Plug-In API Programming Guide

Sun™ ONE Directory Server

Version 5.2

816-6702-10
June 2003

Contents

List of Code Examples

About This Guide

Sun™ ONE Directory Server 5.2 is a powerful and scalable distributed directory server based on the industry-standard Lightweight Directory Access Protocol (LDAP). Sun ONE Directory Server software is part of the Sun Open Net Environment (Sun ONE), Sun's standards-based software vision, architecture, platform, and expertise for building and deploying Services On Demand.

Sun ONE Directory Server is the cornerstone for building a centralized and distributed data repository that can be used in your intranet, over your extranet with your trading partners, or over the public Internet to reach your customers.

Purpose of This Guide

This guide demonstrates how to develop *server plug-ins*, libraries registered with Directory Server that customize and extend server capabilities. This guide also lists what has changed since the last release, so you can upgrade 4.x plug-ins for use with Directory Server 5.2.

Prerequisites

To make the most of this document, you must be familiar with Directory Server and with programming in the C language.

Typographical Conventions

This section explains the typographical conventions used in this book.

`Monospaced font` - This typeface is used for literal text, such as the names of attributes and object classes when they appear in text. It is also used for URLs, filenames, and examples.

Italic font - This typeface is used for emphasis, for new terms, and for text that you must substitute for actual values, such as placeholders in path names.

The greater-than symbol (>) is used as a separator when naming an item in a menu or sub-menu. For example, Object > New > User means that you should select the User item in the New sub-menu of the Object menu.

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

Default Paths and Filenames

All path and filename examples in the Sun ONE Directory Server product documentation are one of the following two forms:

- *ServerRoot/...* - The *ServerRoot* is the location of the Sun ONE Directory Server product. This path contains the shared binary files of Directory Server, Sun ONE Administration Server, and command line tools.

  The actual *ServerRoot* path depends on your platform, your installation, and your configuration. The default path depends on the product platform and packaging as shown in Table 1.

- *ServerRoot*`/slapd-`*serverID*`/...` - The *serverID* is the name of the Directory Server instance that you defined during installation or configuration. This path contains database and configuration files that are specific to the given instance.

NOTE Paths specified in this manual use the forward slash format of UNIX and commands are specified without file extensions. If you are using a Windows version of Sun ONE Directory Server, use the equivalent backslash format. Executable files on Windows systems generally have the same names with the `.exe` or `.bat` extension.

Table 1 Default *ServerRoot* Paths

Product Version	*ServerRoot* **Path**
Solaris Packages[1]	`/var/mps/serverroot` - After configuration, this directory contains links to the following locations: • `/etc/ds/v5.2` (static configuration files) • `/usr/admserv/mps/admin` (Sun ONE Administration Server binaries) • `/usr/admserv/mps/console` (Server Console binaries) • `/usr/ds/v5.2` (Directory Server binaries)
Compressed Archive Installation on Solaris and Other UNIX Systems	`/var/Sun/mps`
Zip Installation on Windows Systems	`C:\Program Files\Sun\MPS`

1. If you are working on the Solaris Operating Environment and are unsure which version of the Sun ONE Directory Server software is installed, check for the existence a key package such as `SUNWdsvu` using the `pkginfo` command. For example: `pkginfo | grep SUNWdsvu`.

Directory Server instances are located under *ServerRoot*`/slapd-`*serverID*`/`, where *serverID* represents the server identifier given to the instance on creation. For example, if you gave the name `dirserv` to your Directory Server, then the actual path would appear as shown in Table 2. If you have created a Directory Server instance in a different location, adapt the path accordingly.

Table 2 Default Example `dirserv` Instance Locations

Product Version	**Instance Location**
Solaris Packages	`/var/mps/serverroot/slapd-dirserv`
Compressed Archive Installation on Solaris and Other UNIX Systems	`/usr/Sun/mps/slapd-dirserv`
Zip Installation on Windows Systems	`C:\Program Files\Sun\MPS\slapd-dirserv`

Downloading Directory Server Tools

Some supported platforms provide native tools for accessing Directory Server. More tools for testing and maintaining LDAP directory servers, download the Sun ONE Directory Server Resource Kit (DSRK). This software is available at the following location:

```
http://wwws.sun.com/software/download/
```

Installation instructions and reference documentation for the DSRK tools is available in the *Sun ONE Directory Server Resource Kit Tools Reference*.

For developing directory client applications, you may also download the iPlanet Directory SDK for C and the iPlanet Directory SDK for Java from the same location.

Additionally, Java Naming and Directory Interface (JNDI) technology supports accessing the Directory Server using LDAP and DSML v2 from Java applications. Information about JNDI is available from:

```
http://java.sun.com/products/jndi/
```

The JNDI Tutorial contains detailed descriptions and examples of how to use JNDI. It is available at:

```
http://java.sun.com/products/jndi/tutorial/
```

Suggested Reading

Sun ONE Directory Server product documentation includes the following documents delivered in both HTML and PDF:

- *Sun ONE Directory Server Getting Started Guide* - Provides a quick look at many key features of Directory Server 5.2.
- *Sun ONE Directory Server Deployment Guide* - Explains how to plan directory topology, data structure, security, and monitoring, and discusses example deployments.
- *Sun ONE Directory Server Installation and Tuning Guide* - Covers installation and upgrade procedures, and provides tips for optimizing Directory Server performance.
- *Sun ONE Directory Server Administration Guide* - Gives the procedures for using the console and command-line to manage your directory contents and configure every feature of Directory Server.

- *Sun ONE Directory Server Reference Manual* - Details the Directory Server configuration parameters, commands, files, error messages, and schema.
- *Sun ONE Directory Server Plug-In API Programming Guide* - Demonstrates how to develop Directory Server plug-ins.
- *Sun ONE Directory Server Plug-In API Reference* - Details the data structures and functions of the Directory Server plug-in API.
- *Sun ONE Server Console Server Management Guide* - Discusses how to manage servers using the Sun ONE Administration Server and Java based console.
- *Sun ONE Directory Server Resource Kit Tools Reference* - Covers installation and features of the Sun ONE Directory Server Resource Kit, including many useful tools.

Other useful information can be found on the following Web sites:

- Product documentation online: `http://docs.sun.com/coll/S1_DirectoryServer_52`
- Sun software: `http://wwws.sun.com/software/`
- Sun ONE Services: `http://www.sun.com/service/sunps/sunone/`
- Sun Support Services: `http://www.sun.com/service/support/`
- Sun ONE for Developers: `http://sunonedev.sun.com/`
- Training: `http://suned.sun.com/`

Chapter 1

Before You Start

If you are not yet familiar with Sun ONE Directory Server software, take the time to read *Sun ONE Directory Server Getting Started Guide*.

Next, read this short chapter. It helps you decide whether to implement a server plug-in, and what to do next if you choose to implement one.

When to Implement a Server Plug-In

Many Sun ONE Directory Server native product features rely on *server plug-ins*, libraries of functions registered with the server to perform key parts of specific directory service operations. In publishing the Sun ONE Directory Server plug-in API, the Directory Server team offers you an interface to extend server capabilities for needs not met in the currently available releases of the product.

Enhancing Server Capabilities

If you must enhance server capabilities because some required feature — such as custom authentication, encryption, validation, pattern matching, or notification — has not yet been implemented in the product, the plug-in API may render that enhancement possible. If you can instead use standard features of the product, avoid creating a plug-in.

Maintaining Plug-Ins

Know that creating your own plug-in links your software solution to a specific product release. The plug-in API may evolve from release to release to accommodate new features. As you choose to upgrade and take advantage of new features, you may need to update your Directory Server plug-ins to account for the changes, as described in the license.

> You may use the header files and class libraries solely to create and distribute programs to interface with the Software's APIs. You may also use libraries to create and distribute program "plug-ins" to interface with the Software's plug-in APIs. You may not modify the header files or libraries. You acknowledge that Sun makes no direct or implied guarantees that the plug-in APIs will be backward-compatible in future releases of the Software. In addition, you acknowledge that Sun is under no obligation to support or to provide upgrades or error corrections to any derivative plug-ins.

When providing technical support, Sun technical support personnel will request that you reproduce the problem *after disactivating your custom plug-ins.*

Sun Professional Services

For information about what Sun Professional Services has to offer, refer to

```
http://www.sun.com/service/sunps/sunone/
```

Sun Professional Services can help you make the right deployment decisions, and can help you deploy the right solution for your specific situation, whether or not the solution includes custom plug-ins.

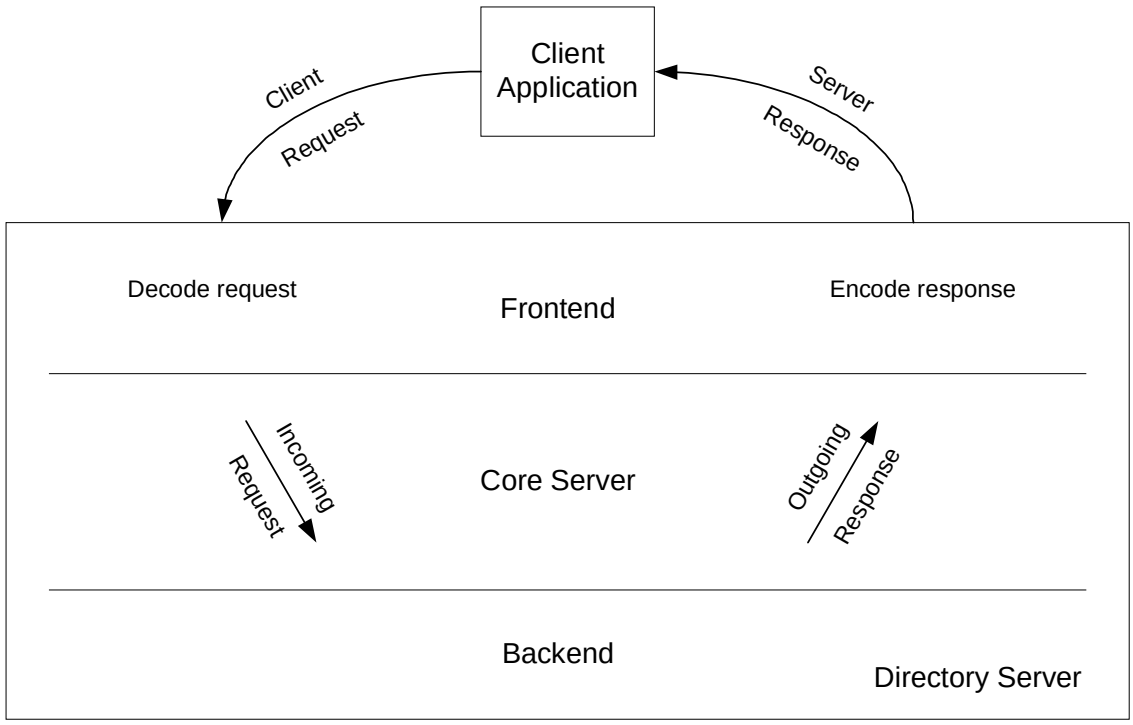
How Plug-Ins Interact With the Server

All client requests to Directory Server instances undergo particular processing sequences. The exact sequence depends on the operation requested, but overall the sequences are similar. The way plug-ins handle requests is independent of how the client encodes the request. In other words, client request data appears essentially the same to plug-ins regardless of the client access protocol used to communicate with Directory Server.

In general, client request processing proceeds as follows. First, Directory Server frontend decodes the request. The core server handles the decoded request, calling the necessary backend functions to find or store entries for example. The primary function of the Directory Server backend is to read entries from and write entries to the database. Unless abandoned, a client request results in a response originating in the backend. The frontend formats the response and sends it to the client.

Figure 1-1 illustrates how client request data moves through Directory Server.

Figure 1-1 Client Request Processing



As a rule, plug-ins register functions to be called by Directory Server at specific points in the client request processing sequence. A plug-in has a specific type, such as `preoperation` or `postoperation`. A `preoperation` plug-in registers functions handling incoming requests before they are processed in the server. For example, a `preoperation` plug-in may register a *pre-bind function* called before the core server begins processing the client bind. The pre-bind function might redefine the way the client authenticates to Directory Server, enabling authentication against an external

database. A `postoperation` plug-in registers functions called after a request is processed. For example, a `postoperation` plug-in may register a post-modification function called to log the information about the modification request. Table 1-2 provides examples for other plug-in types.

The two internal operation plug-in types form an exception to the rule that functions are called to perform client request processing. Internal operation plug-ins implement functions triggered for internal Directory Server operations. Internal operations are initiated not by client requests, but instead by Directory Server itself or by another plug-in called by Directory Server. Exercise great caution when manipulating internal operation data!

Table 1-1 summarizes when Directory Server calls documented plug-in function types.

Table 1-1 Server Plug-In Types

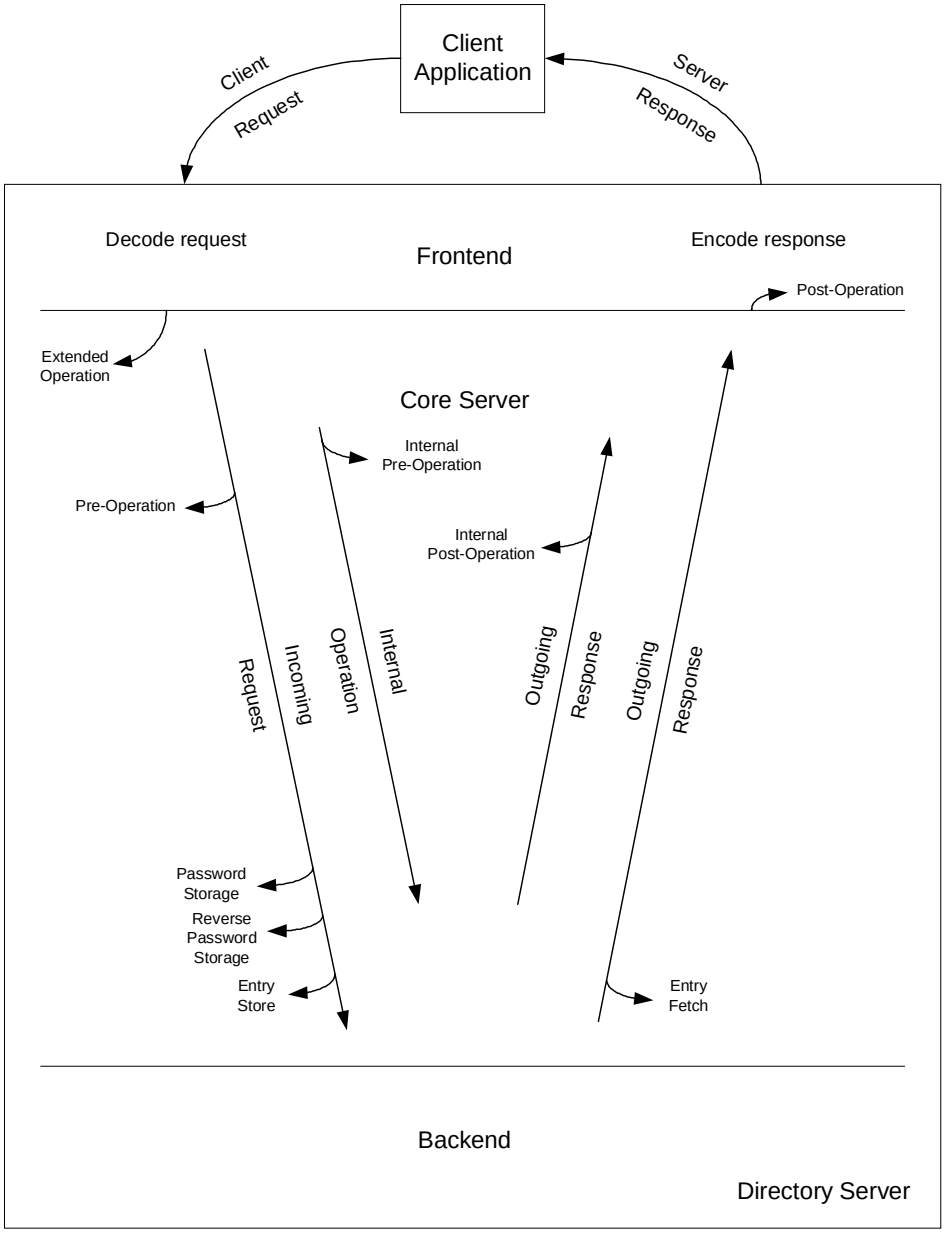
Plug-In Type	When Called
Entry store / fetch	Immediately before writing a directory entry to or immediately after reading a directory entry from the directory database
	`nsslapd-pluginType: ldbmentryfetchstore`
Extended operation	Before processing an LDAP v3 extended operation
	`nsslapd-pluginType: extendedop`
Internal post-operation	After completing an operation initiated by Directory Server, for which no corresponding client request exists
	`nsslapd-pluginType: internalpostoperation`
Internal pre-operation	Before performing an operation initiated by Directory Server, for which no corresponding client request exists
	`nsslapd-pluginType: internalpreoperation`
Matching rule	When finding search result candidates based on an extensible match filter (search operations only)
	`nsslapd-pluginType: matchingrule`
Object	Depends on the functions registered
	`nsslapd-pluginType: object`
Password storage scheme	When storing an encoded `userPassword` value that cannot be decrypted
	`nsslapd-pluginType: pwdstoragescheme`

Table 1-1 Server Plug-In Types *(Continued)*

Plug-In Type	When Called
Post-operation	After completing an operation requested by a client `nsslapd-pluginType: postoperation`
Pre-operation	Before performing an operation requested by a client `nsslapd-pluginType: preoperation`
Reversible password storage scheme	When storing an encoded `userPassword` value that must be decrypted during processing `nsslapd-pluginType: reverpwdstoragescheme`

Figure 1-2 illustrates where in the processing sequence Directory Server calls several of the plug-in types described in Table 1-1.

Figure 1-2 Plug-In Entry Points



Notice that post-operation return codes do not affect Directory Server processing.

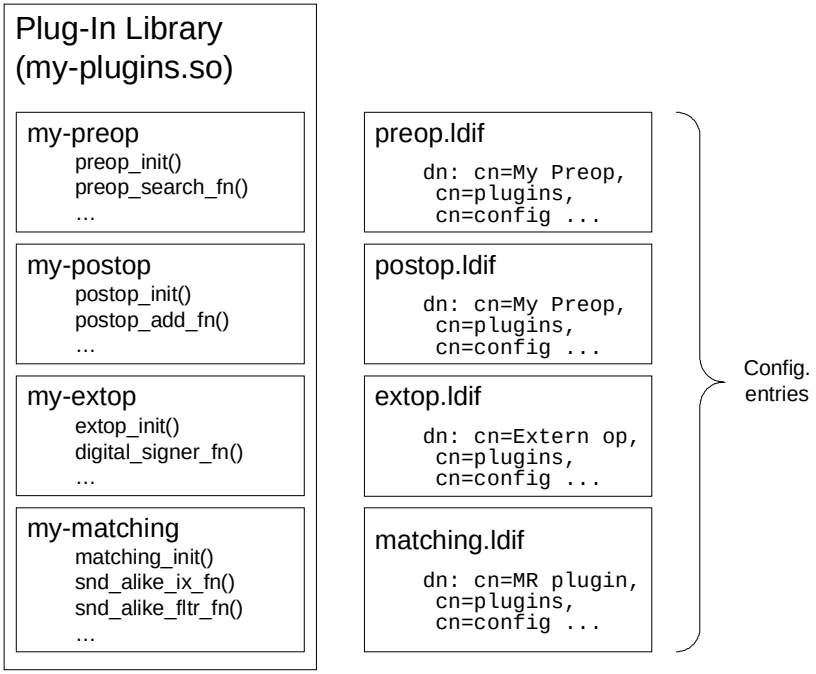
Plug-in type and other configuration information is typically specified statically in a configuration entry.

You may include multiple plug-ins in a single shared library, but you must specify individual, server-specific configuration entries for each plug-in registered statically. This means that if you want both pre- and post-operation functions, you typically write two plug-ins and register each with Directory Server. Both plug-ins may be contained in the same shared library. Both plug-ins may communicate private data across the same operation or connection, as well, using the interface provided. It is also possible to write one plug-in that initializes other plug-ins if that is required for your deployment.

The sample plug-ins delivered with the product follow the principle of building all plug-ins into one library, then registering plug-ins individually.

Figure 1-3 shows multiple plug-ins residing in the same library.

Figure 1-3 Multiple Plug-Ins in a Shared Library



Each plug-in includes a registration routine, called by Directory Server at startup. The registration routine explicitly registers each plug-in function with Directory Server.

TIP Plug-ins libraries are linked with Directory Server. Plug-ins execute in the same memory address space as Directory Server itself. They have direct access to data structures used by Directory Server. As a result, plug-ins can corrupt memory used by Directory Server. Such memory corruption can result in database corruption. This is why you *never* use an untested plug-in on a production Directory Server.

Example Uses

Selecting the right plug-in type for the job is an art, rather than a science. The following table suggests example uses for documented plug-in types.

Table 1-2 Plug-In Example Uses, By Type

Plug-In Type	Example Uses
Entry store / fetch	Encoding and decoding entire plug-in entries
	Auditing or logging each entry written to or read from the directory as it is written to disk
Extended operation	Adding client services unavailable in LDAP v3 such as digital signatures in requests and responses
Internal post-operation	Auditing results of internal operations initiated by another plug-in
Internal pre-operation	Preempting internal operations initiated by another plug-in
Matching rule	Offering enhanced sounds-like matching for directory searches
Object	Developing a plug-in that registers a group of other plug-ins with Directory Server
Password storage scheme, Reversible password storage scheme	Using a custom algorithm for password encryption instead of one of the algorithms supported by the standard product
Post-operation	Associating alerts and alarms sent after particular operations
	Auditing changes to specific entries
	Performing cleanup after an operation

Table 1-2 Plug-In Example Uses, By Type *(Continued)*

Plug-In Type	Example Uses
Pre-operation	Handling custom authentication methods external to the directory
	Forcing syntax checking for attribute values before adding or modifying an entry
	Adding attributes to or deleting attributes from a request
	Preprocessing client request content to translate requests from legacy applications
	Approving or rejecting the content of a client modification request before processing it

The list of example uses is by no means exhaustive, but is instead intended to help you brainstorm solutions.

Where to Go From Here

We devote most of the rest of this guide to examples demonstrating the specific plug-in types.

TIP *Never* develop plug-ins on a production server, but instead on a test server used specifically for plug-in development.

Prepare Your Development Environment

If you have not yet installed Sun ONE Directory Server software and have not installed C language development tools for use during plug-in development, install them now. Without development tools and a functioning Directory Server, you cannot use the examples discussed here.

Learn About Plug-In Development

If you have not yet written a plug-in for the current release, refer to Chapter 3, “Getting Started With Directory Server Plug-Ins” for a demonstration of how to build a simple plug-in and register it with Directory Server.

Upgrade Existing Plug-Ins

If you already have plug-ins developed for a previous release and need to upgrade them for use with the current release, refer to Chapter 2, “What’s New.”

Try a Sample Plug-In

Examples for several plug-in types are provided with the product under *ServerRoot*/`plugins/slapd/slapi/examples/`. Subsequent chapters demonstrate the use of the sample plug-ins.

Find Details

Refer to the *Sun ONE Directory Server Plug-In API Reference* for details about particular data structures, functions, and parameter block semantics.

Chapter 2

What's New

This chapter covers changes to the plug-in API since the 4.x releases. If you maintain Directory Server plug-ins originally developed for a previous release, we recommend upgrading such plug-ins to accommodate new features.

Sun Professional Services consultants can help. For information about what Sun Professional Services has to offer, refer to

```
http://www.sun.com/service/sunps/sunone/
```

We strongly recommend you work closely with Sun Professional Services consultants to develop and to maintain your Directory Server plug-ins.

Deprecated and Changed Features

This section lists features deprecated since the 4.x releases. Where possible, we recommend replacement functionality.

Handling Deprecation

`SLAPI_DEPRECATED` has been defined in `slapi-plugin.h` to highlight what has been superseded. If you cannot change core plug-in code, you may decide to recompile after including `slapi-plugin-compat4.h` to access deprecated features for this release. For example:

```
#include "slapi-plugin-compat4.h"
```

In general, you must *at minimum* recompile your plug-ins to have them work with Directory Server 5.2. Where possible, replace deprecated code with new alternatives before recompiling and testing.

Registering Plug-Ins

Plug-in directives have been deprecated. Use configuration entries instead. Refer to “Plugging Libraries into Directory Server,” on page 61 for details on using configuration entries to load and configure plug-ins.

Function Parameters

Many existing function parameters have become `const` to reflect that parameter values are not changed when reading information.

Data Types

Consider using opaque data types wherever possible. Refer to the *Sun ONE Directory Server Plug-In API Reference* for information on data types available.

Plug-In Types

The `SLAPI_PLUGIN_DATABASE` plug-in (`database`) type has been deprecated.

Access Control

`slapi_acl_verify_aci_syntax()` now takes a pointer to a `Slapi_PBlock` as its first parameter. You must modify plug-in code to account for this change.

Attributes

Table 2-1 lists deprecated and replacement functions for handling attributes.

Table 2-1 Replacement Functions for Handling Attributes

Deprecated Function	Replacement Function
`slapi_attr_get_values()`	`slapi_attr_get_bervals_copy()`
`slapi_attr_get_oid()`	`slapi_attr_get_oid_copy()`
`slapi_compute_add_search_rewriter()`	`slapi_compute_add_search_rewriter_ex()`

Refer to the *Sun ONE Directory Server Plug-In API Reference* for descriptions of the replacement functions.

Controls

`slapi_get_supported_controls()` is not thread-safe and has been deprecated. Use `slapi_get_supported_controls_copy()` instead.

`SLAPI_OPERATION_*` identifiers have changed to `unsigned long` values.

Entries

Table 2-2 lists deprecated and replacement functions for handling entries.

Table 2-2 Replacement Functions for Handling Entries

Deprecated Function	Replacement Function
`slapi_entry_add_values()`	`slapi_entry_add_values_sv()`
	`slapi_entry_add_valueset()`
`slapi_entry_attr_hasvalue()`	`slapi_entry_attr_has_syntax_value()`
`slapi_entry_attr_merge()`	`slapi_entry_attr_merge_sv()`
	`slapi_entry_merge_values_sv()`
`slapi_entry_attr_replace()`	`slapi_entry_attr_replace_sv()`
`slapi_entry_delete_values()`	`slapi_entry_delete_values_sv()`

Refer to the *Sun ONE Directory Server Plug-In API Reference* for descriptions of the replacement functions.

Error Codes

`slapi-plugin.h` now includes `ldap_msg.h`, which lists unique error numbers used by Directory Server.

Functions for dealing with backends that returning `int` error codes can now also return `SLAPI_FAIL_RETRY`. The old values are also still used.

Directory Server now distinguishes between SLAPI_BIND_FAIL, indicating that the bind failed and the server prepares a result to send, and SLAPI_BIND_ANONYMOUS, indicating that the bind succeeded as an anonymous bind.

Extended Operations

slapi_get_supported_extended_ops() is not thread-safe and has been deprecated. Use slapi_get_supported_extended_ops_copy() instead.

Filters

slapi_filter_get_type() has been deprecated. Use slapi_filter_get_attribute_type() instead, which works for all simple filter choices.

Internal Operations

Table 2-3 lists deprecated and replacement functions for handling internal operations.

Table 2-3 Replacement Functions for Handling Internal Operations

Deprecated Function	Replacement Function
slapi_add_entry_internal()	slapi_add_internal_pb()
slapi_add_internal()	slapi_add_internal_pb()
slapi_delete_internal()	slapi_delete_internal_pb()
slapi_modify_internal()	slapi_modify_internal_pb()
slapi_modrdn_internal()	slapi_modrdn_internal_pb()
slapi_search_internal()	slapi_search_internal_pb()
slapi_search_internal_callback()	slapi_search_internal_callback_pb()

Refer to the *Sun ONE Directory Server Plug-In API Reference* for descriptions of the replacement functions.

Logging

Table 2-4 lists deprecated and replacement functions for logging.

Table 2-4 Replacement Functions for Logging

Deprecated Function	Replacement Function
`slapi_log_error()`	`slapi_log_error_ex()`
	`slapi_log_info_ex()`
	`slapi_log_warning_ex()`
`slapi_is_loglevel_set()`	none available

Refer to “Getting Started With Directory Server Plug-Ins,” on page 51 and all example code delivered with the product for demonstrations of how to use the new logging interface.

Parameter Block Arguments

Table 2-5 lists deprecated and replacement parameter block arguments.

Table 2-5 Replacement Parameter Block Arguments

Deprecated Argument	Replacement Argument
`SLAPI_CHANGENUMBER`	none available
`SLAPI_CONN_AUTHTYPE`	`SLAPI_CONN_AUTHMETHOD`
`SLAPI_CONN_CLIENTIP`	`SLAPI_CONN_CLIENTNETADDR`
`SLAPI_CONN_SERVERIP`	`SLAPI_CONN_SERVERNETADDR`
`SLAPI_LOG_OPERATION`	none available
`SLAPI_PLUGIN_DB_ABANDON_FN`	none (`database` plug-in type deprecated)
`SLAPI_PLUGIN_DB_ABORT_FN`	none (`database` plug-in type deprecated)
`SLAPI_PLUGIN_DB_ADD_FN`	none (`database` plug-in type deprecated)
`SLAPI_PLUGIN_DB_ARCHIVE2DB_FN`	none (`database` plug-in type deprecated)
`SLAPI_PLUGIN_DB_BEGIN_FN`	none (`database` plug-in type deprecated)
`SLAPI_PLUGIN_DB_BIND_FN`	none (`database` plug-in type deprecated)

Table 2-5 Replacement Parameter Block Arguments *(Continued)*

Deprecated Argument	Replacement Argument
SLAPI_PLUGIN_DB_COMMIT_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_COMPARE_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_CONFIG_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_DB2ARCHIVE_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_DB2INDEX_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_DB2LDIF_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_DELETE_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_ENTRY_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_ENTRY_RELEASE_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_FLUSH_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_FREE_RESULT_SET_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_LDIF2DB_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_MODIFY_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_MODRDN_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_NEXT_SEARCH_ENTRY_EXT_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_NEXT_SEARCH_ENTRY_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_NO_ACL	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_REFERRAL_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_RESULT_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_SEARCH_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_SEQ_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_SIZE_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_TEST_FN	none (database plug-in type deprecated)
SLAPI_PLUGIN_DB_UNBIND_FN	none (database plug-in type deprecated)
SLAPI_REQUESTOR_ISUPDATEDN	none available

Password Handling

slapi_pw_find() has been deprecated. Use slapi_pw_find_valueset() instead.

SASL Binds

`slapi_get_supported_saslmechanisms()` is not thread-safe and has been deprecated. Use `slapi_get_supported_saslmechanisms_copy()` instead.

New Features

This section summarizes features added since the previous release. It does not include features reserved for internal use.

Plug-In API Version 3

The header file `slapi-plugin.h` identifies the current plug-in API as version 3. Plug-ins supporting API version 3 indicate the version in their description using `SLAPI_PLUGIN_VERSION_03` or currently `SLAPI_PLUGIN_CURRENT_VERSION`.

Plug-In Types

A variety of new plug-in types have been created, many for internal use. Plug-in types not reserved for internal use are supported. If your preferred plug-in type lacks documentation, do not hesitate to offer your feedback through the `Send Comments` link at `http://docs.sun.com/`.

Plug-In Configuration Entries

Configuration entries rather than directives are now used to configure Directory Server plug-ins. Refer to “Plugging Libraries into Directory Server,” on page 61 for details on using configuration entries to load and configure plug-ins.

Use of NSPR 4.x

The Netscape Portable Runtime (NSPR) API allows compliant applications to use system facilities such as threads, thread synchronization, I/O, interval timing, atomic operations, and several other low-level services in a platform-independent manner.

For information on the version of NSPR used at the time of this writing, refer to:

http://www.mozilla.org/projects/nspr/release-notes/nspr412.html

Attributes

Table 2-6 lists new flags defined for use with existing functions for handling attributes.

Table 2-6 New Flags for Handling Attributes

Function	Flags Added
slapi_attr_get_flags()	SLAPI_ATTR_FLAG_STD_ATTR
	SLAPI_ATTR_FLAG_OBSOLETE
	SLAPI_ATTR_FLAG_COLLECTIVE
	SLAPI_ATTR_FLAG_NOUSERMOD
slapi_attr_type_cmp()	SLAPI_TYPE_CMP_BASE
	SLAPI_TYPE_CMP_EXACT
	SLAPI_TYPE_CMP_SUBTYPE

The old flags are still available. Refer to comments in slapi-plugin.h for hints on how to use the flags.

The following functions have been added to handle attributes and their values.

slapi_attr_add_value()

slapi_attr_first_value()

slapi_attr_get_numvalues()

slapi_attr_get_valueset()

slapi_attr_init()

slapi_attr_next_value()

slapi_attr_set_valueset()

slapi_attr_syntax_normalize()

slapi_attr_types_equivalent()

Refer to the *Sun ONE Directory Server Plug-In API Reference* for descriptions.

Backends

The following functions have been added to handle `Slapi_Backend` structures.

`slapi_be_exist()`

`slapi_be_get_name()`

`slapi_be_get_readonly()`

`slapi_be_getsuffix()`

`slapi_be_gettype()`

`slapi_be_is_flag_set()`

`slapi_be_issuffix()`

`slapi_be_logchanges()`

`slapi_be_private()`

`slapi_be_select()`

`slapi_be_select_by_instance_name()`

`slapi_free_suffix_list()`

`slapi_get_first_backend()`

`slapi_get_next_backend()`

`slapi_get_suffix_list()`

`slapi_is_root_suffix()`

Refer to the *Sun ONE Directory Server Plug-In API Reference* for descriptions.

Controls

The following functions have been added.

`slapi_build_control()`

`slapi_build_control_from_berval()`

`slapi_dup_control()`

Refer to the *Sun ONE Directory Server Plug-In API Reference* for descriptions.

Data Structures

The following data are now defined in `slapi-plugin.h` to wrap key objects used by Directory Server plug-ins.

`Slapi_Attr`

`Slapi_Backend`

`Slapi_ComponentId`

`Slapi_Connection`

`Slapi_DN`

`Slapi_Entry`

`Slapi_Filter`

`Slapi_MatchingRuleEntry`

`Slapi_Mod`

`Slapi_Mods`

`Slapi_Operation`

`Slapi_PBlock`

`Slapi_RDN`

`Slapi_Value`

`Slapi_ValueSet`

Refer to the *Sun ONE Directory Server Plug-In API Reference* for descriptions.

Distinguished Names (DNs)

The following functions have been added to handle `Slapi_DN` structures.

`slapi_dn_normalize_to_end()`

`slapi_dn_plus_rdn()`

`slapi_moddn_get_newdn()`

`slapi_sdn_compare()`

`slapi_sdn_copy()`

`slapi_sdn_done()`

`slapi_sdn_dup()`

`slapi_sdn_free()`

```
slapi_sdn_get_backend_parent()
slapi_sdn_get_dn()
slapi_sdn_get_ndn()
slapi_sdn_get_ndn_len()
slapi_sdn_get_parent()
slapi_sdn_get_rdn()
slapi_sdn_is_rdn_component()
slapi_sdn_isempty()
slapi_sdn_isgrandparent()
slapi_sdn_isparent()
slapi_sdn_issuffix()
slapi_sdn_new()
slapi_sdn_new_dn_byref()
slapi_sdn_new_dn_byval()
slapi_sdn_new_dn_passin()
slapi_sdn_new_ndn_byref()
slapi_sdn_new_ndn_byval()
slapi_sdn_scope_test()
slapi_sdn_set_dn_byref()
slapi_sdn_set_dn_byval()
slapi_sdn_set_dn_passin()
slapi_sdn_set_ndn_byref()
slapi_sdn_set_ndn_byval()
slapi_sdn_set_parent()
slapi_sdn_set_rdn()
```

Refer to the *Sun ONE Directory Server Plug-In API Reference* for descriptions.

Entries

Table 2-7 lists new flags defined for use with existing functions for handling entries.

Table 2-7 New Flags for Handling Entries

Function	Flags Added
slapi_str2entry()	SLAPI_STR2ENTRY_TOMBSTONE_CHECK
	SLAPI_STR2ENTRY_IGNORE_STATE
	SLAPI_STR2ENTRY_INCLUDE_VERSION_STR
	SLAPI_STR2ENTRY_EXPAND_OBJECTCLASSES
	SLAPI_STR2ENTRY_NOT_WELL_FORMED_LDIF

The old flags are still available. Refer to comments in slapi-plugin.h for hints on how to use the flags.

The following functions have been added to handle entries.

slapi_entry2str_with_options()

slapi_entry_add_string()

slapi_entry_add_value()

slapi_entry_attr_add()

slapi_entry_attr_get_long()

slapi_entry_attr_get_uint()

slapi_entry_attr_get_ulong()

slapi_entry_attr_remove()

slapi_entry_attr_set_int()

slapi_entry_attr_set_uint()

slapi_entry_attr_set_ulong()

slapi_entry_delete_string()

slapi_entry_get_dn_const()

slapi_entry_get_ndn()

slapi_entry_get_sdn()

slapi_entry_get_sdn_const()

slapi_entry_get_uniqueid()

slapi_entry_has_children()

slapi_entry_init()

`slapi_entry_set_sdn()`

`slapi_entry_size()`

`slapi_is_rootdse()`

Refer to the *Sun ONE Directory Server Plug-In API Reference* for descriptions.

Filters

The following functions have been added to handle `Slapi_Filter` structures.

`slapi_filter_apply()`

`slapi_filter_compare()`

`slapi_filter_get_attribute_type()`

`slapi_filter_has_extension()`

`slapi_filter_test_simple()`

`slapi_find_matching_paren()`

`slapi_vattr_filter_test()`

Refer to the *Sun ONE Directory Server Plug-In API Reference* for descriptions.

Internal Operations

The following new functions are intended for use with the updated interface.

`slapi_add_entry_internal_set_pb()`

`slapi_add_internal_set_pb()`

`slapi_delete_internal_set_pb()`

`slapi_modify_internal_set_pb()`

`slapi_rename_internal_set_pb()`

`slapi_search_internal_get_entry()`

`slapi_search_internal_set_pb()`

Refer to “Performing Internal Operations,” on page 135 for a demonstration of how to use the updated interface.

Memory Management

Table 2-8 lists new functions that have been added to manage memory.

Table 2-8 New Functions for Managing Memory

Data Structure	Memory Management Functions
`char *`	`slapi_ch_array_free()`
	`slapi_ch_free_string()`
`Slapi_Attr`	`slapi_attr_dup()`
	`slapi_attr_free()`
	`slapi_attr_new()`
`Slapi_DN`	`slapi_sdn_dup()`
	`slapi_sdn_free()`
	`slapi_sdn_new()`
	`slapi_sdn_new_dn_byref()`
	`slapi_sdn_new_dn_byval()`
	`slapi_sdn_new_dn_passin()`
	`slapi_sdn_new_ndn_byref()`
	`slapi_sdn_new_ndn_byval()`
`Slapi_Filter`	`slapi_filter_free()`
`Slapi_MatchingRuleEntry`	`slapi_matchingrule_free()`
	`slapi_matchingrule_new()`
`Slapi_Mod`	`slapi_mod_free()`
	`slapi_mod_new()`
`Slapi_Mods`	`slapi_mods_free()`
	`slapi_mods_new()`
`Slapi_PBlock`	`slapi_pblock_destroy()`
	`slapi_pblock_new()`

Table 2-8 New Functions for Managing Memory *(Continued)*

Data Structure	Memory Management Functions
Slapi_RDN	slapi_rdn_free()
	slapi_rdn_new()
	slapi_rdn_new_dn()
	slapi_rdn_new_sdn()
	slapi_rdn_new_rdn()
Slapi_Value	slapi_valuearray_free()
	slapi_value_dup()
	slapi_value_free()
	slapi_value_new()
	slapi_value_new_berval()
	slapi_value_new_value()
	slapi_value_new_string()
	slapi_value_new_string_passin()
Slapi_ValueSet	slapi_valueset_free()
	slapi_valueset_new()

Refer to the *Sun ONE Directory Server Plug-In API Reference* for descriptions.

Modification Structures

The following functions have been added to handle Slapi_Mod structures.

slapi_mod_add_value()

slapi_mod_done()

slapi_mod_dump()

slapi_mod_free()

slapi_mod_get_first_value()

slapi_mod_get_next_value()

slapi_mod_get_num_values()

slapi_mod_get_operation()

slapi_mod_get_type()

slapi_mod_init()

slapi_mod_init_byref()

slapi_mod_init_byval()

slapi_mod_init_passin()

slapi_mod_isvalid()

slapi_mod_new()

slapi_mod_remove_value()

slapi_mod_set_operation()

slapi_mod_set_type()

The following functions have been added to handle Slapi_Mods structures.

slapi_entry2mods()

slapi_mods2entry()

slapi_mods_add()

slapi_mods_add_ldapmod()

slapi_mods_add_mod_values()

slapi_mods_add_modbvps()

slapi_mods_add_smod()

slapi_mods_add_string()

slapi_mods_done()

slapi_mods_dump()

slapi_mods_free()

slapi_mods_get_first_smod()

slapi_mods_get_next_mod()

slapi_mods_get_next_smod()

slapi_mods_get_num_mods()

slapi_mods_init()

slapi_mods_init_byref()

slapi_mods_init_passin()

slapi_mods_insert_after()

slapi_mods_insert_at()

`slapi_mods_insert_before()`

`slapi_mods_insert_smod_at()`

`slapi_mods_insert_smod_before()`

`slapi_mods_iterator_backone()`

`slapi_mods_new()`

`slapi_mods_remove()`

Refer to the *Sun ONE Directory Server Plug-In API Reference* for descriptions.

Object Extensions

Object extensions offer a new way of passing data through Directory Server from plug-in to plug-in. The following macros define extensible objects.

`SLAPI_EXT_CONNECTION`

`SLAPI_EXT_OPERATION`

`SLAPI_EXT_ENTRY`

`SLAPI_EXT_MTNODE`

The following new functions and associated constructor and destructor callbacks comprise the object extension interface.

`slapi_extension_constructor_fnptr`

`slapi_extension_destructor_fnptr`

`slapi_get_object_extension()`

`slapi_register_object_extension()`

`slapi_set_object_extension()`

Refer to the *Sun ONE Directory Server Plug-In API Reference* for descriptions.

Operations

`slapi_op_get_type()` and `slapi_op_is_flag_set()` have been added.

Parameter Block Arguments

The following parameter block arguments are new.

SLAPI_CLIENT_DNS

SLAPI_CONFIG_DIRECTORY

SLAPI_CONN_CERT

SLAPI_CONN_CLIENTNETADDR

SLAPI_CONN_IS_REPLICATION_SESSION

SLAPI_CONN_IS_SSL_SESSION

SLAPI_CONN_SERVERNETADDR

SLAPI_CONTROLS_ARG

SLAPI_DESTROY_CONTENT

SLAPI_IS_INTERNAL_OPERATION

SLAPI_IS_REPLICATED_OPERATION

SLAPI_ORIGINAL_TARGET

SLAPI_ORIGINAL_TARGET_DN

SLAPI_PLUGIN_ENTRY_FETCH_FUNC

SLAPI_PLUGIN_ENTRY_STORE_FUNC

SLAPI_PLUGIN_IDENTITY

SLAPI_PLUGIN_INTERNAL_POST_ADD_FN

SLAPI_PLUGIN_INTERNAL_POST_DELETE_FN

SLAPI_PLUGIN_INTERNAL_POST_MODIFY_FN

SLAPI_PLUGIN_INTERNAL_POST_MODRDN_FN

SLAPI_PLUGIN_INTERNAL_PRE_ADD_FN

SLAPI_PLUGIN_INTERNAL_PRE_DELETE_FN

SLAPI_PLUGIN_INTERNAL_PRE_MODIFY_FN

SLAPI_PLUGIN_INTERNAL_PRE_MODRDN_FN

SLAPI_PLUGIN_POSTSTART_FN

SLAPI_PLUGIN_PWD_STORAGE_SCHEME_CMP_FN

SLAPI_PLUGIN_PWD_STORAGE_SCHEME_DB_PWD

SLAPI_PLUGIN_PWD_STORAGE_SCHEME_DEC_FN

SLAPI_PLUGIN_PWD_STORAGE_SCHEME_ENC_FN

SLAPI_PLUGIN_PWD_STORAGE_SCHEME_NAME

SLAPI_PLUGIN_PWD_STORAGE_SCHEME_USER_PWD

SLAPI_RESULT_CODE

SLAPI_RESULT_MATCHED

SLAPI_RESULT_TEXT

Refer to the *Sun ONE Directory Server Plug-In API Reference* for descriptions.

Relative Distinguished Names (RDNs)

The following functions have been added to handle Slapi_RDN structures.

slapi_rdn_add()

slapi_rdn_compare()

slapi_rdn_contains()

slapi_rdn_contains_attr()

slapi_rdn_done()

slapi_rdn_free()

slapi_rdn_get_first()

slapi_rdn_get_index()

slapi_rdn_get_index_attr()

slapi_rdn_get_next()

slapi_rdn_get_nrdn()

slapi_rdn_get_num_components()

slapi_rdn_get_rdn()

slapi_rdn_init()

slapi_rdn_init_dn()

slapi_rdn_init_rdn()

slapi_rdn_init_sdn()

slapi_rdn_isempty()

slapi_rdn_new()

slapi_rdn_new_dn()

slapi_rdn_new_rdn()

slapi_rdn_new_sdn()

slapi_rdn_remove()

`slapi_rdn_remove_attr()`

`slapi_rdn_remove_index()`

`slapi_rdn_set_dn()`

`slapi_rdn_set_rdn()`

`slapi_rdn_set_sdn()`

`slapi_sdn_add_rdn()`

Refer to the *Sun ONE Directory Server Plug-In API Reference* for descriptions.

UTF8 Encoding

The following functions have been added to handle UTF8 encoding and decoding.

`slapi_has8thBit()`

`slapi_UTF8CASECMP()`

`slapi_UTF8ISLOWER()`

`slapi_UTF8ISUPPER()`

`slapi_UTF8NCASECMP()`

`slapi_UTF8STRTOLOWER()`

`slapi_UTF8STRTOUPPER()`

`slapi_UTF8TOLOWER()`

`slapi_UTF8TOUPPER()`

Refer to the *Sun ONE Directory Server Plug-In API Reference* for descriptions.

Values

The following functions have been added to handle `Slapi_Value` structures.

`slapi_value_compare()`

`slapi_value_dup()`

`slapi_value_free()`

`slapi_value_get_berval()`

`slapi_value_get_int()`

`slapi_value_get_length()`

`slapi_value_get_long()`

`slapi_value_get_string()`

`slapi_value_get_uint()`

`slapi_value_get_ulong()`

`slapi_value_init()`

`slapi_value_init_berval()`

`slapi_value_init_string()`

`slapi_value_init_string_passin()`

`slapi_value_new()`

`slapi_value_new_berval()`

`slapi_value_new_string()`

`slapi_value_new_string_passin()`

`slapi_value_new_value()`

`slapi_value_set()`

`slapi_value_set_berval()`

`slapi_value_set_int()`

`slapi_value_set_string()`

`slapi_value_set_string_passin()`

`slapi_value_set_value()`

`slapi_valuearray_free()`

Refer to the *Sun ONE Directory Server Plug-In API Reference* for descriptions.

Value Sets

The following functions have been added to handle `Slapi_Valueset` structures.

`slapi_valueset_add_value()`

`slapi_valueset_count()`

`slapi_valueset_done()`

`slapi_valueset_find()`

`slapi_valueset_first_value()`

`slapi_valueset_free()`

`slapi_valueset_init()`

`slapi_valueset_new()`

`slapi_valueset_next_value()`

`slapi_valueset_set_from_smod()`

`slapi_valueset_set_valueset()`

Refer to the *Sun ONE Directory Server Plug-In API Reference* for descriptions.

Virtual Attributes

The following functions have been added to handle virtual attributes.

`slapi_entry_vattr_find()`

`slapi_vattr_attr_free()`

`slapi_vattr_attrs_free()`

`slapi_vattr_is_registered()`

`slapi_vattr_list_attrs()`

`slapi_vattr_values_free()`

`slapi_vattr_values_get_ex()`

Refer to the *Sun ONE Directory Server Plug-In API Reference* for descriptions.

Chapter 3

Getting Started With Directory Server Plug-Ins

This chapter provides an introduction to creating Directory Server plug-ins and enabling the server to use them. It covers the mechanics of writing and compiling plug-ins, configuring your Directory Server to recognize and load plug-ins, and generating log entries. It begins with a minimal but complete plug-in example to give you an idea of how the pieces fit together.

TIP You can find complete code examples including all code covered here under *ServerRoot*`/plugins/slapd/slapi/examples/`.

The location of *ServerRoot* depends on how Directory Server was installed. Refer to "Default Paths and Filenames," on page 14 for details.

If you maintain plug-ins developed for a 4.x release of Directory Server, refer to Chapter 2, "What's New," for information about what has changed since that release.

An Example Plug-In

This section demonstrates a plug-in that logs a famous greeting. You may want to perform the steps outlined here to appreciate the concepts covered in this chapter.

Find the Code

On the host where the directory is installed, have a look in the *ServerRoot*/plugins/slapd/slapi/examples/ directory. The example code covered in this section is *ServerRoot*/plugins/slapd/slapi/examples/hello.c.

Review the Plug-In

Code Example 3-1 logs Hello, World! at Directory Server startup.

Code Example 3-1 Hello, World! Plug-In (hello.c)

```
#include "slapi-plugin.h"

Slapi_PluginDesc desc = {
    "Hello, World",                  /* plug-in identifier      */
    "Sun Microsystems, Inc.",        /* vendor name             */
    "5.2",                           /* plug-in revision number */
    "My first plug-in"               /* plug-in description     */
};

/* Log a greeting at server startup if info logging is on for plug-ins */
int
hello()
{
    slapi_log_info_ex(
        SLAPI_LOG_INFO_AREA_PLUGIN,    /* Log if info logging is  */
        SLAPI_LOG_INFO_LEVEL_DEFAULT,  /* set for plug-ins.       */
        SLAPI_LOG_NO_MSGID,            /* No client at startup    */
        SLAPI_LOG_NO_CONNID,           /* No conn.  at startup    */
        SLAPI_LOG_NO_OPID,             /* No op.    at startup    */
        "hello() in My first plug-in", /* Origin of this message  */
        "Hello, World!\n"              /* Informational message   */
    );
    return 0;
}

/* Register the plug-in with the server */
#ifdef _WIN32
__declspec(dllexport)
#endif
int
hello_init(Slapi_PBlock * pb)
{
    int rc = 0;                      /* 0 means success         */
    rc |= slapi_pblock_set(          /* Plug-in API version     */
        pb,
        SLAPI_PLUGIN_VERSION,
        SLAPI_PLUGIN_CURRENT_VERSION
    );
    rc |= slapi_pblock_set(          /* Plug-in description     */
```

Code Example 3-1 Hello, World! Plug-In (`hello.c`) *(Continued)*

```
        pb,
        SLAPI_PLUGIN_DESCRIPTION,
        (void *) &desc
    );
    rc |= slapi_pblock_set(            /* Startup function        */
        pb,
        SLAPI_PLUGIN_START_FN,
        (void *) hello
    );
    return rc;
}
```

To log the greeting, the plug-in includes a function to be called at Directory Server startup. The plug-in also includes an initialization function to register the plug-in description, supported API version, and startup function with Directory Server.

The startup function specifies the message is from a plug-in (`SLAPI_LOG_INFO_AREA_PLUGIN`) and that it should be logged when informational logging is activated (`SLAPI_LOG_INFO_LEVEL_DEFAULT`). For this log message no client connection information is available (`SLAPI_LOG_NO_MSGID`, `SLAPI_LOG_NO_CONNID`, `SLAPI_LOG_NO_OPID`), as Directory Server calls the function at startup before any clients have connected. The function specifies where the log message originates (`"hello() in My first plug-in"`). Finally, it provides the famous log message itself.

The initialization function is named `hello_init()`. This function modifies the parameter block, `pb`, with the function `slapi_pblock_set()` to register the plug-in API version supported, the plug-in description, and the functions offered to Directory Server by this plug-in. As required for all plug-in initialization functions, `hello_init()` returns `0` on success, `-1` on failure. The function `slapi_pblock_set()` returns `0` if successful, `-1` otherwise. It is therefore not necessary to set the return code to `-1` explicitly in Code Example 3-1.

Build It

Build the plug-in as a shared object, `libtest-plugin.so` or `libtest-plugin.sl`, or dynamic link library, `testplugin.dll`, depending on your platform. For example, in the Solaris Operating Environment:

```
$ make
```

Use the `Makefile` or `Makefile64` in *ServerRoot*`/plugins/slapd/slapi/examples/` to compile and link the code for a 32-bit Directory Server. This builds and links all plug-in examples into the same library. Refer to “64-Bit Plug-In Locations,” on page 66 for details on building a plug-in when using a 64-bit Directory Server.

Plug It In

To get Directory Server to recognize our plug-in we add a configuration entry for the plug-in to the directory configuration.

Updating Directory Server Configuration

Start by copying the plug-in configuration entry in LDIF format in the comment at the top of `hello.c` to a separate file, `hello.ldif`, as shown in Code Example 3-2.

Code Example 3-2 Configuration Entry (`hello.c`)

```
dn: cn=Hello World,cn=plugins,cn=config
objectClass: top
objectClass: nsSlapdPlugin
objectClass: extensibleObject
cn: Hello World
nsslapd-pluginPath: <ServerRoot>/plugins/slapd/slapi/examples/<LibName>
nsslapd-pluginInitfunc: hello_init
nsslapd-pluginType: object
nsslapd-pluginId: Hello, World
nsslapd-pluginEnabled: on
nsslapd-pluginVersion: 5.2
nsslapd-pluginVendor: Sun Microsystems, Inc.
nsslapd-pluginDescription: My first plug-in
```

Change the value of `nsslapd-pluginPath` to correspond to the absolute path to the plug-in library. Directory Server requires an absolute path, not a relative path.

Notice that `nsslapd-pluginInitfunc` identifies the plug-in initialization function.

With Directory Server running, add the plug-in configuration entry to the directory:

```
$ ldapmodify -a -p port -D "cn=Directory Manager" -w password -f hello.ldif
```

With Directory Server still running, turn on logging for plug-in informational messages as shown in Code Example 3-3.

Code Example 3-3 Turning on Informational Logging for Plug-Ins

```
$ ldapmodify -p port -D "cn=Directory Manager" -w password
dn: cn=config
changetype: modify
replace: nsslapd-infolog-area
nsslapd-infolog-area: 65536
^D
```

65536 is equivalent to 0x10000, the value of SLAPI_LOG_INFO_AREA_PLUGIN defined in slapi-plugin.h. If your plug-in informational log messages use log level SLAPI_LOG_INFO_LEVEL_EXTRA instead of SLAPI_LOG_INFO_LEVEL_DEFAULT, you must also set the value of nsslapd-infolog-level on cn=config to a non-zero value such as 1 to view such messages in the log.

Restarting Directory Server

After changing the Directory Server configuration by adding the plug-in configuration entry to the directory and turning on logging for plug-in informational messages, you must restart Directory Server for the server to register the plug-in.

```
$ ServerRoot/slapd-serverID/restart-slapd
```

Checking the Log

Once Directory Server has started, search the error log, *ServerRoot*/slapd-*serverID*/logs/errors, for Hello, World! You should find an entry similar to:

```
[16/Sep/2002:12:44:19 +0200] - INFORMATION - hello() in My first plug-in - conn=-1
op=-1 msgId=-1 -  Hello, World!
```

Finally, turn off informational logging for plug-in messages to avoid slowing down your Directory Server.

Writing Directory Server Plug-Ins

This section focuses on the basics of coding a Directory Server plug-in. It covers the key tasks of using the right header file to specify use of the plug-in API, including an initialization function called by Directory Server when registering plug-in functionality, setting parameter block values, and registering plug-in functions.

Include the Header File for the Plug-In API

In *ServerRoot*/plugins/slapd/slapi/include/slapi-plugin.h, the plug-in API is defined. Observe that the header file includes ldap.h, the entry point for the standard and extended LDAP C APIs, and includes ldap_msg.h, the list of error message identifiers used by Directory Server.

In general, interfaces exposed by Directory Server are specified under *ServerRoot*/plugins/slapd/slapi/include/. For details about specific features of the API, refer to the *Sun ONE Directory Server Plug-In API Reference*.

To use the API, include slapi-plugin.h in the declaration section of your plug-in source:

```
#include "slapi-plugin.h"
```

As a rule, use appropriate macros in your Makefile or project file to tell the linker to look for header files under *ServerRoot*/plugins/slapd/slapi/include/.

Write Your Plug-In Functions

Directory Server calls plug-in functions in the context of a Directory Server operation, when a bind, add, search, modify, or delete is performed, for example. You need not export these functions, as they become available in the appropriate scope when registered with Directory Server at startup. The main body of this guide covers writing such functions.

A plug-in function prototype looks similar to that of any other locally used function. Many plug-in functions are passed a parameter block. For example:

```
int prebind_auth(Slapi_PBlock * pb); /* External authentication. */
```

You may also use additional helper functions not registered with Directory Server. This guide does not cover additional helper functions, but some of the sample plug-ins delivered with the product include such functions.

Use Appropriate Return Codes

In general, return 0 from your plug-in functions when they complete successfully. For some functions that search for matches, 0 means a successful match, -1 means no match found. For pre-operation functions, 0 means Directory Server should continue processing the operation. When you want the server to stop processing the operation after your pre-operation function completes, return a positive value such as 1.

When plug-in functions do not complete successfully, send a result to the client if appropriate, log an error message, and return a non-zero value, such as the LDAP result code from *ServerRoot*/plugins/slapd/slapi/include/ldap-standard.h, for example.

Write an Initialization Function

All plug-ins must include an initialization function that registers the plug-in version compatibility, the plug-in description, plug-in functions, and any other data required by the plug-in. The initialization function takes a pointer to a Slapi_PBlock structure as a parameter. Refer to Code Example 3-1 on page 52 for an example.

The initialization function returns 0 if everything registers successfully and -1 if registration fails for any configuration information or function. A return code of -1 from a plug-in initialization function prevents the server from starting.

Directory Server may call the initialization function more than once during startup.

Set Configuration Information Through the Parameter Block

Recall that Directory Server passes a parameter block pointer to the plug-in initialization function. This parameter block holds configuration information and may hold other data from Directory Server. It is the structure into which you add configuration information and pointers to plug-in functions using calls to slapi_pblock_set().

TIP To read parameter values, use slapi_pblock_get(). To write parameter values, use slapi_pblock_set(). Use of other functions may crash the server.

Specifying Compatibility

Specifying compatibility with the plug-in API version as defined in slapi-plugin.h and described in the *Sun ONE Directory Server Plug-In API Reference*. If you are creating a new plug-in for example, use SLAPI_PLUGIN_CURRENT_VERSION or SLAPI_PLUGIN_VERSION_03. For example, to specify that a plug-in supports the current version, version 3, of the plug-in API:

```
slapi_pblock_set(pb,SLAPI_PLUGIN_VERSION, SLAPI_PLUGIN_VERSION_03);
```

Here, `pb` is the parameter block pointer passed to the initialization function, and `SLAPI_PLUGIN_VERSION` signifies you are setting plug-in API version compatibility in the parameter block.

Specifying the Plug-In Description

Specify the plug-in description in a `Slapi_PluginDesc` structure, as specified in `slapi-plugin.h` and described in the *Sun ONE Directory Server Plug-In API Reference*. Call `slapi_pblock_set()` to register the description:

```
slapi_pblock_set(pb, SLAPI_PLUGIN_DESCRIPTION,(void *) &desc);
```

where `desc` is a `Slapi_PluginDesc` structure containing the plug-in description. Refer to Code Example 3-1 on page 52 for an example.

Set Pointers to Functions Through the Parameter Block

Register plug-in functions according to what operation Directory Server is going to perform, is performing, or has just performed. In other words, Directory Server typically calls plug-in functions before, during, or after a standard operation.

Macros that specify when Directory Server should call plug-ins have the form `SLAPI_PLUGIN_*_FN`. As the sample plug-ins demonstrate, the macro used to register a plug-in function depends entirely on what the function is supposed to do. For example, to register a plug-in function `foo()` to be called at Directory Server startup:

```
slapi_pblock_set(pb, SLAPI_PLUGIN_START_FN, (void *) foo);
```

Here, `pb` is the parameter block pointer passed to the initialization function, and `SLAPI_PLUGIN_START_FN` signifies Directory Server calls `foo()` at startup. Note `foo()` returns `0` on success.

Where to Find Examples

Find sample plug-ins under *ServerRoot*`/plugins/slapd/slapi/examples/`.

Building Directory Server Plug-Ins

This section details how to build plug-in binaries. It deals with compilation and linking requirements for Directory Server plug-ins.

Include the Header File for the Plug-In API

The compiler needs to be able to locate the header files such as `slapi-plugin.h`. Find header files under *ServerRoot*`/plugins/slapd/slapi/include/`.

Link the Plug-In as a Shared Object or Dynamic Link Library

Link plug-ins so the plug-in library is reentrant and portable and can be shared. Code Example 3-4 shows one way of building a plug-in library for use with a 64-bit Directory Server running in the Solaris Operating Environment.

Code Example 3-4 `Makefile` for a 64-Bit Solaris Sample Plug-In Library (`Makefile64`)

```
INCLUDE_FLAGS = -I../include
CFLAGS = $(INCLUDE_FLAGS) -D_REENTRANT -KPIC -xarch=v9 -DUSE_64
LDFLAGS = -G
DIR64 = 64

OBJS = dns.o entries.o hello.o internal.o testpwdstore.o testsaslbind.o
testextendedop.o testpreop.o testpostop.o testentry.o testbind.o testgetip.o

all: MKDIR64 $(DIR64)/libtest-plugin.so

MKDIR64:
    @if [ ! -d $(DIR64) ]; then mkdir $(DIR64); fi

$(DIR64)/libtest-plugin.so: $(OBJS)
    $(LD) $(LDFLAGS) -o $@ $(OBJS)

.c.o:
    $(CC) $(CFLAGS) -c $<

clean:
    -rm -f $(OBJS) libtest-plugin.so $(DIR64)/libtest-plugin.so
```

The `CFLAGS -xarch=v9 and -DUSE_64` specify that the compiler build the library for use with a 64-bit server. Notice also that the 64-bit library is placed in a directory ending in `64/`. An extra directory is required for 64-bit plug-ins, as a 64-bit server adds the name of the extra directory to the path name when searching for the library. Refer to "64-Bit Plug-In Locations," on page 66 for further information.

Code Example 3-5 shows 32-bit equivalent of the `Makefile`.

Code Example 3-5 `Makefile` for a 32-Bit Solaris Sample Plug-In Library (`Makefile`)

```
INCLUDE_FLAGS = -I../include
CFLAGS = $(INCLUDE_FLAGS) -D_REENTRANT -KPIC
LDFLAGS = -G

OBJS = dns.o entries.o hello.o internal.o testpwdstore.o testsaslbind.o
testextendedop.o testpreop.o testpostop.o testentry.o testbind.o testgetip.o

all: libtest-plugin.so

libtest-plugin.so: $(OBJS)
    $(LD) $(LDFLAGS) -o $@ $(OBJS)

.c.o:
    $(CC) $(CFLAGS) -c $<

clean:
    -rm -f $(OBJS) libtest-plugin.so
```

Notice that both 32-bit and 64-bit versions of the plug-in library may coexist on the same host.

Where to Find Examples

Build rules are in *ServerRoot*`/plugins/slapd/slapi/examples/Makefile` and in *ServerRoot*`/plugins/slapd/slapi/examples/Makefile64`. Refer to these files for the recommended setup for compilation and linking on your platform.

Plugging Libraries into Directory Server

This section covers server plug-in configuration and explains how to load plug-ins so Directory Server initializes them and registers the functionality they provide. It touches on specifying dependencies between Directory Server plug-ins.

Create a Configuration Entry for Your Plug-In

Plug-in configuration entries specify information Directory Server requires. It uses the information to load a plug-in at Directory Server startup, to identify the plug-in to Directory Server, and to pass parameters to the plug-in at load time.

Example Plug-In Configuration Entry

Directory Server can be configured to use an Attribute Value Uniqueness plug-in, whose configuration entry is shown in Code Example 3-2, to ensure unique attribute values across multiple master servers. This plug-in is part of the Sun ONE Directory Server Resource Kit.

Code Example 3-6 Sample Configuration Entry (`dse.ldif`)

```
dn: cn=Attribute Value Uniqueness,cn=plugins,cn=config
objectClass: top
objectClass: nsSlapdPlugin
objectClass: extensibleObject
cn: Attribute Value Uniqueness
nsslapd-pluginPath: ServerRoot/lib/valueunique-plugin.so
nsslapd-pluginInitfunc: valueunique_init
nsslapd-pluginType: preoperation
nsslapd-pluginEnabled: off
nsslapd-pluginarg0: attribute=cn
nsslapd-pluginarg1: suffix=dc=example,dc=com
nsslapd-pluginarg2: dcsleaf
nsslapd-plugin-depends-on-type: database
nsslapd-pluginId: UniqueValue
nsslapd-pluginVersion: 5.2
nsslapd-pluginVendor: Sun Microsystems, Inc.
nsslapd-pluginDescription: Unique attribute values across multiple servers
```

View entries like this in *ServerRoot*`/slapd-`*serverID*`/config/dse.ldif` and in the comments of examples under *ServerRoot*`/plugins/slapd/slapi/examples/`.

Configuration Entry Attributes

Edit your plug-in configuration entry as LDIF. To complete the configuration entry, you must include at minimum the information identified as required in Table 3-1.

Table 3-1 Configuration Entry Attributes

Attribute	Description	Required/Optional
`cn`	Common name for the plug-in, corresponding to the name registered by the plug-in	Required
`nsslapd-pluginEnabled`	Status of your plug-in after Directory Server startup, either `on` or `off`	Required
`nsslapd-pluginInitFunc`	Name of the initialization function called during Directory Server startup as it appears in the plug-in code	Required
`nsslapd-pluginPath`	Full path to the library containing the plug-in, not including 64-bit directory for 64-bit plug-in libraries Refer to "64-Bit Plug-In Locations," on page 66 for details.	Required
`nsslapd-pluginType`	Plug-in type that defines the types of functions the plug-in implements Refer to "Types and Dependencies," on page 63 for details.	Required
`nsslapd-depends-on-named`	One or more plug-in dependencies on plug-in names (RDN) such as `State Change Plugin` or `Multimaster Replication Plugin` Refer to "Types and Dependencies," on page 63 for details.	Optional
`nsslapd-depends-on-type`	One or more plug-in dependencies on plug-in types such as `preoperation` or `database` Refer to "Types and Dependencies," on page 63 for details.	Optional
`nsslapd-pluginArg[0-9]+`	A parameter passed to the plug-in at Directory Server startup such as `suffix=dc=example,dc=com` Refer to "Parameters Specified in the Configuration Entry," on page 64 for details.	Optional

Table 3-1 Configuration Entry Attributes *(Continued)*

Attribute	Description	Required/Optional
`nsslapd-pluginDescription`	One-line description as specified in the `spd_description` field of the plug-in `Slapi_PluginDesc` structure	Optional
`nsslapd-pluginId`	Identifier as specified in the `spd_id` field of the plug-in `Slapi_PluginDesc` structure	Optional
`nsslapd-pluginVendor`	Vendor name as specified in the `spd_vendor` field of the plug-in `Slapi_PluginDesc` structure	Optional
`nsslapd-pluginVersion`	Version number as specified in the `spd_version` field of the plug-in `Slapi_PluginDesc` structure	Optional

The rest of this section covers types, dependencies, and parameters.

Types and Dependencies

The plug-in *type* tells Directory Server which type of plug-in functions may be registered by a plug-in. Plug-in types correspond to the principle type of operation a plug-in performs. Refer to “Server Plug-In Types,” on page 22 and “Plug-In Example Uses, By Type,” on page 26 for short explanations of plug-in types. The *Sun ONE Directory Server Plug-In API Reference* also describes plug-in types. The *Sun ONE Directory Server Plug-In API Reference* lists `nsslapd-pluginType` attribute values corresponding to the type identifiers specified in `slapi-plugin.h`.

Determining plug-in type is straightforward when you know what the plug-in must do. For example, if you write a plug-in to handle authentication of a request before Directory Server processes the bind for that request, then the type is `preoperation`. In other words, the plug-in does something to the request *prior to* the bind operation. Such a plug-in therefore registers at least a pre-bind plug-in function to process the request prior to the bind. If, on the other hand, you write a plug-in to encode passwords and compare incoming passwords with encoded passwords, then the type is `pwdstoragescheme`. Such a plug-in registers at least password encoding and compare functions. In both cases, plug-in type follows plug-in function.

Plug-in *dependencies* tell Directory Server a plug-in requires one or more other plug-ins in order to function properly. Directory Server resolves dependencies as it loads plug-ins. Therefore, Directory Server fails to register a plug-in if a plug-in it depends on cannot be registered.

Specify plug-in dependencies by name, using `nsslapd-depends-on-named`, or by type, using `nsslapd-depends-on-type`. Here, plug-in name refers to the plug-in Relative Distinguished Name (RDN) from the configuration entry. Plug-in type is a type identifier from the list of types in the *Sun ONE Directory Server Plug-In API Reference*. Both `nsslapd-depends-on-named` and `nsslapd-depends-on-type` can take multiple values.

A plug-in configuration entry may specify either kind or both kinds of dependencies. Use the configuration entry to identify dependencies on specific plug-ins by *name*. Identify dependencies on a type of plug-ins by *type*. If a dependency identifies a plug-in by name, Directory Server only registers the plug-in after it manages to register the named plug-in. If a dependency identifies a plug-in by type, Directory Server only registers the plug-in after it manages to register *all* plug-ins of the specified type.

Parameters Specified in the Configuration Entry

Your plug-in can retrieve parameters specified in the configuration entry attributes of the form `nsslapd-pluginArg[0-9]+`. If you include parameters in the configuration entry, then:

- `SLAPI_PLUGIN_ARGC` specifies the number of parameters
- `SLAPI_PLUGIN_ARGV` contains the parameters themselves

Use `slapi_pblock_get()` to retrieve the arguments as in Code Example 3-7 from *ServerRoot*`/plugins/slapd/slapi/examples/testextendedop.c`, where the configuration entry includes the following line:

```
nsslapd-pluginarg0: 1.2.3.4
```

Directory Server makes this argument, an OID, available to the plug-in through the parameter block passed to the plug-in initialization function.

Code Example 3-7 Configuration Entry with Arguments (`testextendedop.c`)

```
#include "slapi-plugin.h"

Slapi_PluginDesc expdesc = {
    "test-extendedop",                /* plug-in identifier       */
    "Sun Microsystems, Inc.",         /* vendor name              */
    "5.2",                            /* plug-in revision number  */
    "Sample extended operation plug-in"/* plug-in description      */
};

/* Register the plug-in with the server.                          */
#ifdef _WIN32
__declspec(dllexport)
#endif
```

Code Example 3-7 Configuration Entry with Arguments (`testextendedop.c`) *(Continued)*

```
int
testexop_init(Slapi_PBlock * pb)
{
    char ** argv;                       /* Args from configuration  */
    int     argc;                       /* entry for plug-in.       */
    char ** oid_list;                   /* OIDs supported           */
    int     rc = 0;                     /* 0 means success          */
    int     i;

    /* Get the arguments from the configuration entry.              */
    rc |= slapi_pblock_get(pb, SLAPI_PLUGIN_ARGV, &argv);
    rc |= slapi_pblock_get(pb, SLAPI_PLUGIN_ARGC, &argc);
    if (rc != 0) {
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
            SLAPI_LOG_INFO_LEVEL_DEFAULT,
            SLAPI_LOG_NO_MSGID,
            SLAPI_LOG_NO_CONNID,
            SLAPI_LOG_NO_OPID,
            "testexop_init in test-extendedop plug-in",
            "Could not get plug-in arguments.\n"
        );
        return (rc);
    }

    /* Extended operation plug-ins may handle a range of OIDs. */
    oid_list = (char **)slapi_ch_malloc((argc + 1) * sizeof(char *));
    for (i = 0; i < argc; ++i) {
        oid_list[i] = slapi_ch_strdup(argv[i]);
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
            SLAPI_LOG_INFO_LEVEL_DEFAULT,
            SLAPI_LOG_NO_MSGID,
            SLAPI_LOG_NO_CONNID,
            SLAPI_LOG_NO_OPID,
            "testexop_init in test-extendedop plug-in",
            "Registering plug-in for extended operation %s.\n",
            oid_list[i]
        );
    }
    oid_list[argc] = NULL;

    rc |= slapi_pblock_set(             /* Plug-in API version      */
        pb,
        SLAPI_PLUGIN_VERSION,
        SLAPI_PLUGIN_CURRENT_VERSION
    );
    rc |= slapi_pblock_set(             /* Plug-in description      */
        pb,
        SLAPI_PLUGIN_DESCRIPTION,
        (void *) &expdesc
    );
    rc |= slapi_pblock_set(             /* Extended op. handler     */
        pb,
        SLAPI_PLUGIN_EXT_OP_FN,
```

Code Example 3-7 Configuration Entry with Arguments (`testextendedop.c`) *(Continued)*

```
        (void *) test_extendedop
    );
    rc |= slapi_pblock_set(            /* List of OIDs handled     */
        pb,
        SLAPI_PLUGIN_EXT_OP_OIDLIST,
        oid_list
    );
    return (rc);
}
```

You can specify multiple parameters using multiple attributes, `nsslapd-pluginArg0`, `nsslapd-pluginArg1`, `nsslapd-pluginArg2`, and so forth.

64-Bit Plug-In Locations

Directory Server searches for plug-in libraries using the value of `nsslapd-pluginPath` in the plug-in configuration entry. 64-bit servers search for 64-bit libraries by adding a directory name to the end of the path name.

For example, if the value of `nsslapd-pluginPath` is set in the configuration entry as follows:

```
nsslapd-pluginPath: /myplugins/mylib.so
```

A 64-bit Directory Server running in Solaris Operating Environment searches for a 64-bit plug-in library named:

```
/myplugins/64/mylib.so
```

A 32-bit Directory Server searches for 32-bit plug-in libraries using the exact path specified as the value of `nsslapd-pluginPath`. A 32-bit Directory Server using the same configuration entry would search for the plug-in library named:

```
/myplugins/mylib.so
```

Directory Server fails to start when it cannot find a plug-in library specified in the configuration entry.

Modify the Directory Server Configuration

With Directory Server running, add your plug-in configuration entry using the `ldapmodify` command:

```
$ ldapmodify -a -p port -D "cn=Directory Manager" -w password -f file.ldif
```

Here, *file.ldif* contains the plug-in configuration entry. If everything works as expected, Directory Server adds your plug-in configuration entry to *ServerRoot*/slapd-*serverID*/config/dse.ldif.

TIP Edit dse.ldif carefully as errors in this file may prevent Directory Server from starting.

Furthermore, direct changes to dse.ldif only take effect if Directory Server is not running when the changes are made.

Restart Directory Server

Directory Server does not load plug-ins dynamically. Instead, you must restart the server. Directory Server then registers your plug-in at startup.

Set logging so Directory Server logs plug-in informational messages. Refer to “Set Appropriate Log Level in the Directory Server Configuration,” on page 70 for details. After setting the log level appropriately, you can have the server copy log messages to standard output:

```
$ ServerRoot/slapd-serverID/stop-slapd
$ ServerRoot/slapd-serverID/start-slapd -d 65536
```

When everything works fine, restart without the -d option or turn off plug-in log messages to avoid slowing down Directory Server. You can adjust logging levels through the Logs node under the Configuration tab of Directory Server Console, for example.

Logging Plug-In Messages

This section shows you how to generate messages in the Directory Server logs, how to set the log level such that Directory Server writes particular types of messages to the log, and then how to find the messages logged.

Three Levels of Message Severity

The plug-in API provides log functions for logging fatal error messages, warnings and information. Error messages and warnings are always logged. Logging of informational messages is turned off by default. Recall you can adjust log levels through the console or on the command line while Directory Server is running.

TIP Logging slows down Directory Server, as Directory Server waits for the system to write each log message to disk.

Use logging when you need it. Turn it off when you do not.

Refer to the *Sun ONE Directory Server Plug-In API Reference* for details on each of the logging functions.

Fatal Errors

For fatal errors, use `slapi_log_error_ex()`. In many cases, you may return `-1` after you log the message to indicate to Directory Server that a serious problem has occurred.

Code Example 3-8 Logging a Fatal Error Message

```
#include "slapi-plugin.h"
#include "example-com-error-ids.h" /* example.com unique
                                      error IDs file       */
int
foobar(Slapi_PBlock * pb)
{
    char * error_cause;
    int    apocalypse = 1;         /* Expect the worst.    */

    /* ... */

    if (apocalypse) {              /* Server to crash soon */
        slapi_log_error_ex(
            EXCOM_SERVER_MORIBUND, /* Unique error ID      */
            SLAPI_LOG_NO_MSGID,
            SLAPI_LOG_NO_CONNID,
            SLAPI_LOG_NO_OPID,
            "example.com: foobar in baz plug-in",
            "cannot write to file system: %s\n",
            error_cause
        );
        return -1;
    }
    return 0;
}
```

In Code Example 3-8, `foobar()` logs a error as Directory Server is about to crash.

TIP If the plug-ins are internationalized, use macros rather than literal strings for the last two arguments to `slapi_log_*_ex()` functions.

Warnings

For serious situations that require attention and involve messages that should always be logged, use `slapi_log_warning_ex()`.

In Code Example 3-9, `foobar()` logs a warning because the disk is nearly full.

Code Example 3-9 Logging a Warning Message

```
#include "slapi-plugin.h"
#include "example-com-warning-ids.h" /* example.com unique
                                        warning IDs file  */
int
foobar()
{
    int disk_use_percentage;

    /* ... */

    if (disk_use_percentage >= 95) {
        slapi_log_warning_ex(
            EXCOM_DISK_FULL_WARN,    /* unique warning ID */
            SLAPI_LOG_NO_MSGID,
            SLAPI_LOG_NO_CONNID,
            SLAPI_LOG_NO_OPID,
            "example.com: foobar in baz plug-in",
            "disk %.0f%% full, find more space\n",
            disk_use_percentage
        );
    }
    return 0;
}
```

Informational Messages

For purely informational or debug messages, use `slapi_log_info_ex()`. This function can be set for default logging, meaning the message is not logged when logging is turned off for plug-ins. It can also be set only when heavy logging is used, meaning the message is logged only when plug-in logging is on and the level is set to log all informational messages.

Refer to Code Example 3-1 on page 52 for an example of how to log an informational message from a plug-in.

Set Appropriate Log Level in the Directory Server Configuration

Log levels may be set for plug-in informational messages, as for a number of other Directory Server subsystems. You can set the log level through the console. You can also set the log level from the command line, using the `ldapmodify` command to change the values of the `nsslapd-infolog-area` and `nsslapd-infolog-level` attributes on the configuration object, `dn: cn=config`. Refer to "Updating Directory Server Configuration," on page 54 for an example.

Setting `nsslapd-infolog-area` to `65536` (the value of `SLAPI_LOG_INFO_AREA_PLUGIN` expressed as a decimal) turns on plug-in informational logging. Leaving `nsslapd-infolog-level` unspecified or setting it to `0` turns on information logging as specified using `SLAPI_LOG_INFO_DEFAULT`. Setting `nsslapd-infolog-level` a non-zero value turns on heavier informational logging, including messages registered with `SLAPI_LOG_INFO_LEVEL_EXTRA`.

Find Messages in the Log

The log file for errors, warnings, and informational messages is *ServerRoot*`/slapd-`*serverID*`/logs/errors`. All messages go to the same log so you can easily determine what happened in chronological order.

Messages consist of a timestamp, followed by an identifier, followed by the other information in the log message.

```
[16/Sep/2002:12:44:10 +0200] - INFORMATION - Backend Database - conn=-1 op=-1
msgId=-1 -  Checkpointing database ...
```

Use the identifiers to filter what you do not need.

Chapter 4

Working with Entries

This chapter covers plug-in API features for handling directory entries, attributes, attribute values, and Distinguished Names (DNs). It also deals with converting entries to and from LDAP Data Interchange Format (LDIF) strings, and with checking whether entries comply with LDAP schema.

Code excerpts used in this chapter can be found in the *ServerRoot*/plugins/slapd/slapi/examples/entries.c sample plug-in and the *ServerRoot*/plugins/slapd/slapi/examples/dns.c sample plug-in.

Refer to the *Sun ONE Directory Server Plug-In API Reference* for complete reference information concerning the plug-in API functions used in this chapter.

Creating Entries

This section demonstrates how to create a new entry in the directory. Create a new entry either by allocating space and creating a wholly new entry, or by duplicating an existing entry and modifying its values.

New Entries

Create a wholly new entry using slapi_entry_alloc() to allocate the memory required, as shown in Code Example 4-1.

Code Example 4-1 Creating a New Entry (entries.c)

```
#include "slapi-plugin.h"

int
test_ldif()
{
```

Code Example 4-1 Creating a New Entry (`entries.c`) *(Continued)*

```
    Slapi_Entry * entry = NULL;        /* Entry to hold LDIF      */

    /* Allocate the Slapi_Entry structure.                        */
    entry = slapi_entry_alloc();

    /* Add code that fills the Slapi_Entry structure.             */

    /* Add code that uses the Slapi_Entry structure.              */

    /* Release memory allocated for the entry.                    */
    slapi_entry_free(entry);

    return (0);
}
```

Copies of Entries

Create a copy of an entry with `slapi_entry_dup()` as shown in Code Example 4-2.

Code Example 4-2 Copying an Existing Entry (`entries.c`)

```
#include "slapi-plugin.h"

int
test_create_entry()
{
    Slapi_Entry * entry = NULL;        /* Original entry          */
    Slapi_Entry * ecopy = NULL;        /* Copy entry              */

    entry = slapi_entry_alloc();

    /* Add code that fills the new entry before making a copy.    */

    ecopy = slapi_entry_dup(entry);

    /* Add code that uses the Slapi_Entry structure.              */

    /* Release memory allocated for the entries.                  */
    slapi_entry_free(entry);
    slapi_entry_free(ecopy);

    return (0);
}
```

Converting To and From LDIF Representations

This section shows you how to use the plug-in API to convert from entries represented in LDIF to `Slapi_Entry` structures and from `Slapi_Entry` structures to LDIF strings.

LDIF files appear as a series of human-readable representations of directory entries, optionally with access control instructions as well. Entries represented in LDIF start with a line for the distinguished name, and continue with optional lines for the attributes as shown in Code Example 4-3.

Code Example 4-3 LDIF Syntax Representing an Entry

```
dn:[:] dn-value\n
[attribute:[:] value\n]
[attribute:[:] value\n]
[single-space continued-value\n]*
...
```

A double colon, `::`, indicates that the *value* is base-64 encoded. This makes it possible, for example, to have a line break in the midst of an attribute value.

As shown in Code Example 4-3, LDIF lines can be folded by leaving a single space at the beginning of the continued line.

Sample LDIF files can be found under *ServerRoot*`/slapd-`*serverID*`/ldif/`. Code Example 4-4 shows two entries represented in LDIF.

Code Example 4-4 LDIF Representation of Two Entries (`Example-Plugin.ldif`)

```
dn: ou=People,dc=example,dc=com
objectclass: top
objectclass: organizationalUnit
ou: People
aci: (target = "ldap:///ou=People,dc=example,dc=com")
 (targetattr = "userpassword")(version 3.0; acl
 "Allow people to change their password";
 allow(write)(userdn = "ldap:///self");)

dn: uid=yyorgens,ou=People,dc=example,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
uid: yyorgens
givenName: Yolanda
```

Code Example 4-4 LDIF Representation of Two Entries (`Example-Plugin.ldif`) *(Continued)*

```
sn: Yorgenson
cn: Yolanda Yorgenson
mail: yyorgens@example.com
userPassword: yyorgens
secretary: uid=bcubbins,ou=People,dc=example,dc=com
```

Refer to the *Sun ONE Directory Server Reference Manual* for details on LDIF syntax.

Converting an LDIF String to a Slapi_Entry Structure

This can be done using `slapi_str2entry()`. The function takes as its two arguments the string to convert and an `int` holding flags of the form `SLAPI_STR2ENTRY_*` in `slapi-plugin.h`. It returns a pointer to a `Slapi_Entry` if successful, `NULL` otherwise, as in Code Example 4-5.

Code Example 4-5 Converting To and From LDIF Strings (`entries.c`)

```
#include "slapi-plugin.h"

#define LDIF_STR "dn: dc=example,dc=com\nobjectclass: \
    top\nobjectclass: domain\ndc: example\n"

int
test_ldif()
{
    char        * ldif  = NULL;        /* Example LDIF string     */
    Slapi_Entry * entry = NULL;        /* Entry to hold LDIF      */
    char        * str   = NULL;        /* String to hold entry    */
    int           len;                 /* Length of entry as LDIF */

    /* LDIF to Slapi_Entry                                        */
    entry = slapi_entry_alloc();
    ldif  = slapi_ch_strdup(LDIF_STR);
    entry = slapi_str2entry(ldif, SLAPI_STR2ENTRY_ADDRDNVALS);
    slapi_ch_free_string(&ldif);
    if (entry == NULL) return (-1);

    /* Slapi_Entry to LDIF                                        */
    str = slapi_entry2str(entry, &len);
    if (str == NULL) return (-1);
    slapi_log_info_ex(
        SLAPI_LOG_INFO_AREA_PLUGIN,
        SLAPI_LOG_INFO_LEVEL_DEFAULT,
        SLAPI_LOG_NO_MSGID,
        SLAPI_LOG_NO_CONNID,
        SLAPI_LOG_NO_OPID,
        "test_ldif in test-entries plug-in",
        "\nOriginal entry:\n%sEntry length: %d\n", str, len
```

Code Example 4-5 Converting To and From LDIF Strings (`entries.c`) *(Continued)*

```
    );
    slapi_entry_free(entry);
    return (0);
}
```

Here `SLAPI_STR2ENTRY_ADDRDNVALS` adds any missing Relative Distinguished Name (RDN) values, as specified in `slapi-plugin.h` where supported flags for `slapi_str2entry()` are listed.

Converting a Slapi_Entry Structure to an LDIF String

This can be done using `slapi_entry2str()`. This function takes as its two arguments the entry to convert and an `int` to hold the length of the string returned. It returns a `char *` to the LDIF if successful, `NULL` otherwise, as in Code Example 4-5.

Getting Entry Attributes and Attribute Values

This section demonstrates how to iterate through real attributes of an entry. Real attributes contrast with virtual attributes generated by Directory Server for advanced entry management using roles and Class of Service (CoS). You may use this technique when looking through the list of attributes belonging to an entry. If you know the entry contains a particular attribute, `slapi_entry_attr_find()` returns a pointer to the attribute directly.

Iteration through attribute values can be done using `slapi_attr_first_value()` and `slapi_attr_next_value()` together as shown in Code Example 4-6.

Code Example 4-6 Iterating Through Attributes of an Entry (`testpreop.c`)

```
#include "slapi-plugin.h"

int
testpreop_add(Slapi_PBlock * pb)
{
    Slapi_Entry * entry;              /* Entry to add           */
    Slapi_Attr  * attribute;          /* Entry attributes       */
    Slapi_Value * value;              /* Attribute values       */
    int         i, rc = 0;
```

Code Example 4-6 Iterating Through Attributes of an Entry (`testpreop.c`) *(Continued)*

```
    char        * tmp;                /* Holder for new desc.    */
    const char  * string;             /* Holder for current desc.*/

    rc |= slapi_pblock_get(pb, SLAPI_ADD_ENTRY, &entry);

    rc |= slapi_entry_attr_find(entry, "description", &attribute);
    if (rc == 0) {                /* Found a description, so...  */
        for (                         /* ...loop for value...    */
            i = slapi_attr_first_value(attribute, &value);
            i != -1;
            i = slapi_attr_next_value(attribute, i, &value)
        ) {                           /* ...prepend "ADD ".      */
            string = slapi_value_get_string(value);
            tmp    = slapi_ch_malloc(5+strlen(string));
            strcpy(tmp, "ADD ");
            strcat(tmp+4, string);
            slapi_value_set_string(value, tmp);
            slapi_ch_free((void **)&tmp);
        }
    }
    return 0;
}
```

When working with virtual attributes such as those used for CoS, you may use `slapi_vattr_list_attrs()` to obtain the complete list of both real and virtual attributes belonging to an entry.

Adding and Removing Attribute Values

This section demonstrates how to add and remove attributes and their values.

Adding Attribute Values

If the attribute name and value are strings, you may be able to use `slapi_entry_add_string()`, as in Code Example 4-7 that builds an entry by first setting the DN, then adding attributes with string values.

Code Example 4-7 Adding String Attribute Values (`entries.c`)

```
#include "slapi-plugin.h"

int
test_create_entry()
{
```

Code Example 4-7 Adding String Attribute Values (`entries.c`) *(Continued)*

```
    Slapi_Entry * entry = NULL;        /* Original entry          */

    entry = slapi_entry_alloc();
    slapi_entry_set_dn(entry, slapi_ch_strdup("dc=example,dc=com"));
    slapi_entry_add_string(entry, "objectclass", "top");
    slapi_entry_add_string(entry, "objectclass", "domain");
    slapi_entry_add_string(entry, "dc", "example");

    /* Add code using the entry you created.....                */

    slapi_entry_free(entry);

    return (0);
}
```

When working with values that already exist or are not strings, you may use `slapi_entry_add_values_sv()` to add new values to an attribute of an entry, or `slapi_entry_attr_merge_sv()` to merge new values with the existing values of an attribute. The function `slapi_entry_add_values_sv()` returns an error when duplicates of the new values already exist; `slapi_entry_attr_merge_sv()` does not. Code Example 4-8 demonstrates `slapi_entry_attr_merge_sv()`.

Code Example 4-8 Merging Attribute Values (`entries.c`)

```
#include "slapi-plugin.h"

int
test_ldif()
{
    Slapi_Entry * entry = NULL;        /* Entry to hold LDIF      */
    Slapi_Value * values[2];           /* Attribute values        */

    entry = slapi_entry_alloc();

    /* Add code to transform an LDIF representation into an entry.*/

    /* Add a description by setting the value of the attribute.
     * Although this is overkill when manipulating string values,
     * it can be handy when manipulating binary values.          */
    values[0] = slapi_value_new_string("Description for the entry.");
    values[1] = NULL;
    if (slapi_entry_attr_merge_sv(entry,"description",values) != 0)
        /* Merge did not work if we arrive here. */ ;

    slapi_entry_free(entry);
```

Code Example 4-8 Merging Attribute Values (`entries.c`) *(Continued)*

```
    return (0);
}
```

If the attribute exists in the entry, `slapi_entry_attr_merge_sv()` merges the specified values into the set of existing values. If the attribute does not exist, `slapi_entry_attr_merge_sv()` adds the attribute with the specified values.

Removing Attribute Values

Remove attribute values using `slapi_entry_delete_string()` or `slapi_entry_delete_values_sv()`.

Verifying Schema Compliance for an Entry

This section demonstrates how to check that an entry is valid with respect to the directory schema known to Directory Server. Verify schema compliance for an entry using `slapi_entry_check()`. Its two arguments are a pointer to a parameter block and a pointer to the entry, as shown in Code Example 4-9.

Code Example 4-9 Checking Schema Compliance (`entries.c`)

```
#include "slapi-plugin.h"

int
test_create_entry()
{
    Slapi_Entry * entry = NULL;        /* Original entry          */
    Slapi_Entry * ecopy = NULL;        /* Copy entry              */

    entry = slapi_entry_alloc();

    /* Add code to fill the entry, setting the DN and attributes. */

    ecopy = slapi_entry_dup(entry);
    slapi_entry_set_dn(ecopy, slapi_ch_strdup("dc=example,dc=org"));
    slapi_entry_add_string(ecopy, "description", "A copy of the orig.");

    /* Does the resulting copy comply with our schema?            */
    if (slapi_entry_schema_check(NULL, ecopy) == 0) {
        /* Resulting entry does comply. */ ;
    } else {
        /* Resulting entry does not comply. */ ;
    }
```

Code Example 4-9 Checking Schema Compliance (`entries.c`) *(Continued)*

```
    slapi_entry_free(entry);
    slapi_entry_free(ecopy);

    return (0);
}
```

Notice the parameter block pointer argument is `NULL`. Leave the parameter block pointer argument `NULL` unless the plug-in is used in a replicated environment and you do not want schema compliance verification carried out for replicated operations.

Handling Entry Distinguished Names (DNs)

This section demonstrates how to work with Distinguished Names to:

- Determine the parent DN of a given DN
- Determine the suffix DN with which the entry DN is associated
- Identify whether a root suffix is served by the backend
- Set the DN of the entry
- Normalize the DN of the entry for comparison with other entries
- Determine whether the entry DN is that of the directory superuser

Getting the Parent and Suffix DNs

This can be done using `slapi_dn_parent()`. This function returns the parent, whereas `slapi_dn_beparent()` returns the suffix associated with the entry, as shown in Code Example 4-10.

Code Example 4-10 Determining the Parent and Suffix of an Entry (`dns.c`)

```
#include "slapi-plugin.h"

int
test_dns(Slapi_PBlock * pb)
{
    char * bind_DN;                    /* DN being used to bind    */
    char * parent_DN;                  /* DN of parent entry       */
```

Code Example 4-10 Determining the Parent and Suffix of an Entry (dns.c) *(Continued)*

```
    char * suffix_DN;                  /* DN of suffix entry       */
    int    connId, opId, rc = 0;
    long   msgId;

    rc |= slapi_pblock_get(pb, SLAPI_BIND_TARGET,     &bind_DN);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_MSGID, &msgId);
    rc |= slapi_pblock_get(pb, SLAPI_CONN_ID,         &connId);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_ID,    &opId);
    if (rc != 0) return (-1);

    /* Get the parent DN of the DN being used to bind.             */
    parent_DN = slapi_dn_parent(bind_DN);
    slapi_log_info_ex(
        SLAPI_LOG_INFO_AREA_PLUGIN,
        SLAPI_LOG_INFO_LEVEL_DEFAULT,
        msgId,
        connId,
        opId,
        "test_dns in test-dns plug-in",
        "Parent DN of %s: %s\n", bind_DN, parent_DN
    );

    /* Get the suffix DN of the DN being used to bind.             */
    suffix_DN = slapi_dn_beparent(pb, bind_DN);
    if (suffix_DN != NULL) {
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
            SLAPI_LOG_INFO_LEVEL_DEFAULT,
            msgId,
            connId,
            opId,
            "test_dns in test-dns plug-in",
            "Suffix for user (%s) is (%s).\n", bind_DN, suffix_DN
        );
    }

    return rc;
}
```

Notice the suffix and parent DNs may be the same if the tree is shallow. For example, if the `bind_DN` here is `uid=yyorgens,ou=People,dc=example,dc=com`, then the parent is `ou=People,dc=example,dc=com`, and so is the suffix.

Checking Whether a Suffix is Served Locally

This can be done using `slapi_dn_isbesuffix()`, which takes as its two arguments a parameter block and a `char *` to the root suffix DN, as demonstrated in Code Example 4-11.

Code Example 4-11 Checking Whether a Suffix is Local (`dns.c`)

```
#include "slapi-plugin.h"

int
test_dns(Slapi_PBlock * pb)
{
    char * bind_DN;                    /* DN being used to bind    */
    char * parent_DN;                  /* DN of parent entry       */
    char * suffix_DN;                  /* DN of suffix entry       */

    slapi_pblock_get(pb, SLAPI_BIND_TARGET, &bind_DN);

    /* Get the suffix DN of the DN being used to bind.             */
    suffix_DN = slapi_dn_beparent(pb, bind_DN);

    /* Climb the tree to the top suffix and check if it is local.  */
    while (suffix_DN != NULL && !slapi_dn_isbesuffix(pb, suffix_DN)) {
        suffix_DN = slapi_dn_parent(suffix_DN);
    }
    if (suffix_DN != NULL) {
        /* Suffix is served locally. */ ;
    } else {
        /* Suffix is not served locally. */ ;
    }

    return 0;
}
```

Notice we use `slapi_dn_isbesuffix()` to climb the tree. Notice also that `slapi_dn_parent()` keeps gnawing away at the DN until it is `NULL`. The parent does not necessarily correspond to an actual entry.

Getting and Setting Entry DNs

This can be done using `slapi_entry_get_dn()` and `slapi_entry_set_dn()`. Code Example 4-9 on page 78 demonstrates changing the DN of a copy of an entry, using `slapi_entry_set_dn()`. Notice the use of `slapi_ch_strdup()` to ensure the old DN in the copy is not used.

Normalizing a DN

Before comparing DNs, you may want to normalize the DNs to prevent the comparison from failing due to extra white space or different capitalization. You may normalize a DN using `slapi_dn_normalize()`, or `slapi_dn_normalize_case()` as shown in Code Example 4-12.

Code Example 4-12 Normalizing a DN (dns.c)

```
#include "slapi-plugin.h"

int
test_norm()
{
    char * test_DN;                   /* Original, not normalized */
    char * copy_DN;                   /* Copy that we normalize   */

    test_DN = "dc=Example,     dc=COM";/* Prior to normalization...*/

    /* When normalizing the DN with slapi_dn_normalize() and
     * slapi_dn_normalize_case(), the DN is changed in place.
     * Use slapi_ch_strdup() to work on a copy.                   */
    copy_DN = slapi_ch_strdup(test_DN);
    copy_DN = slapi_dn_normalize_case(copy_DN);

    return 0;
}
```

The function `slapi_dn_normalize_case()` works *directly* on the `char *` DN passed as an argument to perform "case ignore matching" as specified, but not defined, for X.500. Use `slapi_ch_strdup()` to make a copy first if you do not want to modify the original.

Is This the Directory Manager?

Answer this question using `slapi_dn_isroot()`. The bind for the directory superuser, however, as for anonymous users, is handled as a special case. For such users, the server front end handles the bind *before* calling pre-operation bind plug-ins.

Chapter 5

Extending Client Request Handling

This chapter covers support in the plug-in API for modifying what Directory Server does before or after carrying out operations for clients.

Although related to the bind operation, authentication merits its own discussion in Chapter 6, “Handling Authentication.”

Pre-Operation and Post-Operation Plug-Ins

Before and after performing processing client requests and before and after sending information to clients, Directory Server calls *pre-operation* and *post-operation* plug-in functions, respectively.

Pre-Operation Plug-Ins

Pre-operation plug-ins are called before a client request is processed. Use pre-operation plug-ins to validate attribute values, and to add to and delete from attributes provided in a request, for example.

Post-Operation Plug-Ins

Post-operation plug-ins are called after a client request has been processed, whether or not the operation completes successfully. Use post-operation plug-ins to send alerts and alarms, and to perform auditing and clean up work, for example.

NOTE Directory Server calls pre- and post-operation plug-ins only when an external client is involved. (Internally, Directory Server also performs operations no external client has requested.)

To make this possible, Directory Server distinguishes between external operations and internal operations. External operations respond to incoming client requests. Internal operations are those performed without a corresponding client request.

Operations supporting the use of pre- and post-operation plug-in functions include abandon, add, bind, compare, delete, modify, modify RDN, search, send entry, send referral, send result, and unbind, regardless of Directory Server front end contacted by the client application.

As is the case for other plug-in functions, `slapi_pblock_set()` is used in the plug-in initialization function to register pre- and post-operation plug-in functions. `slapi_pblock_set()` is described in the *Sun ONE Directory Server Plug-In API Reference*.

Registration Identifiers

Pre- and post-operation plug-in registrations use IDs of the form `SLAPI_PLUGIN_PRE_`*`operation`*`_FN` or `SLAPI_PLUGIN_POST_`*`operation`*`_FN`, where *operation* is one of `ABANDON`, `ADD`, `BIND`, `COMPARE`, `DELETE`, `ENTRY` (sending entries to the client), `MODIFY`, `MODRDN`, `REFERRAL` (sending referrals to the client), `RESULT` (sending results to the client), `SEARCH`, or `UNBIND`.

NOTE Pre- and post-operation plug-ins may also register functions to run at Directory Server startup, using `SLAPI_PLUGIN_START_FN`, and at Directory Server shutdown, using `SLAPI_PLUGIN_STOP_FN`.

The *Sun ONE Directory Server Plug-In API Reference* describes these IDs. Refer also to *ServerRoot*`/plugins/slapd/slapi/include/slapi-plugin.h` for the complete list of IDs used at build time.

Code Example 5-1 demonstrates how the functions are registered with Directory Server using the appropriate registration IDs.

Code Example 5-1 Registering Post-Operation Functions (`testpostop.c`)

```
#include "slapi-plugin.h"

Slapi_PluginDesc postop_desc = {
    "test-postop",                      /* plug-in identifier         */
    "Sun Microsystems, Inc.",           /* vendor name                */
    "5.2",                              /* plug-in revision number    */
    "Sample post-operation plug-in"     /* plug-in description        */
};

static Slapi_ComponentId * postop_id;  /* Used to set log            */

/* Register the plug-in with the server. */
#ifdef _WIN32
__declspec(dllexport)
#endif
int
testpostop_init(Slapi_PBlock * pb)
{
    int rc = 0;                         /* 0 means success            */
    rc |= slapi_pblock_set(             /* Plug-in API version        */
        pb,
        SLAPI_PLUGIN_VERSION,
        SLAPI_PLUGIN_CURRENT_VERSION
    );
    rc |= slapi_pblock_set(             /* Plug-in description        */
        pb,
        SLAPI_PLUGIN_DESCRIPTION,
        (void *) &postop_desc
    );
    rc |= slapi_pblock_set(             /* Open log at startup        */
        pb,
        SLAPI_PLUGIN_START_FN,
        (void *) testpostop_set_log
    );
    rc |= slapi_pblock_set(             /* Post-op add function       */
        pb,
        SLAPI_PLUGIN_POST_ADD_FN,
        (void *) testpostop_add
    );
    rc |= slapi_pblock_set(             /* Post-op modify function    */
        pb,
        SLAPI_PLUGIN_POST_MODIFY_FN,
        (void *) testpostop_mod
    );
    rc |= slapi_pblock_set(             /* Post-op delete function    */
        pb,
        SLAPI_PLUGIN_POST_DELETE_FN,
        (void *) testpostop_del
    );
    rc |= slapi_pblock_set(             /* Post-op modrdn function    */
        pb,
        SLAPI_PLUGIN_POST_MODRDN_FN,
        (void *) testpostop_modrdn
    );
```

Code Example 5-1 Registering Post-Operation Functions (`testpostop.c`) *(Continued)*

```
    rc |= slapi_pblock_set(           /* Close log on shutdown      */
        pb,
        SLAPI_PLUGIN_CLOSE_FN,
        (void *) testpostop_free_log
    );
    /* Plug-in identifier is required for internal search.          */
    rc |= slapi_pblock_get(pb, SLAPI_PLUGIN_IDENTITY, &postop_id);
    return (rc);
}
```

In addition to its post-operation functions, the plug-in shown in Code Example 5-1 registers a function to open a log file at startup and close the log file at shutdown. For details, refer to *ServerRoot*`/plugins/slapd/slapi/examples/testpostop.c`.

Finding Examples

Find plug-in examples under *ServerRoot*`/plugins/slapd/slapi/examples/`.

Extending the Bind Operation

This section shows how to develop functions called by Directory Server before client bind operations.

TIP Pre-bind plug-in functions are often used to handle extensions to authentication.

For details, refer to Chapter 6, “Handling Authentication.”

Setting Up an Example Suffix

If you have not done so already, prepare some example data by creating a directory suffix, `dc=example,dc=com`, whose users we load from an LDIF file, *ServerRoot*`/slapd-`*serverID*`/ldif/Example-Plugin.ldif`. You may do this using either by using the console, or client tools on the command line. Using the command line for example:

1. Prepare an LDIF file specifying the backend and suffix for Example.com.

Code Example 5-2 LDIF File for an Example.com Suffix

```
dn: cn="dc=example,dc=com",cn=mapping tree,cn=config
objectclass: top
objectclass: extensibleObject
objectclass: nsMappingTree
nsslapd-state: backend
nsslapd-backend: example
cn: dc=example,dc=com

dn: cn=example,cn=ldbm database, cn=plugins, cn=config
objectClass: top
objectClass: extensibleObject
objectClass: nsBackendInstance
nsslapd-suffix: dc=example,dc=com
```

2. Add the suffix to the directory.

   ```
   $ ldapmodify -a -p port -D "cn=directory manager" -w passwd -f file
   ```

   Here, *file* is the LDIF file from Code Example 5-2.

3. Stop Directory Server.

   ```
   $ ServerRoot/slapd-serverID/stop-slapd
   ```

4. Load *ServerRoot*/slapd-*serverID*/ldif/Example-Plugin.ldif.

   ```
   $ ServerRoot/slapd-serverID/ldif2db -s "dc=example,dc=com" \
   -i ServerRoot/slapd-serverID/ldif/Example-Plugin.ldif
   ```

5. Start Directory Server.

   ```
   $ ServerRoot/slapd-serverID/start-slapd
   ```

Logging the Authentication Method

Code Example 5-3 logs the bind authentication method. Refer to *ServerRoot*/plugins/slapd/slapi/examples/testpreop.c for complete example code.

Code Example 5-3 Logging the Authentication Method (testpreop.c)

```
#include "slapi-plugin.h"

int
```

Code Example 5-3 Logging the Authentication Method (`testpreop.c`) *(Continued)*

```
testpreop_bind(Slapi_PBlock * pb)
{
    char * auth;                        /* Authentication type     */
    char * dn;                          /* Target DN               */
    int    method;                      /* Authentication method   */
    int    connId, opId, rc = 0;
    long   msgId;

    /* Get target DN for bind and authentication method used.     */
    rc |= slapi_pblock_get(pb, SLAPI_BIND_TARGET,     &dn);
    rc |= slapi_pblock_get(pb, SLAPI_BIND_METHOD,     &method);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_MSGID, &msgId);
    rc |= slapi_pblock_get(pb, SLAPI_CONN_ID,         &connId);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_ID,    &opId);
    if (rc == 0) {
        switch (method) {
            case LDAP_AUTH_NONE:   auth = "No authentication";
                break;
            case LDAP_AUTH_SIMPLE: auth = "Simple authentication";
                break;
            case LDAP_AUTH_SASL:   auth = "SASL authentication";
                break;
            default: auth = "Unknown authentication method";
                break;
        }
    } else {
        return (rc);
    }

    /* Log target DN and authentication method info.              */
    slapi_log_info_ex(
        SLAPI_LOG_INFO_AREA_PLUGIN,
        SLAPI_LOG_INFO_LEVEL_DEFAULT,
        msgId,
        connId,
        opId,
        "testpreop_bind in test-preop plug-in",
        "Target DN: %s\tAuthentication method: %s\n", dn, auth
    );
    return (rc);
}
```

This plug-in function sets the `auth` message based on the authentication `method`. It does nothing to affect the way Directory Server processes the bind.

Registering the Plug-In

If you have not already done so, build the example plug-in library and activate both plug-in informational logging and the `Test Preop` plug-in.

1. Build the plug-in.

   Hint Use *ServerRoot*`/plugins/slapd/slapi/examples/Makefile`.

2. Load the plug-in configuration entry.

   Hint Start with the configuration entry specified in the introductory comments of *ServerRoot*`/plugins/slapd/slapi/examples/testpreop.c`.

Code Example 5-4 Configuration Entry (`testpreop.c`)

```
dn: cn=Test Preop,cn=plugins,cn=config
objectClass: top
objectClass: nsSlapdPlugin
objectClass: extensibleObject
cn: Test Preop
nsslapd-pluginPath: <ServerRoot>/plugins/slapd/slapi/examples/<LibName>
nsslapd-pluginInitfunc: testpreop_init
nsslapd-pluginType: preoperation
nsslapd-pluginEnabled: on
nsslapd-plugin-depends-on-type: database
nsslapd-pluginId: test-preop
nsslapd-pluginVersion: 5.2
nsslapd-pluginVendor: Sun Microsystems, Inc.
nsslapd-pluginDescription: Sample pre-operation plug-in
```

3. Restart Directory Server.
4. Turn on informational logging for plug-ins.

   Refer to “Set Appropriate Log Level in the Directory Server Configuration,” on page 70 for details.

Generating a Bind Log Message

If you use the console, it generates messages whenever the `admin` user binds. For something different:

1. Bind as Plinner Blinn.

```
$ ldapsearch -p port -b "dc=example,dc=com" \
-D "uid=pblinn,ou=People,dc=example,dc=com" -w pblinn "(uid=*)"
```

2. Search *ServerRoot*/slapd-*serverID*/logs/errors for the resulting message from the testpreop_bind() function.

   Ignoring housekeeping information for the entry, you find something like this:

   ```
   Target DN: uid=pblinn,ou=people,dc=example,dc=comAuthentication
   method: Simple authentication
   ```

For a discussion of less trivial pre-bind plug-in functions, refer to Chapter 6, "Handling Authentication."

Bypassing Bind Processing in Directory Server

When the plug-in returns 0, Directory Server continues to process the bind. To bypass Directory Server bind processing, set SLAPI_CONN_DN in the parameter block, and return a non-zero value, such as 1.

Normal Directory Server Behavior

Directory Server follows the LDAP bind model. At minimum, it authenticates the client and sends a bind response to indicate the status of authentication. Refer to RFC 2251, *Lightweight Directory Access Protocol (v3)*, and RFC 1777, *Lightweight Directory Access Protocol*, for details.

Extending the Search Operation

This section shows how to develop functionality called by Directory Server before LDAP search operations.

Logging Who Requests a Search

The first example discussed here logs the DN of the client requesting the search. Refer to *ServerRoot*/plugins/slapd/slapi/examples/testbind.c for complete example code.

Before using the plug-in function as described here, set up the example suffix and register the plug-in as described under “Setting Up an Example Suffix,” on page 86 and “Registering the Plug-In,” on page 89, loading the configuration entry from the introductory comments of `testbind.c`. (The plug-in named `Test Bind` also includes our pre-search function.) Code Example 5-4 shows a similar configuration entry.

Adjust `nsslapd-pluginPath` to fit how the product is installed on your system. Notice the value of `nsslapd-pluginType`: `preoperation`.

The `test_search()` function logs the request, as shown in Code Example 5-5.

Code Example 5-5 Getting the DN of the Client Requesting a Search (`testbind.c`)

```
#include "slapi-plugin.h"

int
test_search(Slapi_PBlock * pb)
{
    char * requestor_dn;              /* DN of client searching    */
    int    is_repl;                   /* Is this replication?      */
    int    is_intl;                   /* Is this an internal op?   */
    int    connId, opId, rc = 0;
    long   msgId;

    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_MSGID,         &msgId);
    rc |= slapi_pblock_get(pb, SLAPI_CONN_ID,                 &connId);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_ID,            &opId);
    rc |= slapi_pblock_get(pb, SLAPI_REQUESTOR_DN,            &requestor_dn);
    rc |= slapi_pblock_get(pb, SLAPI_IS_REPLICATED_OPERATION, &is_repl);
    rc |= slapi_pblock_get(pb, SLAPI_IS_INTERNAL_OPERATION,   &is_intl);
    if (rc != 0) return (rc);

    /* Do not interfere with replication and internal operations.   */
    if (is_repl || is_intl) return 0;

    if (requestor_dn != NULL && *requestor_dn != '\0') {
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
            SLAPI_LOG_INFO_LEVEL_DEFAULT,
            msgId,
            connId,
            opId,
            "test_search in test-bind plug-in",
            "Search requested by %s\n", requestor_dn
        );
    } else {
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
            SLAPI_LOG_INFO_LEVEL_DEFAULT,
            msgId,
            connId,
            opId,
```

Code Example 5-5 Getting the DN of the Client Requesting a Search (`testbind.c`) *(Continued)*

```
            "test_search in test-bind plug-in",
            "Search requested by anonymous client.\n"
        );
    }
    return (rc);
}
```

With the plug-in active in the server, perform a search.

```
$ ldapsearch -D "uid=hfuddnud,ou=People,dc=example,dc=com" \
-w hfuddnud -p port -b "dc=example,dc=com" "(uid=pblinn)"
```

Search *ServerRoot*`/slapd-`*serverID*`/logs/errors` for the resulting message. The last field of the log entry shows:

```
Search requested by uid=hfuddnud,ou=people,dc=example,dc=com
```

Breaking Down a Search Filter

The second example discussed here breaks down a search filter, logging the component parts of the filter. For complete example code, refer to *ServerRoot*`/plugins/slapd/slapi/examples/testpreop.c`.

LOG Macros for Compact Code

The code for the `SLAPI_PLUGIN_PRE_SEARCH_FN` function, `testpreop_search()`, is preceded by three macros. (Search `testpreop.c` for `#define LOG1(format)`.) The purpose of these macros is only to render the subsequent logging statements more compact, making the code easier to decipher. The digits in `LOG1`, `LOG2`, and `LOG3` reflect the number of parameters each macro takes. The actual number of parameters passed to the log functions varies, as the log functions let you format strings in the style of `printf()`.

Parameter Block Contents

The parameter block passed to the pre-search function contains information about the target and scope of the search. Code Example 5-6 shows how to obtain this information. The scope, as specified in RFC 2251, *Lightweight Directory Access Protocol (v3)*, can be the base object, which is the base DN itself, a single level below the base DN, or the whole subtree below the base DN.

Code Example 5-6 Logging Base and Scope for a Search (`testpreop.c`)

```
#include "slapi-plugin.h"

int
testpreop_search(Slapi_PBlock * pb)
{
    char          *  base       = NULL;/* Base DN for search      */
    int              scope;            /* Base, 1 level, subtree  */
    int              rc = 0;

    rc |= slapi_pblock_get(pb, SLAPI_SEARCH_TARGET,    &base);
    rc |= slapi_pblock_get(pb, SLAPI_SEARCH_SCOPE,     &scope);
    if (rc == 0) {
        switch (scope) {
        case LDAP_SCOPE_BASE:
            LOG2("Target DN: %s\tScope: LDAP_SCOPE_BASE\n", base);
            break;
        case LDAP_SCOPE_ONELEVEL:
            LOG2("Target DN: %s\tScope: LDAP_SCOPE_ONELEVEL\n", base);
            break;
        case LDAP_SCOPE_SUBTREE:
            LOG2("Target DN: %s\tScope: LDAP_SCOPE_SUBTREE\n", base);
            break;
        default:
            LOG3("Target DN: %s\tScope: unknown value %d\n", base, scope);
            break;
        }
    } /* Continue processing... */

    return 0;
}
```

In writing a plug-in mapping X.500 search targets to LDAP search targets, for example, you could choose to parse and transform the base DN for searches.

The parameter block also contains information about when aliases are resolved during the search, how much Directory Server time and resources the search is set to consume, and which attribute types are to be returned.

Peeking Into the Search Filter

Of particular interest here is the search filter you obtain from the parameter block. `testpreop_search()` gets the filter from the parameter block both as a `Slapi_Filter` structure (`filter`) and as a string (`filter_str`). Depending on what kind of search filter is passed to Directory Server and retrieved in the

`Slapi_Filter` structure, the plug-in breaks the filter down in different ways. If the function were post-search rather than pre-search, you would likely want to access the results returned through the parameter block. Refer to the *Sun ONE Directory Server Plug-In API Reference* for details on how to access the results.

The plug-in API offers several functions for manipulating the search filter. `testpreop_search()` uses `slapi_filter_get_choice()` to determine the kind of filter used, `slapi_filter_get_subfilt()` to break down substrings used in the filter, `slapi_filter_get_type()` to find the attribute type when searching for the presence of an attribute, and `slapi_filter_get_ava()` to obtain attribute types and values used for comparisons.

Code Example 5-7 shows a code excerpt that retrieves some information from the search filter.

Code Example 5-7 Retrieving Filter Information (`testpreop.c`)

```
#include "slapi-plugin.h"

int
testpreop_search(Slapi_PBlock * pb)
{
    char          *  attr_type  = NULL;/* For substr and compare  */
    char          *  substr_init= NULL;/* For substring filters   */
    char          *  substr_final=NULL;
    char          ** substr_any = NULL;
    int              i, rc =0;
    Slapi_Filter  *  filter;

    rc |= slapi_pblock_get(pb, SLAPI_SEARCH_FILTER,    &filter);
    if (rc == 0) {
        filter_type = slapi_filter_get_choice(filter);
        switch (filter_type) {
        case LDAP_FILTER_AND:
        case LDAP_FILTER_OR:
        case LDAP_FILTER_NOT:
            LOG1("Search filter: complex boolean\n");
            break;
        case LDAP_FILTER_EQUALITY:
            LOG1("Search filter: LDAP_FILTER_EQUALITY\n");
            break;
        case LDAP_FILTER_GE:
            LOG1("Search filter: LDAP_FILTER_GE\n");
            break;
        case LDAP_FILTER_LE:
            LOG1("Search filter: LDAP_FILTER_LE\n");
            break;
        case LDAP_FILTER_APPROX:
            LOG1("Search filter: LDAP_FILTER_APPROX\n");
            break;
        case LDAP_FILTER_SUBSTRINGS:
            LOG1("Search filter: LDAP_FILTER_SUBSTRINGS\n");
```

Code Example 5-7 Retrieving Filter Information (`testpreop.c`) *(Continued)*

```
            slapi_filter_get_subfilt(
                filter,
                &attr_type,
                &substr_init,
                &substr_any,
                &substr_final
            );
            if (attr_type != NULL)
                LOG2("\tAttribute type: %s\n", attr_type);
            if (substr_init != NULL)
                LOG2("\tInitial substring: %s\n", substr_init);
            if (substr_any != NULL)
                for (i = 0; substr_any[i] != NULL; ++i) {
                    LOG3("\tSubstring# %d: %s\n", i, substr_any[i]);
                }
            if (substr_final != NULL)
                LOG2("\tFinal substring: %s\n", substr_final);
            break;
        case LDAP_FILTER_PRESENT:
            LOG1("Search filter: LDAP_FILTER_PRESENT\n");
            slapi_filter_get_attribute_type(filter, &attr_type);
            LOG2("\tAttribute type: %s\n", attr_type);
            break;
        case LDAP_FILTER_EXTENDED:
            LOG1("Search filter: LDAP_FILTER_EXTENDED\n");
            break;
        default:
            LOG2("Search filter: unknown type %d\n", filter_type);
            break;
        }
    }
    return (rc);
}
```

Before using the plug-in function as described here, set up the example suffix and register the plug-in as described under “Setting Up an Example Suffix,” on page 86 and “Registering the Plug-In,” on page 89.

After the example suffix and plug-in have been loaded into the directory, run a search to generate some output in *ServerRoot*`/slapd-`*serverID*`/logs/errors`.

```
$ ldapsearch -p port -b "dc=example,dc=com" "(uid=*go*iv*)"
```

Ignoring the housekeeping information in the error log, the search filter breakdown happens as shown in Code Example 5-8.

Code Example 5-8 Search Filter Breakdown

```
*** PREOPERATION SEARCH PLUG-IN - START ***
Target DN: dc=example,dc=com       Scope: LDAP_SCOPE_SUBTREE
Dereference setting: LDAP_DEREF_NEVER
Search filter: LDAP_FILTER_SUBSTRINGS
        Attribute type: uid
        Substring# 0: go
        Substring# 1: iv
String representation of search filter: (uid=*go*iv*)
*** PREOPERATION SEARCH PLUG-IN - END ***
```

When Building a Filter

Notice the plug-in API provides `slapi_str2filter()` to convert strings to `Slapi_Filter` data types and `slapi_filter_join()` to join simple filters with boolean operators `AND`, `OR`, or `NOT`. Deallocate the filter using `slapi_filter_free()`. Refer to the *Sun ONE Directory Server Plug-In API Reference* for details.

Normal Directory Server Behavior

Directory Server gets a list of candidate entries, then iterates through the list to check which entries match the search criteria. It then puts the set of results in the parameter block. In most cases, searching from within a plug-in is best done with `slapi_search_internal*()` calls. Refer to Chapter 7, “Performing Internal Operations,” for details.

Extending the Compare Operation

This section shows how to develop functionality called by Directory Server before a client compare operation. Code Example 5-9 logs the target DN and attribute of the entry with which to compare values. For complete example code, refer to *ServerRoot*`/plugins/slapd/slapi/examples/testpreop.c`.

Code Example 5-9 Plug-In Comparison Function (`testpreop.c`)

```
#include "slapi-plugin.h"

int
testpreop_cmp(Slapi_PBlock * pb)
{
```

Code Example 5-9 Plug-In Comparison Function (`testpreop.c`) *(Continued)*

```
    char * dn;                          /* Target DN              */
    char * attr_type;                   /* Type of attr to compare */
    /* Attribute value could be lots of things, even a binary file.
     * Here, we do not try to retrieve the value to compare.      */
    int    connId, opId, rc = 0;
    long   msgId;

    rc |= slapi_pblock_get(pb, SLAPI_COMPARE_TARGET,  &dn);
    rc |= slapi_pblock_get(pb, SLAPI_COMPARE_TYPE,    &attr_type);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_MSGID, &msgId);
    rc |= slapi_pblock_get(pb, SLAPI_CONN_ID,         &connId);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_ID,    &opId);
    if (rc == 0) {
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
            SLAPI_LOG_INFO_LEVEL_DEFAULT,
            msgId,
            connId,
            opId,
            "testpreop_cmp in test-preop plug-in",
            "Target DN: %s\tAttribute type: %s\n", dn, attr_type
        );
    }

    return (rc);
}
```

The plug-in function can access the attribute value to use for the comparison as well. The attribute value in the parameter block is in a `berval` structure, meaning it could be binary data such as a JPEG image, and no attempt is made to write the value to the logs.

Code Example 5-10 shows the `slapi_pblock_get()` call used to obtain the attribute value.

Code Example 5-10 Obtaining the Attribute Value

```
#include "slapi-plugin.h"

int
my_compare_fn(Slapi_PBlock * pb)
{
    int             rc = 0;
    struct berval * attr_val;

    /* Obtain the attribute value from the parameter block */
    rc |= slapi_pblock_get(pb, SLAPI_COMPARE_VALUE, &attr_val);
```

Code Example 5-10 Obtaining the Attribute Value *(Continued)*

```
    if (rc != 0) {
        rc = LDAP_OPERATIONS_ERROR;
        slapi_send_ldap_result(pb, rc, NULL, NULL, 0, NULL);
        return 0;
    }

    /* Do something with the value here.                    */

    return 0;
}
```

Before using the plug-in function as described here, set up the example suffix and register the plug-in as described under “Setting Up an Example Suffix,” on page 86 and “Registering the Plug-In,” on page 89.

Try the example plug-in function using the `ldapcompare` tool from the Sun ONE Directory Server Resource Kit. Refer to “Downloading Directory Server Tools,” on page 16 for further information.

Code Example 5-11 Performing a Comparison

```
$ ldapcompare -p port sn:Blinn "uid=pblinn,ou=People,dc=example,dc=com"
comparing type: "sn" value: "Blinn" in entry "uid=pblinn,ou=People,dc=example,dc=com"
compare TRUE
```

The log entry in *ServerRoot*`/slapd-`*serverID*`/logs/errors` results as follows, not including housekeeping information at the beginning of the log entry:

```
Target DN: uid=pblinn,ou=people,dc=example,dc=com Attribute type: sn
```

Extending the Add Operation

This section shows how to develop functionality called by Directory Server before and after a client add operation.

Prepending a String to an Attribute

The first example discussed here prepends the string "ADD " to the description attribute of the entry to be added to the directory. For complete example code, refer to *ServerRoot*/plugins/slapd/slapi/examples/testpreop.c.

Before using the plug-in function as described here, set up the example suffix and register the plug-in as described under "Setting Up an Example Suffix," on page 86 and "Registering the Plug-In," on page 89.

To try the pre-operation add function, add an entry for Bartholomew's cousin, Quentin, who recently joined Example.com. Quentin's LDIF entry, quentin.ldif, reads as shown in Code Example 5-12.

Code Example 5-12 LDIF for Quentin Cubbins

```
dn: uid=qcubbins,ou=People,dc=example,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
uid: qcubbins
givenName: Quentin
sn: Cubbins
cn: Quentin Cubbins
mail: qcubbins@example.com
userPassword: qcubbins
secretary: uid=bcubbins,ou=People,dc=example,dc=com
description: Entry for Quentin Cubbins
```

Code Example 5-13 performs the prepend.

Code Example 5-13 Prepending ADD to the Description of the Entry (testpreop.c)

```
#include "slapi-plugin.h"

int
testpreop_add(Slapi_PBlock * pb)
{
    Slapi_Entry * entry;              /* Entry to add            */
    Slapi_Attr  * attribute;          /* Entry attributes        */
    Slapi_Value * value;              /* Attribute values        */
    int         i;
    char        * tmp;                /* Holder for new desc.    */
    const char  * string;             /* Holder for current desc.*/
    int         connId, opId, rc = 0;
    long        msgId;
```

Code Example 5-13 Prepending ADD to the Description of the Entry (`testpreop.c`) *(Continued)*

```
    /* Get the entry. */
    rc |= slapi_pblock_get(pb, SLAPI_ADD_ENTRY,       &entry);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_MSGID, &msgId);
    rc |= slapi_pblock_get(pb, SLAPI_CONN_ID,         &connId);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_ID,    &opId);

    /* Prepend ADD to value of description attribute.             */
    rc |= slapi_entry_attr_find(entry, "description", &attribute);
    if (rc == 0) {                 /* Found a description, so...  */
        for (                          /* ...loop for value...    */
            i = slapi_attr_first_value(attribute, &value);
            i != -1;
            i = slapi_attr_next_value(attribute, i, &value)
        ) {                            /* ...prepend "ADD ".      */
            string = slapi_value_get_string(value);
            tmp    = slapi_ch_malloc(5+strlen(string));
            strcpy(tmp, "ADD ");
            strcpy(tmp+4, string);
            slapi_value_set_string(value, tmp);
            slapi_ch_free((void **)&tmp);
        }
    } else {                           /* entry has no desc.      */
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
            SLAPI_LOG_INFO_LEVEL_DEFAULT,
            msgId,
            connId,
            opId,
            "testpreop_add in test-preop plug-in",
            "Entry has no description attribute.\n"
        );
    }

    return 0;
}
```

Add Quentin's entry to the directory. For example, if the entry is in `quentin.ldif`:

```
$ ldapmodify -a -p port -D "cn=directory manager" \
-w password -f quentin.ldif
```

At this point, we can search the directory for Quentin's entry.

Code Example 5-14 Searching for the Entry

```
$ ldapsearch -L -p port -b "dc=example,dc=com" "(uid=qcubbins)"
dn: uid=qcubbins,ou=People,dc=example,dc=com
objectClass: top
```

Code Example 5-14 Searching for the Entry *(Continued)*

```
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
uid: qcubbins
givenName: Quentin
sn: Cubbins
cn: Quentin Cubbins
mail: qcubbins@example.com
secretary: uid=bcubbins,ou=People,dc=example,dc=com
description: ADD Entry for Quentin Cubbins
```

Notice the value of the description attribute.

Delete Quentin's entry so you can use it again later as an example.

```
$ ldapdelete -p port -D "cn=directory manager" -w password \
"uid=qcubbins,ou=People,dc=example,dc=com"
```

Turn off the pre-operation plug-in to avoid prepending `ADD` to all the entries you add. Do so from the console on the `Configuration` tab page by selecting the node bearing the common name of the plug-in and clearing the appropriate checkbox. Restart Directory Server (`Tasks` tab page in the console) for the change to take effect.

Logging the Entry to Add

The second example discussed here logs the entry to add. For complete example code, refer to *ServerRoot*`/plugins/slapd/slapi/examples/testpostop.c`.

Before using the plug-in function as described here, set up the example suffix and register the plug-in as described under "Extending the Bind Operation," on page 86, loading the configuration entry from the introductory comment of `testpostop.c`. Code Example 5-15 shows that configuration entry.

Code Example 5-15 Configuration Entry (`testpostop.c`)

```
dn: cn=Test Postop,cn=plugins,cn=config
objectClass: top
objectClass: nsSlapdPlugin
objectClass: extensibleObject
cn: Test Postop
nsslapd-pluginPath: <ServerRoot>/plugins/slapd/slapi/examples/<LibName>
nsslapd-pluginInitfunc: testpostop_init
nsslapd-pluginType: postoperation
```

Code Example 5-15 Configuration Entry (`testpostop.c`) *(Continued)*

```
nsslapd-pluginEnabled: on
nsslapd-plugin-depends-on-type: database
nsslapd-pluginId: test-postop
nsslapd-pluginVersion: 5.2
nsslapd-pluginVendor: Sun Microsystems, Inc.
nsslapd-pluginDescription: Sample post-operation plug-in
```

Adjust `nsslapd-pluginPath` to fit how the product is installed on your system. Notice the value of `nsslapd-pluginType`: `postoperation`.

The `testpostop_add( )` function logs the DN of the added entry and also writes the entry to a log created by the plug-in, called `changelog.txt`. The location of `changelog.txt` depends on the platform, as shown in Code Example 5-16. The changelog.txt file managed by the plug-in is not related to other change logs managed by Directory Server.

Code Example 5-16 Tracking Added Entries (`testpostop.c`)

```
#include "slapi-plugin.h"

int
testpostop_add(Slapi_PBlock * pb)
{
    char        * dn;                 /* DN of entry to add         */
    Slapi_Entry * entry;              /* Entry to add               */
    int           is_repl = 0;        /* Is this replication?       */
    int           connId, opId, rc = 0;
    long          msgId;

    rc |= slapi_pblock_get(pb, SLAPI_ADD_TARGET,              &dn);
    rc |= slapi_pblock_get(pb, SLAPI_ADD_ENTRY,               &entry);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_MSGID,         &msgId);
    rc |= slapi_pblock_get(pb, SLAPI_CONN_ID,                 &connId);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_ID,            &opId);
    rc |= slapi_pblock_get(pb, SLAPI_IS_REPLICATED_OPERATION, &is_repl);

    if (rc != 0) return (rc);
    slapi_log_info_ex(
        SLAPI_LOG_INFO_AREA_PLUGIN,
        SLAPI_LOG_INFO_LEVEL_DEFAULT,
        msgId,
        connId,
        opId,
        "testpostop_add in test-postop plug-in",
        "Added entry (%s)\n", dn
    );

    /* In general, do not interfere in replication operations.       */
```

Code Example 5-16 Tracking Added Entries (`testpostop.c`) *(Continued)*

```
    /* Log the DN and the entry to the change log file.              */
    if (!is_repl) write_changelog(_ADD, dn, (void *) entry, 0);

    return (rc);
}
```

After activating the plug-in, add Quentin's entry as specified in "Prepending a String to an Attribute," on page 99.

```
$ ldapmodify -a -p port -D "cn=directory manager" \
-w password -f quentin.ldif
```

Search *ServerRoot*`/slapd-`*serverID*`/logs/errors` for the log message. Not including housekeeping information:

```
Added entry (uid=qcubbins,ou=people,dc=example,dc=com)
```

Notice also the entry logged to `changelog.txt`, which resembles the LDIF added, except that it has time stamps and the clear text password attached for internal use as shown in Code Example 5-17.

Code Example 5-17 Sample `changelog.txt` Entry After the Add

```
time: 20020506172445
dn: uid=qcubbins,ou=people,dc=example,dc=com
changetype: add
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
uid: qcubbins
givenName: Quentin
sn: Cubbins
cn: Quentin Cubbins
mail: qcubbins@example.com
secretary: uid=bcubbins,ou=People,dc=example,dc=com
userPassword: qcubbins
creatorsName: cn=directory manager
modifiersName: cn=directory manager
createTimestamp: 20020506152444Z
modifyTimestamp: 20020506152444Z
unhashed#user#password: qcubbins
```

Extending the Modify Operation

This section shows how to develop functionality called by Directory Server after a client modify operation. A pre-operation plug-in, not demonstrated here, can be found in *ServerRoot*/plugins/slapd/slapi/examples/testpreop.c. Refer to *ServerRoot*/plugins/slapd/slapi/examples/testpostop.c for the source code discussed here.

Before using the plug-in function as described here, set up Directory Server as described under “Logging the Entry to Add,” on page 101, making sure to add Quentin’s entry.

The testpostop_mod() function logs the DN of the modified entry and also writes the entry to a log managed by the plug-in, called changelog.txt. The location of changelog.txt depends on the platform, as shown in the source code.

Code Example 5-18 performs the logging.

Code Example 5-18 Tracking Modified Entries (testpostop.c)

```
#include "slapi-plugin.h"

int
testpostop_mod(Slapi_PBlock * pb)
{
    char    *  dn;                    /* DN of entry to modify      */
    LDAPMod ** mods;                  /* Modifications to apply     */
    int        is_repl = 0;           /* Is this replication?       */
    int        connId, opId, rc = 0;
    long       msgId;

    rc |= slapi_pblock_get(pb, SLAPI_MODIFY_TARGET,           &dn);
    rc |= slapi_pblock_get(pb, SLAPI_MODIFY_MODS,             &mods);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_MSGID,         &msgId);
    rc |= slapi_pblock_get(pb, SLAPI_CONN_ID,                 &connId);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_ID,            &opId);
    rc |= slapi_pblock_get(pb, SLAPI_IS_REPLICATED_OPERATION, &is_repl);

    if (rc != 0) return (rc);
    slapi_log_info_ex(
        SLAPI_LOG_INFO_AREA_PLUGIN,
        SLAPI_LOG_INFO_LEVEL_DEFAULT,
        msgId,
        connId,
        opId,
        "testpostop_mod in test-postop plug-in",
        "Modified entry (%s)\n", dn
    );

    /* In general, do not interfere in replication operations.       */
    /* Log the DN and the modifications made to the change log file. */
    if (!is_repl) write_changelog(_MOD, dn, (void *) mods, 0);
```

Code Example 5-18 Tracking Modified Entries (`testpostop.c`) *(Continued)*

```
    return (rc);
}
```

First, check that Quentin's entry is in the directory. Quentin's entry is specified in "Prepending a String to an Attribute," on page 99.

Code Example 5-19 Checking the Directory for an Entry

```
$ ldapsearch -L -p port -b "dc=example,dc=com" "(uid=qcubbins)"
dn: uid=qcubbins,ou=People,dc=example,dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
uid: qcubbins
givenName: Quentin
sn: Cubbins
cn: Quentin Cubbins
mail: qcubbins@example.com
secretary: uid=bcubbins,ou=People,dc=example,dc=com
```

After the plug-in is activated in Directory Server, modify Quentin's mail address.

Code Example 5-20 Modifying a Mail Address

```
$ ldapmodify -p port -D "cn=directory manager" -w password
dn: uid=qcubbins,ou=People,dc=example,dc=com
changetype: modify
replace: mail
mail: quentin@example.com
^D
```

Search *ServerRoot*`/slapd-`*serverID*`/logs/errors` for the log message. Minus housekeeping information:

```
Modified entry (uid=qcubbins,ou=people,dc=example,dc=com)
```

Notice also the information logged to `changelog.txt` as shown in Code Example 5-21.

Code Example 5-21 Sample `changelog.txt` Entries After the Modify

```
time: 20020506181305
dn: uid=qcubbins,ou=people,dc=example,dc=com
changetype: modify
replace: mail
mail: quentin@example.com
-
replace: modifiersname
modifiersname: cn=directory manager
-
replace: modifytimestamp
modifytimestamp: 20020506161305Z
-
```

Extending the Rename Operation

This section demonstrates functionality called by Directory Server after a client modify RDN operation. A pre-operation plug-in, not demonstrated here, can be found in *ServerRoot*`/plugins/slapd/slapi/examples/testpreop.c`. Refer to *ServerRoot*`/plugins/slapd/slapi/examples/testpostop.c` for the source code discussed here.

Before using the plug-in function as described here, set up Directory Server as described under “Logging the Entry to Add,” on page 101, making sure to add Quentin’s entry.

The `testpostop_modrdn()` function logs the DN of the modified entry and also writes the entry to a log managed by the plug-in, called `changelog.txt`. The location of `changelog.txt` depends on the platform, as shown in the source code.

Code Example 5-22 performs the logging.

Code Example 5-22 Tracking Renamed Entries (`testpostop.c`)

```
#include "slapi-plugin.h"

int
testpostop_modrdn( Slapi_PBlock *pb )
{
    char * dn;                          /* DN of entry to rename      */
    char * newrdn;                      /* New RDN                    */
    int    dflag;                       /* Delete the old RDN?        */
```

Code Example 5-22 Tracking Renamed Entries (`testpostop.c`) *(Continued)*

```
    int    is_repl = 0;                /* Is this replication?       */
    int    connId, opId, rc = 0;
    long   msgId;

    rc |= slapi_pblock_get(pb, SLAPI_MODRDN_TARGET,           &dn);
    rc |= slapi_pblock_get(pb, SLAPI_MODRDN_NEWRDN,           &newrdn);
    rc |= slapi_pblock_get(pb, SLAPI_MODRDN_DELOLDRDN,        &dflag);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_MSGID,         &msgId);
    rc |= slapi_pblock_get(pb, SLAPI_CONN_ID,                 &connId);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_ID,            &opId);
    rc |= slapi_pblock_get(pb, SLAPI_IS_REPLICATED_OPERATION, &is_repl);

    if (rc != 0) return (rc);
    slapi_log_info_ex(
        SLAPI_LOG_INFO_AREA_PLUGIN,
        SLAPI_LOG_INFO_LEVEL_DEFAULT,
        msgId,
        connId,
        opId,
        "testpostop_modrdn in test-postop plug-in",
        "Modrdn entry (%s)\n", dn
    );

    /* In general, do not interfere in replication operations.       */
    /* Log the DN of the renamed entry, its new RDN, and the flag
     * indicating whether the old RDN was removed to the change log. */
    if (!is_repl) write_changelog(_MODRDN, dn, (void *) newrdn, dflag);

    return (rc);
}
```

Check that Quentin's entry is in the directory as shown in Code Example 5-19 on page 105.

Last weekend, Quentin decided to change his given name to Fred. His user ID is now `fcubbins`, so his entry must be renamed. With this plug-in activated in Directory Server, modify the entry.

Code Example 5-23 Renaming an Entry

```
$ ldapmodify -p port -D "cn=directory manager" -w password
dn: uid=qcubbins,ou=People,dc=example,dc=com
changetype: modify
replace: givenName
givenName: Fred

dn: uid=qcubbins,ou=People,dc=example,dc=com
changetype: modify
replace: mail
```

Code Example 5-23 Renaming an Entry *(Continued)*

```
mail: fcubbins@example.com

dn: uid=qcubbins,ou=People,dc=example,dc=com
changetype: modify
replace: cn
cn: Fred Cubbins

dn: uid=qcubbins,ou=People,dc=example,dc=com
changetype: modrdn
newrdn: uid=fcubbins
deleteoldrdn: 1
^D
```

Search *ServerRoot*/slapd-*serverID*/logs/errors for the log message. Not including housekeeping information:

```
Modrdn entry (uid=qcubbins,ou=people,dc=example,dc=com)
```

Notice also the information logged to changelog.txt as shown in Code Example 5-24.

Code Example 5-24 Sample changelog.txt Entry after Rename

```
time: 20020506184432
dn: uid=qcubbins,ou=people,dc=example,dc=com
changetype: modrdn
newrdn: uid=fcubbins
deleteoldrdn: 1
```

Extending the Delete Operation

This section shows how to develop functionality called by Directory Server after a client delete operation. A pre-operation plug-in, not demonstrated here, can be found in *ServerRoot*/plugins/slapd/slapi/examples/testpreop.c. Refer to *ServerRoot*/plugins/slapd/slapi/examples/testpostop.c for the source code discussed here.

Before using the plug-in function as described here, set up Directory Server as described under “Logging the Entry to Add,” on page 101, making sure to add Quentin’s entry.

The `testpostop_modrdn()` function logs the DN of the modified entry and also writes the entry to a log managed by the plug-in, called `changelog.txt`. The location of `changelog.txt` depends on the platform, as shown in the source code.

Code Example 5-25 performs the logging.

Code Example 5-25 Tracking Entry Deletion (`testpostop.c`)

```
#include "slapi-plugin.h"

int
testpostop_del( Slapi_PBlock *pb )
{
    char * dn;                          /* DN of entry to delete      */
    int    is_repl = 0;                 /* Is this replication?       */
    int    connId, opId, rc = 0;
    long   msgId;

    rc |= slapi_pblock_get(pb, SLAPI_DELETE_TARGET,           &dn);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_MSGID,         &msgId);
    rc |= slapi_pblock_get(pb, SLAPI_CONN_ID,                 &connId);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_ID,            &opId);
    rc |= slapi_pblock_get(pb, SLAPI_IS_REPLICATED_OPERATION, &is_repl);

    if (rc != 0) return (rc);
    slapi_log_info_ex(
        SLAPI_LOG_INFO_AREA_PLUGIN,
        SLAPI_LOG_INFO_LEVEL_DEFAULT,
        msgId,
        connId,
        opId,
        "testpostop_del in test-postop plug-in",
        "Deleted entry (%s)\n", dn
    );

    /* In general, do not interfere in replication operations.       */
    /* Log the DN of the deleted entry to the change log.            */
    if (!is_repl) write_changelog(_DEL, dn, NULL, 0);

    return (rc);
}
```

First, check that Quentin's entry is in the directory as shown in Code Example 5-19 on page 105.

Quentin's name may be Fred if you have renamed it as described in "Extending the Rename Operation," on page 106.

Imagine last Thursday, Quentin shouted copious verbal abuse at a key customer, embarrassing his brother Bartholomew. As a result Quentin has just been fired from Example.com. With this plug-in activated in Directory Server, delete his entry.

```
$ ldapdelete -p port -D "cn=directory manager" -w password \
"uid=qcubbins,ou=People,dc=example,dc=com"
```

Search *ServerRoot*/slapd-*serverID*/logs/errors for the log message. Not including housekeeping information:

```
Deleted entry (uid=qcubbins,ou=people,dc=example,dc=com)
```

Notice also the information logged to changelog.txt as shown in Code Example 5-26.

Code Example 5-26 Sample changelog.txt Entry after Deletion

```
time: 20020506185404
dn: uid=qcubbins,ou=people,dc=example,dc=com
changetype: delete
```

Intercepting Information Sent to the Client

This section demonstrates functionality called by Directory Server before sending a result code, a referral, or an entry to the client application. Refer to *ServerRoot*/plugins/slapd/slapi/examples/testpreop.c for the source code discussed here.

Code Example 5-27 logs the operation number and user DN.

Code Example 5-27 Logging and Responding to the Client (testpreop.c)

```
#include "slapi-plugin.h"

int
testpreop_send(Slapi_PBlock * pb)
{
    Slapi_Operation * op;             /* Operation in progress   */
    char            * connDn;         /* Get DN from connection  */
    int               connId, opId, rc = 0;
    long              msgId;

    rc |= slapi_pblock_get(pb, SLAPI_OPERATION,       &op);
    rc |= slapi_pblock_get(pb, SLAPI_CONN_DN,         &connDn);
    rc |= slapi_pblock_get(pb, SLAPI_CONN_ID,         &connId);
```

Code Example 5-27 Logging and Responding to the Client (`testpreop.c`) *(Continued)*

```
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_MSGID, &msgId);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_ID,    &opId);
    if (rc == 0) {
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
            SLAPI_LOG_INFO_LEVEL_DEFAULT,
            msgId,
            connId,
            opId,
            "testpreop_send in test-preop plug-in",
            "Operation: %d\tUser: %s\n", slapi_op_get_type(op), connDn
        );
    }

    return (rc);
}
```

Operation identifiers are defined in the plug-in header file *ServerRoot*`/plugins/slapd/slapi/include/slapi-plugin.h`. Search for `SLAPI_OPERATION_*`.

Before using the plug-in function as described here, set up the example suffix and register the plug-in as described under “Setting Up an Example Suffix,” on page 86 and “Registering the Plug-In,” on page 89.

With the plug-in activated in Directory Server, perform a search as Yolanda Yorgenson:

```
$ ldapsearch -L -p port -b "dc=example,dc=com" \
-D "uid=yyorgens,ou=people,dc=example,dc=com" -w yyorgens \
"(uid=bcubbins)"
```

Search *ServerRoot*`/slapd-`*serverID*`/logs/errors` for the log messages. Minus housekeeping information, the first message reflects the bind result:

```
Operation: 1 User: uid=yyorgens,ou=people,dc=example,dc=com
```

The next message reflects the search:

```
Operation: 4 User: uid=yyorgens,ou=people,dc=example,dc=com
```

Inside plug-in functions, use `slapi_op_get_type()` to determine the type of an operation. Refer to the *Sun ONE Directory Server Plug-In API Reference* documentation on `slapi_op_get_type()` or to the plug-in header file *ServerRoot*`/plugins/slapd/slapi/include/slapi-plugin.h` for a list of operation types.

Chapter 6

Handling Authentication

This chapter explains how to write a plug-in that adds to, bypasses, or replaces the authentication mechanisms supported for Directory Server 5.2. It demonstrates use of existing mechanisms for authentication. You must adapt the code examples to meet your particular authentication requirements.

CAUTION The examples alone do *not* provide secure authentication methods.

This chapter contains the following sections:

- How Authentication Works
- How a Plug-In Can Modify Authentication
- Developing a Simple Authentication Plug-In
- Developing a SASL Authentication Plug-In

How Authentication Works

This section identifies authentication methods available and describes how Directory Server handles authentication to identify clients. Consider the Directory Server model described here when writing plug-ins to modify the mechanism.

Support for Standard Methods

Directory Server supports the two authentication methods described in RFC 2251: simple authentication, rendered more secure through the use of Secure Socket Layer (SSL) for transport; and Simple Authentication and Security Layer (SASL). RFC 2222, *Simple Authentication and Security Layer (SASL)*, describes the technology. Through SASL, Directory Server supports Kerberos authentication to the LDAP server and the Directory System Agent (DSA, an X.500 term) as described in RFC 1777, *Lightweight Directory Access Protocol*.

Identifying Clients During the Bind

For LDAP clients, Directory Server keeps track of client identity through the DN the client used to connect, and through the authentication method and external credentials the client uses to connect. The parameter block holds the relevant client connection information. The DN can be accessed through the `SLAPI_CONN_DN` and `SLAPI_CONN_AUTHTYPE` parameters to `slapi_pblock_set()` and `slapi_pblock_get()`.

For DSML clients connecting over HTTP, Directory Server performs identity mapping for the bind as described in the *Sun ONE Directory Server Administration Guide*. As a result, plug-ins have the same view of the client bind, regardless of the front end protocol.

Bind Processing in Directory Server

Much happens before Directory Server calls a pre-operation bind plug-in. For example, Directory Server completes authentication for anonymous binds and binds by root and replication users before calling pre-operation bind functions, thus completing the bind without calling the plug-in. To process the bind, Directory Server:

1. Parses the bind request
2. Determines the authentication method
3. Determines whether the bind DN is handled locally
4. Adds request information to the parameter block
5. Determines whether to handle the bind in the front end or to call pre-operation bind plug-in functions

6. Performs the bind or not, using information about the bind DN entry from the server back end

The rest of this section provides more detail for each step.

Parsing the Bind Request

When a bind request arrives, Directory Server retrieves the DN, the authentication method used, and any credentials sent such as a password. This may involve a mapping if the client accesses the server using DSML over HTTP. If the request calls for SASL authentication (`LDAP_AUTH_SASL`), Directory Server retrieves the SASL mechanism. Next, Directory Server normalizes the DN retrieved from the request. It also retrieves any LDAP v3 controls included in the request.

Determining the Authentication Method

If the method is simple authentication, and the DN or credentials are empty or missing, Directory Server assumes the client is binding anonymously, sets `SLAPI_CONN_DN` to `NULL` and `SLAPI_CONN_AUTHTYPE` to `SLAPD_AUTH_NONE`, and sends an `LDAP_SUCCESS` result to the client. If the method is SASL authentication, Directory Server first determines whether the mechanism is supported. If not, it sends an `LDAP_AUTH_METHOD_NOT_SUPPORTED` result to the client.

Determining Whether the DN Is Handled Locally

If the DN retrieved is not server locally, and Directory Server is configured for default referral, it sends an `LDAP_REFERRAL` result to the client. Otherwise, it sends an `LDAP_NO_SUCH_OBJECT` result to the client.

Adding Request Information to the Parameter Block

Directory Server puts bind request information into the parameter block:

- `SLAPI_BIND_TARGET` to the DN retrieved from the request
- `SLAPI_BIND_METHOD` to the authentication method requested
- `SLAPI_BIND_CREDENTIALS` to the credentials retrieved from the request
- `SLAPI_BIND_MECHANISM` to the name of the SASL mechanism, if applicable

Determining Whether to Handle the Bind or Call a Plug-In

Directory Server authenticates bind requests from the directory manager (root user whose DN is specified as the value of `nsslapd-rootdn`) or the replication manager (DN used for binds related to replication), sending an `LDAP_SUCCESS` result on success or an `LDAP_INVALID_CREDENTIALS` result on failure.

After completing all this work, the server calls pre-operation bind functions.

Directory Server does not call pre-operation bind functions for bind requests from the directory manager, the replication manager, nor for anonymous binds.

Performing the Backend Bind

Assuming any pre-operation bind functions called return `0`, Directory Server completes the bind operation, checking the bind against the information in the directory database, setting `SLAPI_CONN_DN` and `SLAPI_CONN_AUTHTYPE` appropriately. If necessary, Directory Server sends a "password expiring" control (OID 2.16.840.1.113730.3.4.5) result to the client. It always sends an `LDAP_SUCCESS` result on success or some other (non-zero) result on failure.

How a Plug-In Can Modify Authentication

A pre-operation bind function can modify Directory Server authentication in one of two ways. The plug-in either:

- Completely bypasses the comparison of incoming authentication information to authentication information stored in the directory database, or
- Implements a custom SASL mechanism.

Bypassing Authentication

If the plug-in bypasses the comparison of authentication information in the client request to authentication information stored in the directory, then the plug-in returns a non-zero value, such as 1, to prevent the server from completing the bind after the pre-operation bind function returns. You may opt for this solution if you store all authentication information outside the directory, and for some reason cannot adequately map to information available in the directory through LDAP or the plug-in API. In addition to the other testing and validation performed on the plug-in, make sure you verify that it works well with server access control mechanisms.

Refer to "Developing a Simple Authentication Plug-In," on page 117 for an example.

Using Custom SASL Mechanisms

If the plug-in implements a custom SASL mechanism then clients using the mechanism must support it as well. Refer to the documentation for the iPlanet Directory SDK for C for example for details on developing client applications to support specific bind requirements. An SDK for the Java language is also available.

Refer to “Developing a SASL Authentication Plug-In,” on page 125 for an example.

Developing a Simple Authentication Plug-In

This section shows how a pre-operation bind plug-in can use the plug-in API to authenticate a user. The example discussed here obtains the appropriate bind information from the parameter block, then handles the authentication if the method requested is `LDAP_AUTH_SIMPLE`, but allows the server to continue the bind if the authentication succeeds.

Notice some binds are performed by the front end before pre-operation bind functions are called. Refer to “Determining the Authentication Method,” on page 115 and “Determining Whether to Handle the Bind or Call a Plug-In,” on page 115 for details.

Finding the Example

Code Example 6-1 shows the code excerpt from the source file *ServerRoot*`/plugins/slapd/slapi/examples/testbind.c`.

Code Example 6-1 Pre-Operation Bind Function (`testbind.c`)

```
#include "slapi-plugin.h"

int
test_bind(Slapi_PBlock * pb)
{
    char          *  dn;              /* Target DN                */
    int              method;          /* Authentication method    */
    struct berval *  credentials;     /* Client SASL credentials  */
    Slapi_DN      *  sdn      = NULL;  /* DN used in internal srch  */
    char          *  attrs[2] = {      /* Look at userPassword only */
                              SLAPI_USERPWD_ATTR,
                              NULL
                          };
    Slapi_Entry   *  entry    = NULL;  /* Entry returned by srch    */
    Slapi_Attr    *  attr     = NULL;  /* Pwd attr in entry found   */
```

Code Example 6-1 Pre-Operation Bind Function (`testbind.c`) *(Continued)*

```
    int             is_repl;          /* Is this replication?      */
    int             is_intl;          /* Is this an internal op?   */
    int             connId, opId, rc = 0;
    long            msgId;

    /* Obtain the bind information from the parameter block.        */
    rc |= slapi_pblock_get(pb, SLAPI_BIND_TARGET,             &dn);
    rc |= slapi_pblock_get(pb, SLAPI_BIND_METHOD,             &method);
    rc |= slapi_pblock_get(pb, SLAPI_BIND_CREDENTIALS,        &credentials);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_MSGID,         &msgId);
    rc |= slapi_pblock_get(pb, SLAPI_CONN_ID,                 &connId);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_ID,            &opId);
    rc |= slapi_pblock_get(pb, SLAPI_IS_REPLICATED_OPERATION, &is_repl);
    rc |= slapi_pblock_get(pb, SLAPI_IS_INTERNAL_OPERATION,   &is_intl);

    if (rc != 0) {
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
            SLAPI_LOG_INFO_LEVEL_DEFAULT,
            SLAPI_LOG_NO_MSGID,
            SLAPI_LOG_NO_CONNID,
            SLAPI_LOG_NO_OPID,
            "test_bind in test-bind plug-in",
            "Could not get parameters for bind operation (error %d).\n", rc
        );
        slapi_send_ldap_result(pb, LDAP_OPERATIONS_ERROR, NULL, NULL, 0, NULL);
        return (LDAP_OPERATIONS_ERROR);/* Server aborts bind here.  */
    }

    /* The following code handles simple authentication, where a
     * user offers a bind DN and a password for authentication.
     *
     * Handling simple authentication is a matter of finding the
     * entry corresponding to the bind DN sent in the request,
     * then if the entry is found, checking whether the password
     * sent in the request matches a value found on the
     * userPassword attribute of the entry.                        */

    /* Avoid interfering with replication or internal operations.   */
    if (!is_repl && !is_intl) switch (method) {
    case LDAP_AUTH_SIMPLE:

        /* Find the entry specified by the bind DN...               */
        sdn = slapi_sdn_new_dn_byref(dn);
        rc |= slapi_search_internal_get_entry(
            sdn,
            attrs,
            &entry,
            plugin_id
            );
        slapi_sdn_free(&sdn);

        if (rc != 0 || entry == NULL) {
            slapi_log_info_ex(
                SLAPI_LOG_INFO_AREA_PLUGIN,
```

Code Example 6-1 Pre-Operation Bind Function (`testbind.c`) *(Continued)*

```
                SLAPI_LOG_INFO_LEVEL_DEFAULT,
                msgId,
                connId,
                opId,
                "test_bind in test-bind plug-in",
                "Could not find entry: %s\n", dn
            );
            rc = LDAP_NO_SUCH_OBJECT;
            slapi_send_ldap_result(pb, rc, NULL, NULL, 0, NULL);
            return (rc);
        } else {
        /* ...check credentials against the userpassword...         */
            Slapi_Value    *  credval; /* Value of credentials      */
            Slapi_ValueSet * pwvalues; /* Password attribute values */

            rc |= slapi_entry_attr_find(
                entry,
                SLAPI_USERPWD_ATTR,
                &attr
            );

            if (attr == NULL) {

                slapi_log_info_ex(
                    SLAPI_LOG_INFO_AREA_PLUGIN,
                    SLAPI_LOG_INFO_LEVEL_DEFAULT,
                    msgId,
                    connId,
                    opId,
                    "test_bind in test-bind plug-in",
                    "Entry %s has no userpassword.\n",
                    dn
                );
                rc = LDAP_INAPPROPRIATE_AUTH;
                slapi_send_ldap_result(pb, rc, NULL, NULL, 0, NULL);
                return (rc);
            }

            rc |= slapi_attr_get_valueset(
                attr,
                &pwvalues
            );
            if (rc != 0 || slapi_valueset_count(pwvalues) == 0) {
                slapi_log_info_ex(
                    SLAPI_LOG_INFO_AREA_PLUGIN,
                    SLAPI_LOG_INFO_LEVEL_DEFAULT,
                    msgId,
                    connId,
                    opId,
                    "test_bind in test-bind plug-in",
                    "Entry %s has no %s attribute values.\n",
                    dn, SLAPI_USERPWD_ATTR
                );
                rc = LDAP_INAPPROPRIATE_AUTH;
                slapi_send_ldap_result(pb, rc, NULL, NULL, 0, NULL);
```

Code Example 6-1 Pre-Operation Bind Function (`testbind.c`) *(Continued)*

```
                            slapi_valueset_free(pwvalues);
                            return (rc);
        }

        credval = slapi_value_new_berval(credentials);

        rc = slapi_pw_find_valueset(pwvalues, credval);

        slapi_value_free(&credval);
                    slapi_valueset_free(pwvalues);

        if (rc != 0) {
            slapi_log_info_ex(
                SLAPI_LOG_INFO_AREA_PLUGIN,
                SLAPI_LOG_INFO_LEVEL_DEFAULT,
                msgId,
                connId,
                opId,
                "test_bind in test-bind plug-in",
                "Credentials are not correct.\n"
            );
            rc = LDAP_INVALID_CREDENTIALS;
            slapi_send_ldap_result(pb, rc, NULL, NULL, 0, NULL);
            return (rc);
        }
    }

    /* ...if successful, set authentication for the connection. */
    if (rc != 0) return (rc);
    rc |= slapi_pblock_set(pb, SLAPI_CONN_DN, slapi_ch_strdup(dn));
    rc |= slapi_pblock_set(pb, SLAPI_CONN_AUTHMETHOD, SLAPD_AUTH_SIMPLE);
    if (rc != 0) {
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
            SLAPI_LOG_INFO_LEVEL_DEFAULT,
            msgId,
            connId,
            opId,
            "test_bind in test-bind plug-in",
            "Failed to set connection info.\n"
        );
        rc = LDAP_OPERATIONS_ERROR;
        slapi_send_ldap_result(pb, rc, NULL, NULL, 0, NULL);
        return (rc);
    } else {
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
            SLAPI_LOG_INFO_LEVEL_DEFAULT,
            msgId,
            connId,
            opId,
            "test_bind in test-bind plug-in",
            "Authenticated: %s\n", dn
        );
```

Code Example 6-1 Pre-Operation Bind Function (`testbind.c`) *(Continued)*

```
            /* Now that authentication succeeded, the plug-in
             * returns a value greater than 0, even though the
             * authentication has been successful. A return
             * code > 0 tells the server not to continue
             * processing the bind. A return code of 0, such
             * as LDAP_SUCCESS tells the server to continue
             * processing the operation.                          */
            slapi_send_ldap_result(pb, LDAP_SUCCESS, NULL, NULL, 0, NULL);
            rc = 1;
        }
        break;

    /* This plug-in supports only simple authentication.          */
    case LDAP_AUTH_NONE:
        /* Anonymous binds are handled by the front-end before
         * pre-bind plug-in functions are called, so we should
         * never reach this part of the code.                     */
    case LDAP_AUTH_SASL:
    default:
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
            SLAPI_LOG_INFO_LEVEL_DEFAULT,
            msgId,
            connId,
            opId,
            "test_bind in test-bind plug-in",
            "Plug-in does not handle auth. method: %d\n", method
        );
        rc = 0;                         /* Let server handle bind.   */
        break;
    }

    return (rc);                        /* Server stops processing
                                         * the bind if rc > 0.       */
}
```

The bulk of the code processes the `LDAP_AUTH_SIMPLE` case, where the plug-in uses the DN and the password to authenticate the user requesting the bind through plug-in API calls.

Seeing It Work

We demonstrate the plug-in by turning on informational logging for plug-ins and reading the log messages written by the plug-in at different stages in its operation. Before using the plug-in, we load some example users and data, because we cannot demonstrate the functionality while binding as the root user whose DN is specified as the value of `nsslapd-rootdn`. As described in "Determining Whether to Handle the Bind or Call a Plug-In," on page 115, Directory Server processes the directory manager bind without calling the pre-operation bind functions.

Setting Up an Example Suffix

If you have not done so already, prepare some data by creating a directory suffix, `dc=example,dc=com`, whose users we load from an LDIF file, *`ServerRoot`*`/slapd-`*`serverID`*`/ldif/Example-Plugin.ldif`. Refer to "Setting Up an Example Suffix," on page 86 for instructions.

Building and Registering the Plug-In

1. If you have not already done so, build the plug-in.

   Hint Use *`ServerRoot`*`/plugins/slapd/slapi/examples/Makefile`.

2. Edit the value of `nsslapd-pluginPath` in the configuration entry taken from the introductory comment in `testbind.c`.

Code Example 6-2 Configuration Entry (`testbind.c`)

```
dn: cn=Test Bind,cn=plugins,cn=config
objectClass: top
objectClass: nsSlapdPlugin
objectClass: extensibleObject
cn: Test Bind
nsslapd-pluginPath: <ServerRoot>/plugins/slapd/slapi/examples/<LibName>
nsslapd-pluginInitfunc: testbind_init
nsslapd-pluginType: preoperation
nsslapd-pluginEnabled: on
nsslapd-plugin-depends-on-type: database
nsslapd-pluginId: test-bind
nsslapd-pluginVersion: 5.2
nsslapd-pluginVendor: Sun Microsystems, Inc.
nsslapd-pluginDescription: Sample bind pre-operation plug-in
```

   For 64-bit plug-ins, refer to "64-Bit Plug-In Locations," on page 66.

3. Save the configuration entry as `testbind.ldif`.

4. Load the configuration entry into the directory.

```
$ ldapmodify -a -p port -D "cn=directory manager" -w password -f testbind.ldif
```

5. Turn on informational logging for plug-ins.

   Hint `Configuration` tab page. Select `Logs` node. Set `Log Level`.

6. Restart Directory Server.

When Directory Server Bypasses the Plug-In

The example suffix contains a number of people. If we look up the entry for one of those people, Arabella Godiva, as either the directory manager (`nsslapd-rootdn`) user or as an anonymous client, the `test_bind()` plug-in function is never called. The plug-in therefore never logs informational messages to the `errors` log.

Code Example 6-3 Bypassing the Plug-In

```
$ ldapsearch -L -p port -D "cn=directory manager" -w password \
-b "dc=example,dc=com" "(uid=agodiva)" uid
dn: uid=agodiva,ou=People,dc=example,dc=com
uid: agodiva
$ ldapsearch -L -p port -b "dc=example,dc=com" "(uid=agodiva)" uid
dn: uid=agodiva,ou=People,dc=example,dc=com
uid: agodiva
$ grep test_bind ServerRoot/slapd-serverID/logs/errors
$
```

Here we see the server bypasses pre-operation bind plug-ins when special users request a bind.

Binding as an Example.com User

First, check what happens in the errors log when we bind as Arabella Godiva using the correct password, `agodiva`, as shown in Code Example 6-4.

Code Example 6-4 Bind Involving the Plug-In

```
$ ldapsearch -L -p port -D "uid=agodiva,ou=People,dc=example,dc=com" \
-w agodiva -b "dc=example,dc=com" "(uid=agodiva)" uid
dn: uid=agodiva,ou=People,dc=example,dc=com
uid: agodiva
$ grep test_bind ServerRoot/slapd-serverID/logs/errors
```

Code Example 6-4 Bind Involving the Plug-In *(Continued)*

```
[07/May/2002:16:50:16 +0200] - INFORMATION - test_bind in test-bind plug-in -
conn=7 op=0 msgId=1 -  Authenticated: uid=agodiva,ou=people,dc=example,dc=com
```

What happens when we bind as Arabella Godiva, but get the password wrong?

Code Example 6-5 Wrong Password

```
$ ldapsearch -L -p port -D "uid=agodiva,ou=People,dc=example,dc=com" \
-w AAARGH -b "dc=example,dc=com" "(uid=agodiva)" uid
ldap_simple_bind_s: Invalid credentials
$ grep test_bind ServerRoot/slapd-serverID/logs/errors
...
[07/May/2002:16:51:18 +0200] - INFORMATION - test_bind in test-bind plug-in -
conn=8 op=0 msgId=1 -  Credentials are not correct.
```

We see here the LDAP result interpreted correctly by the command-line client, and the plug-in message to the same effect written to the `errors` log.

Delete Arabella's password and try again.

Code Example 6-6 No Password

```
$ ldapmodify -p port -D "cn=directory manager" -w password
dn: uid=agodiva,ou=people,dc=example,dc=com
changetype: modify
delete: userpassword
^D
modifying entry uid=agodiva,ou=people,dc=example,dc=com
$ ldapsearch -L -p port -D "uid=agodiva,ou=People,dc=example,dc=com" \
-b "dc=example,dc=com" "(uid=agodiva)" uid
Bind Password:
ldap_simple_bind_s: Inappropriate authentication
$ grep test_bind ServerRoot/slapd-serverID/logs/errors
[07/May/2002:16:54:25 +0200] - INFORMATION - test_bind in test-bind plug-in -
conn=12 op=0 msgId=1 -  Entry uid=agodiva,ou=people,dc=example,dc=com has no
userpassword attribute values
```

We see the LDAP result displayed correctly by the command-line client, and the plug-in message giving more information about what went wrong during Arabella's attempt to bind (`no userpassword attribute values`).

Developing a SASL Authentication Plug-In

This section shows how a pre-bind operation plug-in can implement a custom Simple Authentication and Security Layer (SASL) mechanism that enables mutual authentication without affecting existing authentication mechanisms.

The example plug-in responds to SASL bind requests with a mechanism called `my_sasl_mechanism`. Binds with this mechanism always succeed. The example is intended only to illustrate how a SASL authentication plug-in works, not to provide a secure mechanism for use in production.

Finding Examples

This section covers the examples *ServerRoot*`/plugins/slapd/slapi/examples/testsaslbind.c` and *ServerRoot*`/plugins/slapd/slapi/examples/clients/saslclient.c`.

Before using the plug-in function as described here, set up the example suffix and register the plug-in as described under "Developing a Simple Authentication Plug-In," on page 117, loading the configuration entry from the introductory comment of `testsaslbind.c`. Code Example 6-7 shows the entry.

Code Example 6-7 Configuration Entry (`testsaslbind.c`)

```
dn: cn=Test SASL bind,cn=plugins,cn=config
objectclass: top
objectclass: nsSlapdPlugin
objectclass: extensibleObject
cn: SASL bind test
nsslapd-pluginPath: <ServerRoot>/plugins/slapd/slapi/examples/<LibName>
nsslapd-plugininitfunc: testsasl_init
nsslapd-plugintype: preoperation
nsslapd-pluginenabled: on
nsslapd-pluginid: test-saslbind
nsslapd-pluginversion: 5.2
nsslapd-pluginvendor: Sun Microsystems, Inc.
nsslapd-plugindescription: Sample SASL pre-bind plug-in
```

Adjust `nsslapd-pluginPath` to fit how the product is installed on your system. For 64-bit plug-ins, refer to "64-Bit Plug-In Locations," on page 66.

Registering the SASL Mechanism

Register the SASL mechanism in the same function that registers the plug-in, using `slapi_register_supported_saslmechanism()`. Code Example 6-8 shows the function that registers both the plug-in and the SASL mechanism.

Code Example 6-8 Registering a Custom SASL Plug-In (`testsaslbind.c`)

```
#include "slapi-plugin.h"

Slapi_PluginDesc saslpdesc = {
    "test-saslbind",                /* plug-in identifier          */
    "Sun Microsystems, Inc.",       /* vendor name                 */
    "5.2",                          /* plug-in revision number     */
    "Sample SASL pre-bind plug-in"  /* plug-in description         */
};

#define TEST_MECHANISM  "my_sasl_mechanism"
#define TEST_AUTHMETHOD SLAPD_AUTH_SASL TEST_MECHANISM

/* Register the plug-in with the server.                           */
#ifdef _WIN32
__declspec(dllexport)
#endif
int
testsasl_init(Slapi_PBlock * pb)
{
    int rc = 0;                     /* 0 means success             */
    rc |= slapi_pblock_set(
        pb,
        SLAPI_PLUGIN_VERSION,
        SLAPI_PLUGIN_CURRENT_VERSION
    );
    rc |= slapi_pblock_set(         /* Plug-in description         */
        pb,
        SLAPI_PLUGIN_DESCRIPTION,
        (void *) &saslpdesc
    );
    rc |= slapi_pblock_set(         /* Pre-op bind SASL function   */
        pb,
        SLAPI_PLUGIN_PRE_BIND_FN,
        (void *) testsasl_bind
    );

    /* Register the SASL mechanism.                                */
    slapi_register_supported_saslmechanism(TEST_MECHANISM);
    return (rc);
}
```

Refer to the *Sun ONE Directory Server Plug-In API Reference* for details on using `slapi_register_supported_saslmechanism()`.

The plug-in function `testsasl_bind()` first obtains the client DN, method, SASL mechanism, and credentials. If the client does not ask to use the `TEST_MECHANISM`, the function returns `0` so the server can process the bind. Code Example 6-9 shows how this is done.

Code Example 6-9 Handling the `my_sasl_mechanism` Bind (`testsaslbind.c`)

```
#include "slapi-plugin.h"

#define TEST_MECHANISM  "my_sasl_mechanism"
#define TEST_AUTHMETHOD SLAPD_AUTH_SASL TEST_MECHANISM

int
testsasl_bind(Slapi_PBlock * pb)
{
    char          * dn;                /* Target DN                    */
    int             method;            /* Authentication method        */
    char          * mechanism;         /* SASL mechanism               */
    struct berval * credentials;       /* SASL client credentials      */
    struct berval   svrcreds;          /* SASL server credentials      */
    int             connId, opId, rc = 0;
    long            msgId;

    rc |= slapi_pblock_get(pb, SLAPI_BIND_TARGET,        &dn);
    rc |= slapi_pblock_get(pb, SLAPI_BIND_METHOD,        &method);
    rc |= slapi_pblock_get(pb, SLAPI_BIND_CREDENTIALS,   &credentials);
    rc |= slapi_pblock_get(pb, SLAPI_BIND_SASLMECHANISM, &mechanism);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_MSGID,    &msgId);
    rc |= slapi_pblock_get(pb, SLAPI_CONN_ID,            &connId);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_ID,       &opId);
    if (rc == 0) {
        if (mechanism == NULL || strcmp(mechanism, TEST_MECHANISM) != 0)
            return(rc);                /* Client binding another way. */
    } else {                           /* slapi_pblock_get() failed!  */
        return(rc);
    }

    /* Using our SASL mechanism that always succeeds, set conn. info. */
    rc |= slapi_pblock_set(pb, SLAPI_CONN_DN,         slapi_ch_strdup(dn));
    rc |= slapi_pblock_set(pb, SLAPI_CONN_AUTHMETHOD, TEST_AUTHMETHOD);
    if (rc != 0) {                     /* Failed to set conn. info!   */
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
        SLAPI_LOG_INFO_LEVEL_DEFAULT,
            msgId,
            connId,
            opId,
            "testsasl_bind in test-saslbind plug-in",
            "slapi_pblock_set() for connection information failed.\n"
        );
        slapi_send_ldap_result(pb, LDAP_OPERATIONS_ERROR, NULL, NULL, 0, NULL);
```

Code Example 6-9 Handling the `my_sasl_mechanism` Bind (`testsaslbind.c`) *(Continued)*

```
        return(LDAP_OPERATIONS_ERROR); /* Server tries other mechs.   */
    }

    /* Set server credentials.                                       */
    svrcreds.bv_val = "my credentials";
    svrcreds.bv_len = sizeof("my credentials") - 1;

    rc |= slapi_pblock_set(pb, SLAPI_BIND_RET_SASLCREDS, &svrcreds);
    if (rc != 0) {                    /* Failed to set credentials!  */
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
        SLAPI_LOG_INFO_LEVEL_DEFAULT,
            msgId,
            connId,
            opId,
            "testsasl_bind in test-saslbind plug-in",
            "slapi_pblock_set() for server credentials failed.\n"
        );
        rc |= slapi_pblock_set(pb, SLAPI_CONN_DN,         NULL);
        rc |= slapi_pblock_set(pb, SLAPI_CONN_AUTHMETHOD, SLAPD_AUTH_NONE);
        return(LDAP_OPERATIONS_ERROR); /* Server tries other mechs.   */
    }

    /* Send credentials to client.                                   */
    slapi_log_info_ex(
        SLAPI_LOG_INFO_AREA_PLUGIN,
        SLAPI_LOG_INFO_LEVEL_DEFAULT,
        msgId,
        connId,
        opId,
        "testsasl_bind in test-saslbind plug-in",
        "Authenticated: %s\n", dn
    );
    slapi_send_ldap_result(pb, LDAP_SUCCESS, NULL, NULL, 0, NULL);
    return 1;                         /* Server stops processing the
                                       * bind if the plug-in returns
                                       * a value greater than 0.     */
}
```

Code Example 6-9 sets the DN for the client connection and the authentication method, as Directory Server does for a bind. Setting `SLAPI_CONN_DN` and `SLAPI_CONN_AUTHMETHOD` is the key step of the bind. Without these values in the parameter block, Directory Server handles the client request as if the bind were anonymous, leaving the client with access rights corresponding to an anonymous bind. Next, according to the SASL model, Directory Server sends credentials to the client as shown.

Developing the Client

To test our plug-in, we need a client that uses `my_sasl_mechanism` to bind. The example discussed here uses the mechanism, and the client code, `saslclient.c`, is delivered with the product. The iPlanet Directory SDK for C, available separately as described in “Downloading Directory Server Tools,” on page 16, is used to build the client.

The client authenticates using the example SASL mechanism for a synchronous bind to Directory Server. Code Example 6-10 shows how authentication proceeds on the client side for Soapy Cooper.

Code Example 6-10 Client Using `my_sasl_mechanism` (`clients/saslclient.c`)

```
#include "ldap.h"

/* Use our fake SASL mechanism.                                          */
#define MECH "my_sasl_mechanism"

/* Global variables for client connection information.
 * You may set these here or on the command line.                        */
static char * host = "localhost";      /* Server hostname                */
static int    port = 389;              /* Server port                    */
/* Load <ServerRoot>/slapd-<ServerID>/ldif/Example-Plugin.ldif
 * before trying the plug-in with this default user.                     */
static char * user = "uid=scooper,ou=people,dc=example,dc=com";
/* New value for userPassword                                            */
static char * npwd = "23skidoo";

/* Check for host, port, user and new password as arguments.             */
int get_user_args(int argc, char ** argv);

int
main(int argc, char ** argv)
{
    LDAP          * ld;                /* Handle to LDAP connection  */
    LDAPMod         modPW, * mods[2];  /* For modifying the password */
    char          * vals[2];           /* Value of modified password */
    struct berval   cred;              /* Client bind credentials    */
    struct berval * srvCred;           /* Server bind credentials    */
    int             ldapVersion;

    /* Use default hostname, server port, user, and new password
     * unless they are provided as arguments on the command line.    */
    if (get_user_args(argc, argv) != 0) return 1; /* Usage error     */

    /* Get a handle to an LDAP connection.                           */
    printf("Getting the handle to the LDAP connection...\n");
    if ((ld = ldap_init(host, port)) == NULL) {
        perror("ldap_init");
        return 1;
    }
```

Code Example 6-10 Client Using `my_sasl_mechanism` (`clients/saslclient.c`) *(Continued)*

```
    /* By default, the LDAP version is set to 2.                    */
    printf("Setting the version to LDAP v3...\n");
    ldapVersion = LDAP_VERSION3;
    ldap_set_option(ld, LDAP_OPT_PROTOCOL_VERSION, &ldapVersion);

    /* Authenticate using the example SASL mechanism.               */
    printf("Bind DN is %s...\n", user);
    printf("Binding to the server using %s...\n", MECH);
    cred.bv_val = "magic";
    cred.bv_len = sizeof("magic") - 1;
    if (ldap_sasl_bind_s(ld, user, MECH, &cred, NULL, NULL, &srvCred)) {
        ldap_perror(ld, "ldap_sasl_bind_s");
        return 1;
    }

    /* Display the credentials returned by the server.              */
    printf("Server credentials: %s\n", srvCred->bv_val);

    /* Modify the user's password.                                  */
    printf("Modifying the password...\n");
    modPW.mod_op     = LDAP_MOD_REPLACE;
    modPW.mod_type   = "userpassword";
    vals[0]          = npwd;
    vals[1]          = NULL;
    modPW.mod_values = vals;

    mods[0] = &modPW; mods[1] = NULL;

    if (ldap_modify_ext_s(ld, user, mods, NULL, NULL)) {
        ldap_perror(ld, "ldap_modify_ext_s");
        return 1;
    }

    /* Finish up.                                                   */
    ldap_unbind(ld);
    printf("Modification was successful.\n");
    return 0;
}
```

Trying It Out

The client changes Soapy's password by binding to Directory Server, then requesting a modification to the `userPassword` attribute value. To be able to read the new password as clear text, set the password storage scheme for user passwords to `No encryption (CLEAR)` using the console. (**Hint** `Configuration` tab page. `Data` node. Use the `Password encryption` drop-down list.)

As the directory manager, change Soapy Cooper's password to `newpassword`. Verify the change, and that the password appears in clear text.

Code Example 6-11 Modifying a Password

```
$ ldapmodify -p port -D "cn=directory manager" -w password
dn: uid=scooper,ou=People,dc=example,dc=com
changetype: modify
replace: userpassword
userpassword: newpassword
^D
modifying entry uid=scooper,ou=People,dc=example,dc=com
$ ldapsearch -L -p port -D "cn=directory manager" -w password \
-b "dc=example,dc=com" "(uid=scooper)"
dn: uid=scooper,ou=People,dc=example,dc=com
mail: scooper@example.com
uid: scooper
secretary: uid=bcubbins,ou=People,dc=example,dc=com
givenName: Soapy
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
sn: Cooper
cn: Soapy Cooper
userPassword: newpassword
```

The client can then change Soapy's password to `soapy` after using the example SASL mechanism to bind.

With the plug-in active in the server, compile the client, `saslclient.c`. One way of doing this with the iPlanet Directory SDK for C involves adding a `saslclient` target to `lib/ldapcsdk/examples/Makefile`.

Code Example 6-12 `Makefile` Additions to Build the SASL Client

```
saslclient:     saslclient.o
    $(CC) -o saslclient saslclient.o $(LIBS)
```

Next, copy `saslclient.c` to the `lib/ldapcsdk/examples/` directory and build the client. The `Makefile` works with GNU Make.

Code Example 6-13 Building the SASL Client

```
$ cd sdk_install_dir/lib/ldapcsdk/examples
$ cp ServerRoot/plugins/slapd/slapi/examples/clients/saslclient.c \
sdk_install_dir/lib/ldapcsdk/examples
$ gmake saslclient
```

Next, run the client to perform the bind and the password modification.

Code Example 6-14 Using the SASL Client

```
$ ./saslclient -p port
Using the following connection info:
        host:    localhost
        port:    port
        bind DN: uid=scooper,ou=people,dc=example,dc=com
        new pwd: 23skidoo
Getting the handle to the LDAP connection...
Setting the version to LDAP v3...
Bind DN is uid=scooper,ou=people,dc=example,dc=com...
Binding to the server using my_sasl_mechanism...
Server credentials: my credentials
Modifying the password...
Modification was successful.
```

On the Directory Server side, we see a message in the `errors` log showing that the plug-in authenticated the client:

```
[10/May/2002:14:50:48 +0200] - INFORMATION - testsasl_bind in
test-saslbind plug-in - conn=13 op=0 msgId=1 -  Authenticated:
uid=scooper,ou=people,dc=example,dc=com
```

As directory manager, we can see Soapy's password has changed from `newpassword` to `soapy`.

Code Example 6-15 Results of Password Modification after SASL Bind

```
$ ldapsearch -L -p port -D "cn=directory manager" -w password \
-b "dc=example,dc=com" "(uid=scooper)"
dn: uid=scooper,ou=People,dc=example,dc=com
mail: scooper@example.com
uid: scooper
secretary: uid=bcubbins,ou=People,dc=example,dc=com
```

Code Example 6-15 Results of Password Modification after SASL Bind *(Continued)*

```
givenName: Soapy
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
sn: Cooper
cn: Soapy Cooper
userPassword: soapy
```

We have thus demonstrated that our example SASL mechanism provides an alternative way of binding to the server.

Chapter 7

Performing Internal Operations

This chapter covers how to perform search, add, modify, modify RDN, and delete operations for which no corresponding client requests exist.

Finding the Example

The plug-in code used in this chapter can be found in *ServerRoot*/plugins/slapd/slapi/examples/internal.c.

Before Using the Example

Before using the example plug-in `internal.c` create a new suffix, `dc=example,dc=com`, then load the entries from the LDIF example file *ServerRoot*/slapd-*serverID*/ldif/Example-Plugin.ldif into the directory. Refer to "Setting Up an Example Suffix," on page 86 for instructions.

Refer to the *Sun ONE Directory Server Plug-In API Reference* for complete reference information concerning the plug-in API functions used in this chapter.

This chapter includes the following sections:

- Using Internal Operations
- Internal Add
- Internal Modify
- Internal Rename (Modify RDN)
- Internal Search
- Internal Delete

Using Internal Operations

This section demonstrates use of internal search, add, modify, modify RDN, and delete operations.

When to Use Internal Operations

Internal operations are *internal* in the sense that they are initiated not by external requests from clients, but instead internally by plug-ins. Use internal operation calls when your plug-in needs Directory Server to perform an operation for which no client request exists.

Caveats

You set up the parameter blocks and handle all memory management directly when developing with internal operations. Debugging against optimized binaries such as those delivered with the product can be tedious. Review the code carefully. You may want to work with a partner who can point out errors you miss. Memory management mistakes around internal operation calls lead more quickly to inscrutable segmentation faults than other calls in the plug-in API.

Furthermore, internal operations result in updates to some internal caches but not others. For example, changes to access control instructions cause updates to the access control cache. Internal operation changes to attributes used in CoS and roles do not cause CoS and roles caches to be updated.

The rest of this chapter concerns individual internal operations.

Internal Add

For an internal add, you allocate space for a parameter block, set it up for the add with `slapi_add_entry_internal_set_pb()` so that the entry is in the parameter block, invoke `slapi_add_internal_pb()`, and free the parameter block. The internal add consumes the entry passed in to the server through the parameter block.

Code Example 7-1 demonstrates the process for an internal add.

Code Example 7-1 Internal Add Operation (`internal.c`)

```
#include "slapi-plugin.h"

/* Internal operations require an ID for the plug-in.                    */
static Slapi_ComponentId * plugin_id     = NULL;

int
test_internal()
{
    Slapi_DN       * sdn;             /* DN holder for internal ops    */
    Slapi_Entry    * entry;           /* Entry holder for internal ops */
    Slapi_PBlock   * pb;              /* PBlock for internal ops       */
    char           * str = NULL;      /* String holder for internal ops*/
    int              len;             /* Length of LDIF from entry     */
    int              rc;              /* Return code; 0 means success. */

    /* Check that the example suffix exists.                            */
    sdn = slapi_sdn_new_dn_byval("dc=example,dc=com");
    if (slapi_be_exist(sdn)) {
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
            SLAPI_LOG_INFO_LEVEL_DEFAULT,
            SLAPI_LOG_NO_MSGID,
            SLAPI_LOG_NO_CONNID,
            SLAPI_LOG_NO_OPID,
            "test_internal in test-internal plug-in",
            "Suffix (%s) does not exist, exiting.\n",
            slapi_sdn_get_dn(sdn)
        );
        slapi_sdn_free(&sdn);
        return (0);
    }
    slapi_sdn_free(&sdn);

    /*
     * Add an entry for Quentin Cubbins to the example suffix
     * using slapi_add_entry_internal().                                */
    entry = slapi_entry_alloc();
    slapi_entry_set_dn(               /* slapi_entry_set_dn()           */
        entry,                        /* requires a copy of the DN.     */
        slapi_ch_strdup("uid=qcubbins,ou=People,dc=example,dc=com")
    );
                                      /* slapi_entry_add_string()       */
                                      /* does not require a copy.       */
    slapi_entry_add_string(entry, "objectclass", "top");
    slapi_entry_add_string(entry, "objectclass", "person");
    slapi_entry_add_string(entry, "objectclass", "organizationalPerson");
    slapi_entry_add_string(entry, "objectclass", "inetOrgPerson");
    slapi_entry_add_string(entry, "uid", "qcubbins");
    slapi_entry_add_string(entry, "givenName", "Quentin");
    slapi_entry_add_string(entry, "sn", "Cubbins");
    slapi_entry_add_string(entry, "cn", "Quentin Cubbins");
    slapi_entry_add_string(entry, "mail", "qcubbins@example.com");
    slapi_entry_add_string(entry, "userPassword", "qcubbins");
    slapi_entry_add_string(entry, "secretary",
```

Code Example 7-1 Internal Add Operation (`internal.c`) *(Continued)*

```
        "uid=bcubbins,ou=People,dc=example,dc=com");

    pb = slapi_pblock_new();           /* Make our own PBlock...        */
    rc = slapi_add_entry_internal_set_pb(
        pb,
        entry,
        NULL,                          /* No controls                   */
        plugin_id,
        SLAPI_OP_FLAG_NEVER_CHAIN      /* Never chain this operation.   */
    );
    if (rc != 0) {
        slapi_pblock_destroy(pb);
        return (-1);
    }

    str = slapi_entry2str(entry, &len);/* Entry as LDIF for the log.
                                        * Note we have to capture this
                                        * before the internal add, during
                                        * which the entry is consumed.  */
    rc = slapi_add_internal_pb(pb);    /* Entry consumed here           */
                                       /* ... get status ...            */
    slapi_pblock_get(pb, SLAPI_PLUGIN_INTOP_RESULT, &rc);
    if (rc != LDAP_SUCCESS) {
        slapi_pblock_destroy(pb);
        return (-1);
    }

    slapi_pblock_destroy(pb);

    slapi_log_info_ex(
        SLAPI_LOG_INFO_AREA_PLUGIN,
        SLAPI_LOG_INFO_LEVEL_DEFAULT,
        SLAPI_LOG_NO_MSGID,
        SLAPI_LOG_NO_CONNID,
        SLAPI_LOG_NO_OPID,
        "test_internal in test-internal plug-in",
        "\nAdded entry:\n%sEntry length: %d\n", str, len
    );

    return (0);
}
```

Notice the internal operation requires a `plugin_id`. As shown, the plug-in ID is a global variable. It is set during plug-in initialization, using

```
slapi_pblock_get(pb, SLAPI_PLUGIN_IDENTITY, &plugin_id);
```

on the parameter block, `pb`, passed to the plug-in initialization function. Refer to the `internal_init()` function in `internal.c` for a sample implementation.

Internal Modify

For internal modify, you first set up an array of LDAPMod modifications containing information about the attribute types you want to modify, and containing the attribute values. Then, similar to internal add, you allocate space for a parameter block, set it up using slapi_modify_internal_set_pb(), invoke the modify operation using slapi_modify_internal_pb(), and finally free the memory used. Code Example 7-2 demonstrates internal modification of a user mail address.

Code Example 7-2 Internal Modify Operation (internal.c)

```
#include "slapi-plugin.h"

static Slapi_ComponentId * plugin_id     = NULL;

int
test_internal()
{
    Slapi_Entry    * entry;            /* Entry holder for internal ops */
    Slapi_PBlock   * pb;               /* PBlock for internal ops       */
    LDAPMod          mod_attr;         /* Attribute to modify           */
    LDAPMod        * mods[2];          /* Array of modifications        */
    char           * mail_vals[]  =    /* New mail address              */
                       {"quentin@example.com", NULL};
    int              rc;               /* Return code; 0 means success. */

    /* Modify Quentin's entry after his email address changes from
     * qcubbins@example.com to quentin@example.com.                     */
    mod_attr.mod_type   = "mail";
    mod_attr.mod_op     = LDAP_MOD_REPLACE;
    mod_attr.mod_values = mail_vals;   /* mail: quentin@example.com     */
    mods[0]             = &mod_attr;
    mods[1]             = NULL;

    pb = slapi_pblock_new();           /* Set up a PBlock...            */
    rc = slapi_modify_internal_set_pb(
        pb,
        "uid=qcubbins,ou=people,dc=example,dc=com",
        mods,
        NULL,                          /* No controls                   */
        NULL,                          /* DN rather than unique ID      */
        plugin_id,
        SLAPI_OP_FLAG_NEVER_CHAIN      /* Never chain this operation.   */
    );
    if (rc != 0) {
        slapi_pblock_destroy(pb);
        return (-1);
    }

    rc = slapi_modify_internal_pb(pb); /* Unlike internal add,          */
                                       /* nothing consumed here         */
                                       /* ... get status ...            */
    slapi_pblock_get(pb, SLAPI_PLUGIN_INTOP_RESULT, &rc);
```

Code Example 7-2 Internal Modify Operation (`internal.c`) *(Continued)*

```
    if (rc != LDAP_SUCCESS) {
        slapi_pblock_destroy(pb);
        return (-1);
    }

    slapi_pblock_destroy(pb);          /* ... clean up the PBlock.      */

    slapi_log_info_ex(
        SLAPI_LOG_INFO_AREA_PLUGIN,
        SLAPI_LOG_INFO_LEVEL_DEFAULT,
        SLAPI_LOG_NO_MSGID,
        SLAPI_LOG_NO_CONNID,
        SLAPI_LOG_NO_OPID,
        "test_internal in test-internal plug-in",
        "\nModified attribute: %s\nNew value: %s\n",
        mod_attr.mod_type, mail_vals[0]
    );

    return (0);
}
```

Notice the data in `mod_attr` and `mail_vals` is still available for use after the modification. Unlike internal add, internal modify does not consume the data you set in the parameter block.

Internal Rename (Modify RDN)

For internal modify RDN, you allocate space for a parameter block, set it up using `slapi_rename_internal_set_pb()`, invoke the operation using `slapi_modrdn_internal_pb()` — notice that the first is `rename`, the second `modrdn` — and finally free the parameter block. Code Example 7-3 demonstrates an internal modify RDN.

Code Example 7-3 Internal Rename Operation (`internal.c`)

```
#include "slapi-plugin.h"

static Slapi_ComponentId * plugin_id     = NULL;

int
test_internal()
{
    Slapi_PBlock   * pb;               /* PBlock for internal ops       */
    int              rc;               /* Return code; 0 means success. */
```

Code Example 7-3 Internal Rename Operation (`internal.c`) *(Continued)*

```
    pb = slapi_pblock_new();           /* Set up a PBlock again...      */
    rc = slapi_rename_internal_set_pb(
        pb,
        "uid=qcubbins,ou=people,dc=example,dc=com",
        "uid=fcubbins",
        "ou=people,dc=example,dc=com",
        1,                             /* Delete old RDN                */
        NULL,                          /* No controls                   */
        NULL,                          /* DN rather than unique ID      */
        plugin_id,
        SLAPI_OP_FLAG_NEVER_CHAIN      /* Never chain this operation.   */
    );

    if (rc != LDAP_SUCCESS) {
        slapi_pblock_destroy(pb);
        return (-1);
    }

    rc = slapi_modrdn_internal_pb(pb); /* Like internal modify,         */
                                       /* nothing consumed here.        */
                                       /* ... get status ...            */
    slapi_pblock_get(pb, SLAPI_PLUGIN_INTOP_RESULT, &rc);
    if (rc != LDAP_SUCCESS) {
        slapi_pblock_destroy(pb);
        return (-1);
    }

    slapi_pblock_destroy(pb);          /* ... cleaning up.              */

    slapi_log_info_ex(
        SLAPI_LOG_INFO_AREA_PLUGIN,
        SLAPI_LOG_INFO_LEVEL_DEFAULT,
        SLAPI_LOG_NO_MSGID,
        SLAPI_LOG_NO_CONNID,
        SLAPI_LOG_NO_OPID,
        "test_internal in test-internal plug-in",
        "\nNew entry RDN: %s\n", "uid=fcubbins"
    );

    return (0);
}
```

Internal modify RDN does not consume the data you set in the parameter block.

Internal Search

For an internal search, use callbacks to retrieve what the server finds. The callbacks let you retrieve what would be sent back to a client application were the operation initiated by an external request: LDAP result codes, entries found, and referrals.

You set up the callback data you want to retrieve, and write the function called back by the server. Code Example 7-4 shows how the example plug-in `internal.c` uses a callback to retrieve an LDIF representation of the first entry found during an internal search using `slapi_entry2str()` in the callback function.

Code Example 7-4 Search Callback (`internal.c`)

```
#include "slapi-plugin.h"

struct cb_data {                        /* Data returned from search     */
    char * e_str;                       /* Entry as LDIF                 */
    int    e_len;                       /* Length of LDIF                */
};

int
test_internal_entry_callback(Slapi_Entry * entry, void * callback_data)
{
    struct cb_data * data;
    int              len;

    /* This callback could do something more interesting with the
     * data such as build an array of entries returned by the search.
     * Here we simply log the result.                                    */
    data        = (struct cb_data *)callback_data;
    data->e_str = slapi_entry2str(entry, &len);
    data->e_len = len;
    slapi_log_info_ex(
        SLAPI_LOG_INFO_AREA_PLUGIN,
        SLAPI_LOG_INFO_LEVEL_DEFAULT,
        SLAPI_LOG_NO_MSGID,
        SLAPI_LOG_NO_CONNID,
        SLAPI_LOG_NO_OPID,
        "test_internal_entry_callback in test-internal plug-in",
        "\nFound entry: %sLength: %d\n", data->e_str, data->e_len
    );
    return (-1);                        /* Stop at the first entry.      */
}                                       /* To continue, return 0.        */
```

The callback here stops the search at the first entry. Your plug-in may have to deal with more than one entry being returned by the search, so consider how you want to allocate space for your data depending on what your plug-in does.

With the callback data and function implemented, you are ready to process the internal search. First, allocate space for the parameter block and your callback data, and set up the parameter block using `slapi_search_internal_pb_set()`. Next, invoke the search using `slapi_search_internal_pb()`, and also set up the callback using `slapi_search_internal_callback_pb()`. Finally, free the space allocated when you are finished.

Code Example 7-5 demonstrates an internal search.

Code Example 7-5 Internal Search Operation (`internal.c`)

```
#include "slapi-plugin.h"

static Slapi_ComponentId * plugin_id     = NULL;

int
test_internal()
{
    Slapi_PBlock   * pb;               /* PBlock for internal ops       */
    char           * srch_attrs[] =    /* Attr. to get during search    */
                       {LDAP_ALL_USER_ATTRS, NULL};
    struct cb_data   callback_data;    /* Data returned from search     */
    int              rc;               /* Return code; 0 means success. */

    slapi_log_info_ex(
        SLAPI_LOG_INFO_AREA_PLUGIN,
        SLAPI_LOG_INFO_LEVEL_DEFAULT,
        SLAPI_LOG_NO_MSGID,
        SLAPI_LOG_NO_CONNID,
        SLAPI_LOG_NO_OPID,
        "test_internal in test-internal plug-in",
        "\nSearching with base DN %s, filter %s...\n",
        "dc=example,dc=com", "(uid=fcubbins)"
    );

    pb = slapi_pblock_new();           /* Set up a PBlock...            */
    rc = slapi_search_internal_set_pb(
        pb,
        "dc=example,dc=com",           /* Base DN for search            */
        LDAP_SCOPE_SUBTREE,            /* Scope                         */
        "(uid=fcubbins)",              /* Filter                        */
        srch_attrs,                    /* Set to get all user attrs.    */
        0,                             /* Return attrs. and values      */
        NULL,                          /* No controls                   */
        NULL,                          /* DN rather than unique ID      */
        plugin_id,
        SLAPI_OP_FLAG_NEVER_CHAIN      /* Never chain this operation.   */
    );

    if (rc != LDAP_SUCCESS) {
        slapi_pblock_destroy(pb);
        return (-1);
    }
```

Code Example 7-5 Internal Search Operation (`internal.c`) *(Continued)*

```
    /* Internal search puts results into the PBlock, but we use callbacks
     * to get at the data as it is turned up by the search. In this case,
     * what we want to get is the entry found by the search.           */
    rc  = slapi_search_internal_pb(pb);
    rc |= slapi_search_internal_callback_pb(
        pb,
        &callback_data,
        NULL,                           /* No result callback          */
        test_internal_entry_callback,
        NULL                            /* No referral callback        */
    );

                                        /* ... get status ...          */
    slapi_pblock_get(pb, SLAPI_PLUGIN_INTOP_RESULT, &rc);
    if (rc != LDAP_SUCCESS) {
        slapi_pblock_destroy(pb);
        return -1;
    }
                                        /* Free the search results when
                                         * finished with them.         */
    slapi_free_search_results_internal(pb);
    slapi_pblock_destroy(pb);           /* ... done cleaning up.       */

    return (0);
}
```

Here we allocate and free `callback_data` locally. You may want to manage memory differently if you pass the data to another plug-in function.

Internal Delete

For internal delete, you allocate space for the parameter block, set it up using `slapi_delete_internal_set_pb()`, invoke the delete using `slapi_delete_internal_pb()`, and finally free the parameter block. Code Example 7-6 demonstrates an internal delete.

Code Example 7-6 Internal Delete Operation (`internal.c`)

```
#include "slapi-plugin.h"

static Slapi_ComponentId * plugin_id     = NULL;

int
test_internal()
{
```

Code Example 7-6 Internal Delete Operation (`internal.c`) *(Continued)*

```
    Slapi_PBlock   * pb;              /* PBlock for internal ops       */
    int              rc;              /* Return code; 0 means success. */

    pb = slapi_pblock_new();          /* Set up a PBlock...            */
    rc = slapi_delete_internal_set_pb(
        pb,
        "uid=fcubbins,ou=people,dc=example,dc=com",
        NULL,                         /* No controls                   */
        NULL,                         /* DN rather than unique ID      */
        plugin_id,
        SLAPI_OP_FLAG_NEVER_CHAIN     /* Never chain this operation.   */
    );

    if (rc != LDAP_SUCCESS) {
        slapi_pblock_destroy(pb);
        return -1;
    }

    rc = slapi_delete_internal_pb(pb); /* Does not consume any         */
                                      /* data set in the PBlock.       */
                                      /* ... get status ...            */
    slapi_pblock_get(pb, SLAPI_PLUGIN_INTOP_RESULT, &rc);
    if (rc != LDAP_SUCCESS) {
        slapi_pblock_destroy(pb);
        return -1;
    }

    slapi_pblock_destroy(pb);         /* ... cleaning up.              */

    slapi_log_info_ex(
        SLAPI_LOG_INFO_AREA_PLUGIN,
        SLAPI_LOG_INFO_LEVEL_DEFAULT,
        SLAPI_LOG_NO_MSGID,
        SLAPI_LOG_NO_CONNID,
        SLAPI_LOG_NO_OPID,
        "test_internal in test-internal plug-in",
        "\nDeleted entry with DN: %s\n",
        "uid=fcubbins,ou=people,dc=example,dc=com"
    );

    return (0);
}
```

As mentioned in the excerpt, internal delete does not consume the data you set in the parameter block.

Chapter 8

Writing Entry Store and Entry Fetch Plug-Ins

This chapter covers how to write plug-ins that modify how Directory Server operates when writing entries to and reading entries from the directory database.

This chapter includes the following sections:

- Calling Entry Store and Entry Fetch Plug-Ins
- Writing a Plug-In to Encrypt Entries

Calling Entry Store and Entry Fetch Plug-Ins

This section describes when entry store and entry fetch plug-ins are called and how entries are expected to be handled by the plug-in.

The server calls entry store plug-in functions before writing data to a directory database. It calls entry fetch plug-in functions after reading data from the directory database. Entry store and entry fetch plug-ins may therefore typically be used to encrypt and decrypt directory entries, or to perform auditing work.

An LDIF String, Not a Parameter Block

Unlike other types of plug-ins, entry store and entry fetch plug-in functions do not take a parameter block as an argument. Instead, they take two parameters, the LDIF string representation of an entry and the length of the string. Directory Server uses the modified LDIF string after the plug-in returns successfully. An example prototype for an entry store plug-in function:

```
int my_entrystore_fn(char ** entry, unsigned int * length);
```

Plug-in functions may manipulate the string as necessary. Entry store and entry fetch plug-ins return `0` on success. When the function returns, Directory Server expects `entry` and `length` to contain the modified versions of the parameters.

On Windows Platforms

Export the plug-in function itself with `__declspec(dllexport)` and include it in the appropriate `.def` file.

Writing a Plug-In to Encrypt Entries

This section demonstrates how to write a plug-in that scrambles directory entries written to the directory database and de-scrambles directory entries read from the directory database.

CAUTION The example does *not* constitute a secure entry storage scheme.

Code Example 8-1 demonstrates how entry store and entry fetch plug-ins are registered with Directory Server.

Code Example 8-1 Registering Entry Store and Entry Fetch Plug-Ins (`testentry.c`)

```
#include "slapi-plugin.h"

Slapi_PluginDesc entrypdesc = {
    "test-entry",                       /* plug-in identifier      */
    "Sun Microsystems, Inc.",           /* vendor name             */
    "5.2",                              /* plug-in revision number */
    "Sample entry store/fetch plug-in" /* plug-in description     */
};

int
testentry_init(Slapi_PBlock *pb)
{
    int rc = 0;                         /* 0 means success         */
    rc |= slapi_pblock_set(             /* Plug-in API version     */
        pb,
        SLAPI_PLUGIN_VERSION,
        SLAPI_PLUGIN_CURRENT_VERSION
    );
    rc |= slapi_pblock_set(             /* Plug-in description     */
        pb,
        SLAPI_PLUGIN_DESCRIPTION,
        (void *) &entrypdesc
```

Code Example 8-1 Registering Entry Store and Entry Fetch Plug-Ins (`testentry.c`) *(Continued)*

```
    );
    rc |= slapi_pblock_set(            /* Entry store function    */
        pb,
        SLAPI_PLUGIN_ENTRY_STORE_FUNC,
        (void *) testentry_scramble
    );
    rc |= slapi_pblock_set(            /* Entry fetch function    */
        pb,
        SLAPI_PLUGIN_ENTRY_FETCH_FUNC,
        (void *) testentry_unscramble
    );
    return rc;
}
```

Finding the Examples

This chapter refers to the plug-in function examples in *ServerRoot*`/plugins/slapd/slapi/examples/testentry.c`.

Code Example 8-2 shows the entry store scrambling function used in this chapter, and called by Directory Server before writing an entry to the database.

Code Example 8-2 Entry Fetch Scrambler (`testentry.c`)

```
#include "slapi-plugin.h"

#ifdef _WIN32
typedef unsigned int uint;
#endif

#ifdef _WIN32
__declspec(dllexport)
#endif
int
testentry_scramble(char ** entry, uint * len)
{
    uint i;

    /* Scramble using bitwise exclusive-or on each character.     */
    for (i = 0; i < *len - 1; i++) { (*entry)[i] ^= 0xaa; }

    slapi_log_info_ex(
        SLAPI_LOG_INFO_AREA_PLUGIN,
        SLAPI_LOG_INFO_LEVEL_DEFAULT,
        SLAPI_LOG_NO_MSGID,
        SLAPI_LOG_NO_CONNID,
        SLAPI_LOG_NO_OPID,
        "testentry_scramble in test-entry plug-in",
```

Code Example 8-2 Entry Fetch Scrambler (`testentry.c`) *(Continued)*

```
        "Entry data scrambled.\n"
    );

    return 0;
}
```

Code Example 8-3 shows the entry fetch de-scrambling function used in this chapter, and called by the server after reading an entry from the database.

Code Example 8-3 Entry Fetch De-Scrambler (`testentry.c`)

```
#include "slapi-plugin.h"

#ifdef _WIN32
typedef unsigned int uint;
#endif

#ifdef _WIN32
__declspec(dllexport)
#endif
int
testentry_unscramble(char ** entry, uint * len)
{
    uint i;

    /* Return now if the entry is not scrambled.                  */
    if (!strncmp(*entry, "dn:", 3)) { return 0; }

    /* Unscramble using bitwise exclusive-or on each character.   */
    for (i = 0; i < *len - 1; i++) { (*entry)[i] ^= 0xaa; }

    slapi_log_info_ex(
        SLAPI_LOG_INFO_AREA_PLUGIN,
        SLAPI_LOG_INFO_LEVEL_DEFAULT,
        SLAPI_LOG_NO_MSGID,
        SLAPI_LOG_NO_CONNID,
        SLAPI_LOG_NO_OPID,
        "testentry_unscramble in test-entry plug-in",
        "Entry data unscrambled.\n"
    );

    return 0;
}
```

Notice the symmetry between the two functions. The scrambling mask, `0xaa` or `10101010` in binary, makes the transformation simple to understand but not secure. A secure encryption mechanism may be significantly more complicated.

Trying It Out

We demonstrate the plug-in by showing the difference between scrambled and unscrambled data in the database itself. We therefore do not enable the plug-in immediately, but instead add an entry to a new directory suffix before scrambling and observe the results in the database on disk. Then we remove the entry, enable the scrambling plug-in, add the same entry again, and finally observe the results after the entry has been scrambled.

Setting Up an Example Suffix

If you have not done so already, prepare some data by creating a directory suffix, `dc=example,dc=com`, whose users we load from an LDIF file, *`ServerRoot`*`/slapd-`*`serverID`*`/ldif/Example-Plugin.ldif`. Refer to “Setting Up an Example Suffix,” on page 86 for instructions.

Looking for Strings in the Database Before Scrambling

Here, we add an entry for Quentin Cubbins to our example suffix before registering the entry store-fetch plug-in with Directory Server. We see that Quentin’s mail address is visible in the database holding mail address attribute values. Quentin’s entry, `quentin.ldif`, appears as shown in Code Example 8-4.

Code Example 8-4 LDIF Representation of Quentin’s Entry

```
dn: uid=qcubbins,ou=People,dc=example,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
uid: qcubbins
givenName: Quentin
sn: Cubbins
cn: Quentin Cubbins
mail: qcubbins@example.com
userPassword: qcubbins
secretary: uid=bcubbins,ou=People,dc=example,dc=com
```

Add Quentin’s entry to the directory. For example, if the entry is in `quentin.ldif`:

```
$ ldapmodify -a -p port -D "cn=directory manager" -w password -f quentin.ldif
```

Now look for strings in the directory database file for the mail attribute values.

Code Example 8-5 Attribute Values in a Database File Before Scrambling

```
$ cd ServerRoot/slapd-serverID/db/example/
$ strings example_mail.db3 | grep example.com
=qcubbins@example.com
=agodiva@example.com
=hfuddnud@example.com
=pblinn@example.com
=scooper@example.com
=bcubbins@example.com
=yyorgens@example.com
```

Notice that Quentin's mail address is clearly visible to a user who manages to gain access to the database files. If this were not a mail address, but instead a credit card number, we could have a possible security issue.

Looking for Strings in the Database After Scrambling

Here, we add an entry for Quentin Cubbins to our example suffix after registering the entry store-fetch plug-in with Directory Server. We see that Quentin's mail address is no longer visible in the database holding mail address attribute values.

Before loading the plug-in, delete Quentin's entry. For example:

```
$ ldapdelete -p port -D "cn=directory manager" -w password \
"uid=qcubbins,ou=People,dc=example,dc=com"
```

Next, load the plug-in configuration entry into the directory, and then restart the server. To load the configuration entry, first save it to a file, `testentry.ldif`. Code Example 8-6 shows the configuration entry.

Code Example 8-6 Configuration Entry (`testentry.c`)

```
dn: cn=Test entry,cn=plugins,cn=config
objectClass: top
objectClass: nsSlapdPlugin
objectClass: extensibleObject
cn: Test entry
nsslapd-pluginPath: ServerRoot/plugins/slapd/slapi/examples/LibName
nsslapd-pluginInitfunc: testentry_init
nsslapd-pluginType: ldbmentryfetchstore
nsslapd-pluginEnabled: on
nsslapd-plugin-depends-on-type: database
```

Code Example 8-6 Configuration Entry (`testentry.c`) *(Continued)*

```
nsslapd-pluginId: test-entry
nsslapd-pluginVersion: 5.2
nsslapd-pluginVendor: Sun Microsystems, Inc.
nsslapd-pluginDescription: Sample entry store/fetch plug-in
```

After editing *ServerRoot* to correspond to your system and saving `testentry.ldif`, modify the directory configuration to include the entry:

```
$ ldapmodify -a -p port -D "cn=directory manager" -w password \
-f testentry.ldif
```

Restart Directory Server for the modification to take effect. For example:

```
$ ServerRoot/slapd-serverID/start-slapd
```

With the entry store-fetch plug-in active, add Quentin's entry back into the directory:

```
$ ldapmodify -a -p port -D "cn=directory manager" -w password \
-f quentin.ldif
```

Now search again for strings in the directory database file for the mail attribute values.

Code Example 8-7 Attribute Values in a Database File After Scrambling

```
$ cd ServerRoot/slapd-serverID/db/example/
$ strings example_mail.db3 | grep example.com
=agodiva@example.com
=hfuddnud@example.com
=pblinn@example.com
=scooper@example.com
=bcubbins@example.com
=yyorgens@example.com
```

Notice that Quentin's mail address value is now not visible in the directory database. Directory users having appropriate access rights (anonymous in this simple example case) can still view the attribute during a search. The attribute and its value are *emphasized* in Code Example 8-8.

Code Example 8-8 De-Scrambled Search Results

```
$ ldapsearch -L -p port -b "dc=example,dc=com" "(uid=qcubbins)"
dn: uid=qcubbins,ou=People,dc=example,dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
uid: qcubbins
givenName: Quentin
sn: Cubbins
cn: Quentin Cubbins
mail: qcubbins@example.com
secretary: uid=bcubbins,ou=People,dc=example,dc=com
```

In this way, we see that entry fetch-store plug-ins affect only the way entries are stored, and not the directory frontend.

Writing Extended Operation Plug-Ins

This chapter covers how to write plug-ins to take advantage of LDAP version 3 extended operations, defined in RFC 2251 as allowing “additional operations to be defined for services not available elsewhere in this protocol, for instance digitally signed operations and results.”

This chapter includes the following sections:

- Calling Extended Operation Plug-Ins
- Implementing an Extended Operation Plug-In

Calling Extended Operation Plug-Ins

This section describes how extended operations are handled by Directory Server and what requirements are placed on extended operation plug-ins.

Directory Server identifies extended operations by object identifiers (OIDs). Clients request the operation by sending an extended operation request, specifying:

- The OID of the extended operation
- Data specific to the extended operation

Upon receiving an extended operation request, Directory Server calls the plug-in registered to handle the OID sent in the request. The plug-in function handling the request can obtain the OID and operation-specific data, do its processing, and then send a response to the client containing an OID and any additional data pertaining the extended operation.

When implementing extended operation support, you need to determine the OID to use. The Internet Assigned Numbers Authority can help with registration of official OIDs. If you select your own, know that the key is to avoid OIDs that conflict with other OIDs, such as those defined by the LDAP schema used by Directory Server.

Directory Server determines which extended operation plug-ins handle which extended operation OIDs at startup. At startup, Directory Server calls the initialization function for each plug-in. Extended operation plug-ins register the extended operation OIDs they handle as part of their initialization function, as described in “Initializing the Extended Operation Plug-In,” on page 158.

The configuration entry for a plug-in may include parameters that Directory Server passes to the plug-in initialization function at startup, as described in “Parameters Specified in the Configuration Entry,” on page 64. You can write the initialization function in such a way that if OIDs are determined after the plug-in functionality is written, they may be passed as configuration entry arguments, for example.

Implementing an Extended Operation Plug-In

This section demonstrates how to implement a basic extended operation plug-in and client application to trigger the extended operation.

The client sends an extended operation request to the server using the OID that identifies the extended operation, in this case `1.2.3.4`. The client also sends a value that holds a string. The example plug-in responds to an extended operation request by sending the client an OID and a modified version of the string the client sent with the request.

Finding the Examples

The rest of this chapter refers to the plug-in code in *ServerRoot*`/plugins/slapd/slapi/examples/testextendedop.c`, and client code *ServerRoot*`/plugins/slapd/slapi/examples/clients/reqextop.c`.

An Example Plug-In

This section explains how our extended operation plug-in works.

Registering the Plug-In

Before using the plug-in function as described here, build and then register the plug-in. Refer to “Building Directory Server Plug-Ins,” on page 59 for build hints. To register the plug-in, start with the configuration entry from the introductory comment of `testextendedop.c`. Code Example 9-1 shows the entry.

Code Example 9-1 Configuration Entry (`testextendedop.c`)

```
dn: cn=Test ExtendedOp,cn=plugins,cn=config
objectClass: top
objectClass: nsSlapdPlugin
objectClass: extensibleObject
cn: Test ExtendedOp
nsslapd-pluginPath: <ServerRoot>/plugins/slapd/slapi/examples/<LibName>
nsslapd-pluginInitfunc: testexop_init
nsslapd-pluginType: extendedop
nsslapd-pluginEnabled: on
nsslapd-plugin-depends-on-type: database
nsslapd-pluginId: test-extendedop
nsslapd-pluginVersion: 5.2
nsslapd-pluginVendor: Sun Microsystems, Inc.
nsslapd-pluginDescription: Sample extended operation plug-in
nsslapd-pluginarg0: 1.2.3.4
```

Notice the OID `1.2.3.4` is passed as an argument through the configuration entry. The configuration entry could specify more than one `nsslapd-pluginarg` attribute if the plug-in supported multiple extended operations, each identified by a distinct OID, for example.

To register the plug-in, copy the example configuration entry to a file named `testextendedop.ldif`, and then adjust `nsslapd-pluginPath` to fit how the product is installed on your system.

With the correct configuration entry in `testextendedop.ldif`, load the plug-in. For example:

```
$ ldapmodify -a -p port -D "cn=directory manager" -w password \
-f testextendedop.ldif
```

Turn on plug-in logging using the console if you want Directory Server to log informational messages for the plug-in.

With the plug-in configuration entry correctly loaded in the Directory Server configuration, restart Directory Server to activate the plug-in.

Initializing the Extended Operation Plug-In

As for other plug-in types, extended operation plug-ins include an initialization function that registers other functions in the plug-in with Directory Server. For extended operation plug-ins, this initialization function also registers the OIDs handled by the plug-in. It does this by setting `SLAPI_PLUGIN_EXT_OP_OIDLIST` in the parameter block Directory Server passes to the initialization function.

Code Example 9-2 demonstrates how the OID list is built and registered.

Code Example 9-2 Registering Plug-In Functions and OIDs (`testextendedop.c`)

```
#include "slapi-plugin.h"

Slapi_PluginDesc expdesc = {
    "test-extendedop",                 /* plug-in identifier       */
    "Sun Microsystems, Inc.",          /* vendor name              */
    "5.2",                             /* plug-in revision number  */
    "Sample extended operation plug-in"/* plug-in description      */
};

#ifdef _WIN32
__declspec(dllexport)
#endif
int
testexop_init(Slapi_PBlock * pb)
{
    char ** argv;                      /* Args from configuration  */
    int     argc;                      /* entry for plug-in.       */
    char ** oid_list;                  /* OIDs supported           */
    int     rc = 0;                    /* 0 means success          */
    int     i;

    /* Get the arguments from the configuration entry.             */
    rc |= slapi_pblock_get(pb, SLAPI_PLUGIN_ARGV, &argv);
    rc |= slapi_pblock_get(pb, SLAPI_PLUGIN_ARGC, &argc);
    if (rc != 0) {
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
            SLAPI_LOG_INFO_LEVEL_DEFAULT,
            SLAPI_LOG_NO_MSGID,
            SLAPI_LOG_NO_CONNID,
            SLAPI_LOG_NO_OPID,
            "testexop_init in test-extendedop plug-in",
            "Could not get plug-in arguments.\n"
        );
        return (rc);
    }

    /* Extended operation plug-ins may handle a range of OIDs.     */
    oid_list = (char **)slapi_ch_malloc((argc + 1) * sizeof(char *));
    for (i = 0; i < argc; ++i) {
        oid_list[i] = slapi_ch_strdup(argv[i]);
        slapi_log_info_ex(
```

Code Example 9-2 Registering Plug-In Functions and OIDs (`testextendedop.c`) *(Continued)*

```
            SLAPI_LOG_INFO_AREA_PLUGIN,
            SLAPI_LOG_INFO_LEVEL_DEFAULT,
            SLAPI_LOG_NO_MSGID,
            SLAPI_LOG_NO_CONNID,
            SLAPI_LOG_NO_OPID,
            "testexop_init in test-extendedop plug-in",
            "Registering plug-in for extended operation %s.\n",
            oid_list[i]
        );
    }
    oid_list[argc] = NULL;

    rc |= slapi_pblock_set(            /* Plug-in API version      */
        pb,
        SLAPI_PLUGIN_VERSION,
        SLAPI_PLUGIN_CURRENT_VERSION
    );
    rc |= slapi_pblock_set(            /* Plug-in description      */
        pb,
        SLAPI_PLUGIN_DESCRIPTION,
        (void *) &expdesc
    );
    rc |= slapi_pblock_set(            /* Extended op. handler     */
        pb,
        SLAPI_PLUGIN_EXT_OP_FN,
        (void *) test_extendedop
    );
    rc |= slapi_pblock_set(            /* List of OIDs handled     */
        pb,
        SLAPI_PLUGIN_EXT_OP_OIDLIST,
        oid_list
    );
    return (rc);
}
```

Notice here we extract OIDs from the arguments passed by Directory Server from the configuration entry into the parameter block, using `slapi_ch_strdup()` on each `argv[]` element. The OID list is then built by allocating space for the array using `slapi_ch_malloc()` and placing the OIDs in each `oid_list[]` element. We then register the plug-in OID list using `SLAPI_PLUGIN_EXT_OP_OIDLIST`. We register the extended operation handler function, `test_extendedop()`, using `SLAPI_PLUGIN_EXT_OP_FN` as shown.

Refer to the *Sun ONE Directory Server Plug-In API Reference* for more information about parameter block arguments and plug-in API functions.

Handling the Extended Operation

The plug-in function `test_extendedop()` gets the OID and value for the operation from the client request. It then sends the client a response, as shown in Code Example 9-3.

Code Example 9-3 Responding to an Extended Operation Request (`testextendedop.c`)

```
#include "slapi-plugin.h"

int
test_extendedop(Slapi_PBlock * pb)
{
    char          * oid;               /* Client request OID       */
    struct berval * client_bval;       /* Value from client        */
    char          * result;            /* Result to send to client */
    char          * tmp_msg;
    struct berval * result_bval;       /* Encoded result           */
    int             connId, opId, rc = 0;
    long            msgId;

    /* Identify the request for logging.                           */
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_MSGID, &msgId);
    rc |= slapi_pblock_get(pb, SLAPI_CONN_ID,         &connId);
    rc |= slapi_pblock_get(pb, SLAPI_OPERATION_ID,    &opId);
    if (rc != 0) {
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
            SLAPI_LOG_INFO_LEVEL_DEFAULT,
            SLAPI_LOG_NO_MSGID,
            SLAPI_LOG_NO_CONNID,
            SLAPI_LOG_NO_OPID,
            "test_extendedop in test-extendedop plug-in",
            "Could not identify message, connection or operation.\n"
        );
    }

    /* Get the OID and the value included in the request.          */
    rc |= slapi_pblock_get(pb, SLAPI_EXT_OP_REQ_OID,   &oid );
    rc |= slapi_pblock_get(pb, SLAPI_EXT_OP_REQ_VALUE, &client_bval);
    if (rc != 0) {
        tmp_msg = "Could not get OID and value from request";
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
            SLAPI_LOG_INFO_LEVEL_DEFAULT,
            msgId,
            connId,
            opId,
            "test_extendedop in test-extendedop plug-in",
            "%s\n", tmp_msg
        );
        slapi_send_ldap_result(
            pb,                        /* PBlock for request       */
            LDAP_OPERATIONS_ERROR,     /* LDAP result code         */
            NULL,                      /* For LDAP_NO_SUCH_OBJECT  */
```

Code Example 9-3 Responding to an Extended Operation Request (`testextendedop.c`) *(Continued)*

```
            tmp_msg,                    /* Text message for client  */
            0,                          /* Number of entries sent   */
            NULL                        /* URL for referral         */
        );
        return (SLAPI_PLUGIN_EXTENDED_SENT_RESULT);
    } else {
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
            SLAPI_LOG_INFO_LEVEL_DEFAULT,
            msgId,
            connId,
            opId,
            "test_extendedop in test-extendedop plug-in",
            "Request with OID: %s  Value from client: %s\n",
            oid, client_bval->bv_val
        );
    }

    /*
     * Set the value to return to the client, depending on what your
     * plug-in function does. Here, we return the value sent by the
     * client, prefixed with the string "Value from client: ".     */
    tmp_msg             = "Value from client: ";
    result              = (char *)slapi_ch_malloc(
        client_bval->bv_len + strlen(tmp_msg) + 1);
    sprintf(result, "%s%s", tmp_msg, client_bval->bv_val);
    result_bval         = (struct berval *)slapi_ch_malloc(
        sizeof(struct berval));
    result_bval->bv_val = result;
    result_bval->bv_len = strlen(result_bval->bv_val);

    /*
     * Prepare the PBlock to return and OID and value to the client.
     * Here, we demonstrate that the plug-in may return a different
     * OID than the one sent by the client. You may, for example,
     * use the different OID to indicate something to the client.  */
    rc |= slapi_pblock_set(pb, SLAPI_EXT_OP_RET_OID,   "5.6.7.8");
    rc |= slapi_pblock_set(pb, SLAPI_EXT_OP_RET_VALUE, result_bval);
    if (rc != 0) {
        slapi_ch_free((void **)&result);
        tmp_msg = "Could not set results to return to client.";
        slapi_log_info_ex(
            SLAPI_LOG_INFO_AREA_PLUGIN,
            SLAPI_LOG_INFO_LEVEL_DEFAULT,
            msgId,
            connId,
            opId,
            "test_extendedop in test-extendedop plug-in",
            "%s\n", tmp_msg
        );
        slapi_send_ldap_result(
            pb,
            LDAP_OPERATIONS_ERROR,
            NULL,
            tmp_msg,
```

Code Example 9-3 Responding to an Extended Operation Request (`testextendedop.c`) *(Continued)*

```
            0,
            NULL
        );
        return (SLAPI_PLUGIN_EXTENDED_SENT_RESULT);
    }

    /* Send the result to the client.                                */
    slapi_send_ldap_result(
        pb,
        LDAP_SUCCESS,
        NULL,
        "Extended operation successful!",
        0,
        NULL
    );
    slapi_log_info_ex(
        SLAPI_LOG_INFO_AREA_PLUGIN,
        SLAPI_LOG_INFO_LEVEL_DEFAULT,
        msgId,
        connId,
        opId,
        "test_extendedop in test-extendedop plug-in",
        "OID sent to client: %s  Value sent to client: %s\n",
        "5.6.7.8", result
    );
    slapi_ch_free((void **)&result);

    /* Tell the server we sent the result.                           */
    return (SLAPI_PLUGIN_EXTENDED_SENT_RESULT);
}
```

Notice how the function obtains the OID and value from the request using `SLAPI_EXT_OP_REQ_OID` and `SLAPI_EXT_OP_REQ_VALUE`. It then uses `slapi_ch_malloc()` to construct a string to return to the client through the pointer to a `berval` structure, `result_bval`. A different extended operation plug-in might do something entirely different at this point.

Also notice the function sends a different OID back to the client than the OID in the client request. The OID sent back can be used to indicate a particular result to the client, for example. The function uses `slapi_send_ldap_result()` to indicate success and send the OID and value to the client, frees the memory allocated, then returns `SLAPI_PLUGIN_EXTENDED_SENT_RESULT` to indicate to Directory Server that processing of the plug-in function is complete.

If the function had not sent a result code to the client, it would return an LDAP result code to Directory Server, which Directory Server would then send to the client.

If the function cannot handle the extended operation with the specified OID, it returns SLAPI_PLUGIN_EXTENDED_NOT_HANDLED, after which Directory Server sends an LDAP_PROTOCOL_ERROR result code to the client.

Developing the Client

To test our plug-in, we need a client that requests an extended operation with OID 1.2.3.4. The example discussed here, reqextop.c, is delivered with the product. The *Sun ONE LDAP C SDK*, available separately as described in "Downloading Directory Server Tools," on page 16, is used to build the client.

The client sets up a short string to send to Directory Server in the extended operation request. It then gets an LDAP connection that supports LDAP version 3, and binds to Directory Server. The client then sends an extended operation request and displays the result on STDOUT. Code Example 9-4 shows how the extended operation request is handled on the client side.

Code Example 9-4 Client Requesting an Extended Operation (clients/reqextop.c)

```
#include <stdlib.h>
#include "ldap.h"

/* Global variables for client connection information.
 * You may set these here or on the command line.                    */
static char * host     = "localhost";  /* Server hostname            */
static int    port     = 389;          /* Server port                */
static char * bind_DN  = "cn=Directory Manager"; /* DN to bind as    */
static char * bind_pwd = "23skidoo";   /* Password for bind DN       */

/* Check for connection info as command line arguments.              */
int get_user_args(int argc, char ** argv);

int
main(int argc, char ** argv)
{
    /* OID of the extended operation that you are requesting         */
    const char    * oidrequest = "1.2.3.4"; /* Ext op OID            */
    char          * oidresult;        /* OID in reply from server   */
    struct berval   valrequest;       /* Request sent               */
    struct berval * valresult;        /* Reply received             */
    LDAP          * ld;               /* Handle to connection       */
    int             version;          /* LDAP version               */

    /* Use default connection arguments unless all four are
     * provided as arguments on the command line.                    */
    if (get_user_args(argc, argv) != 0) return 1; /* Usage error     */

    /* Set up the value that you want to pass to the server          */
    printf("Setting up value to pass to server...\n");
    valrequest.bv_val = "My Value";
```

Code Example 9-4 Client Requesting an Extended Operation (`clients/reqextop.c`) *(Continued)*

```
    valrequest.bv_len = strlen("My Value");

    /* Get a handle to an LDAP connection                          */
    printf("Getting the handle to the LDAP connection...\n");
    if ((ld = ldap_init(host, port)) == NULL) {
        perror("ldap_init");
        ldap_unbind(ld);
        return 1;
    }

    /* Set the LDAP protocol version supported by the client
       to 3. (By default, this is set to 2. Extended operations
       are part of version 3 of the LDAP protocol.)                */
    printf("Resetting version %d to 3.0...\n", version);
    version = LDAP_VERSION3;
    ldap_set_option(ld, LDAP_OPT_PROTOCOL_VERSION, &version);

    /* Authenticate to the directory as the Directory Manager      */
    printf("Binding to the directory...\n");
    if (ldap_simple_bind_s(ld, bind_DN, bind_pwd) != LDAP_SUCCESS) {
        ldap_perror(ld, "ldap_simple_bind_s");
        ldap_unbind(ld);
        return 1;
    }

    /* Initiate the extended operation                             */
    printf( "Initiating the extended operation...\n" );
    if (ldap_extended_operation_s(
            ld,
            oidrequest,
            &valrequest,
            NULL,
            NULL,
            &oidresult,
            &valresult
        ) != LDAP_SUCCESS) {
        ldap_perror(ld, "ldap_extended_operation_s failed: ");
        ldap_unbind(ld);
        return 1;
    }

    /* Get OID and value from result returned by server.           */
    printf("Operation successful.\n");
    printf("\tReturned OID: %s\n", oidresult);
    printf("\tReturned value: %s\n", valresult->bv_val);

    /* Disconnect from the server.                                 */
    ldap_unbind(ld);
    return 0;
}
```

Notice the client identifies the request by OID, and that it resets the protocol version to `LDAP_VERSION3` to ensure extended operation support in the protocol. The value the client sends with the request, `valrequest`, points to a `berval` structure. Also notice that the calls used here for the bind and extended operation are synchronous. Asynchronous versions are also available.

Trying It Out

With the plug-in active in the server, compile the client `reqextop.c`. One way of doing this with the *Sun ONE LDAP C SDK* involves adding a `reqextop` target to `lib/ldapcsdk/examples/Makefile`. For example:

Code Example 9-5 `Makefile` Additions to Build the Extended Operation Client

```
reqextop: reqextop.o
    $(CC) -o reqextop reqextop.o $(LIBS)
```

Next, copy `reqextop.c` to the `lib/ldapcsdk/examples/` directory and build the client. The `Makefile` works with GNU Make. For example:

Code Example 9-6 Building the Extended Operation Client

```
$ cd sdk_install_dir/lib/ldapcsdk/examples
$ cp ServerRoot/plugins/slapd/slapi/examples/clients/reqextop.c \
sdk_install_dir/lib/ldapcsdk/examples
$ gmake reqextop
```

Next, run the client to send the extended operation request and display the result.

Code Example 9-7 Client Side Extended Operation Results

```
$ ./reqextop -p port -w password
Using the following connection info:
        host:    localhost
        port:    port
        bind DN: cn=Directory Manager
        pwd:     password
Setting up value to pass to server...
Getting the handle to the LDAP connection...
```

Code Example 9-7 Client Side Extended Operation Results *(Continued)*

```
Resetting version 2 to 3.0...
Binding to the directory...
Initiating the extended operation...
Operation successful.
        Returned OID: 5.6.7.8
        Returned value: Value from client: My Value
```

On the Directory Server side, if you have turned on logging, messages similar to the following in the `errors` log show that the plug-in handled the client request:

```
[22/May/2002:08:54:15 +0200] - INFORMATION - test_extendedop in
test-extendedop plug-in - conn=0 op=1 msgId=2 -  Request with OID:
1.2.3.4  Value from client: My Value
[22/May/2002:08:54:15 +0200] - INFORMATION - test_extendedop in
test-extendedop plug-in - conn=0 op=1 msgId=2 -  OID sent to client:
5.6.7.8  Value sent to client: Value from client: My Value
```

We have thus demonstrated that our example extended operation plug-in handles requests for the extended operation with OID `1.2.3.4`.

Chapter 10

Writing Matching Rule Plug-Ins

This chapter covers plug-ins that enable Directory Server to handle custom matching rules. The server requires such matching rule plug-ins to support LDAP v3 extensible matching, such as “sounds like” searches.

This chapter includes the following sections:

- How Matching Rule Plug-Ins Work
- An Example Matching Rule Plug-in
- Handling Extensible Match Filters
- Indexing Entries According to a Matching Rule
- Enabling Sorting According to a Matching Rule
- Handling an Unknown Matching Rule

Code excerpts shown in this chapter demonstrate a conceptually simple case exact matching scheme: Case exact matching is a byte comparison between `DirectoryString` attribute values. Despite the straightforward concept, the plug-in code runs to many lines, as it must provide several functions and wrapper functions to enable indexing, searching and sorting.

Ensure you understand how Directory Server uses matching rule plug-ins before trying to implement your own plug-in. Read this chapter, then read the sample plug-in code. Avoid creating a new matching rule plug-in from scratch.

How Matching Rule Plug-Ins Work

This section summarizes what matching rules plug-ins are and how Directory Server handles them.

What a Matching Rule Is

A *matching rule* defines a specific way to compare attribute values having a given syntax. In other words, a matching rule defines the how potentially matching attributes are compared.

Each matching rule is identified by a unique object identifier (OID) string. A client application requesting a search may specify the matching rule OID in the search filter, indicating to Directory Server how to check for a match of two attribute values.

In practice, a client application may want only entries with attribute values that match the value provided exactly. The sample plug-in demonstrates how you might implement a solution for that case. Another client may want to sort entries using the rules for a given locale. Directory Server actually uses a matching rule plug-in to handle locale specific matching.

The *Sun ONE Directory Server Reference Manual* includes a list of matching rules Directory Server supports for internationalized searches. You can also view the list by searching the default schema. For example:

```
$ ldapsearch -p port -D "cn=directory manager" -w password \
-b cn=schema cn=schema matchingRules
```

Requesting a Matching Rule

To request a custom matching rule on a specific attribute, a client application includes the matching rule OID in the search filter. LDAP v3 calls these search filters *extensible match filters*. An extensible match filter looks something like the following:

```
(cn:2.5.13.5:=Quentin)
```

This filter tells the server to search for Quentin in the CN of the entries, using matching rule `2.5.13.5`, which happens to be case exact match. Refer to *Sun ONE Directory Server Getting Started Guide* for further details on searching with extensible match filters.

The case exact matching rule plug-in supporting OID `2.5.13.5` enables Directory Server to perform the search correctly. Directory Server calls code in this matching rule plug-in not only to check for matches during a search, but also to generate indexes to accelerate case exact searches and to sort entries found during such searches.

What a Matching Rule Plug-In Does

A matching rule plug-in may provide code to:

- Check for matches during an extensible match search
- Sort results for extensible match search using a sort control
- Maintain an index to speed an extensible match search (optional)

To enable these capabilities, the plug-in implements matching and indexing routines Directory Server calls to handle requests involving the particular matching rule.

The plug-in also implements factory functions that specify which routine to call when handling a particular matching rule. As a result, a plug-in may support multiple matching rules. Yet, plug-ins implementing only one matching rule also require factory function code to wrap indexing and matching routines. A plug-in therefore requires many lines of code and several functions to handle even a minimal matching rule.

Table 10-1 lists all the functions that matching rule plug-in may implement.

Table 10-1 Functions Defined in Matching Rule Plug-Ins

Type	Parameter Block Identifier	Required?
Filter factory	`SLAPI_PLUGIN_MR_FILTER_CREATE_FN`	Yes
Filter index (used to check an index for matches)	`SLAPI_PLUGIN_MR_FILTER_INDEX_FN`	No
Filter match	`SLAPI_PLUGIN_MR_FILTER_MATCH_FN`	Yes
Filter match reset	`SLAPI_PLUGIN_MR_FILTER_RESET_FN`	If filter must be reset for reuse
Filter object destructor	`SLAPI_PLUGIN_DESTROY_FN`	If needed to free memory
Indexer	`SLAPI_PLUGIN_MR_INDEX_FN`	No
Indexer factory	`SLAPI_PLUGIN_MR_INDEXER_CREATE_FN`	No
Indexer object destructor	`SLAPI_PLUGIN_DESTROY_FN`	If needed to free memory
Plug-in initialization function	not applicable (specified in configuration entry)	Yes
Server shutdown (cleanup) function	`SLAPI_CLOSE_FN`	No
Server startup function	`SLAPI_START_FN`	No

Refer to the *Sun ONE Directory Server Plug-In API Reference* for details on all parameter block identifiers available for use with matching rule plug-ins.

An Example Matching Rule Plug-in

The example code cited in this chapter, from *ServerRoot*/examples/slapd/slapi/examples/matchingrule.c, mimics functionality installed by default with Directory Server. For this reason, you do not need to build and load the plug-in unless you modify it to implement your own matching rule. The example uses a different OID from the plug-in provided with Directory Server, so the sample plug-in does not interfere with the existing plug-in if you do choose to load it.

Matching Rule Plug-In Configuration Entry

Matching rule plug-ins have type matchingrule, as shown in Code Example 10-1.

Code Example 10-1 Configuration Entry Template (matchingrule.c)

```
dn: cn=Common Name,cn=plugins,cn=config
objectClass: top
objectClass: nsSlapdPlugin
objectClass: extensibleObject
cn: Common Name
nsslapd-pluginPath: full path to plug-in library
nsslapd-pluginInitfunc: initialization function
nsslapd-pluginType: matchingRule
nsslapd-pluginId: plug-in identifier
nsslapd-pluginEnabled: on
nsslapd-pluginVersion: plug-in revision number
nsslapd-pluginVendor: vendor name
nsslapd-pluginDescription: plug-in description
```

Directory Server plug-ins may depend on matching rule plug-ins by type. Directory Server cannot load plug-ins unless plug-ins of the type they depend on load without error.

Refer to “Plugging Libraries into Directory Server,” on page 61 for details on loading plug-in configuration entries into Directory Server.

Registering Matching Rule Plug-Ins

Recall that matching rule plug-ins include factory functions. Factory functions themselves provide function pointers to the indexing or filter match routines dynamically. You therefore register only the factory functions as part of the plug-in initialization function.

A plug-in handling both indexing and matching registers both factory functions and a description, as shown in Code Example 10-2.

Code Example 10-2 Registering Matching Rule Factory Functions (`matchingrule.c`)

```
#include "slapi-plugin.h"

static Slapi_PluginDesc plg_desc = {
    "caseExactMatchingRule",           /* Plug-in identifier           */
    "Sun Microsystems, Inc.",          /* Vendor name                  */
    "5.2",                             /* Plug-in revision number      */
    "Case Exact Matching Rule plug-in" /* Plug-in description          */
};

#ifdef _WIN32
__declspec(dllexport)
#endif
int
plg_init(Slapi_PBlock * pb)
{
    int                       rc;

    /* Matching rule factory functions are registered using
     * the parameter block structure. Other functions are then
     * registered dynamically by the factory functions.
     *
     * This means that a single matching rule plug-in may handle
     * a number of different matching rules.                          */
    rc  = slapi_pblock_set(
        pb,
        SLAPI_PLUGIN_MR_INDEXER_CREATE_FN,
        (void *)plg_indexer_create
    );
    rc |= slapi_pblock_set(
        pb,
        SLAPI_PLUGIN_MR_FILTER_CREATE_FN,
        (void *)plg_filter_create
    );
    rc |= slapi_pblock_set(
        pb,
        SLAPI_PLUGIN_DESCRIPTION,
        (void *)&plg_desc
    );

    /* Register the matching rules themselves as in Code Example 10-3. */
```

Code Example 10-2 Registering Matching Rule Factory Functions (`matchingrule.c`) *(Continued)*

```
    return rc;
}
```

In Code Example 10-2, `plg_init()` is the plug-in initialization function, `plg_indexer_create()` is the indexer factory, `plg_filter_create()` is the filter factory and `plg_desc` is the plug-in description structure.

TIP If your plug-in uses private data, set `SLAPI_PLUGIN_PRIVATE` in the parameter block as a pointer to the private data structure.

Register matching rules themselves using `slapi_matchingrule_register()` as shown in Code Example 10-3, rather than using `slapi_pblock_set()` in the initialization function.

Code Example 10-3 Registering a Matching Rule (`matchingrule.c`)

```
#include "slapi-plugin.h"

#define PLG_DESC      "Case Exact Matching on Directory String" \
                      " [defined in X.520]"
#define PLG_NAME      "caseExactMatch"
/* This OID is for examples only and is not intended for reuse. The
 * official OID for this matching rule is 2.5.13.5.                    */
#define PLG_OID       "1.3.6.1.4.1.42.2.27.999.4.1"

#ifdef _WIN32
__declspec(dllexport)
#endif
int
plg_init(Slapi_PBlock * pb)
{
    Slapi_MatchingRuleEntry * mrentry = slapi_matchingrule_new();
    int                       rc;

    /* Register matching rule factory functions as in Code Example 10-2.*/

    /* Matching rules themselves are registered using a
     * Slapi_MatchingRuleEntry structure, not the parameter
     * block structure used when registering plug-in functions.
     *
     * This plug-in registers only one matching rule. Yours
     * may register many matching rules.                                */
    rc |= slapi_matchingrule_set(
        mrentry,
        SLAPI_MATCHINGRULE_DESC,
```

Code Example 10-3 Registering a Matching Rule (`matchingrule.c`) *(Continued)*

```
        (void *)slapi_ch_strdup(PLG_DESC)
    );
    rc |= slapi_matchingrule_set(
        mrentry,
        SLAPI_MATCHINGRULE_NAME,
        (void *)slapi_ch_strdup(PLG_NAME)
    );
    rc |= slapi_matchingrule_set(
        mrentry,
        SLAPI_MATCHINGRULE_OID,
        (void *)slapi_ch_strdup(PLG_OID)
    );
    rc |= slapi_matchingrule_set(
        mrentry,
        SLAPI_MATCHINGRULE_SYNTAX,     /* Here we use DirectoryString. */
        (void *)slapi_ch_strdup("1.3.6.1.4.1.1466.115.121.1.15")
    );
    rc |= slapi_matchingrule_register(mrentry);
    slapi_matchingrule_free(&mrentry, 1);

    return rc;
}
```

In Code Example 10-3:

- `PLG_DESC` is a string describing the matching rule
- `PLG_NAME` is a string identifier for the matching rule
- `PLG_OID` is the object identifier for the matching rule

The sample plug-in `#defines` each of these. If the plug-in implements several matching rules, the initialization function must register each one. Code Example 10-3 registers only one.

Notice plug-ins must register factory functions separately, using `slapi_pblock_set()` as shown in Code Example 10-2 on page 171.

Handling Extensible Match Filters

This section explains and demonstrates how to handle extensible match filters corresponding to a matching rule. All matching rule plug-ins must enable filter matching.

The following steps outline how Directory Server handles an extensible match search.

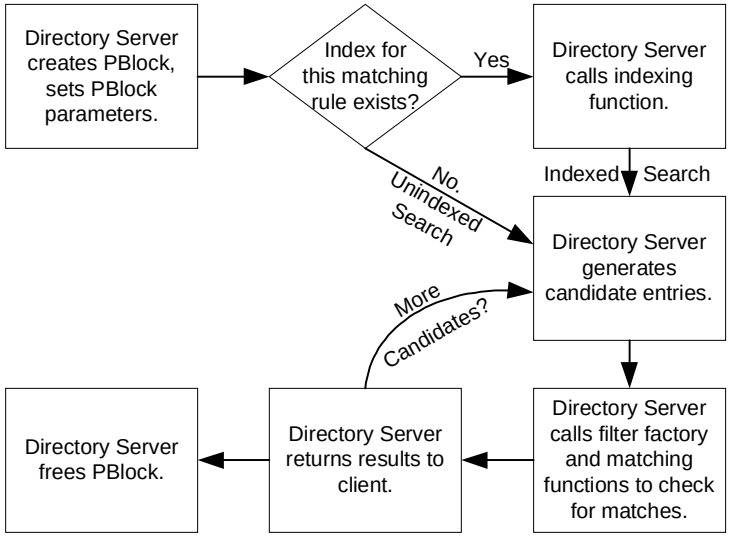
1. When Directory Server receives a search request containing an extensible match filter, it calls the *filter factory function* in the plug-in handling the corresponding matching rule, passing the filter factory a parameter block containing the extensible match filter information.
2. The filter factory function builds a filter object, and sets pointers to the object and the appropriate *filter matching* and *filter index functions*.
3. The server attempts to look up a set of results in an index rather than searching the entire directory. Directory Server only considers all entries in the directory, perhaps slowing down the search considerably, if one of the following is the case:
   - No existing index applies to the search
   - The plug-in specifies no indexer function
   - The plug-in generates no keys for the values specified

   a. In order to read an index, the server calls the filter index function. The filter index function helps the server locate the appropriate indexer function.

   b. Directory Server calls the indexer function to generate the keys. Refer to “Indexer Function,” on page 188 for details on that function.
4. Directory Server builds a list of candidate entries.
5. For each entry in the list of candidates, Directory Server calls the filter matching function. The filter matching function indicates whether an attribute value matches the search value.
6. Directory Server verifies that the entry is in the scope of the search before returning it to the client as a search result. Finally, Directory Server frees memory allocated for the operation.

Figure 10-1 summarizes how Directory Server processes the search request.

Figure 10-1 Directory Server Performing an Extensible Match Search

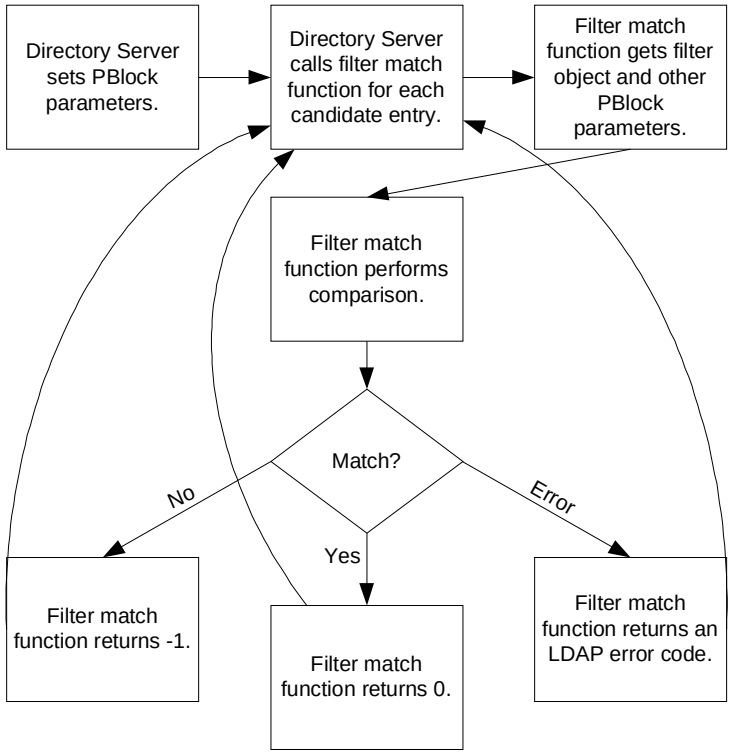


Filter Matching Function

Your filter matching function takes pointers to the filter object, the entry, and the first attribute to check for a match.

Figure 10-2 shows how Directory Server uses the filter matching function.

Figure 10-2 Filter Match Function Context



Notice the filter matching function returns `0` (match), `-1` (no match), or an LDAP error code for each entry processed.

Code Example 10-4 shows the filter match function for the case exact matching rule where a match occurs when the two `berval` structures match.

Code Example 10-4 Filter Match Function (`matchingrule.c`)

```
#include "slapi-plugin.h"

typedef struct plg_filter_t           /* For manipulating filter obj. */
{   char          *  f_type;          /* Attribute type to match      */
    int              f_op;            /* Type of comparison
                                       * (<, <=, ==, >=, >, substr)
                                       * for the filter.              */
    struct berval ** f_values;        /* Array of values to match     */
} plg_filter_t;

/* Check for a match against the filter.
```

Code Example 10-4 Filter Match Function (`matchingrule.c`) *(Continued)*

```
 * Returns:  0 filter matched
 *          -1 filter did not match
 *         > 0 an LDAP Error code                                      */
static int
plg_filter_match(
    void        * obj,                  /* Matching rule object        */
    Slapi_Entry * entry,                /* Entry to match              */
    Slapi_Attr  * attr                  /* Attributes to match         */
)
{
    plg_filter_t  * fobj  = (plg_filter_t *)obj;
    struct berval * mrVal = NULL;       /* Values handled as bervals... */
    Slapi_Value   * sval;               /* ...and as Slapi_Value structs*/
    int             rc    = -1;         /* No match                    */

    if (fobj && fobj->f_type && fobj->f_values && fobj->f_values[0]) {

        mrVal = fobj->f_values[0];

        /* Iterate through the attributes, matching subtypes.          */
        for (; attr != NULL; slapi_entry_next_attr(entry, attr, &attr)) {

            char * type = NULL;         /* Attribute type to check     */

            if ((slapi_attr_get_type(attr, &type) == 0) &&
                (type != NULL)                          &&
                (slapi_attr_type_cmp(fobj->f_type, type, 2) == 0)
            ) { /* slapi_attr_type_cmp(type1, type2, ==>2<==)
                 * matches subtypes, too. Refer to the reference
                 * documentation for details.                          */

                /* Type and subtype match, so iterate through the
                 * values of the attribute.                            */
                int hint = slapi_attr_first_value(attr, &sval);
                while (hint != -1) {
                    const struct berval * val = slapi_value_get_berval(sval);

                    /* The case exact matching rule
                     * compares the two bervals.
                     *
                     * Your matching rule may do
                     * lots of different checks here.                  */
                    rc = slapi_berval_cmp(val, mrVal);
                    if (rc == 0)  {    /* Successful match             */
                    /* If you have allocated memory for a custom
                     * matching rule, do not forget to release it.     */
                       return 0;
                    }

                    hint = slapi_attr_next_value(attr, hint, &sval);
                }
            }
        }
    }
```

Code Example 10-4 Filter Match Function (`matchingrule.c`) *(Continued)*

```
    return rc;
}
```

Notice that checking for a case exact match involves only `slapi_berval_cmp()`, which performs a byte by byte comparison. Your plug-in may do something more complex for the comparison.

Subtype Matches

Notice in the code for iterating through the attributes that `slapi_attr_type_cmp()` takes 2, the value assigned to `SLAPI_TYPE_CMP_SUBTYPE`, as its third argument. This forces comparison of attribute subtypes, such as the locale of an attribute, in addition to attribute types. Refer to the *Sun ONE Directory Server Plug-In API Reference* for details on plug-in API functions and their arguments.

Thread Safety

Directory Server never calls a filter matching function for the same filter object concurrently. Directory Server may, however, call the function concurrently for different filter objects. If you use global variables, ensure that your filter matching function handles them safely.

Filter Index Function

Your filter index function takes a parameter block from Directory Server, and sets pointers in the parameter block enabling the server to read the index.

Code Example 10-5 shows the filter index function for the case exact matching rule where the keys are the same as the values.

Code Example 10-5 Filter Index Function (`matchingrule.c`)

```
#include "slapi-plugin.h"

#define PLG_OID       "1.3.6.1.4.1.42.2.27.999.4.1"
#define SUBSYS        "CaseExactMatching Plugin"

typedef struct plg_filter_t           /* For manipulating filter obj. */
{   char          *  f_type;          /* Attribute type to match      */
    int             f_op;             /* Type of comparison
```

Code Example 10-5 Filter Index Function (`matchingrule.c`) *(Continued)*

```
                                        * (<, <=, ==, >=, >, substr)
                                        * for the filter.                */
    struct berval ** f_values;         /* Array of values to match      */
} plg_filter_t;

static int
plg_filter_index(Slapi_PBlock * pb)
{
    int           rc       = LDAP_UNAVAILABLE_CRITICAL_EXTENSION;
    void         * obj      = NULL;    /* Server lets our plug-in       */
    plg_filter_t * fobj;               /* handle the object type.       */
    int           query_op = SLAPI_OP_EQUAL; /* Only exact matches     */

    if (!slapi_pblock_get(pb, SLAPI_PLUGIN_OBJECT, &obj)) {

        fobj = (plg_filter_t *)obj;

        /* Case exact match requires no modifications to
         * the at this point. Your plug-in may however
         * modify the object, then set it again in the
         * parameter block.                                             */
        rc  = slapi_pblock_set(        /* This is for completeness.     */
            pb,                        /* Your plug-in may modify       */
            SLAPI_PLUGIN_OBJECT,       /* the object if necessary.      */
            fobj
        );
        rc |= slapi_pblock_set(        /* Set attr type to match.       */
            pb,
            SLAPI_PLUGIN_MR_TYPE,
            fobj->f_type
        );
        rc |= slapi_pblock_set(        /* Set fcn to obtain keys.       */
            pb,
            SLAPI_PLUGIN_MR_INDEX_FN,
            (void*)plg_index_entry
        );
        rc |= slapi_pblock_set(        /* Set values to obtain keys.    */
            pb,
            SLAPI_PLUGIN_MR_VALUES,
            fobj->f_values
        );
        rc |= slapi_pblock_set(        /* This is for completeness.     */
            pb,                        /* Your plug-in may set a        */
            SLAPI_PLUGIN_MR_OID,       /* different MR OID than the     */
            PLG_OID                    /* one in the parameter block    */
        );
        rc |= slapi_pblock_set(        /* <, <=, ==, >=, >, substr      */
            pb,                        /* In this case, ==              */
            SLAPI_PLUGIN_MR_QUERY_OPERATOR,
            &query_op
        );
    }
```

Code Example 10-5 Filter Index Function (`matchingrule.c`) *(Continued)*

```
    return rc;
}
```

Code Example 10-5 uses the following variables and macros:

- `fobj->ftype` indicates the attribute type specified in the filter
- `plg_index_entry()` indicates the indexer function for generating keys from values
- `fobj->values` points to the berval array of attribute values
- `PLG_OID` is the object identifier of the matching rule
- `query_op` indicates the query operator from the filter

Input Parameters

A filter index function may get a pointer to the filter object from `SLAPI_PLUGIN_OBJECT` in the parameter block.

Output Parameters

A filter index function should set values for at least the following in the parameter block before returning control to Directory Server.

- `SLAPI_PLUGIN_MR_INDEX_FN` (pointer to the indexer for generating the keys)
- `SLAPI_PLUGIN_MR_OID`
- `SLAPI_PLUGIN_MR_QUERY_OPERATOR`
- `SLAPI_PLUGIN_MR_TYPE`
- `SLAPI_PLUGIN_MR_VALUES`
- `SLAPI_PLUGIN_OBJECT` (if modified by your function)

Refer to the *Sun ONE Directory Server Plug-In API Reference* for details about the parameter block.

Thread Safety

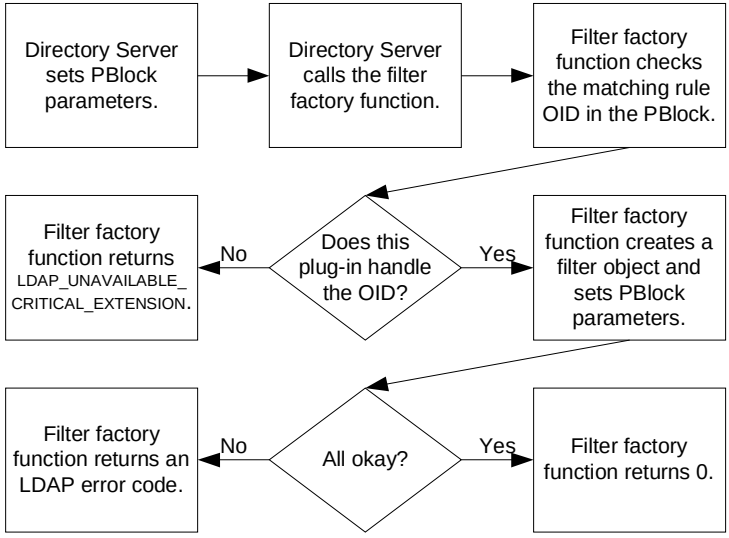
Directory Server never calls a filter index function for the same filter object concurrently. Directory Server may, however, call the function concurrently for different filter objects. If you use global variables, ensure that your function handles them safely.

Filter Factory Function

Your filter factory function takes a parameter block from Directory Server, and sets parameters such that Directory Server can build a list of candidate entries and check them for matches.

Figure 10-3 shows how the filter factory function operates.

Figure 10-3 Filter Factory Function Context



Code Example 10-6 shows the filter factory function for the case exact matching rule.

Code Example 10-6 Filter Factory Function (`matchingrule.c`)

```
#include "slapi-plugin.h"

#define PLG_OID       "1.3.6.1.4.1.42.2.27.999.4.1"
```

Code Example 10-6 Filter Factory Function (`matchingrule.c`) *(Continued)*

```
#define SUBSYS        "CaseExactMatching Plugin"

/* Functions to obtain connection information for logging.            */
static long mypblock_get_msgid( Slapi_PBlock * pb);
static int  mypblock_get_connid(Slapi_PBlock * pb);
static int  mypblock_get_opid(  Slapi_PBlock * pb);

typedef struct plg_filter_t           /* For manipulating filter obj. */
{   char          *  f_type;          /* Attribute type to match      */
    int              f_op;            /* Type of comparison
                                       * (<, <=, ==, >=, >, substr)
                                       * for the filter.              */
    struct berval ** f_values;        /* Array of values to match     */
} plg_filter_t;

static int
plg_filter_create(Slapi_PBlock * pb)
{
    int             rc       = LDAP_UNAVAILABLE_CRITICAL_EXTENSION;
    char          * mr_oid   = NULL;   /* MR OID from the server       */
    char          * mr_type  = NULL;   /* Attr type to match           */
    struct berval * mr_value = NULL;   /* Attr value to match          */
    plg_filter_t  * fobj     = NULL;   /* Object to create             */

    if (
        slapi_pblock_get(pb, SLAPI_PLUGIN_MR_OID, &mr_oid) ||
        (mr_oid == NULL)
    ) {
        slapi_log_error_ex(
            -1, /* errorId */
            mypblock_get_msgid(pb),
            mypblock_get_connid(pb),
            mypblock_get_opid(pb),
            SUBSYS,
            SLAPI_INTERNAL_ERROR,
            "plg_filter_create failed: NULL OID values are invalid.\n"
        );
        }
    else if (strcmp(mr_oid, PLG_OID) == 0) {
        /* The MR OID from the server is handled by this plug-in.      */
        if (
            (slapi_pblock_get(pb, SLAPI_PLUGIN_MR_TYPE,  &mr_type) == 0)  &&
            (mr_type  != NULL)                                            &&
            (slapi_pblock_get(pb, SLAPI_PLUGIN_MR_VALUE, &mr_value) == 0) &&
            (mr_value != NULL)
        ) { /* ...provide a pointer to a filter match function.        */
            int op            = SLAPI_OP_EQUAL;
            fobj              = (plg_filter_t *)slapi_ch_calloc(
                                    1,
                                    sizeof (plg_filter_t)
                                );
            fobj->f_type      = slapi_ch_strdup(mr_type);
            fobj->f_op        = op;
            fobj->f_values    = (struct berval **)slapi_ch_malloc(
                                    2 * sizeof(struct berval *)
```

Code Example 10-6 Filter Factory Function (`matchingrule.c`) *(Continued)*

```
                                );
            fobj->f_values[0] = slapi_ch_bvdup(mr_value);
            fobj->f_values[1] = NULL;

            rc  = slapi_pblock_set(        /* Set object destructor.   */
                pb,
                SLAPI_PLUGIN_DESTROY_FN,
                (void *)plg_filter_destroy
            );
            rc |= slapi_pblock_set(        /* Set object itself.       */
                pb,
                SLAPI_PLUGIN_OBJECT,
                (void  *)fobj
            );
            rc |= slapi_pblock_set(        /* Set filter match fcn.    */
                pb,
                SLAPI_PLUGIN_MR_FILTER_MATCH_FN,
                (void *)plg_filter_match
            );
            rc |= slapi_pblock_set(        /* Set sorting function.    */
                pb,
                SLAPI_PLUGIN_MR_FILTER_INDEX_FN,
                (void *)plg_filter_index
            );

            if (rc == 0) {
                slapi_log_info_ex(
                    SLAPI_LOG_INFO_AREA_PLUGIN,
                    SLAPI_LOG_INFO_LEVEL_DEFAULT,
                    mypblock_get_msgid(pb),
                    mypblock_get_connid(pb),
                    mypblock_get_opid(pb),
                    SUBSYS,
                    "plg_filter_create (oid %s; type %s) OK\n",
                    mr_oid, mr_type
                );
            } else {
                slapi_log_error_ex(
                    -1, /* errorId */
                    mypblock_get_msgid(pb),
                    mypblock_get_connid(pb),
                    mypblock_get_opid(pb),
                    SUBSYS,
                    SLAPI_INTERNAL_ERROR,
                    "plg_filter_create (oid %s; type %s) - pblock error \n",
                    mr_oid, mr_type
                );
            }
        } else { /* Missing parameters in the pblock                    */
            slapi_log_error_ex(
                -1,
                mypblock_get_msgid(pb),
                mypblock_get_connid(pb),
                mypblock_get_opid(pb),
                SUBSYS,
```

Code Example 10-6 Filter Factory Function (`matchingrule.c`) *(Continued)*

```
                "Parameter errors ",
                "plg_filter_create: invalid input pblock.\n"
            );
        }
    }

    return rc;
}
```

Notice this function returns `LDAP_UNAVAILABLE_CRITICAL_EXTENSION` if it does not handle the matching rule OID in the parameter block. Refer to "Handling an Unknown Matching Rule," on page 195 for details on this behavior.

Input Parameters

A filter factory function may get values for the following from the parameter block.

- `SLAPI_PLUGIN_MR_OID`
- `SLAPI_PLUGIN_MR_TYPE`
- `SLAPI_PLUGIN_MR_VALUE`
- `SLAPI_PLUGIN_PRIVATE` (if your plug-in uses private data specified in the plug-in initialization function)

Output Parameters

An indexer factory function should set values for the following in the parameter block before returning control to Directory Server.

- `SLAPI_PLUGIN_DESTROY_FN` (pointer to the destructor for the filter object)
- `SLAPI_PLUGIN_MR_FILTER_INDEX_FN` (pointer to the filter index function)
- `SLAPI_PLUGIN_MR_FILTER_MATCH_FN` (pointer to the filter match function)
- `SLAPI_PLUGIN_MR_FILTER_RESET_FN` (if required, pointer to a filter reset function)
- `SLAPI_PLUGIN_MR_FILTER_REUSABLE` (if required to specify that the filter may be reused)
- `SLAPI_PLUGIN_OBJECT` (pointer to the filter object)

Refer to the *Sun ONE Directory Server Plug-In API Reference* for details about the parameter block.

Thread Safety

This function must be thread safe. Directory Server may call this function concurrently.

Filter Object Destructor

A filter object destructor function frees memory allocated for a filter object set up by the filter factory function. Directory Server calls the destructor after operation completes, passing the parameter block indicating the filter object as the value of `SLAPI_PLUGIN_OBJECT`.

Code Example 10-7 shows an example object and destructor.

Code Example 10-7 Filter Object and Destructor (`matchingrule.c`)

```
#include "slapi-plugin.h"

typedef struct plg_filter_t           /* For manipulating filter obj. */
{   char           *  f_type;         /* Attribute type to match      */
    int               f_op;          /* Type of comparison
                                       * (<, <=, ==, >=, >, substr)
                                       * for the filter.              */
    struct berval ** f_values;        /* Array of values to match     */
} plg_filter_t;


/* Free memory allocated for the filter object.                        */
static int
plg_filter_destroy(Slapi_PBlock * pb)
{
    void         * obj  = NULL;       /* Server lets our plug-in      */
    plg_filter_t * fobj = NULL;       /* handle the object type.      */

    if (!slapi_pblock_get(pb, SLAPI_PLUGIN_OBJECT, &obj))
    {
        fobj = (plg_filter_t *)obj;
        if (fobj){
            slapi_ch_free((void **)&fobj->f_type);
                if (fobj->f_values){
                    ber_bvecfree(fobj->f_values);
                }
            slapi_ch_free((void **)&fobj);
        }
    }
```

Code Example 10-7 Filter Object and Destructor (`matchingrule.c`) *(Continued)*

```
    return 0;
}
```

Thread Safety

Directory Server never calls a destructor for the same object concurrently.

Indexing Entries According to a Matching Rule

This section covers providing plug-in support for a directory index based on a matching rule. Matching rule plug-ins are not required to enable indexing for the matching rules they support.

NOTE Directory Server can use your plug-in for extensible match searches even if you provide no indexing capability.

Without support for indexing, however, Directory Server must generate search results from the entire directory, potentially ruining search performance.

Directory indexes speed up client searches. Directory Server can build a list of search result entries by looking through the index rather than the entire directory. When directories contain many entries, indexes offer a considerable search performance boost. Directory administrators therefore configure the directory to maintain indexes for attributes searched regularly.

Refer to the *Sun ONE Directory Server Administration Guide* for details on how to manage directory indexes.

How Directory Server Handles the Index

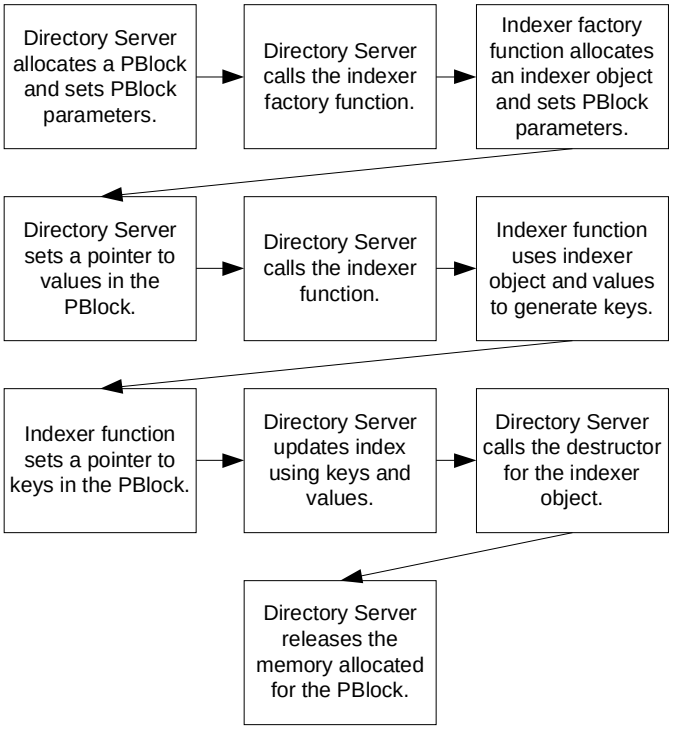
To maintain an index for a custom matching rule supported by your plug-in, the server requires an *indexer function* capable of translating attribute values to index keys. Directory Server calls this indexer function when the creating the index, and subsequently when adding, modifying, or deleting attribute values.

To read an index for a custom matching rule, Directory Server requires a *filter index function*. The filter factory function provides a pointer to this function. Refer to “Handling Extensible Match Filters,” on page 173 for details.

Directory Server relies on the *indexer factory function* in your plug-in to set up an indexing object and provide a pointer to the appropriate indexer function.

Figure 10-4 shows how Directory Server performs indexing based on a matching rule.

Figure 10-4 Directory Server Maintaining an Index Using a Matching Rule



The following steps summarize the process shown in Figure 10-4.

1. Directory Server creates a parameter block, setting parameters to indicate the matching rule OID and attribute type to match. Directory Server then calls the indexer factory function.

2. The indexer factory function examines the parameter block, allocates an indexer object if necessary, and sets pointers to the indexer and filter index functions and if necessary a destructor to free the indexer object before returning control to Directory Server.
3. Directory Server sets the parameter block to indicate attribute values to translate to keys and calls the indexer function.
4. The indexer function translates values to keys. After the indexer returns, Directory Server uses the keys to update the index.
5. Directory Server frees the parameter block and if necessary frees the indexer object using the destructor provided.

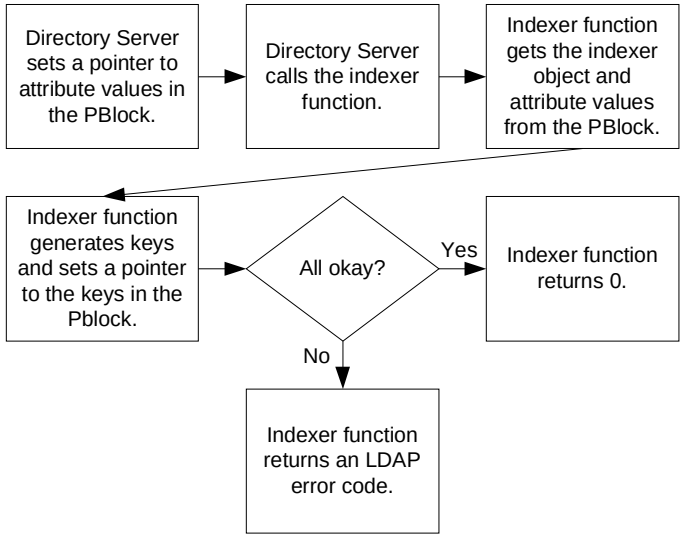
Be aware that Directory Server may call indexer factory functions concurrently. Indexer factory functions must therefore be thread safe.

Indexer Function

An indexer function takes a parameter block from Directory Server containing a pointer to a `berval` array of attribute values, generates a `berval` array of corresponding keys from the values, and sets a pointer in the parameter block to the keys.

Figure 10-5 shows how Directory Server uses the indexer function.

Figure 10-5 Indexer Function Context



Code Example 10-8 shows the indexer function for the case exact matching rule where the keys are the same as the values.

Code Example 10-8 Indexer Function (`matchingrule.c`)

```
#include "slapi-plugin.h"

typedef struct plg_filter_t           /* For manipulating filter obj. */
{   char          *  f_type;          /* Attribute type to match      */
    int             f_op;            /* Type of comparison
                                      * (<, <=, ==, >=, >, substr)
                                      * for the filter.               */
    struct berval ** f_values;        /* Array of values to match     */
} plg_filter_t;

static int
plg_index_entry(Slapi_PBlock * pb)
{
    int             rc     = LDAP_OPERATIONS_ERROR;
    void          *  obj    = NULL;    /* Server lets our plug-in      */
    plg_filter_t  *  fobj;             /* handle the object type.      */
    struct berval ** values;           /* Values from server           */

    if (slapi_pblock_get(pb, SLAPI_PLUGIN_OBJECT, &obj) == 0) {
        fobj = (plg_filter_t *)obj;
        if (
            (slapi_pblock_get(pb, SLAPI_PLUGIN_MR_VALUES, &values) == 0) &&
            (values != NULL)
        ) {
```

Code Example 10-8 Indexer Function (`matchingrule.c`) *(Continued)*

```
            /* The case exact match builds the index keys
             * from the values by copying the values because
             * the keys and values are the same.
             *
             * Your matching rule may do something quite
             * different before setting the keys associated
             * with the values in the parameter block.                */
            rc = slapi_pblock_set(     /* Set keys based on values.    */
                pb,
                SLAPI_PLUGIN_MR_KEYS,
                slapi_ch_bvecdup(values)
            );
        }
    }

    return rc;
}
```

In Code Example 10-8, `obj` points to the indexer object, and `values` points to the `berval` array of attribute values. Notice that the function returns `LDAP_OPERATIONS_ERROR` on failure. Technically `LDAP_OPERATIONS_ERROR` indicates bad sequencing of LDAP operations; for historical reasons, we use it in this context to indicate an internal error.

Input Parameters

An indexer function may get values for the following from the parameter block.

- `SLAPI_PLUGIN_MR_VALUES`
- `SLAPI_PLUGIN_MR_OBJECT`

Output Parameter

An indexer function should generate the berval array of keys, and set `SLAPI_PLUGIN_MR_KEYS` in the parameter block before returning control to Directory Server. Refer to the *Sun ONE Directory Server Plug-In API Reference* for details about the parameter block.

Thread Safety

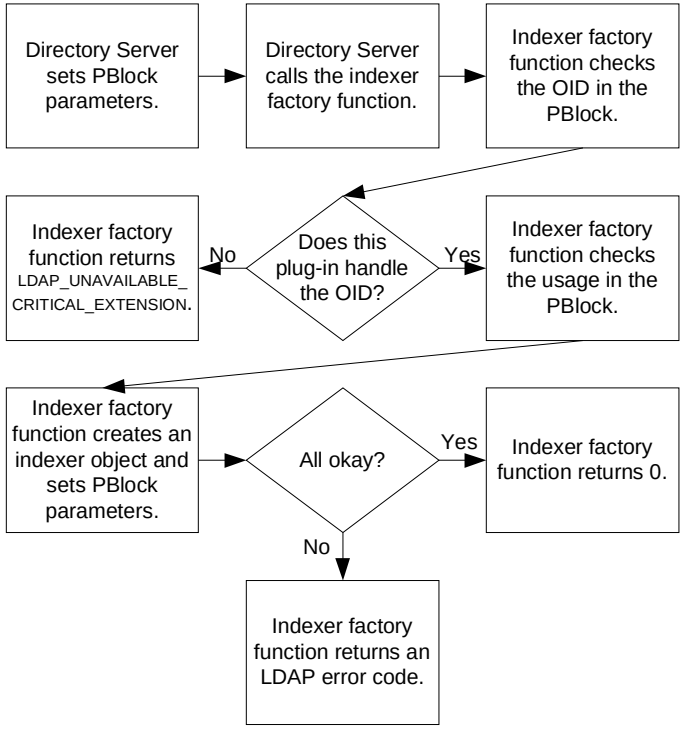
Directory Server never calls an indexer for the same indexer object concurrently. Directory Server may, however, call the function concurrently for different indexer objects. If you use global variables, ensure that your function handles them safely.

Indexer Factory Function

Your indexer factory function takes a parameter block from Directory Server, and sets parameters such that Directory Server can update an index or sort results based on a matching rule.

Figure 10-6 shows how Directory Server uses the indexer factory function.

Figure 10-6 Indexer Factory Function Context



Code Example 10-9 shows the indexer factory function for the case exact matching rule.

Code Example 10-9 Indexer Factory Function (`matchingrule.c`)

```
#include "slapi-plugin.h"

#define PLG_OID       "1.3.6.1.4.1.42.2.27.999.4.1"
#define SUBSYS        "CaseExactMatching Plugin"
```

Code Example 10-9 Indexer Factory Function (`matchingrule.c`) *(Continued)*

```
/* Functions to obtain connection information for logging.             */
static long mypblock_get_msgid( Slapi_PBlock * pb);
static int  mypblock_get_connid(Slapi_PBlock * pb);
static int  mypblock_get_opid(  Slapi_PBlock * pb);

static int
plg_indexer_create(Slapi_PBlock * pb)
{
    int    rc     = LDAP_UNAVAILABLE_CRITICAL_EXTENSION; /* Init failed*/
    char * mr_oid = NULL;              /* MR OID from the server       */

    if (slapi_pblock_get(pb, SLAPI_PLUGIN_MR_OID, mr_oid) ||
        (mr_oid == NULL)) {
        slapi_log_error_ex(
            -1, /* errorId */
            mypblock_get_msgid(pb),
            mypblock_get_connid(pb),
            mypblock_get_opid(pb),
            SUBSYS,
            SLAPI_INTERNAL_ERROR,
            "plg_indexer_create failed: NULL OID values are invalid.\n"
        );
    } else if (strcmp(mr_oid, PLG_OID) == 0) {
        if ( /* The MR OID from the server is handled by this plug-in. */
            (slapi_pblock_set(         /* This is for completeness.    */
                pb,                    /* Your plug-in may set a       */
                SLAPI_PLUGIN_MR_OID,   /* different OID than the one   */
                PLG_OID                /* provided by the server.      */
            ) == 0) &&
            (slapi_pblock_set(         /* Provide an appropriate       */
                pb,                    /* indexer function pointer.    */
                SLAPI_PLUGIN_MR_INDEX_FN,
                (void *)plg_index_entry
            ) == 0) &&
            (slapi_pblock_set(         /* Set the object destructor.   */
                pb,
                SLAPI_PLUGIN_DESTROY_FN,
                (void *)plg_filter_destroy
            ) == 0)) {
                rc = LDAP_SUCCESS;
        } else {
            slapi_log_error_ex(
                -1, /* errorId */
                mypblock_get_msgid(pb),
                mypblock_get_connid(pb),
                mypblock_get_opid(pb),
                SUBSYS,
                SLAPI_INTERNAL_ERROR,
                "plg_indexer_create failed %d \n", rc
            );
        }
    }
```

Code Example 10-9 Indexer Factory Function (`matchingrule.c`) *(Continued)*

```
    return rc;
}
```

In Code Example 10-9, `PLG_OID` is the object identifier for the matching rule, `plg_index_entry()` is the indexer, and `plg_filter_destroy()` is the destructor. Notice that the function returns `LDAP_UNAVAILABLE_CRITICAL_EXTENSION` on failure.

Input Parameters

An indexer factory function may read the values for the following from the parameter block.

- `SLAPI_PLUGIN_MR_OID`
- `SLAPI_PLUGIN_MR_TYPE`
- `SLAPI_PLUGIN_MR_USAGE` (indicates whether results must be sorted)
- `SLAPI_PLUGIN_PRIVATE` (if your plug-in uses private data you specify in the plug-in initialization function)

Output Parameters

An indexer factory function should set values for at least the following in the parameter block before returning control to Directory Server.

- `SLAPI_PLUGIN_DESTROY_FN` (pointer to the indexer object destructor)
- `SLAPI_PLUGIN_MR_INDEX_FN` (pointer to the indexer)
- `SLAPI_PLUGIN_OBJECT` (pointer to the indexer object)

Refer to the *Sun ONE Directory Server Plug-In API Reference* for details about the parameter block.

Thread Safety

This function must be thread safe. Directory Server may call this function concurrently.

Indexer Object Destructor

An indexer object destructor function frees memory allocated for an indexer object set up by the indexer factory function. Directory Server calls the destructor after an index operation completes.

Indexer object destructors take a parameter block as their only argument, as do filter object destructors. The parameter block holds a pointer to the object in `SLAPI_PLUGIN_OBJECT`. Refer to the Code Example 10-7 on page 185 for details.

Thread safety

Directory Server never calls a destructor for the same object concurrently.

Enabling Sorting According to a Matching Rule

Clients may request that Directory Server sort results from an extensible match search. This section explains how to enable sorting based on a matching rule.

Directory Server performs sorting as a variation of indexing, using the keys generated by an indexer function to sort results. The following steps describe the process.

1. Directory Server creates a parameter block as for indexing, setting `SLAPI_PLUGIN_MR_USAGE` to `SLAPI_PLUGIN_MR_USAGE_SORT`, before passing the parameter block to your indexer factory function.
2. Your indexer factory function should set parameters in the parameter block as for indexing.

   If your sort function is different from your normal indexer function, ensure that your function checks the value of `SLAPI_PLUGIN_MR_USAGE`, and then sets `SLAPI_MR_INDEXER_FN` accordingly.
3. Directory Server sets `SLAPI_PLUGIN_MR_VALUES` in the parameter block as a pointer to the values to be sorted. Directory Server then passes the parameter block the indexer function.
4. Directory Server sorts results based on the keys your indexer function set in `SLAPI_PLUGIN_MR_KEYS`.
5. Directory Server frees memory allocated for the operation.

Refer to “Indexing Entries According to a Matching Rule,” on page 186 for details on how matching rule plug-ins can allow Directory Server to perform indexing based on a matching rule.

Handling an Unknown Matching Rule

This section explains how Directory Server finds a plug-in for a matching rule OID not known to Directory Server.

Internal List of Correspondences

Directory Server identifies matching rules by OID. It keeps an internal list of which matching rule plug-ins handle which OIDs.

Directory Server uses the internal list to determine which plug-in to call when it receives extensible match filter search request including a matching rule OID. Directory Server initially builds the list as matching rule plug-ins register OIDs they handle using `slapi_matchingrule_register()` in the plug-in initialization function.

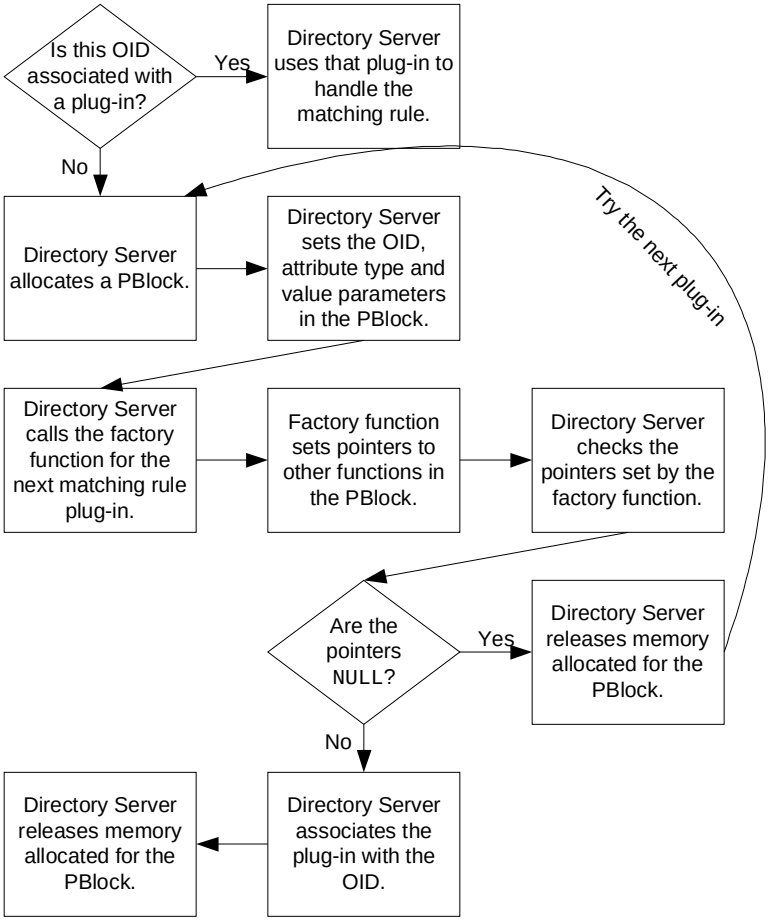
OIDs Not in the Internal List

When it encounters a matching rule OID not in the list, Directory Server queries each matching rule plug-in through registered factory functions. For each matching rule plug-in, Directory Server passes a parameter block containing the OID to the matching rule factory function. It then checks whether the factory function has in return provided a pointer in the parameter block to the appropriate indexing or matching function for the OID.

If the factory function sets a pointer to the appropriate function, Directory Server assumes the plug-in supports the matching rule.

Figure 10-7 summarizes how Directory Server checks whether a plug-in handles a specific filter match operation. The process for indexing closely resembles the process for matching.

Figure 10-7 Finding a Matching Rule Plug-In for an Unknown OID



The following steps summarize the steps in Figure 10-7.

1. Directory Server finds it has no match in the internal list of plug-ins to handle the OID. It therefore creates a parameter block, and sets appropriate parameters for the factory including `SLAPI_PLUGIN_MR_OID`.
2. Directory Server calls the factory function. The factory function either sets a pointer in the parameter block to the appropriate function to handle the matching rule, or it leaves the pointer `NULL`.

3. Directory Server checks the parameter block after the factory function returns. If the factory function has set a function pointer in the parameter block, Directory Server updates its internal list, indicating that the plug-in supports the matching rule operation associated with the OID. Otherwise, the pointer remains `NULL`, and Directory Server tries the process again on the next matching rule plug-in registered.

   If after calling all matching rule plug-ins, Directory Server has found no plug-in to handle the matching rule OID, it returns `LDAP_UNAVAILABLE_CRITICAL_EXTENSION` to the client.

4. Directory Server frees the memory allocated for the operation.

Be aware that Directory Server calls plug-in factory functions for the purpose of extending the internal list of correspondences.

Chapter 11

Writing Password Storage Scheme Plug-Ins

This chapter covers how to write plug-ins that let you modify how Directory Server stores password attribute values.

This chapter includes the following sections:

- Calling Password Storage Scheme Plug-Ins
- Writing a Password Storage Scheme Plug-In

Calling Password Storage Scheme Plug-Ins

This section describes the circumstances in which Directory Server calls password storage scheme plug-ins and how password values are expected to be handled by the plug-ins.

Two Types

Two types of password storage scheme plug-ins work with Directory Server, `pwdstoragescheme` and `reverpwdstoragescheme`. The former is one-way, in that once the server encodes and stores a password, the password is not decoded. The `pwdstoragescheme` type therefore includes plug-in functions only for encoding passwords to be stored and for comparing incoming passwords with encoded, stored passwords. The latter is reversible, in that the plug-in allows Directory Server both to encode and to decode passwords. The `reverpwdstoragescheme` type therefore includes encode, compare, and decode plug-in functions.

NOTE This chapter covers the one-way type `pwdstoragescheme` plug-ins.

Pre-Installed Schemes

Existing schemes delivered with Directory Server are provided as password storage scheme plug-ins. Search `cn=config` for entries whose DN contains `cn=Password Storage Schemes`. The default password storage scheme uses the Salted Secure Hashing Algorithm (SSHA).

You can change the password storage scheme used to encode user passwords. Refer to the *Sun ONE Directory Server Administration Guide* for instructions.

Affects Password Attribute Values

Password storage scheme plug-in functions act on password (`userPassword`) attribute values. Directory Server registers password storage scheme plug-ins at startup. After startup, any registered, enabled password storage scheme plug-in can then be used to encode password values, and to compare incoming passwords to the encoded values. Which plug-in Directory Server invokes depends on the password storage scheme used for the entry in question.

Invoked for Add and Modify Requests

Add and modify requests can imply that Directory Server encode an input password, and then store it in the directory. First, Directory Server determines the storage scheme for the password value. Next, it invokes the plug-in encode function for the appropriate scheme. The encode function returns the encoded password to Directory Server.

Invoked for Bind Requests

Bind requests imply that Directory Server compares an input password value to a stored password value. As for add and modify requests, Directory Server determines the storage scheme for the password value. Next, it invokes the plug-in compare function for the appropriate scheme. The compare scheme returns an `int` that communicates to Directory Server whether the two passwords match as described in “Comparing a Password,” on page 203.

Part of a Password Policy

Password storage scheme plug-ins typically do no more than encode passwords and compare input passwords with stored, encoded passwords. In other words, they represent only a part of a comprehensive password policy. Refer to the *Sun ONE Directory Server Deployment Guide* for suggestions on designing secure directory services.

Writing a Password Storage Scheme Plug-In

This section demonstrates how to write a plug-in that encodes passwords and allows Directory Server to compare stored passwords with passwords provided by a client application.

CAUTION The example does *not* constitute a secure password storage scheme.

The source for the example plug-in referenced in this section is *ServerRoot*`/plugins/slapd/slapi/examples/testpwdstore.c`. For encoding and comparing, the plug-in performs an exclusive or with 42 on each character of the password.

Encoding a Password

When Directory Server calls a password storage scheme plug-in encode function, it passes that function an input password `char *` and expects an encoded password `char *` in return. The prototype for our example encode function, `xorenc()`, is:

```
static char * xorenc(char * pwd);
```

Allocate space for the encoded password using `slapi_ch_malloc()` rather than regular `malloc()`. Directory Server can then terminate with an “out of memory” message if allocation fails. Free memory using `slapi_ch_free()`.

By convention, we prefix the encoded password with the common name, of the password storage scheme, enclosed in braces, `{` and `}`. In other words, our example plug-in has `cn: XOR`.

Our name is declared in the example:

```
static char * name = "XOR";   /* Storage scheme name */
```

We return encoded strings prefixed with {XOR}, and register that name with Directory Server.

Code Example 11-1 Encoding a `userPassword` Value (`testpwdstore.c`)

```
#include "slapi-plugin.h"

static char * name           ="XOR";   /* Storage scheme name */

#define PREFIX_START '{'
#define PREFIX_END   '}'

static char *
xorenc(char * pwd)
{
    char * tmp    = NULL;              /* Used for encoding   */
    char * head   = NULL;              /* Encoded password    */
    char * cipher = NULL;              /* Prefix, then pwd    */
    int i, len;

    /* Allocate space to build the encoded password          */
    len = strlen(pwd);
    tmp = slapi_ch_malloc(len + 1);
    if (tmp == NULL) return NULL;

    memset(tmp, '\0', len + 1);
    head = tmp;

    /* Encode. This example is not secure by any means.      */
    for (i = 0; i < len; i++, pwd++, tmp++) *tmp = *pwd ^ 42;

    /* Add the prefix to the cipher                          */
    if (tmp != NULL) {
        cipher = slapi_ch_malloc(3 + strlen(name) + strlen(head));
        if (cipher != NULL) {
            sprintf(cipher,"%c%s%c%s",PREFIX_START,name,PREFIX_END,head);
        }
    }
    slapi_ch_free((void **) &head);

    return (cipher);                   /* Server frees cipher */
}
```

Notice we free only memory allocated for temporary use. Directory Server frees memory for the `char *` returned, not the plug-in. For details on `slapi_ch_malloc()` and `slapi_ch_free()`, refer to the *Sun ONE Directory Server Plug-In API Reference*.

Comparing a Password

When Directory Server calls a password storage scheme plug-in compare function, it passes that function an input password `char *` and a stored, encoded password `char *` from the directory. The compare function returns `0` if the input password matches the password from the directory. It returns `1` otherwise. The prototype for our example compare function, `xorcmp()`, is therefore:

```
static int xorcmp(char * userpwd, char * dbpwd);
```

Here, `userpwd` is the input password and `dbpwd` is the password from the directory. The compare function must encode the input password to compare the result to the password from the directory.

Code Example 11-2 Comparing a `userPassword` Value (`testpwdstore.c`)

```
#include "slapi-plugin.h"

static int
xorcmp(char * userpwd, char * dbpwd)
{
    /* Check the correspondence of the two char by char       */
    int i, len = strlen(userpwd);
    for (i = 0; i < len; i++) {
        if ((userpwd[i] ^ 42) != dbpwd[i])
            return 1;                  /* Different passwords */
    }
    return 0;                          /* Identical passwords */
}
```

Notice Directory Server strips the prefix from the password before passing it to the compare function. In other words, we need not account for `{XOR}` in this case.

Not all encoding algorithms have such a trivial compare function.

Registering the Plug-In

You must register four password storage scheme specific items with Directory Server: the storage scheme name (used for the prefix), the encode function, the compare function, and the decode function.

Notice we provide no decoding function. In this case, Directory Server never decodes user passwords once they are stored.

Code Example 11-3 Registering a Password Storage Scheme Plug-In (`testpwdstore.c`)

```
#include "slapi-plugin.h"

static char * name           ="XOR";   /* Storage scheme name */

static Slapi_PluginDesc desc = {
    "xor-password-storage-scheme",     /* Plug-in identifier  */
    "Sun Microsystems, Inc.",          /* Vendor name         */
    "5.2",                             /* Revision number     */
    "Exclusive-or example (XOR)"       /* Plug-in description */
};

#ifdef _WIN32
__declspec(dllexport)
#endif
int
xor_init(Slapi_PBlock * pb)
{
    int rc = 0;                        /* 0 means success     */
    rc |= slapi_pblock_set(            /* Plug-in API version */
        pb,
        SLAPI_PLUGIN_VERSION,
        (void *) SLAPI_PLUGIN_CURRENT_VERSION
    );
    rc |= slapi_pblock_set(            /* Plug-in description */
        pb,
        SLAPI_PLUGIN_DESCRIPTION,
        (void *) &desc
    );
    rc |= slapi_pblock_set(            /* Storage scheme name */
        pb,
        SLAPI_PLUGIN_PWD_STORAGE_SCHEME_NAME,
        (void *) name
    );
    rc |= slapi_pblock_set(            /* Encode password     */
        pb,
        SLAPI_PLUGIN_PWD_STORAGE_SCHEME_ENC_FN,
        (void *) xorenc
    );
    rc |= slapi_pblock_set(            /* Compare password    */
        pb,
        SLAPI_PLUGIN_PWD_STORAGE_SCHEME_CMP_FN,
        (void *) xorcmp
    );
    rc |= slapi_pblock_set(            /* Never decode pwd    */
        pb,
        SLAPI_PLUGIN_PWD_STORAGE_SCHEME_DEC_FN,
        NULL
    );
    return rc;
}
```

Creating a Configuration Entry

Build the plug-in if you have not done so already.

The configuration entry for the plug-in resembles other entries under `cn=config` whose DN contains `cn=Password Storage Schemes`. Create an LDIF file similar to that of Code Example 11-4.

Code Example 11-4 Configuration Entry (`testpwdstore.c`)

```
dn: cn=XOR,cn=Password Storage Schemes,cn=plugins,cn=config
objectclass: top
objectclass: nsSlapdPlugin
cn: XOR
nsslapd-pluginPath: <ServerRoot>/plugins/slapd/slapi/examples/<LibName>
nsslapd-pluginInitFunc: xor_init
nsslapd-pluginType: pwdstoragescheme
nsslapd-pluginEnabled: on
nsslapd-pluginId: xor-password-storage-scheme
nsslapd-pluginVersion: 5.2
nsslapd-pluginVendor: Sun Microsystems, Inc.
nsslapd-pluginDescription: Exclusive-or example (XOR)
```

Notice `nsslapd-pluginEnabled` must be set to on. Otherwise, no one can use the password storage scheme. Setting `nsslapd-pluginEnabled` to on does not suffice, however, for the plug-in to be used. In addition to enabling the plug-in, you must configure Directory Server to use the password storage scheme it provides.

Add the entry to the directory configuration. For example:

```
$ ldapmodify -a -p port -D "cn=Directory Manager" -w password -f file.ldif
```

Here, *file.ldif* contains the plug-in configuration entry.

Restart Directory Server so it loads the plug-in at startup:

```
$ ServerRoot/slapd-serverID/restart-slapd
```

At this point, the XOR password storage scheme plug-in should be enabled for Directory Server, and visible in the server configuration.

Trying It Out

This section demonstrates the example plug-in for this chapter. Plug the XOR password storage scheme into Directory Server if you have not done so already.

Perform a Quick Test

Before we do anything involved, quickly check that Directory Server manages to call the plug-in encode function as expected. To perform this quick test, we use the `pwdhash` tool. The `pwdhash` tool can be used to have Directory Server encode a password, then display the result.

Code Example 11-5 Testing the Password Storage Scheme

```
$ cd ServerRoot/bin/slapd/server
$ ./pwdhash -D ServerRoot/slapd-serverID -s XOR password
{XOR}ZKYY]EXN
```

Do not be concerned with the exact value of the resulting encoded password. The output should, however, start with `{XOR}`.

As Directory Server calls the encode function dynamically, we can fix the plug-in library, then try `pwdhash` again without touching Directory Server. That is, if this quick test does not work for you, now is a good time to fix the example.

Add Example Users with Clear Text Passwords

In order to experiment with user passwords, we need some users with passwords. Here, we create a directory suffix, `dc=example,dc=com`, whose users we load from an LDIF file, *`ServerRoot`*`/slapd-`*`serverID`*`/ldif/Example-Plugin.ldif`. The example user passwords appear in clear text in `Example-Plugin.ldif`.

Perform the following steps to load the example into the directory.

1. Open the Directory Server Console.
2. Use the console to create a new root suffix, `dc=example,dc=com`.

   Hint `Configuration` tab page. Select `Data` node. `Object` > `New Root Suffix`.
3. Set the password storage scheme to `No encryption (CLEAR)`.

   Hint Use the `Password encryption` drop-down list.
4. Log in as `cn=Directory Manager`.

   Hint `Console` > `Log in as New User...`

5. Import entries from *ServerRoot*/slapd-*serverID*/ldif/Example-Plugin.ldif.

   Hint Console > Import Databases...

Once the Example-Plugin.ldif entries have been imported, check that the user passwords appear in clear text. The console does not show passwords in clear text, so we use ldapsearch on the command line.

Code Example 11-6 Sample User Entry

```
$ ldapsearch -L -p port -D "cn=directory manager" -w password \
-b dc=example,dc=com uid=yyorgens
dn: uid=yyorgens,ou=People,dc=example,dc=com
mail: yyorgens@example.com
uid: yyorgens
secretary: uid=bcubbins,ou=People,dc=example,dc=com
givenName: Yolanda
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
sn: Yorgenson
cn: Yolanda Yorgenson
userPassword: yyorgens
```

Notice that Yolanda Yorgensons's password is yyorgens.

Encode a Password with the XOR Scheme

Here we use the XOR scheme to encode a new password for Yolanda Yorgenson.

1. As Directory Manager, set the password storage scheme for the suffix to Exclusive-or example (XOR).

   Hint Use the Password encryption drop-down list.

2. Change Yolanda's password to foobar12.

   Hint Directory tab page. Select example/People. Double-click yyorgens.

3. View Yolanda's newly encoded password.

Code Example 11-7 User Entry after Password Change

```
$ ldapsearch -L -p port -D "cn=directory manager" -w password \
-b dc=example,dc=com uid=yyorgens
dn: uid=yyorgens,ou=People,dc=example,dc=com
mail: yyorgens@example.com
uid: yyorgens
secretary: uid=bcubbins,ou=People,dc=example,dc=com
givenName: Yolanda
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
sn: Yorgenson
cn: Yolanda Yorgenson
userPassword: {XOR}ZKYY]EXN
```

Notice that Yolanda Yorgenson's password is XOR encoded.

Compare an XOR-Encoded Password

Yolanda has the right to search other entries under `dc=example,dc=com`. Here we search for Bartholomew Cubbins's entry as `yyorgens`.

Code Example 11-8 Binding with the New Password

```
$ ldapsearch -L -p port -b dc=example,dc=com \
-D "uid=yyorgens,ou=People,dc=example,dc=com" -w foobar12 \
uid=bcubbins
dn: uid=bcubbins,ou=People,dc=example,dc=com
mail: bcubbins@example.com
uid: bcubbins
facsimileTelephoneNumber: +1 234 567 8910
givenName: Bartholomew
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
sn: Cubbins
cn: Bartholomew Cubbins
userPassword: bcubbins
```

We know Directory Server uses a plug-in to check Yolanda's password during the bind. In this case, Directory Server must have used the XOR plug-in, because we saw that Yolanda's password was XOR encoded. If the whole process appears to work, we can conclude that our compare function works, too.

Index

F

H

I

L

M

N

O

P

R

S

U

V