-directory which contains the following:

- Server-side classes (Servlets, JavaBean components and related Java class files) must be stored in the `WEB-INF/classes` directory.
- Auxiliary JARs (tag libraries and any utility libraries called by server-side classes) must be stored in the `WEB-INF/lib` directory.

- web.xml—the web component deployment descriptor is stored in the WEB-INF directory.
- Tag library descriptor files

When modifying the files included in Identity Server WARs, customers are changing web components and thus, customizing their deployment.

NOTE Be aware of any loss of the customized data during patch or upgrade.

WARs And Their Contents

Identity Server contains a number of WARs that can be modified to customize an Identity Server deployment. The WARs themselves are located in *IdentityServer_base*/`SUNWam` and include:

- `console.war`—files pertaining to the Identity Server console application.
- `password.war`—files pertaining to the Identity Server password reset service.
- `services.war`—contains files pertaining to Identity Server services.

The following sections detail the files within each WAR that can be modified and those that SHOULD NOT be modified.

console.war

The following sections detail the modifiable and non-modifiable documents contained within `console.war`. The path names are based on the directory structure discussed in Packaging Web Components.

console.war Modifiable Files

These directories contain files that can be modified.

- `web.xml` and related XML files used for constructructing it are located in *IdentityServer_base*/`SUNWam/web-apps/applications/WEB-INF/`.
- Modifiable JavaScript files are located in *IdentityServer_base*/`SUNWam/web-apps/applications/console/js/`.
- Modifiable JSP are located in the following directories dependant upon the service that deploys them:

    - ❍ *IdentityServer_base*/SUNWam/web-apps/applications/console/auth/
    - ❍ *IdentityServer_base*/SUNWam/web-apps/applications/console/federation/
    - ❍ *IdentityServer_base*/SUNWam/web-apps/applications/console/policy/
    - ❍ *IdentityServer_base*/SUNWam/web-apps/applications/console/service/
    - ❍ *IdentityServer_base*/SUNWam/web-apps/applications/console/session/
    - ❍ *IdentityServer_base*/SUNWam/web-apps/applications/console/user/

- Modifiable image files are located in *IdentityServer_base*/SUNWam/web-apps/applications/console/images/.
- Modifiable stylesheets are located in *IdentityServer_base*/SUNWam/web-apps/applications/console/css/.

console.war Non-Modifiable Files

These directories contain files that SHOULD NOT be modified.

- JARs are located in *IdentityServer_base*/SUNWam/web-apps/applications/WEB-INF/lib/.
- Tag Library Descriptor (.tld) files are located in *IdentityServer_base*/SUNWam/web-apps/applications/WEB-INF/.

password.war

The following sections detail the modifiable and non-modifiable documents contained within password.war. The path names are based on the directory structure discussed in Packaging Web Components.

password.war Modifiable Files

These directories contain files that can be modified.

- web.xml and related XML files used for constructructing it are located in *IdentityServer_base*/SUNWam/web-apps/password/WEB-INF/.
- Modifiable JSP are located in *IdentityServer_base*/SUNWam/web-apps/password/password/ui/.
- Modifiable image files are located in *IdentityServer_base*/SUNWam/web-apps/password/password/images/.

- Modifiable stylesheets are located in *IdentityServer_base*/`SUNWam/web-apps/password/password/css/`.

password.war Non-Modifiable Files

These directories contain files that SHOULD NOT be modified.

- Non-modifiable JARs are located in *IdentityServer_base*/`SUNWam/web-apps/password/WEB-INF/lib/`.
- Non-modifiable tag library descriptor (.tld) files are located in *IdentityServer_base*/`SUNWam/web-apps/password/WEB-INF/`.

services.war

The following sections detail the modifiable and non-modifiable documents contained within `services.war`. The path names are based on the directory structure discussed in Packaging Web Components.

services.war Modifiable Files

These directories contain files that can be modified.

- `web.xml` and related XML files used for constructructing it are located in *IdentityServer_base*/`SUNWam/web-apps/services/WEB-INF/`.
- JavaScript files are located in *IdentityServer_base*/`SUNWam/web-apps/services/js/`.
- JSP are located in the following directories dependant upon the service that requires the customization:
  - *IdentityServer_base*/`SUNWam/web-apps/services/config/auth/default/`
  - *IdentityServer_base*/`SUNWam/web-apps/services/config/federation/default/`
- Image files are located in the following directories dependant upon the service to which the images apply:
  - *IdentityServer_base*/`SUNWam/web-apps/services/images/`
  - *IdentityServer_base*/`SUNWam/web-apps/services/fed_images/`
  - *IdentityServer_base*/`SUNWam/web-apps/services/login_images/`
- Stylesheets are located in the following directories dependant upon the service to which they apply:

- *IdentityServer_base*/`SUNWam/web-apps/services/css/`.
- *IdentityServer_base*/`SUNWam/web-apps/services/fed_css/`.

services.war Non-Modifiable Files

These directories contain files that SHOULD NOT be modified.

- Non-modifiable JARs are located in *IdentityServer_base*/`SUNWam/web-apps/services/WEB-INF/lib/`.
- Non-modifiable Tag Library Descriptor (.tld) files are located in *IdentityServer_base*/`SUNWam/web-apps/services/WEB-INF/`.

Updating Modified WARs

Once a file within a WAR is modified, the WAR itself needs to be updated with the newly modified file. Following is the procedure to update a WAR.

1. cd *IdentityServer_base* / `SUNWam`

   This is the directory in which the WARs are kept.

2. jar -uvf *WARfilename*.war <*path_to_modified_file*>

   The `-uvf` option replaces the old file with the newly modified file. For example:

   ```
   jar -uvf console.war newfile/index.html
   ```

   replaces the `index.html` file in console.war with the `index.html` file located in *IdentityServer_base*/`SUNWam/newfile`.

3. `rm newfile/index.html`

   Delete the modified file.

Redeploying Modified WARs

Once updated, the WARs need be redeployed to their web container. The web container provides services such as request dispatching, security, concurrency, and life cycle management. It also gives the web components access to the J2EE APIs. The following procedures are specific to each particular WAR and web container. After redeploying the war files, all related servers need to be restarted.

NOTE The BEA WebLogic Server 6.1 and Sun ONE Application Server web containers do not requireWARs to be exploded. They are deployed as WARs.

BEA WebLogic Server 6.1

The following commands are used on BEA WebLogic Server 6.1 to redeploy Identity Server WARs.

NOTE `amconsole`, `amserver` and `ampassword` are the default console, server and password deploy URIs, respectively.

To Deploy console.war On WebLogic

```
java weblogic.deploy -url protocol://server_host:server_port -component
amconsole:WL61 _server_name deploy WL61_admin_password amconsole
IdentityServer_base/SUNWam/console.war
```

To Deploy services.war on WebLogic

```
java weblogic.deploy -url protocol://server_host:server_port -component
amserver:WL61 _server_name deploy WL61_admin_password amserver
IdentityServer_base/SUNWam/services.war
```

To Deploy password.war on WebLogic

```
java weblogic.deploy -url protocol://server_host:server_port -component
ampassword:WL61 _server_name deploy WL61_admin_password ampassword
IdentityServer_base/SUNWam/password.war
```

NOTE For more complete information on the Java utility `weblogic.deploy` and its options, see the BEA WebLogic Server 6.1 documentation.

Sun ONE Application Server 7.0

The following commands are used on Sun ONE Application Server 7.0 to redeploy Identity Server WARs.

To Deploy console.war On Sun ONE Application Server

asadmin deploy -u *S1AS_administrator* -w *S1AS_administrator_password* -H *console_server_host* -p *S1AS_server_port* --type web *secure_flag* --contextroot *console_deploy_uri* --name `amconsole` --instance *S1AS_instance* *IdentityServer_base* / `SUNWam` / console.war

To Deploy services.war On Sun ONE Application Server

asadmin deploy -u *S1AS_administrator* -w *S1AS_administrator_password* -H *server_host* -p *S1AS_server_port* --type web *secure_flag* --contextroot *server_deploy_uri* --name amserver --instance *S1AS_instance* *IdentityServer_base* / `SUNWam` / services.war

To Deploy password.war on Sun ONE Application Server

asadmin deploy -u *S1AS_administrator* -w *S1AS_administrator_password* -H *console_server_host* -p *S1AS_administrator_server_port* --type web *secure_flag* --contextroot *password_deploy_uri* --name ampassword --instance *S1AS_instance* *IdentityServer_base* / `SUNWam` / password.war

NOTE For more complete information on the `asadmin` deploy command and its options, see the Sun ONE Application Server 7.0 Developer’s Guide.

IBM WebSphere Application Server

For detailed instructions on how to deploy WARs in an IBM WebSphere Application Server container, see the documentation at `http://www-3.ibm.com/software/webservers/studio/doc/v40/studioguide/en/html/sdsscenario1.html`.

Appendix D

Notification Service

Sun™ ONE Identity Server Notification Service allows for session notifications to be sent to remote web containers. It is necessary to enable this service for use by SDK applications running remotely from the Identity Server server itself. This chapter explains how to enable a remote web container to receive the notifications. It contains the following sections:

- Overview
- Enabling The Notification Service

Overview

The Notification Service allows for session notifications to be sent to web containers that are running the Identity Server SDK remotely. The notifications apply to the Session, Policy and Naming Services only. In addition, the remote application must be running in a web container. The purpose of the notifications would be:

- To sync up the client side cache of the respective services.
- To enable more real time updates on the clients. (Polling is used in absense of notifications.)
- No client application changes are required to support notifications.

Enabling The Notification Service

Following are the steps to configure the remote SSO SDK to receive session notifications. Setting up clients to receive notifications

1. Install Identity Server on Machine 1.

2. Install Sun ONE Web Server on Machine 2.

3. Install the `SUNWamsdk` on the same machine as the Web Server.

   For instructions on installing the Identity Server SDK remotely, see the *Sun Java™ Enterprise System 2003Q4 Installation Guide*.

4. Ensure that the following are true concerning the machine where the SDK is installed.

   a. Ensure that the right access permissions are set for the /*remote_SDK_server*/`SUNWam/lib` and /*remote_SDK_server*/`SUNWam/locale` directories on the server where the SDK is installed.

      These directories contains the files and jars on the remote server.

   b. Ensure that the following permissions are set in the Grant section of the `server.policy` file of the Web Server.

      `server.policy` is in the `config` directory of the Web Server installation. These permissions can be copied and pasted, if necessary:

      ```
      permission java.security.SecurityPermission
      "putProviderProperty.Mozilla-JSS"
      ```

      ```
      permission java.security.SecurityPermission
      "insertProvider.Mozilla-JSS";
      ```

   c. Ensure that the correct classpath is set in `server.xml`.

      `server.xml` is also in the `config` directory of the Web Server installation. A typical classpath would be:

      ```
      <JAVA javahome="/export/home/ws61/bin/https/jdk"
      serverclasspath="/export/home/ws61/bin/https/jar/webserv-rt.
      jar:${java.home}/lib/tools.jar:/export/home/ws61/bin/https/j
      ar/webserv-ext.jar:/export/home/ws61/bin/https/jar/webserv-j
      stl.jar:/export/home/ws61/bin/https/jar/nova.jar"
      classpathsuffix="::/IS_CLASSPATH_BEGIN_DELIM://usr/share/lib
      /xalan.jar:/export/SUNWam/lib/xmlsec.jar://usr/share/lib/xer
      cesImpl.jar://usr/share/lib/sax.jar://usr/share/lib/dom.jar:
      /export/SUNWam/lib/dom4j.jar:/export/SUNWam/lib/jakarta-log4
      j-1.2.6.jar:/usr/share/lib/jaxm-api.jar:/usr/share/lib/saaj-
      api.jar://usr/share/lib/jaxrpc-api.jar://usr/share/lib/jaxrp
      c-impl.jar:/export/SUNWam/lib/jaxm-runtime.jar:/usr/share/li
      b/saaj-impl.jar:/export/SUNWam/lib:/export/SUNWam/locale://u
      sr/share/lib/mps/jss3.jar:/export/SUNWam/lib/am_sdk.jar:/exp
      ```

```
ort/SUNWam/lib/am_services.jar:/export/SUNWam/lib/am_sso_pro
vider.jar:/export/SUNWam/lib/swec.jar:/export/SUNWam/lib/acm
ecrypt.jar:/export/SUNWam/lib/iaik_ssl.jar://usr/share/lib/j
axp-api.jar://usr/share/lib/mail.jar://usr/share/lib/activat
ion.jar:/export/SUNWam/lib/servlet.jar:/export/SUNWam/lib/am
_logging.jar:/usr/share/lib/commons-logging.jar:/IS_CLASSPAT
H_END_DELIM:" envclasspathignored="true" debug="false"
debugoptions="-Xdebug
-Xrunjdwp:transport=dt_socket,server=y,suspend=n"
javacoptions="-g" dynamicreloadinterval="2">
```

5. Use the SSO samples installed on the remote SDK server for confiugration purposes.

   a. Change to the /*remote_SDK_server*/`SUNWam/samples/sso` directory.

   b. Run `gmake`.

   c. Copy the generated class files from /*remote_SDK_server*/`SUNWam/samples/sso` to /*remote_SDK_server*/`SUNWam/lib/`.

6. Copy the encryption value of `am.encryption.pwd` from the `AMConfig.properties` file installed with Identity Server to the `AMConfig.properties` file on the remote server to which the SDK was installed.

   The value of `am.encryption.pwd` is used for encrypting and decrypting passwords.

7. Login into Identity Server as `amadmin`.

   `http://`*identity_server_host*`:3000/amconsole`

8. Execute the servlet by entering `http://`*remote_SDK_host*`:58080/servlet/SSOTokenSampleServlet` into the browser location field and validating the `SSOToken`.

   `SSOTokenSampleServlet` is used for validating a session token and adding a listener. Executing the servlet will print out the following message:

   ```
   SSOToken host name: 192.18.149.33 SSOToken Principal name:
   uid=amAdmin,ou=People,dc=red,dc=iplanet,dc=com Authentication
   type used: LDAP IPAddress of the host: 192.18.149.33 The token
   id is AQIC5wM2LY4SfcyURnObg7vEgdkb+32T43+RZN30Req/BGE= Property:
   Company is - Sun Microsystems Property: Country is - USA SSO
   Token Validation test Succeeded
   ```

9. Set the property `com.iplanet.am.notification.url=` in `AMConfig.properties` of the remote machine as follows:

   `com.iplanet.am.notification.url=http://`*remote_SDK_server*`:58080/servlet/com.iplanet.services.comm.server.PLLRequestServlet`

10. Restart the Web Server.

11. Login into Identity Server as `amadmin`.

    `http://`*identity_server_host*`:3000/amconsole`

12. Execute the servlet by entering `http://`*remote_SDK_host*`:58080/servlet/SSOTokenSampleServlet` into the browser location field and validating the `SSOToken` again.

    When the machine on which the remote SDK is running receives the notification, it will call the respective listener when the session state is changed.

Appendix E

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 used by Identity Server to help manage its 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
- Class Of Service

Overview

Because Identity Server needs an underlying data store, it has been built to work with Sun ONE Directory Server. They are complementary in architecture and design data. Use of Directory Server, 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. A *role* is an identity grouping mechanism; an *access control instruction* (ACI) defines rules to allow or deny access to Directory Server data, and *class of service* is an attribute grouping mechanism.

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 12-1 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 12-2 below, the DN identifies Code Example 12-1 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 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 12-2 on page 482 includes this virtual attribute when returned by Directory Server only.

Code Example 12-2 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 ACI that define administrator permissions to directory data. These ACI 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 level of access. For a list of roles created for each Identity Server object configured, see "Access Control Instructions" on page 484.

NOTE Managed groups in Identity Server are modeled almost the same as roles. They add an attribute to an LDAP entry to make the entry a member of the dynamic group

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 482.

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

Control over access to directory information is implemented in Identity Server using roles. Users inherit access permissions based on their role membership and parent organization. Identity Server installs pre-configured administrator roles that define different levels of permission for administrators to access directory information; 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. Policy is another form of access control which are created and used in Identity Server but apply to web resources not Directory Server data.

These default roles, when possessed by a user entry, define that user's level of access to Directory Server data. 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 entries in the organization and read access to all entries in the organization, 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 485.

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 in Identity Server for use with administrator-type roles follow a different format than those defined using 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 the ACI allows.
- 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 486 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 ACI 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=Top 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 263 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.

Index