Netscape Directory Server Programmer's Guide

Netscape Directory Server

Version 3.0


Recycled and Recyclable Paper

The Team:
Engineering:
Marketing:
Publications:
Quality Assurance:
Technical Support:

Version 1.0

Printed in USA
99 98 97 10 9 8 7 6 5 4 3 2 1

Netscape Communications Corporation 501 East Middlefield Road, Mountain View, CA 94043

Contents

Getting Started

This book describes writing server plug-ins to customize and extended the capabilities of the Netscape Directory Server.

Sections:
- What's New in This Version
- What You Should Already Know
- Where to Find Directory Server Information
- Document Conventions
- Sample code

What's New in This Version

This section briefly summarizes the changes to the API (see "Changes to the API" on page xi) and to this online manual (see "Changes to the Manual" on page xiv).

Changes to the API

The following changes were made to the API between the Netscape Directory Server 3.x release and the 4.0 release of the server:

- The syntax for the `plugin` directive in the `slapd.*.conf` files has changed in the 4.0 release. See "Editing the slapd.conf File" on page 40 for details.
- New return codes have been defined for database functions. See "Writing Database Plug-In Functions" on page 82 for more information.
- The `slapi_entry_dup()` function passes in a `const Slapi_Entry *` argument (rather than an `Slapi_Entry *` argument).
- You can use the following new API functions when working with values of attributes in entries:

- `slapi_entry_attr_delete()` (see page 272)
- `slapi_entry_attr_get_charptr()` (see page 274)
- `slapi_entry_attr_get_int()` (see page 275)
- `slapi_entry_attr_hasvalue()` (see page 276)
- `slapi_entry_attr_replace()` (see page 278)
- `slapi_entry_attr_set_charptr()` (see page 279)
- `slapi_entry_attr_set_long()` (see page 280)

- You can use the `Slapi_MatchingRuleEntry` structure (see page 213) and the following new API functions when working with matching rule plug-ins:
  - `slapi_matchingrule_free()` (see page 310)
  - `slapi_matchingrule_get()` (see page 311)
  - `slapi_matchingrule_new()` (see page 312)
  - `slapi_matchingrule_register()` (see page 313)
  - `slapi_matchingrule_set()` (see page 314)
  - `slapi_matchingrule_unregister()` (see page 315)
- The `slapi_dn_issuffix()` function and the `slapi_dn_isparent()` function both pass `const` arguments now.
- You can use the following new API functions for locking and working with condition variables:
  - `slapi_new_mutex()` (see page 323)
  - `slapi_lock_mutex()` (see page 307)
  - `slapi_unlock_mutex()` (see page 348)
  - `slapi_destroy_mutex()` (see page 258)
  - `slapi_new_condvar()` (see page 322)

- `slapi_wait_condvar()` (see page 351)
- `slapi_notify_condvar()` (see page 324)
- `slapi_destroy_condvar()` (see page 257)

The `Slapi_Mutex` structure (see page 214) represents a mutex and the `Slapi_CondVar` structure represents a condition variable (see page 210).

- If you need to connect to another LDAP server from your plug-in, you can use the following API functions to initialize the LDAP session and to end the LDAP session:
  - `slapi_ldap_init()` (see page 304)
  - `slapi_ldap_unbind()` (see page 306)
- The `SLAPI_BE_MONITORDN` parameter (used in the Netscape Directory Server 3.x releases to specify the DN for monitoring the back-end) is no longer available for use in the Netscape Directory Server 4.0 release.
- If you are integrating your own back-end database with the server, you can define and register a function for generating indexes for the database. See "Generating Indexes for the Database" on page 118 for details.
- If you are writing a database import function for your own back-end, you should check the following new parameters that the front-end can specify:
  - The `SLAPI_LDIF2DB_NOATTRINDEXES` parameter specifies whether or not indexes for the database should be generated automatically when the database is created.
  - The `SLAPI_LDIF2DB_INCLUDE` parameter specifies the suffixes and DNs of entries in the LDIF file that should be included in the database.
  - The `SLAPI_LDIF2DB_EXCLUDE` parameter specifies the suffixes and DNs of entries in the LDIF file that should be excluded from the database.

  For more information, see the section "Importing an LDIF File into the Database" on page 114.

- If you are writing your own back-end database functions, you can use the `SLAPI_OPERATION_NOTES` parameter to specify notes that you want appended to access log entries when the server sends a result or entry to an LDAP client.

  For more information, see the section “Adding Notes to Access Log Entries” on page 50.

Changes to the Manual

The following changes were made between the 3.0 version and the 4.0 release of the manual:

- Added Chapter 12, “Writing Matching Rule Plug-Ins”.
- Corrected description of the database search functions and added more material under “Processing an LDAP Search Operation” on page 101.

What You Should Already Know

This book assumes you have this basic background:

- A general understanding of the Internet and the World Wide Web (WWW).
- A general understanding of the Lightweight Directory Access Protocol (LDAP) and the Netscape Directory Server. This book does not duplicate basic information on server administration or LDAP concepts.
- Programming experience in C or C++.
- If you’re going to write a database plug-in, working knowledge of your database and its programming interfaces.

Where to Find Directory Server Information

Netscape provides a number of manuals documenting the Directory Server and the LDAP protocol:

- For information on installing, configuring, and managing the Netscape Directory Server, see the *Netscape Directory Server Administrator's Guide* and *Managing Netscape Servers*.
- For information on writing LDAP clients with the LDAP C and Java API, see the *Netscape LDAP C SDK Programmer's Guide and Reference* and the *Netscape LDAP Java SDK Programmer's Guide*.
- This book (the *Netscape Directory Server Programmer's Guide and Reference*) explains how to write your own server plug-ins to customize the Netscape Directory Server. You can write plug-ins that validate data before the data is stored in the directory, that notify users when data has changed, or that replace the standard database in the Directory Server with your own database.

Netscape web sites contain helpful information for working with the Netscape Directory Server. Some URLs of particular interest include:

- `http://help.netscape.com/`

  This is Netscape's technical support site. You can read the FAQ on the Netscape Directory Server or search the knowledge base for tips and information.

- `http://developer.netscape.com/one/directory/index.html`

  This is LDAP Directory Developer Central. From this page, you can access the documentation online, download the client SDKs, read the release notes, and find support information and additional resources.

Document Conventions

The Netscape Directory Server runs on Windows NT and a number of different UNIX platforms; the information here applies to all versions. File and directory paths are given in Windows format (with backslashes separating directory names). For Unix versions, the directory paths are the same, except that you use slashes instead of backslashes to separate directories.

This book uses Uniform Resource Locators (URLs) of the form

`http://`*`server.domain/path/file`*`.html`

In these URLs, *server* represents the name of the server on which you run your application, such as `research1` or `www`; *domain* represents your Internet domain name, such as `netscape.com` or `uiuc.edu`; *path* represents the directory structure on the server; and *`file`*`.html` represents an individual filename. In general, items in italics in URLs are variables and items in normal monospace font are literals. If your server has Secure Sockets Layer (SSL) enabled, you would use `https` instead of `http` in the URL.

This book uses the following font conventions:

- `The monospace font` is used for sample code and code listings, API and language elements (such as function names and class names), filenames, pathnames, directory names, HTML tags, and any text that must be typed on the screen. (*`Monospace italic font`* is used for placeholders embedded in code.)
- *Italic type* is used for book titles, emphasis, variables and placeholders, and words used in the literal sense.
- **Boldface type** is used for glossary terms.

Sample code

This book contains sample C code. This code was tested with Netscape Directory Server 3.0 on a Solaris machine.

1

Introduction to Directory Server Plug-Ins

Chapter 4 Quick Start

This chapter provides a few simple examples of server plug-ins that you can compile and implement. Essentially, these examples demonstrate the different types of data available to your plug-in.

Chapter

1

Understanding Server Plug-Ins

This chapter introduces the concept of a server plug-in and discusses the different types of plug-ins that you can write.

- What Are Server Plug-Ins?
- How Server Plug-Ins Work
- Types of Server Plug-Ins

What Are Server Plug-Ins?

If you want to extend the capabilities of the Netscape Directory Server, you can write your own server plug-in. A **server plug-in** is a shared object or library (or a dynamic link library on Windows NT) containing your own functions.

You can write your own plug-in functions to extend the functionality of the directory server in the following ways:

- You can validate data before the server performs an LDAP operation on the data.
- You can perform some action (that you define) after the server successfully completes an LDAP operation. (For example, you can send mail after an operation successfully completes.)

- You can define extended operations (which are part of the LDAP v3 protocol).
- You can provide alternate matching rules when comparing certain attribute values.
- You can set up the server to use your own type of database for storing data.

On startup, the directory server loads your library and calls your functions during the course of processing various LDAP requests.

How Server Plug-Ins Work

Internally, the directory server has hooks that allow you to register your own functions to be called when specific events occur. For example, the directory server calls a registered plug-in function:

- before perfoming an LDAP operation (for example, before an entry is added to the directory)
- when adding, modifying, removing, renaming, or searching for entries in the database
- after performing an LDAP operation (for example, after an entry is added to the directory)
- before writing an entry to the database
- after reading an entry from the database
- when processing an extended operation
- when indexing an attribute and when filtering search result candidates based on an attribute

When you register your plug-in, you specify the type of function and the type of plug-in, which both indicate when the function is called (see “Types of Server Plug-Ins” on page 23 for a listing).

How the Server Calls Plug-In Functions

At specific events (such as before an LDAP add operation is peformed), the server calls any plug-in functions registered for that event. For example, before performing an LDAP add operation, the server calls any functions registered as pre-operation add functions.

In most cases, when the server calls your function, it passes a **parameter block** to your function. This parameter block contains data relevant to the operation. In your server plug-in function, you can access and modify data in this parameter block.

For example, when the directory server receives an LDAP add request, the server does the following:

1. Parses the request and retrieves the new DN and the entry to be added.
2. Places pointers to the DN and entry in the parameter block.
3. Calls any registered pre-operation add functions, passing the parameter block to these functions.
4. Calls the registered database add function (which is responsible for adding the entry to the directory database), passing it the parameter block.
5. Calls any registered post-operation add functions, passing the parameter block to these functions.

What Plug-Ins Return to the Server

If you are writing a function that is invoked before an LDAP operation is performed, you can also prevent the operation from being performed. For example, you can write a function that validates data before a new entry is added to the directory. If the data is not valid, you can prevent the LDAP add operation from occurring and return an error message to the LDAP client.

In some situations, you can also set the data that is used by the server to perform an operation. For example, in a pre-operation add function, you can change an entry before it is added to the directory.

How Database Plug-Ins Work

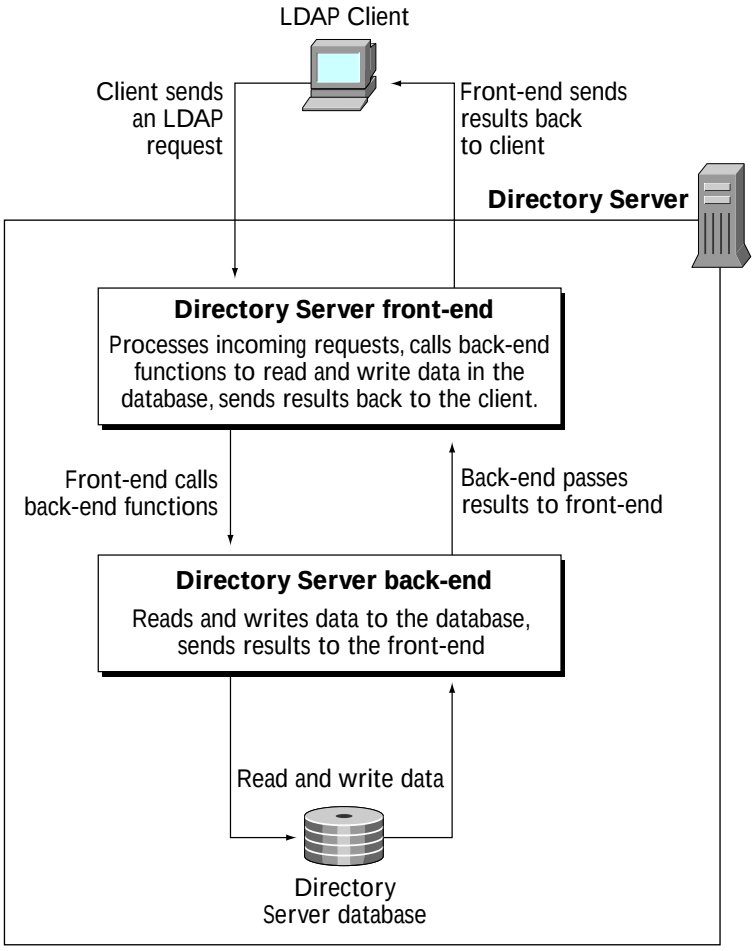
Internally, the Netscape Directory Server consists of two major subsections:

- The front-end receives LDAP requests from clients and processes the requests. When processing requests, the front-end calls functions in the back-end to read and write data. The front-end then sends the results back to the client.
- The back-end reads and writes data to the database containing the directory entries. The back-end abstracts the database from the front-end. The data stored in a back-end is identified by the suffixes that the back-end supports. For example, a back-end that supports the “o=Airius.com” suffix contains directory entries that end with that suffix.

If you want to integrate the directory server with your own database, you just need to add your own back-end to the directory server, configure the back-end to handle the suffix for your directory data, and register your own functions for reading and writing data in the back-end.

Figure 1.1 illustrates the directory server architecture.

Figure 1.1 Directory server architecture



Types of Server Plug-Ins

You can write the following types of server plug-ins for the directory server:

- **Pre-operation/data validation.** The server calls pre-operation/data validation plug-in functions before performing an LDAP operation. The main purpose of this type of plug-in is to validate data before the data can be used in an operation.

  For example, you can write a plug-in that checks each new entry before adding them to the directory.

- **Post-operation/data notification.** The server calls post-operation/data notification plug-in functions after performing an LDAP operation. The main purpose of this type of plug-in is to invoke a function when a particular operation is executed.

  For example, you can write a plug-in that sends email to users if their entries are modified.

- **Database**. You can define a database plug-in to integrate your own database as a back-end to the server. Your plug-in translates requests from the directory server front-end into calls to your database client library. If the directory server front-end requests to add an entry to the directory, your plug-in calls functions in your database client library to insert the data into your database.

  For example, you can write a plug-in that integrates a relational database with the directory server.

- **Entry storage and entry fetch.** The server calls entry storage plug-in functions right before writing data to the default database back-end. The server calls entry fetch plug-in functions after retrieving an entry from the default database back-end.

  Note that these plug-ins only apply to the default back-end database (`ldbm`). If you have defined your own database plug-in, you can define the way that data is written to and read from the database in your database plug-in functions.

  For example, you can create an entry storage plug-in that encrypts an entry before it is saved to the database and an entry fetch plug-in that decrypts an entry after it is read from the database.

- **Extended operation.** The server calls extended operation plug-in functions when the client requests the operation by OID.

- **Syntax.** The server calls syntax plug-in functions when getting a list of possible candidates for a search or when adding or deleting values from certain attribute indexes. Syntax plug-in functions also define the

comparison operations used in searches (for example, a syntax plug-in function might define how the “equals” comparison works for case-insensitive strings).

- **Matching rule.** The server calls matching rule plug-in functions when the client sends a search with an extensible matching search filter.

  You can also write matching rule plug-in functions that the server calls when indexing attributes for the default back-end database. (Note that these are called only if you are using the default back-end database, not if you are using your own database plug-in.)

Figure 1.2 illustrates how some of these different plug-in types fit into the directory server architecture.

Figure 1.2 Architecture of the directory server and Server Plug-Ins

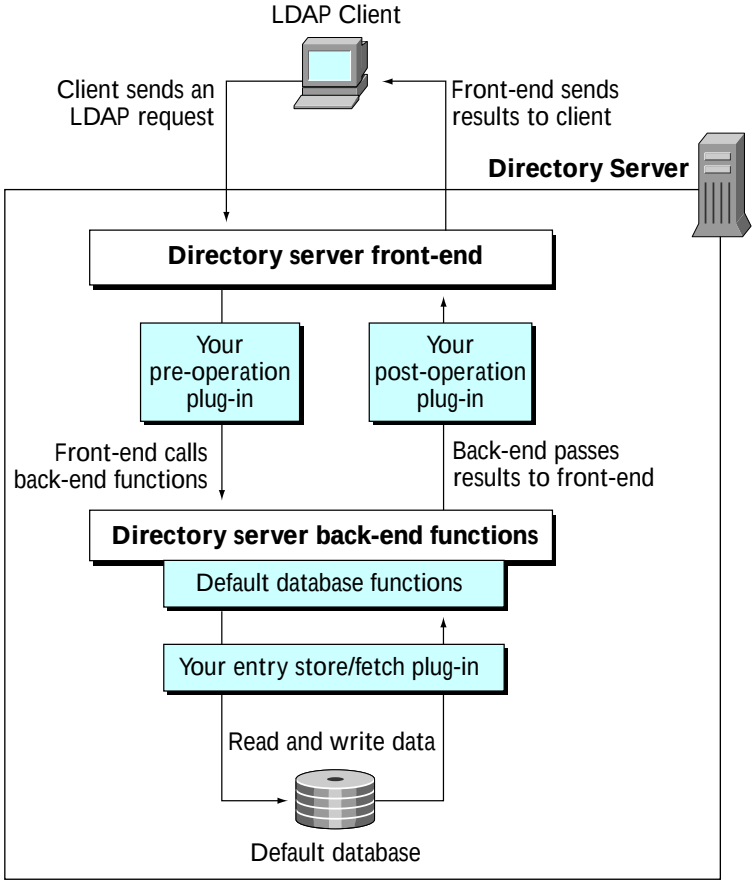
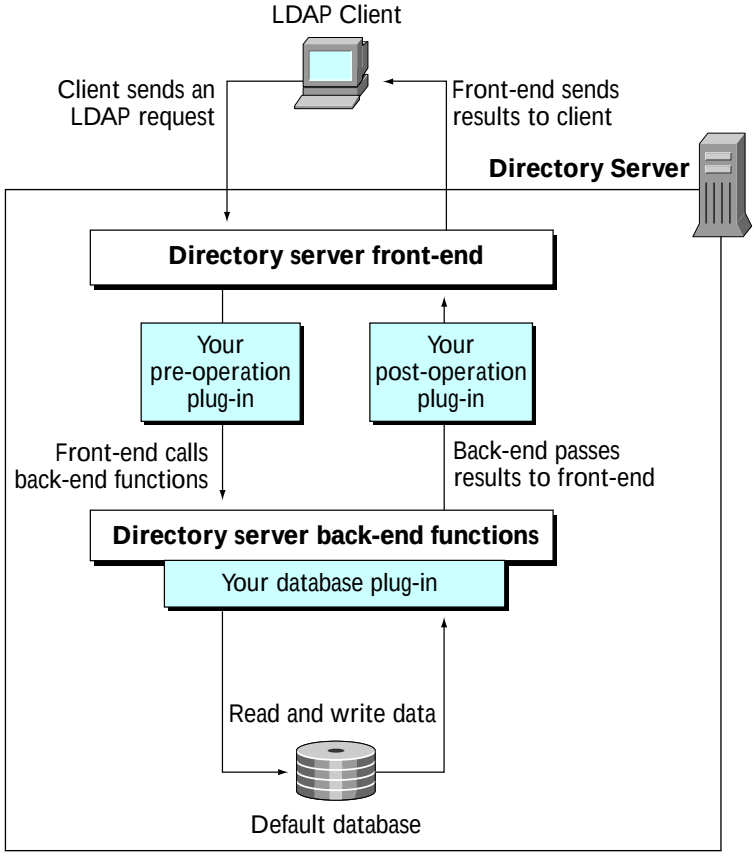


Figure 1.3 illustrates how you can replace the standard database.

Figure 1.3 Using a Plug-In to Replace the Standard Database



Chapter

2

Writing and Compiling Plug-Ins

This chapter explains how to write, compile, and install server plug-ins for the Directory Server.

- Writing a Plug-In Function
- Writing an Initialization Function
- Compiling a Server Plug-In
- Configuring the Server

Writing a Plug-In Function

This section provides general information on writing a server plug-in, including the following:

- Including the API Header File
- Working with Parameter Blocks
- Calling Front-End Functions
- Specifying Return Values

For additional information on specific plug-in types, see the following sections:

- Chapter 5, “Writing Database Plug-Ins”
- Chapter 6, “Writing Pre/Post-Operation Plug-Ins”
- Chapter 10, “Writing Entry Store/Fetch Plug-Ins”
- Chapter 11, “Writing Extended Operation Plug-Ins”

Including the API Header File

Make sure to include the `slapi-plugin.h` header file, which is located in the `<server_root>/plugins/slapd/slapi/include` directory on UNIX and the `<server_root>\plugins\slapd\slapi\include` directory on Windows NT:

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

Make sure that the `ldap.h` and `lber.h` header files (which are included by `slapi-plugin.h`) are in your include path.

Working with Parameter Blocks

The following types of plug-ins pass a parameter block as an argument:

- pre-operation
- post-operation
- database plug-in functions
- matching rule functions for indexing
- factory function for matching rule index functions
- factory function for matching rule filter functions

When invoking these types of plug-in functions, the Directory Server passes a single argument of type `Slapi_PBlock` when calling the plug-in function. Your plug-in function should have the following prototype:

```
int myfunction( Slapi_PBlock pb );
```

`pb` is a parameter block that contains parameters pertaining to the operation or function.

For example, the parameter block for the add operation contains the target DN and the entry to be added, while the parameter block for the bind operation contains the DN of the user, the authentication method, and the user's credentials.

Note the following:

- You can get data from the parameter block in your plug-in function (see "Getting Data from the Parameter Block" on page 31 for details). You can also set values in the parameter block, which the Directory Server still uses to complete the operation (see "Setting Data in the Parameter Block" on page 32).
- Some of the data that you can retrieve from the parameter block includes entries, attributes, distinguished names (DNs), and search filters. If you want to manipulate these, you can call front-end API functions (see "Calling Front-End Functions" on page 33).

  For example, you can call some of the front-end API functions to verify that an entry complies with the schema or to split up a search filter into its components.

Getting Data from the Parameter Block

When the Directory Server calls your plug-in function, it passes any relevant data to your function in a parameter block (`Slapi_PBlock`). To access the data in a parameter block, call the `slapi_pblock_get()` function (see "slapi_pblock_get()" on page 327 for details).

Important In many cases, `slapi_pblock_get()` returns a pointer to the actual data. When you call your own functions to modify the value retrieved by `slapi_pblock_get()`, you are modifying the actual data in the parameter block, not a copy of the data.

In the following example, the `searchdn_preop_search()` function gets the DN of the base DN for the LDAP search operation, normalizes the DN, converts all characters to lowercase, and writes the converted DN to the error log. Note that the actual DN (not a copy of it) is normalized and converted to lowercase.

```
#include <slapi-plugin.h>

...

int

searchdn_preop_search( Slapi_PBlock *pb )
```

```
{
   char *dn;

   /* Indicate the point when the plug-in starts executing */

   slapi_log_error( SLAPI_LOG_PLUGIN, "searchdn_preop_search",
      "*** PREOPERATION SEARCH PLUGIN ***\n");

   /* Get the base DN of the search from the parameter block. */

   slapi_pblock_get( pb, SLAPI_SEARCH_TARGET, &dn );

   /* Normalize the DN (the actual DN, not a copy of it)
    and convert it to lowercase */

   slapi_dn_normalize_case( dn );

   /* Log the normalized DN */

   slapi_log_error( SLAPI_LOG_PLUGIN, "searchdn_preop_search",
      "Normalized DN: %s\n", dn );

   return( 0 );

}
```

`SLAPI_SEARCH_TARGET` identifies the parameter in the parameter block that contains the base DN of the search. For a complete listing of the parameter IDs, see Chapter 15, “Parameter Reference”.

Setting Data in the Parameter Block

You can also change the value of a parameter in the parameter block by calling the `slapi_pblock_set()` function (see “slapi_pblock_set()” on page 329 for details). For example, you call can `slapi_pblock_set()` to change the value of the `SLAPI_PRIVATE` parameter, which stores private data for this plug-in.

In the following example, the `ldif_back_init()` function sets the value of the `SLAPI_PRIVATE` parameter to the context of the database:

```
#include <slapi-plugin.h>

...

int

ldif_back_init( Slapi_PBlock *pb )

{

   LDIF *db; /* context of the database */

   ...
```

```
   /* Allocate space for the database context, which contains
   information about the database and a pointer to the list of entries.
*/
   if ( slapi_pblock_set( pb, SLAPI_PRIVATE, (void *) db ) == -1 ) {
      slapi_log_error( SLAPI_LOG_PLUGIN, "ldif_back_init", "Unable to
                  store database information\n" );
   }
   ...
}
```

In this example, the `slapi_log_error()` function is used to indicate that an error occurred.

`SLAPI_PRIVATE` identifies the parameter in the parameter block that contains private data for use in the database functions. For a complete listing of the parameter IDs, see Chapter 15, “Parameter Reference”.

Calling Front-End Functions

The types of data that you can get from a parameter block include entries, attributes, distinguished names, and search filters. If you want to manipulate these, you can call the front-end API functions provided with the Directory Server. For example, you can call some of the front-end API functions to:

- write messages to the error log
- get the attributes of an entry
- get or set the DN of an entry
- add or delete the values of an attribute
- determine the OID of an attribute
- determine the type of a search filter

For more information on the front-end API, see Chapter 3, “Calling the Front-End API Functions” and Chapter 14, “Function Reference”.

Specifying Return Values

If the function is successful, it should return 0. If it is not successful, it should return a non-zero value, and you should call the front-end API function `slapi_log_error()` to log a message describing the problem (see “Logging Messages” on page 50).

In some cases, you may need to send an LDAP result back to the client. For example, if you are writing a pre-operation bind function and an error occurs, you should return a non-zero value, log an error message, and send the appropriate LDAP result code back to the client. (For information on which result code to send, see the chapter documenting the type of plug-in that you are writing.)

Writing an Initialization Function

Next, you need to write an initialization function for your server plug-in. Your initialization function should do the following:

6. Specify the plug-in version.
7. Specify information about the plug-in.
8. Register your plug-in functions.
9. Return a value to the Directory Server.

Your initialization function should have the following prototype:

```
int my_init_function( Slapi_PBlock pb );
```

Directory Server passes a single argument of type `Slapi_PBlock` when calling your initialization function.

If you are writing an initialization function for a plug-in on Windows NT, make sure to export the initialization functions. Use the following prototype:

```
__declspec( dllexport ) int my_init_function( Slapi_PBlock pb );
```

and specify the initialization function names in a `.def` file.

Note You do not need to export all plug-in functions. Any plug-in functions that you specify in the pblock (see “Registering Your Plug-In Functions” on page 36) do not need to be exported. You only need to export the initialization functions and any functions that are not registered by initialization functions (such as the entry store and entry fetch plug-in functions, which are described in Chapter 10, “Writing Entry Store/Fetch Plug-Ins”).

Specifying the Plug-In Version

You need to specify the version of your plug-in so that the Directory Server can determine whether or not it supports the plug-in.

To specify the plug-in version, call the `slapi_pblock_set()` function, and set the `SLAPI_PLUGIN_VERSION` parameter to the version number. For example:

```
slapi_pblock_set(pb, SLAPI_PLUGIN_VERSION, SLAPI_PLUGIN_VERSION_02);
```

Plug-in version 1 (`SLAPI_PLUGIN_VERSION_01`) is compatible with versions 3.x and 4.x of the Netscape Directory Server. Plug-in version 2 (`SLAPI_PLUGIN_VERSION_02`) is compatible with version 4.x of the Netscape Directory Server.

Specifying Information about the Plug-In

You can specify information about your plug-in (such as a description of the plug-in) in an `Slapi_PluginDesc` structure. Call the `slapi_pblock_set()` function, and set the `SLAPI_PLUGIN_DESCRIPTION` parameter to this structure. For example:

```
/* Specify information about the plug-in */

Slapi_PluginDesc mypdesc = { "test-plugin", "Airius.com", "0.5", "sample
pre-operation plugin" };

...

/* Set this information in the parameter block */

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

In this example, the following information about the server plug-in is specified:

- The unique identifier for the server plug-in is `test-plugin`.

- The vendor of the server plug-in is `Airius.com`.
- The version of the server plug-in is `0.5`. Note that this version is intended to be used for your tracking purposes. (As you make changes to your plug-in, you can change this version number.)

  This is not the same version as the version specified by the `SLAPI_PLUGIN_VERSION` parameter (see "Specifying the Plug-In Version" on page 35), which is used by the Netscape Directory Server to determine if it supports your server plug-in.
- The description of the server plug-in is `sample pre-operation plug-in`.

Registering Your Plug-In Functions

The way in which you register your plug-in function depends on the type of the plug-in function.

For some plug-in types, you do not need to register the plug-in function from within the initialization function. (For example, you register entry store and entry fetch plug-ins by specifying the function names in the `plugin` directive in the `slapd.conf` configuration file.)

For other plug-in types (such as pre-operation plug-ins, post-operation plug-ins, and database plug-ins), the Directory Server gets the pointer to your function from a parameter in the parameter block. You register plug-in functions by calling the `slapi_pblock_set()` function to specify the name of your plug-in function.

These parameters that you can use to specify plug-in functions are listed in "Parameters for Registering Plug-In Functions" on page 354. For example, if you want to register `searchdn_preop_search()` as a pre-operation search function, include the following code in your initialization function:

```
slapi_pblock_set( pb, SLAPI_PLUGIN_PRE_SEARCH_FN,
   (void *) searchdn_preop_search )
```

`SLAPI_PLUGIN_PRE_SEARCH_FN` is the parameter that specifies the pre-operation plug-in function for the LDAP search operation.

Important If you do not register your plug-in functions, the Directory Server will not call your plug-in functions. Make sure to register your plug-in functions.

Note that you can register more than one plug-in function in your initialization function. (In other words, you do not need to define an initialization function for each plug-in function. You should, however, define a different initialization function for each type of plug-in.)

Returning a Value to the Directory Server

If the initialization function is successful, it should return 0. If an error occurs, it should return -1. If the initialization function returns -1, the Directory Server will exit.

Example of an Initialization Function

The following is an example of an initialization function that registers a pre-operation plug-in function `searchdn_preop_search()`. After the initialization function is done, the server will call `searchdn_preop_search()` before each LDAP search operation is executed.

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

...

#ifdef _WIN32

__declspec(dllexport)

#endif

searchdn_preop_init( Slapi_PBlock *pb )

{

   /* Specify the version of the plug-in (set this to "01") */

   if ( slapi_pblock_set( pb, SLAPI_PLUGIN_VERSION,
                          SLAPI_PLUGIN_VERSION_01 ) != 0 ||

   /* Set up the server to call searchdn_preop_search() before
      each LDAP search operation. */

   slapi_pblock_set( pb, SLAPI_PLUGIN_PRE_SEARCH_FN,
                     (void *) searchdn_preop_search ) !=0 ) {

      /* If a problem occurs, log an error message and
         return -1. */
```

```
        slapi_log_error( SLAPI_LOG_PLUGIN, "searchdn_preop_init",
           "Error registering function.\n" );

        return( -1 );

    }

    /* If the plug-in was successfully registered, log a
       message and return 0. */

    slapi_log_error( SLAPI_LOG_PLUGIN, "searchdn_preop_init",
       "Plug-in successfully registered.\n" );

    return( 0 );

}
```

Compiling a Server Plug-In

Keep in mind the following tips when compiling your server plug-in:

- You need to compile and link your code as a shared object or library on UNIX or as a dynamic link library (DLL) on Windows NT. For example, on Solaris, specify `ld -G <objects> -o <shared_object>` as your link command.

  Some compilers require that you provide special flags when compiling code to be used in a DLL or shared object. For example, on Solaris, you must specify the `-Kpic` or `-KPIC` flag (which indicates that you want to generate position-independent code). Consult the documentation for your compiler and linker for details.

- Make sure that the `<server_root>/plugins/slapd/slapi/include` directory is in your include path.

- Link to the `libslapd.lib` import library, which is located in the `<server_root>\plugins\slapd\slapi\lib` directory.

- If you want, you can compile all plug-in functions in a single library. Although you can include different types of plug-in functions in the same library, you need to write separate initialization functions for each type of function. (The next section, “Configuring the Server” on page 39, explains how you need to specify each initialization function in a directive for a specific type of plug-in.)

The following is a sample makefile for Solaris. The example assumes that the source files are located in the <server_root>/plugins/slapd/slapi/examples directory. The server plug-in API header files are located in the relative path ../include.

```
# SOLARIS Makefile for Directory Server plug-in examples

#

CC = cc

LD = ld

INCLUDE_FLAGS = -I../include

CFLAGS = $(INCLUDE_FLAGS) -D_REENTRANT -KPIC

LDFLAGS = -G

OBJS = testsaslbind.o testextendedop.o testpreop.o testpostop.o
testentry.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
```

Configuring the Server

After you compile your server plug-in, you need to configure the Directory Server to load your plug-in. The rest of this section explains the steps you need to take to configure the server.

- Editing the slapd.conf File
- Important Note on Manually Editing the Configuration File
- Passing Extra Arguments to the Plug-Ins
- Setting the Log Level of the Server

Editing the slapd.conf File

To configure the server to load your plug-ins, you need to edit the `slapd.conf` server configuration file. (Before you edit this file, make sure to read the caveats in the section “Important Note on Manually Editing the Configuration File” on page 44.)

Open the `slapd.conf` file in a text editor, and add `plugin` directive to specify the plug-ins that you want to load.

- For the Netscape Directory Server 3.x releases, the `plugin` directive has the following syntax:

```
plugin <plugin_type> <lib_path> <init_function> \
   [<arguments>]
```

- For the Netscape Directory Server 4.x releases, the `plugin` directive has the following syntax:

```
plugin <plugin_type> on|off <name> <lib_path> <init_function> \
   [<arguments>]
```

Note that the directive should not contain any line breaks.

The argument for this directive are explained below:

- `<plugin_type>` identifies the type of the plug-in. See the section “Summary of plugin Directives” on page 42 for a listing of the different plug-in types you can specify in this directive.
- `on|off` specifies whether or not the plug-in is active by default. In the Netscape Directory Server 4.x, you can turn on or turn off plug-ins from the console.
- `<name>` identifies the plug-in. In the Netscape Directory Server 4.x, this is the value of the “cn” attribute in the DN that identifies this plug-in (for example, “dn:cn=name,cn=plugins,cn=config”).
- `<lib_path>` is the absolute path to the library that implements the plug-in.
- `<init_function>` is the name of the initialization function that the server should call to register the plug-in. (See “Writing an Initialization Function” on page 34 for more information.)

- `<arguments>` are any additional argument that you want passed to the initialization function. (See “Passing Extra Arguments to the Plug-Ins” on page 45 for details.)

For example, the following directive configures the server to load the library `mydb.so` and call the initialization function `my_init_fn()` on startup.

- In the Netscape Directory Server 3.x:

```
plugin database /usr/netscape/suitespot/lib/mydb.so my_init_fn
```

- In the Netscape Directory Server 4.x:

```
plugin database on mydbplugin /usr/netscape/suitespot/lib/mydb.so \
   my_init_fn
```

Again, note that the directive should not contain any line breaks.

Determining Where to Add the Directives

Each pre-operation, post-operation, and extended operation plug-in is associated with a back-end. Make sure that the `plugin` directive that registers the plug-in is in the database section for that back-end. (The `plugin` directive should be after the `database` directive and before the next `database` directive, if any.)

Directives registering entry store and entry fetch plug-ins must appear within the database section for the default `ldbm` database. (The `plugin` directive should be after the `database ldbm` directive and before the next `database` directive, if any.)

Specifying the Order of plugin Directives

Plug-ins of different types can be listed in any order. For example:

```
plugin database /usr/netscape/suitespot/lib/plugin.so db_init_fn
plugin preoperation /usr/netscape/suitespot/lib/plugin.so pr_init_fn
```

(In the example above, initialization functions and plug-in functions for both database and preoperation plug-ins have been defined in the same library.)

You can also load different plug-ins of the same type. Plug-ins of the same type are loaded and called in the order in which they are listed in the `slapd.conf` file. For example:

```
plugin preoperation /usr/netscape/suitespot/lib/plugin.so pre_fn_1
plugin preoperation /usr/netscape/suitespot/lib/plugin2.so pre_fn_2
```

In the example above for a given LDAP operation, the pre-operation plug-in function registered by `pre_fn_1()` are called before the pre-operation plug-in function registered by `pre_fn_2()`.

So if `pre_search_fn()` is registered by `pre_fn_1()` and `pre_search_fn2()` is registered by `pre_fn_2()`, the Directory Server calls `pre_search_fn()` before calling `pre_search_fn2()`. Note that if `pre_search_fn()` terminates the LDAP operation by returning a non-zero value, the server will not call `pre_search_fn2()`.

Summary of plugin Directives

The following table summarizes the different types of plug-ins that you can specify in the `slapd.conf` file.

Table 2.1 Directives in slapd.conf for specifying different plug-in types

Directive in slapd.conf	Plug-in Type	Description
`plugin database`	Database	Called by the server when interacting with the back-end database. Example of use: You can define functions that allow the server to store and retrieve data from your own database.
`plugin entryfetch`	Entry fetch	Called by the server after retrieving an entry from the default back-end database. (If you are writing your own database plug-in, you do not need to use this plug-in.) Example of use: If you encrypt data with an entry store plug-in function before saving the data to the database, you can define an entry fetch function that decrypts data after reading it from the database.

Table 2.1 Directives in slapd.conf for specifying different plug-in types

Directive in slapd.conf	Plug-in Type	Description
`plugin entrystore`	Entry store	Called by the server before saving an entry to the default back-end database. (If you are writing your own database plug-in, you do not need to use this plug-in.) Example of use: You can define an entry store function to encrypt data before saving the data to the database.
`plugin extendedop`	Extended operation	Called by the server when receiving a request for an extended operation from a client.
`plugin matchingrule`	Matching rule	Called by the server when receives a search request with an extensible matching search filter from a client. Also called by the server when indexing attributes for the default back-end database. (If you are using your own database plug-in, you do not need to use this type of plug-in for indexing attributes.)
`plugin postoperation`	Post-operation/data notification	Called by the server after performing an LDAP operation. Example of use: You can define a data notification function to send notification to the administrator if certain data has changed.

Table 2.1 Directives in slapd.conf for specifying different plug-in types

Directive in slapd.conf	Plug-in Type	Description
`plugin preoperation`	Pre-operation/data validation	Called by the server before performing an LDAP operation. Example of use: You can define a data validation function to check new entries before they are added to the directory.
`plugin syntax`	Syntax	Called by the server when getting a list of possible candidates for a search, when determining how to compare values in searches, and when adding or deleting values from certain attribute indexes. Example of use: You can define a function that specifies how the “equals” comparison works for case-insensitive strings)

Important Note on Manually Editing the Configuration File

In the administration server, you can apply either hand-edited changes to the configuration file or changes made through the server manager interface -- but not both.

For example, suppose you made some changes using the server manager interface and then hand-edited the configuration file. When you click the Apply button at the top of the server manager interface, the administration server will allow you to apply either the changes made through the interface or the hand-edited changes. You cannot apply both sets of changes.

For this reason, before you open the `slapd.conf` file in a text editor, you should always click the Apply button in the server manager interface and apply any changes made through the interface.

Likewise, after you are done editing the `slapd.conf` file, you should click the Apply button again to apply the manual changes you have just made.

Passing Extra Arguments to the Plug-Ins

You can specify additional arguments at the end of the `plugin` directive. For example:

```
plugin preoperation /usr/lib/myplugin.so my_init_fn arg1 arg2
```

From the initialization function and the plug-in functions, you can get these arguments by getting the following parameters from the parameter block:

- `SLAPI_PLUGIN_ARGC` is an `int` that specifies the number of additional arguments in the directive.
- `SLAPI_PLUGIN_ARGV` is an array of `char*` that specifies each additional argument in the directive.

The following example illustrates how you can pass directive arguments to the initialization function and to the plug-in function. In both functions in the example, the extra arguments are written to the error log.

```
/* Initialization function */

#ifdef _WIN32

__declspec(dllexport)

#endif

int

searchdn_preop_init( Slapi_PBlock *pb )

{

   int argc, cnt;

   char ***argv;

   /* Specify the version of the plug-in (set this to "01") */

   if ( slapi_pblock_set( pb, SLAPI_PLUGIN_VERSION,
                          SLAPI_PLUGIN_VERSION_01 ) != 0 ||

      /* Set up the server to call searchdn_preop_search() before
         each LDAP search operation. */

      slapi_pblock_set( pb, SLAPI_PLUGIN_PRE_SEARCH_FN,
                        (void *) searchdn_preop_search ) !=0 ||

      /* Get the number of additional arguments */

      slapi_pblock_get( pb, SLAPI_PLUGIN_ARGC, &argc ) != 0 ||

      /* Get the array of additional arguments */
```

```
      slapi_pblock_get( pb, SLAPI_PLUGIN_ARGV, &argv ) != 0 ) {

      /* If a problem occurs, log an error message and
         return -1. */

         slapi_log_error( SLAPI_LOG_PLUGIN, "searchdn_preop_init",
            "Error registering function.\n" );

         return( -1 );

   }

   /* If the plug-in was successfully registered, log a
      message and return 0. */

   slapi_log_error( SLAPI_LOG_PLUGIN, "searchdn_preop_init",
      "Plug-in successfully registered.\n" );

   /* Log a note indicating the number of additional arguments found. */

   slapi_log_error( SLAPI_LOG_PLUGIN, "searchdn_preop_init",
      "Read %d additional arguments from configuration file.\n", argc );

   /* Log each additional argument found. */

   for ( cnt = 0; cnt < argc; cnt++ ) {

      slapi_log_error( SLAPI_LOG_PLUGIN, "searchdn_preop_init",
      "-> Argument #%d: %s\n", cnt + 1, argv[cnt] );

   }

   return( 0 );

}


/* Pre-operation plug-in function; invoked before LDAP searches */

int

searchdn_preop_search( Slapi_PBlock *pb )

{

   int argc, cnt;

   char *dn;

   char ***argv;

   /* Indicate the point when the plug-in starts executing */

   slapi_log_error( SLAPI_LOG_PLUGIN, "searchdn_preop_search",
      "*** PREOPERATION SEARCH PLUGIN ***\n");

   /* Get the base DN of the search from the parameter block. */

   slapi_pblock_get( pb, SLAPI_SEARCH_TARGET, &dn );
```

```
   /* Normalize the DN and convert it to lowercase */

   slapi_dn_normalize_case( dn );

   /* Log the normalized DN */

   slapi_log_error( SLAPI_LOG_PLUGIN, "searchdn_preop_search",
      "Normalized DN: %s\n", dn );

   /* Get the number of additional arguments and the
      array of these arguments and write this to the error log. */

   if ( slapi_pblock_get( pb, SLAPI_PLUGIN_ARGC, &argc ) == 0 &&

   slapi_pblock_get( pb, SLAPI_PLUGIN_ARGV, &argv ) == 0 ) {

      slapi_log_error( SLAPI_LOG_PLUGIN, "searchdn_preop_search",
         "Read %d additional parameters from the configuration file.\n",
         argc );

      for ( cnt = 0; cnt < argc; cnt++ ) {

         slapi_log_error( SLAPI_LOG_PLUGIN, "searchdn_preop_search",
            "-> Argument #%d: %s\n", cnt + 1, argv[cnt] );

      }

   }

   return( 0 );

}
```

In the example above, if the `plugin` directive is:

```
   plugin preoperation /usr/ns/plugin.so searchdn_preop_search foo bar
```

the following lines are written to the error log on server startup:

```
[01/Oct/1997:08:41:47 -0700] searchdn_preop_search - Plug-in
successfully registered.

[01/Oct/1997:08:41:47 -0700] searchdn_preop_search - Read 2 additional
parameters from configuration file.

[01/Oct/1997:08:41:47 -0700] searchdn_preop_search - -> Argument #1: foo

[01/Oct/1997:08:41:47 -0700] searchdn_preop_search - -> Argument #2: bar
```

The arguments are also written to the error log when the plug-in function is invoked (before each LDAP search operation is executed).

Setting the Log Level of the Server

If your functions call the `slapi_log_error()` function to write messages to the error log (see “slapi_log_error()” on page 308), you need to make sure that the Directory Server is configured to log messages with the severity level you've specified. (The available severity levels are also documented under “slapi_log_error()” on page 308.)

For example, suppose you call this function in your plug-in:

```
slapi_log_error( SLAPI_LOG_PLUGIN, "searchdn_preop_init",
   "Plug-in successfully registered.\n" );
```

You need to make sure that the Directory Server is configured to log messages with the severity level `SLAPI_LOG_PLUGIN`.

To specify the level of severity at which the Directory Server logs messages, do the following:

10. In the server manager interface, click the Apply button to make sure you've applied any recent manual changes to the `slapd.conf` file.

    It is important that you save any manual edits now. See the section “Important Note on Manually Editing the Configuration File” on page 44 for more details.

11. If it is not already selected, click the Server Preferences tab.

12. Click the LDAP link to display the page on LDAP settings.

13. In the Log Level list box, select the severity levels that you have specified in your `slapi_log_error()` function calls.

    For example, if you use the `SLAPI_LOG_PLUGIN` severity level, choose Plug-Ins from the list. To select more than one level, press the Ctrl key down and click on the additional levels.

14. Click OK.

15. Click the Apply button at the top of the server manager interface.

    Make sure that you apply your changes.

Chapter

3

Calling the Front-End API Functions

This chapter explains how to use the Netscape Directory Server's front-end API functions to accomplish various tasks. You can call these functions in your plug-in to interact with the client (for example, send results or result codes), log messages, and work with entries, attributes, and filters.

In general, the Directory Server includes front-end API functions that allow you to do the following:

- Logging Messages
- Adding Notes to Access Log Entries
- Sending Data to the Client
- Determining if an Operation Was Abandoned
- Working with Entries and Attributes
- Working with Search Filters
- Working with Distinguished Names
- Checking Passwords

The front-end functions are declared in the `slapi-plugin.h` header file.

Logging Messages

To write an error message to the error log, call the `slapi_log_error()` function (for details, see “slapi_log_error()” on page 308). For example, the following function call generates a message in the error log:

```
slapi_log_error( SLAPI_LOG_PLUGIN, "searchdn_preop_search",
   "*** PREOPERATION SEARCH PLUGIN ***\n");
```

This function generates a message similar to the following:

```
[01/Oct/1997:02:24:18 -0700] searchdn_preop_search - *** PREOPERATION
SEARCH PLUGIN ***
```

Make sure that the Directory Server is configured to log messages that have the severity that you specify (for example, `SLAPI_LOG_PLUGIN`). For more information, see “Setting the Log Level of the Server” on page 48.

Adding Notes to Access Log Entries

This feature is available in the Netscape Directory Server 4.0 release and is not available in earlier releases.

When the default back-end database processes a search operation, it attempts to use indexes to narrow down the list of candidates matching the search criteria. If the back-end is unable to use indexes to narrow down the list, it appends the following string to the access log entry for the search:

```
notes="U"
```

If you are writing your own back-end database search function, you can append this note to access log entries by setting the `SLAPI_OPERATION_NOTES` parameter to the flag `SLAPI_OP_NOTE_UNINDEXED`.

This parameter identifies any notes that need to be appended to access log entries. At the current time, `SLAPI_OP_NOTE_UNINDEXED` is the only value that you can set for this parameter. In future releases, additional flags will be defined. You will be able to bitwise OR combinations of flags to specify different combinations of notes.

The server appends these notes and writes out the access log entries whenever sending a result or search entry back to the client.

Sending Data to the Client

When writing the back-end database functions, you may want to communicate some information directly back to the client:

- If you need to send a result code back to the client (for example, to report an error or a successful result to an LDAP operation), call the `slapi_send_ldap_result()` function (see “slapi_send_ldap_result()” on page 341 for details).

  Note that you should only send one result per operation back to the client.

- If you are fulfilling a search request and need to send matching entries back to the client, call the `slapi_send_ldap_search_entry()` function for each entry (see “slapi_send_ldap_search_entry()” on page 343 for details).

- If you need to refer the LDAP request to a different LDAP server, call the `slapi_send_ldap_referral()` function (see “slapi_send_ldap_referral()” on page 339 for details).

For example, the following section of code sends an `LDAP_SUCCESS` status code back to the client.

```
slapi_send_ldap_result( pb, LDAP_SUCCESS, NULL, "The operation was
processed successfully.\n", 0, NULL );
```

Determining if an Operation Was Abandoned

At any point in time, the client can choose to abandon an LDAP operation. When you write your database functions, keep in mind that you should periodically check if the operation has been abandoned.

To determine if an operation has been abandoned, call the `slapi_op_abandoned()` function (see “slapi_op_abandoned()” on page 325 for details). For example:

```
if ( slapi_op_abandoned( pb ) ) {
   slapi_log_error( SLAPI_LOG_PLUGIN, "my_function", "The operation was
abandoned.\n" );
   return 1;
}
```

Working with Entries and Attributes

In certain situations, you will exchange directory entries with the front-end or the client. For example, when the front-end calls your add function, the front-end passes it an entry in the parameter block. When you perform a search operation, you return each matching search entry to the client.

If you need to manipulate these entries, you can call the following front-end routines:

Table 3.1 Front-end functions for manipulating entries and attributes

To do this:	Call this function
Allocate memory for a new entry	`slapi_entry_alloc()`
Copy an entry	`slapi_entry_dup()`
Free an unused entry from memory	`slapi_entry_free()`
Convert an entry to a string representation (in LDIF format) and vice versa	`slapi_entry2str(), slapi_str2entry()`
Get or set the DN for an entry	`slapi_entry_get_dn(), slapi_entry_set_dn()`
Verify that an entry complies with the schema	`slapi_entry_schema_check()`
Get the attributes of an entry	`slapi_entry_first_attr(), slapi_entry_next_attr()`
Find the values for a specified attribute	`slapi_entry_attr_find()`
Merge an attribute and its values into an entry	`slapi_entry_attr_merge()`
Add values to an attribute in an entry	`slapi_entry_add_values()`
Delete values from an attribute in an entry	`slapi_entry_delete_values()`

These functions are described in more detail in the next sections.

Creating a New Entry

In some situations, you may want to create a new entry.

- To allocate memory for an entry, call the `slapi_entry_alloc()` function (see “slapi_entry_alloc()” on page 271 for details).

  This function returns a pointer to a new entry of the opaque datatype `Slapi_Entry`. You can call other front-end functions to set the DN and attributes of this entry.

- To make a copy of an entry, call the `slapi_entry_dup()` function (see “slapi_entry_dup()” on page 282 for details).

  This function returns a pointer to a new entry of the datatype `Slapi_Entry` that contains the copied data.

When you are no longer using the entry, you should free it from memory by calling the `slapi_entry_free()` function (see “slapi_entry_free()” on page 284 for details).

Converting Between Entries and Strings

Entries can be stored in LDIF files. When stored in these files, entries are converted into a string representation. The following format is the LDIF string representation for a directory entry:

```
dn:[:] <dn>\n
[<attr>:[:] <value>\n]
[<attr>:[:] <value>\n]
[<space><continuedvalue>\n]*
...
```

If you want to continue the specification of a value on additional lines (in other words, if the value wraps around to another line), use a single space (the ASCII 32 character) at the beginning of subsequent lines. For example:

```
dn: cn=Jake Lup
```

```
inski, ou=Accoun
ting, o=air
ius.com
```

If a double-colon is used after a type, it means the value after the double-colon is encoded as a base 64 string. Data is sometimes encoded as a base 64 string if (for example) the value contains a non-printing character or newline.

To get the string representation of an entry (and vice versa), call the following functions:

- To convert an entry from the datatype `Slapi_Entry` to its LDIF string representation, call the `slapi_entry2str()` function (see "slapi_entry2str()" on page 268 for details).

  This function returns the LDIF string representation of the entry or NULL if an error occurs. When you no longer need to use the string, you should free it from memory by calling the `slapi_ch_free()` function.

- To convert an LDIF string representation back to an entry of the datatype `Slapi_Entry`, call the `slapi_str2entry()` function (see "slapi_str2entry()" on page 345 for details).

  This function returns an entry of the datatype `Slapi_Entry`. If an error occurred during the conversion process, the function returns NULL instead.

  When you are done working with the entry, you should call the `slapi_entry_free()` function (see "slapi_entry_free()" on page 284 for details).

Note Calling `slapi_str2entry()` modifies the value of the string argument passed into the function. If you still want to use the string representation of the entry, make a copy of the string before calling this function.

Getting and Setting the DN of an Entry

You can call the following two front-end routines to get and set the DN for an entry:

- To get the DN for an entry, call the `slapi_entry_get_dn()` function (see "slapi_entry_get_dn()" on page 285 for details).

- To set the DN for an entry, call the `slapi_entry_set_dn()` function (see “slapi_entry_set_dn()” on page 289 for details).

Verifying Compliance with the Schema

Before you add or modify an entry in the database, you may want to verify that the new or changed entry still complies with the schema.

To see if an entry complies with the schema, call the `slapi_entry_schema_check()` function (see “slapi_entry_schema_check()” on page 288 for details).

Getting the Attributes and Values of an Entry

To get the attributes and values of an entry, call the following functions:

- To iterate through the attributes of an entry, call the `slapi_entry_first_attr()` function to get the first attribute, and then call the `slapi_entry_next_attr()` function to get subsequent attributes (see “slapi_entry_first_attr()” on page 283 and “slapi_entry_next_attr()” on page 286 for details).

  Both functions return a pointer to the current attribute. To get the next attribute in the entry, pass this pointer to the `cookie` parameter of the `slapi_entry_next_attr()` function.

- To get the values of a specific attribute, call the `slapi_entry_attr_find()` function (see “slapi_entry_attr_find()” on page 273 for details).

Adding and Removing Values

You can also call some of the front-end routines to add or remove values in an entry:

- To add new values to an entry, call the `slapi_entry_add_values()` function (see “slapi_entry_add_values()” on page 270 for details).

- To remove values from an entry, call the `slapi_entry_delete_values()` function (see “slapi_entry_delete_values()” on page 281 for details).
- In certain situations, you may want to add an attribute and its values to an entry while not replacing any attribute values that already exist. To do this, call the `slapi_entry_attr_merge()` function (see “slapi_entry_attr_merge()” on page 277 for details).

Working with Search Filters

When a client requests an LDAP search operation, the front-end passes the search filter to the back-end as part of the parameter block. (The filter is passed through the `SLAPI_SEARCH_FILTER` parameter. A string representation of the filter is also available in the `SLAPI_SEARCH_STRFILTER` parameter.)

If you need to manipulate these search filters, you can call the following front-end routines:

Table 3.2 Front-end functions for manipulating filters

To do this:	Call this function
Determine if an entry matches a filter's criteria	`slapi_filter_test()`
Get the filter type	`slapi_filter_get_choice()`
Get the attribute type and value used for comparison in an attribute-value assertion filter (only applicable to `LDAP_FILTER_EQUALITY`, `LDAP_FILTER_GE`, `LDAP_FILTER_LE`, and `LDAP_FILTER_APPROX` searches)	`slapi_filter_get_ava()`
Get the type of attribute that the filter is searching for (only applicable to `LDAP_FILTER_PRESENT` searches)	`slapi_filter_get_type()`

Table 3.2 Front-end functions for manipulating filters

To do this:	Call this function
Get the substring pattern used for the filter (applicable only to LDAP_FILTER_SUBSTRING searches)	slapi_filter_get_subfilt()
Convert a string representation of a filter to a filter of the datatype Slapi_Filter	slapi_str2filter()
Construct a new LDAP_FILTER_AND, LDAP_FILTER_OR, or LDAP_FILTER_NOT filter from other filters	slapi_filter_join()
Get the components of a filter (only applicable to LDAP_FILTER_AND, LDAP_FILTER_OR, and LDAP_FILTER_NOT searches)	slapi_filter_list_first(), slapi_filter_list_next()
Free a filter from memory	slapi_filter_free()

Determining if an Entry Matches a Filter

After retrieving a filter from the SLAPI_SEARCH_FILTER parameter of the parameter block, you can call the slapi_filter_test() function to determine if entries in your database match the filter (see “slapi_filter_test()” on page 299 for details).

Getting the Filter Type

To determine the type of filter you are using, call the `slapi_filter_get_choice()` function (see “slapi_filter_get_choice()” on page 292 for details). This function returns the filter type. The filter type can be one of the following values:

Table 3.3 Types of filters

Filter Type	Description
`LDAP_FILTER_AND`	The search should find entries matching all filters that are specified in this complex filter.
`LDAP_FILTER_OR`	The search should find entries matching any filter specified in this complex filter.
`LDAP_FILTER_NOT`	The search should find entries not matching the specified filter.
`LDAP_FILTER_EQUALITY`	The search should find entries that contain a value equal to the specified attribute value.
`LDAP_FILTER_SUBSTRINGS`	The search should find entries that contain a value matching the specified substrings.
`LDAP_FILTER_GE`	The search should find entries that contain a value greater than or equal to the specified attribute value.
`LDAP_FILTER_LE`	The search should find entries that contain a value less than or equal to the specified attribute value.
`LDAP_FILTER_PRESENT`	The search should find entries that contain the specified attribute.
`LDAP_FILTER_APPROX`	The search should find entries that contain a value approximately matching the specified attribute value.

Getting the Search Criteria

To get the search critieria specified by a search filter, call one of the following functions:

- If the filter type is `LDAP_FILTER_EQUALITY`, `LDAP_FILTER_GE`, `LDAP_FILTER_LE`, or `LDAP_FILTER_APPROX`, you can get the attribute and value used in the filter by calling the `slapi_filter_get_ava()` function (see “slapi_filter_get_ava()” on page 291 for details).
- If the filter type is `LDAP_FILTER_PRESENT`, you can get the attribute type that the filter searches for. To get this attribute, call the `slapi_filter_get_type()` function (see “slapi_filter_get_type()” on page 295 for details).
- If the filter type is `LDAP_FILTER_SUBSTRINGS`, you can get the attribute type that the filter searches for. To get this attribute, call the `slapi_filter_get_subfilt()` function (see “slapi_filter_get_subfilt()” on page 294 for details).
- To get the components of a complex filter of the type `LDAP_FILTER_AND`, `LDAP_FILTER_OR`, or `LDAP_FILTER_NOT`, call the `slapi_filter_list_first()` and `slapi_filter_list_next()` functions (see “slapi_filter_list_first()” on page 297 and “slapi_filter_list_next()” on page 298 for details).

  Both functions return a filter component of the complex filter or a NULL value:

  - If `slapi_filter_list_first()` returns a NULL, the complex filter is not of the type `LDAP_FILTER_AND`, `LDAP_FILTER_OR`, or `LDAP_FILTER_NOT`.
  - If `slapi_filter_list_next()` returns a NULL, the component returned by the call is the last component in the complex filter.

Note Do not free the values returned by the `slapi_filter_get_ava()`, `slapi_filter_get_type()`, and `slapi_filter_get_subfilt()` functions.

Converting a String to a Filter

Search filters can be represented by the datatype `Slapi_Filter` or as strings. (In a parameter block for a search operation, `SLAPI_SEARCH_FILTER` is a filter of the datatype `Slapi_Filter`, and `SLAPI_SEARCH_STRFILTER` is the string representation of that filter.) In general, it is easier to specify a filter as a string than to construct a filter of the type `Slapi_Filter`.

To convert a string representation of a filter into a filter of the datatype `Slapi_Filter`, call the `slapi_str2filter()` function (see “slapi_str2filter()” on page 347 for details).

When you are done working with the filter, you should free it by calling the `slapi_filter_free()` function (see “slapi_filter_free()” on page 290 for details).

Combining Filters

You can use AND, OR and NOT to combine different filters into a complex filter. To do this, call the `slapi_filter_join()` function (see “slapi_filter_join()” on page 296 for details).

Note that filters of the type `LDAP_FILTER_NOT` only have one component. If `ftype` is `LDAP_FILTER_NOT`, pass NULL for the second filter.

The function returns the complex filter you’ve created. When you are done using the complex filter, you should free it by calling `slapi_filter_free()` function (see “slapi_filter_free()” on page 290 for details).

Working with Distinguished Names

In certain situations, the front-end passes DNs to the back-end through the parameter block. For example, when calling the add function, the parameter block includes a parameter that specifies the new DN of the entry to be added.

If you need to manipulate these DNs, you can call the following front-end routines:

Table 3.4 Front-end functions for manipulating DNs

To do this:	Call this function
Determine if a DN is the root DN (the DN of the privileged superuser)	`slapi_dn_isroot()`
Get a copy of the parent DN	`slapi_dn_parent()`, `slapi_dn_beparent()`

Table 3.4 Front-end functions for manipulating DNs

To do this:	Call this function
Determine if a DN is the child of another DN	`slapi_dn_issuffix()`
Determine if a DN is a suffix served by one of the server's back-ends	`slapi_dn_isbesuffix()`
Normalize a DN	`slapi_dn_normalize()`
Convert all characters in a DN to lowercase	`slapi_dn_ignore_case()`
Normalize a DN and convert all characters to lowercase	`slapi_dn_normalize_case()`

Determining if a DN is the Root DN

To determine if a DN is the root DN, call the `slapi_dn_isroot()` function (see "slapi_dn_isroot()" on page 263 for details).

Working with DN Suffixes

A suffix of a DN identifies a subtree in the directory tree where the DN is located. For example, for the following DN:

```
cn=Babs Jensen, ou=Product Development, o=Ace Industries, c=US
```

one of the suffixes is:

```
o=Ace Industry, c=US
```

which indicates that the Babs Jensen entry is located in the Ace Industry subtree in the directory tree.

The `suffix` directive in the `slapd.conf` configuration file determines which DNs are served by a particular back-end. For example, if the suffix directive is:

```
suffix "o=Ace Industry, c=US"
```

the entries within the “o=Ace Industry, c=US” subtree are served by this back-end.

To determine if a value is a suffix for a DN, call the `slapi_dn_issuffix()` function (see “slapi_dn_issuffix()” on page 264 for details).

To determine if a DN is one of the suffixes served by the back-end, call the `slapi_dn_isbesuffix()` function (see “slapi_dn_isbesuffix()” on page 261 for details).

Getting the Parent DN of a DN

To get a copy of the parent DN for a DN, call the `slapi_dn_parent()` function or the `slapi_dn_beparent()` function (see “slapi_dn_isparent()” on page 262 and “slapi_dn_beparent()” on page 259 for details).

These functions return the parent DN of `dn`. If `dn` is a suffix served by the back-end, `slapi_dn_beparent()` returns NULL.

When you are done working with the parent DN, you should free it from memory by calling `free()`.

Normalizing a DN

If you need to compare DNs (for example, if you are searching your database for a particular DN), you may want to normalize the DNs and make sure that the DNs use the same case.

To normalize and convert the case of a DN, you can call the following functions. Note that these functions operate on the actual DN specified in the argument, not a copy of the DN. If you want to modify a copy of the DN, call the `slapi_ch_strdup()` function to make a copy of the DN first (see “slapi_ch_strdup()” on page 253 for details).

- `slapi_dn_normalize()` normalizes the DN (see “slapi_dn_normalize()” on page 265 for details).
- `slapi_dn_ignore_case()` converts all characters in the DN to lowercase (see “slapi_dn_ignore_case()” on page 260 for details).

- `slapi_dn_normalize_case()` normalizes the DN and converts all characters in the DN to lowercase (see “slapi_dn_normalize_case()” on page 266 for details).

Checking Passwords

When the Netscape Directory Server stores the password for an entry in the `userpassword` attribute, it encrypts the password using the scheme specified in the `passwdhash` directive of the `slapd.conf` configuration file. The scheme can be `crypt` or `sha` or `“”` (for cleartext).

If you need to determine if a given password is one of the values of the `userpassword` attribute, you can call the `slapi_pw_find()` function (see “slapi_pw_find()” on page 330 for details). This function determines which password scheme was used to store the password and uses the appropriate comparison function to compare a given value against the encrypted values of the `userpassword` attribute.

Chapter

4

Quick Start

This chapter provides a few simple examples of server plug-ins that you can compile and implement. Essentially, these examples demonstrate the different types of data available to your plug-in.

Example of a Pre-Operation Plug-In

The following example is a pre-operation plug-in for the LDAP search operation. Two functions are included in this example:

- The `test_preop_search()` function logs information about the search, including the base DN of the search, the search scope, and the type of filter used.
- The `test_preop_init()` function is the initialization function that registers `test_preop_search()` as a `SLAPI_PLUGIN_PRE_SEARCH_FN` pre-operation plug-in function for LDAP search operations.

These functions illustrates the data in the parameter block that is available to your function. You can get and manipulate the data by calling the front-end API functions.

Writing the Plug-In Example

The following section of code includes the sample pre-operation search function and the sample initialization function.

```
#include <stdio.h>
#include <string.h>
#include "slapi-plugin.h"

/* function prototypes */
int test_preop_init( Slapi_PBlock *pb );
int test_preop_search( Slapi_PBlock *pb );

/* Description of the plug-in */
Slapi_PluginDesc srchpdesc = { "test-search", "Airius.com", "0.5",
   "sample pre-operation search plugin" };

/* Initialization function
This function registers your plug-in function as a
pre-operation search function in the Directory Server.
You need to specify this initialization function in the
slapd.conf configuration file so that the server calls
this initialization function on startup. */
#ifdef _WIN32
__declspec(dllexport)
#endif
int
test_preop_init( Slapi_PBlock *pb )
{
   /* Specify the version of the plug-in ("01" in this release ) */
   if ( slapi_pblock_set( pb, SLAPI_PLUGIN_VERSION,
      SLAPI_PLUGIN_VERSION_01 ) != 0 ||
      /* Specify the description of the plug-in */
```

```
      slapi_pblock_set( pb, SLAPI_PLUGIN_DESCRIPTION,
         (void *)&srchpdesc ) != 0 ||
      /* Set test_preop_search() as the function to call before
         executing LDAP search operations. */
      slapi_pblock_set( pb, SLAPI_PLUGIN_PRE_SEARCH_FN,
         (void *) test_preop_search ) !=0 ) {
      /* Log an error message and return -1 if a problem occurred */
      slapi_log_error( SLAPI_LOG_PLUGIN, "test_preop_init",
         "Error registering the plug-in.\n" );
      return( -1 );
   }
   /* If successful, log a message and return 0 */
   slapi_log_error( SLAPI_LOG_PLUGIN, "test_preop_init",
      "Plug-in successfully registered.\n" );
   return( 0 );
}

/* Pre-operation plug-in function for LDAP search operations
This function is called by the server before processing an LDAP
search operation. The function gets data about the search request
from the parameter block and prints the data to the error log. */
int
test_preop_search( Slapi_PBlock *pb )
{
   char *base, *filter_str, *attr_type, *substr_init, *substr_final;
   char **substr_any;
   int scope, deref, filter_type, i;
   Slapi_Filter *filter;
   struct berval *bval;
   /* Log a message to indicate when the plug-in function starts */
   slapi_log_error( SLAPI_LOG_PLUGIN, "test_preop_search",
      "*** PREOPERATION SEARCH PLUGIN ***\n");
```

```
/* Get and log the base DN of the search criteria */
if ( slapi_pblock_get( pb, SLAPI_SEARCH_TARGET, &base ) == 0 )
   slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_TARGET",
      "%s\n", base );
/* Get and log the search scope */
if ( slapi_pblock_get( pb, SLAPI_SEARCH_SCOPE, &scope ) == 0 ) {
   switch( scope ) {
   case LDAP_SCOPE_BASE:
      slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_SCOPE",
         "LDAP_SCOPE_BASE\n" );
      break;
   case LDAP_SCOPE_ONELEVEL:
      slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_SCOPE",
         "LDAP_SCOPE_ONELEVEL\n" );
      break;
   case LDAP_SCOPE_SUBTREE:
      slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_SCOPE",
         "LDAP_SCOPE_SUBTREE\n" );
      break;
   default:
      slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_SCOPE",
         "unknown value specified: %d\n", scope );
      break;
   }
}
/* Get and log the alias dereferencing setting */
if ( slapi_pblock_get( pb, SLAPI_SEARCH_DEREF, &deref ) == 0 ) {
   switch( deref ) {
   case LDAP_DEREF_NEVER:
      slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_DEREF",
         "LDAP_DEREF_NEVER\n" );
      break;
```

```
        case LDAP_DEREF_SEARCHING:
            slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_DEREF",
                "LDAP_DEREF_SEARCHING\n" );
            break;
        case LDAP_DEREF_FINDING:
            slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_DEREF",
                "LDAP_DEREF_FINDING\n" );
            break;
        case LDAP_DEREF_ALWAYS:
            slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_DEREF",
                "LDAP_DEREF_ALWAYS\n" );
            break;
        default:
            slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_DEREF",
                "unknown value specified: %d\n", deref );
            break;
        }
    }
    /* Get and log the search filter information */
    if ( slapi_pblock_get( pb, SLAPI_SEARCH_FILTER, &filter ) == 0 ) {
        /* Get and log the filter type */
        filter_type = slapi_filter_get_choice( filter );
        switch( filter_type ) {
        case LDAP_FILTER_AND:
        case LDAP_FILTER_OR:
        case LDAP_FILTER_NOT:
            slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_FILTER",
                "Complex search filter.  See value of
SLAPI_SEARCH_STRFILTER.\n" );
            break;
        case LDAP_FILTER_EQUALITY:
            slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_FILTER",
```

```
            "LDAP_FILTER_EQUALITY\n" );
         break;
      case LDAP_FILTER_GE:
         slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_FILTER",
            "LDAP_FILTER_GE\n" );
         break;
      case LDAP_FILTER_LE:
         slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_FILTER",
            "LDAP_FILTER_LE\n" );
         break;
      case LDAP_FILTER_APPROX:
         slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_FILTER",
            "LDAP_FILTER_APPROX\n" );
         break;
      case LDAP_FILTER_SUBSTRINGS:
         slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_FILTER",
            "LDAP_FILTER_SUBSTRINGS\n" );
         /* For substring filters, get and log the attribute type and
            the substrings in the filter */
         slapi_filter_get_subfilt( filter, &attr_type, &substr_init,
            &substr_any, &substr_final );
         if ( attr_type != NULL )
            slapi_log_error( SLAPI_LOG_PLUGIN, "\tAttribute type",
               "%s\n", attr_type );
         if ( substr_init != NULL )
            slapi_log_error( SLAPI_LOG_PLUGIN, "\tInitial substring",
               "%s\n", substr_init );
         if ( substr_any != NULL ) {
            for ( i = 0; substr_any[i] != NULL; i++ ) {
               slapi_log_error( SLAPI_LOG_PLUGIN, "\tSubstring",
                  "# %d: %s\n", i, substr_any[i] );
            }
```

```
         }
         if ( substr_final != NULL )
            slapi_log_error( SLAPI_LOG_PLUGIN, "\tFinal substring",
               "%s\n", substr_final );
         break;
      case LDAP_FILTER_PRESENT:
         slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_FILTER",
            "LDAP_FILTER_PRESENT\n" );
         /* For presence filters, get and log the attribute type */
         slapi_filter_get_type( filter, &attr_type );
         if ( attr_type != NULL )
            slapi_log_error( SLAPI_LOG_PLUGIN, "\tSearch for presence
of attr",
               "%s\n", attr_type );
         break;
      case LDAP_FILTER_EXTENDED:
         slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_FILTER",
            "LDAP_FILTER_EXTENDED\n" );
         break;
      default:
         slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_FILTER",
            "Unknown filter type: slapi_filter_get_choice returned
%d\n",
            filter_type );
         break;
      }
   }
   /* For comparison filters, get and log the attribute type */
   if ( filter_type == LDAP_FILTER_EQUALITY || LDAP_FILTER_GE ||
LDAP_FILTER_LE ||
      LDAP_FILTER_APPROX ) {
      slapi_filter_get_ava( filter, &attr_type, &bval );
      if ( ( attr_type != NULL ) && ( bval->bv_val != NULL ) ) {
```

```
            slapi_log_error( SLAPI_LOG_PLUGIN, "\tAttribute type",
               "%s\n", attr_type );
            slapi_log_error( SLAPI_LOG_PLUGIN, "\tAttribute value",
               "%s\n", bval->bv_val );
         }
      }
      /* Get and log the string representation of the filter */
      if ( slapi_pblock_get(pb, SLAPI_SEARCH_STRFILTER, &filter_str) == 0 )
         slapi_log_error( SLAPI_LOG_PLUGIN, "SLAPI_SEARCH_STRFILTER",
            "%s\n", filter_str );
      slapi_log_error( SLAPI_LOG_PLUGIN, "test_preop_search",
         "*** DONE ***\n\n" );
      return( 0 );
}
```

Compling the Plug-In Example

On Solaris, you can use the following makefile to compile the example. (This sample makefile assumes that the source code is stored in `srchxmpl.c` and that you are compiling a plug-in named `srchxmpl.so`.)

```
# SOLARIS Makefile for Directory Server plug-in examples
#
CC = cc
LD = ld
INCLUDE_FLAGS = -I../include
CFLAGS = $(INCLUDE_FLAGS) -D_REENTRANT -KPIC
LDFLAGS = -G
OBJS = srchxmpl.o
all: srchxmpl.so
srchxmpl.so: $(OBJS)
   $(LD) $(LDFLAGS) -o $@ $(OBJS)
.c.o:
```

```
   $(CC) $(CFLAGS) -c $<
clean:
   -rm -f $(OBJS) srchxmpl.so
```

On Windows NT, make sure to link to the `libslapd.lib` import library (located under the `<server_root>\plugins\slapd\slapi\lib` directory).

Registering the Plug-In Example

To register this example plug-in, you should do the following:

16. Edit the `slapd.conf` file in a text editor, and add a `plugin` directive after the `database` directive:

```
plugin preoperation <path_to_library> test_preop_init
```

    (Each pre-operation plug-in is associated with a back-end. The directive that registers the plug-in must fall within the database-specific section for that database in the `slapd.conf` file.)

17. In the Administration Server, click the Apply button in the top frame to save and apply your changes.

18. In the server manager, click Server Preferences | LDAP, and verify that the Plugins log level is selected. (The sample source code logs messages to the error log with the Plugins severity level.)

Running the Plug-In Example

Restart the Directory Server and check the error log. (In the server manager, click Server Status | View Error Log.) Perform a few searches against the directory. The pre-operation search plug-in function should write data about the search to the error log.

2

Writing Database and Operation-Based Plug-Ins

Chapter 8 Defining Functions for Database Operations

This chapter explains how to write functions that perform database operations, such as exporting the directory, importing LDIF files, archiving, and restoring database archives.

Chapter 9 Defining Functions for Authentication

This chapter explains how to write a plug-in function to bypass or replace the standard function for authentication with your own function.

Chapter

5

Writing Database Plug-Ins

This chapter explains the architecture of the directory server and how the back-ends work with databases. The chapter also describes how you can add your own back-end to the server database.

- How Directory Server Back-Ends Work
- How Database Plug-Ins Work
- Writing Database Plug-In Functions
- Registering Database Plug-In Functions
- Adding a Back-End to the Server

How Directory Server Back-Ends Work

Directory entries are stored in a back-end database by the directory server. `ldbm`, the standard database provided with the directory server is a general purpose LDAP database designed to be used with directories in a variety of sizes and configurations.

In some cases, you may have special needs for your directory that require a database with different characteristics. In these cases, you can write a database plug-in to integrate your own database with the back-end of the directory server.

The directory server is designed to allow you to specify different databases. The `slapd.conf` server configuration file is organized in the following way:

```
# Directives general to all databases.
...
localhost ldap.airius.com
port 389
...
# Directives specific to the ldbm database.
database ldbm
#########################################################################
# ldbm database definitions
#########################################################################
suffix  o=Airius.com
suffix  "o=Airius.com, c=US"
suffix  o=netscape.com
suffix  dc=rtfm,dc=mcom,dc=com
suffix  cn=schema
...
```

The first section of the `slapd.conf` file contains directives that are general to all databases (for example, the directives specifying the hostname and port number of the directory server).

The `database` directive identifies the beginning of a section specific to that database. The `suffix` directive specifies the suffix handled by that database. (For example, according to the section of the `slapd.conf` file shown above, entries that end with “o=Airius.com” are stored in the `ldbm` database.

When the directory server reads in the `slapd.conf` file and encounters a `database` directive, it creates a back-end in memory and adds it to a list of existing back-ends. The server also reads any `suffix` directives and `plugin` directives after the `database` directive and associates these with the back-end.

If the directory server encounters another `database` directive, the server creates another back-end in memory, and any subsequent directives are associated with this back-end.

The `database` directive marks the beginning and end of a section containing directives specific to a database. For example:

```
...
# Directives specific to the ldbm database.
database ldbm
suffix  o=Airius.com
suffix  "o=Airius.com, c=US"
suffix  o=netscape.com
suffix  dc=rtfm,dc=mcom,dc=com
suffix  cn=schema
...
# Directives specific to the mydb database.
database mydb
suffix  o=test.com
...
# Directives specific to the anotherdb database.
database anotherdb
suffix  o=mycompany.com
...
```

Note The directory server always needs to access the `ldbm` database, even if you plan to store your data in a different back-end database.

How Database Plug-Ins Work

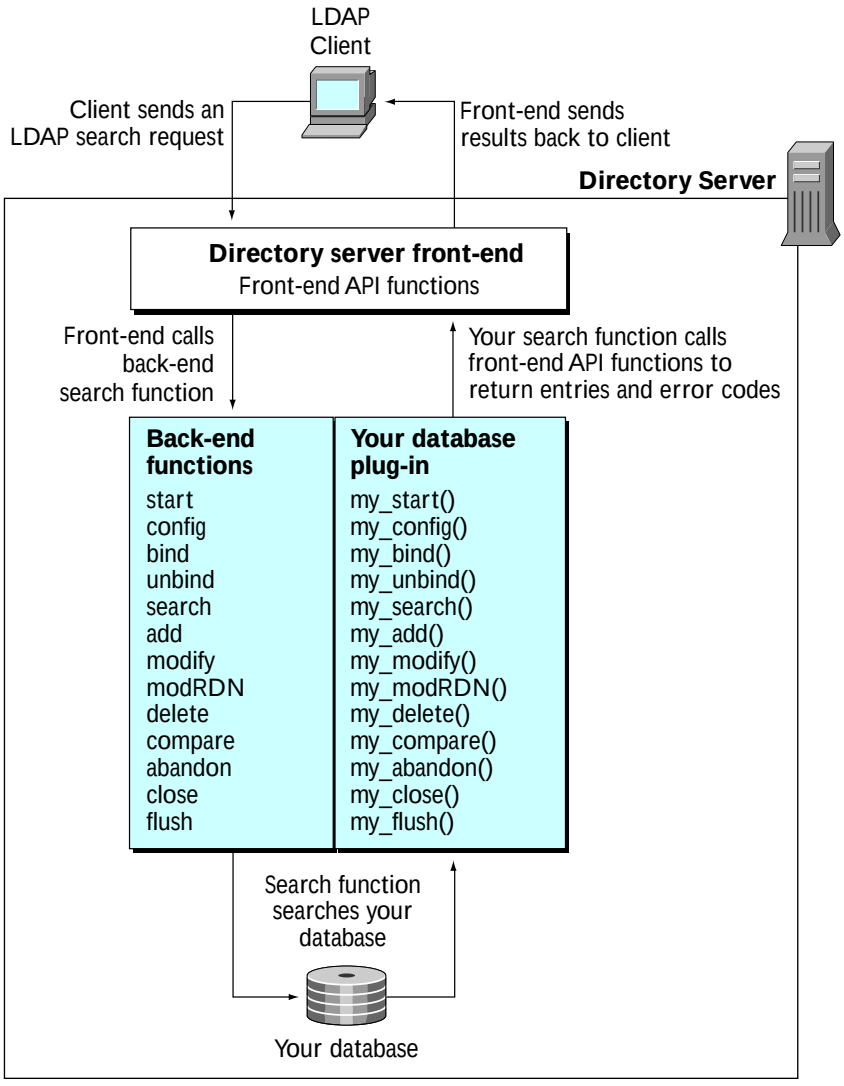
A database plug-in registers the back-end functions that allow the directory server to interact with the database. When processing an LDAP operation that deals with the directory data (such as a search or a modification of an entry), the front-end of the directory server calls the registered back-end functions to access the database.

For example, when processing an LDAP search operation, the front-end of the directory server calls the search function that is registered for that back-end. Or when processing an LDAP add operation, the directory server front-end calls the add function registered for the back-end.

As is the case with other plug-ins, the front-end places data relevant to the operation in a parameter block. When the front-end calls the back-end function, it passes the parameter block to the function.

Figure 5.1 illustrates how the directory server front-end calls back-end functions to access the database and process LDAP operations.

Figure 5.1 How the server calls database plug-in functions



Writing Database Plug-In Functions

Database plug-in functions pass in a parameter block as an argument and return an integer. The server passes information to the plug-in function through parameters in the parameter block. You can call the `slapi_pblock_get()` function to get the values in the parameter block.

If your function successfully completes the requested operation, your function should return 0.

In the Netscape Directory Server 3.x, you should return -1 if an error occurs. In the Netscape Directory Server 4.x, the following return codes have been defined for database plug-in functions:

- `SLAPI_FAIL_GENERAL`

  Your function should return this value if an error occurs and the requested operation cannot be completed.

- `SLAPI_FAIL_DISKFULL`

  Your function should return this value if no disk space is available. If your function returns this value, the server prints a "DISK FULL" message to the error log and attempts to shut itself down.

Registering Database Plug-In Functions

As is the case with other server plug-in functions (see Chapter 2, "Writing and Compiling Plug-Ins"), you register database plug-in functions by writing an initialization function that specifies the functions in a parameter block.

Each function has an ID in the parameter block. You can call the `slapi_pblock_set()` function to specify the name of your function that corresponds to the back-end function.

Table 5.1 lists the directory server back-end functions and the purpose of each function.

Table 5.1 Functions comprising the directory server database back-end

ID in Parameter Block	Description
SLAPI_PLUGIN_DB_CONFIG_FN	Specifies the function called when the directory server reads in and parses the `slapd.conf` configuration file. If the directory server front-end does not recognize a directive, it passes the directive to this back-end function. For information on writing this type of function, see "Reading Configuration Files" on page 98.
SLAPI_PLUGIN_DB_BIND_FN	Specifies the function called when the directory server processes an LDAP bind operation. For information on writing this type of function, see "Processing an LDAP Bind Operation" on page 99.
SLAPI_PLUGIN_DB_UNBIND_FN	Specifies the function called when the directory server processes an LDAP unbind operation. For information on writing this type of function, see "Processing an LDAP Unbind Operation" on page 101.
SLAPI_PLUGIN_DB_SEARCH_FN	Specifies the function called when the directory server processes an LDAP search operation. For information on writing this type of function, see "Processing an LDAP Search Operation" on page 101.
SLAPI_PLUGIN_DB_NEXT_SEARCH_ENTRY_FN	Specifies the function called when the directory server iterates through the next search result in the result set. For information on writing this type of function, see "Iterating through Candidates" on page 104.
SLAPI_PLUGIN_DB_NEXT_SEARCH_ENTRY_EXT_FN	(Netscape Directory Server 4.0) Reserved for future use. This function is a new version of the "next search entry function". This version of the function can include the LDAP result code with the last search entry and can pass the context of the result set (the last result sent) between function calls.

Table 5.1 Functions comprising the directory server database back-end

ID in Parameter Block	Description
SLAPI_PLUGIN_DB_ENTRY_RELEASE_FN	(Netscape Directory Server 4.0) Reserved for future use. Specifies the function called after the server sends the last entry back to the client. This function is intended to free the context of the result set (the last result sent).
SLAPI_PLUGIN_DB_COMPARE_FN	Specifies the function called when the directory server processes an LDAP compare operation. For information on writing this type of function, see "Processing an LDAP Compare Operation" on page 105.
SLAPI_PLUGIN_DB_ADD_FN	Specifies the function called when the directory server processes an LDAP add operation. For information on writing this type of function, see "Processing an LDAP Add Operation" on page 106.
SLAPI_PLUGIN_DB_MODIFY_FN	Specifies the function called when the directory server processes an LDAP modify operation. For information on writing this type of function, see "Processing an LDAP Modify Operation" on page 108.
SLAPI_PLUGIN_DB_MODRDN_FN	Specifies the function called when the directory server processes an LDAP modify RDN operation. For information on writing this type of function, see "Processing an LDAP Modify RDN Operation" on page 109.
SLAPI_PLUGIN_DB_DELETE_FN	Specifies the function called when the directory server processes an LDAP delete operation. For information on writing this type of function, see "Processing an LDAP Delete Operation" on page 111.
SLAPI_PLUGIN_DB_ABANDON_FN	Specifies the function called when the directory server processes an LDAP abandon operation. For information on writing this type of function, see "Processing an LDAP Abandon Operation" on page 112.

Table 5.1 Functions comprising the directory server database back-end

ID in Parameter Block	Description
SLAPI_PLUGIN_CLOSE_FN	Specifies the function called when the directory server is shutting down (this function allows the back-end database to be shut down correctly before the server shuts down).
SLAPI_PLUGIN_DB_FLUSH_FN	Specifies the function called when the directory server flushes any open caches or unwritten information to disk.
SLAPI_PLUGIN_START_FN	Specifies the function called for each back-end on server startup after the `slapd.conf` configuration file is read in.
SLAPI_PLUGIN_DB_SEQ_FN	Specifies the function called when the front-end needs to iterate through database entries sequentially.
SLAPI_PLUGIN_DB_LDIF2DB_FN	Specifies the function called when the front-end needs to convert an LDIF file to database format. For information on writing this type of function, see "Importing an LDIF File into the Database" on page 114.
SLAPI_PLUGIN_DB_DB2LDIF_FN	Specifies the function called when the front-end needs to convert a database to LDIF file format. For information on writing this type of function, see "Exporting the Database to an LDIF File" on page 116.
SLAPI_PLUGIN_DB_ARCHIVE2DB_FN	Specifies the function called when the front-end needs to restore a database from its archive. For information on writing this type of function, see "Restoring the Database from an Archive" on page 117.
SLAPI_PLUGIN_DB_DB2ARCHIVE_FN	Specifies the function called when the front-end needs to archive a database. For information on writing this type of function, see "Saving the Database as an Archive" on page 117.
SLAPI_PLUGIN_DB_SIZE_FN	Specifies the function called to get the size of the database and set as `SLAPI_DBSIZE` in the parameter block. For information on writing this type of function, see "Reporting the Size of the Database" on page 118.

Table 5.1 Functions comprising the directory server database back-end

ID in Parameter Block	Description
SLAPI_PLUGIN_DB_TEST_FN	Specifies the function called to test the back-end database. For information on writing this type of function, see "Testing the Database" on page 118.
SLAPI_PLUGIN_DB_DB2INDEX_FN	Specifies the function called to generate indexes for the existing database. For information on writing this type of function, see "Generating Indexes for the Database" on page 118.

If you do not define a database plug-in function for a back-end function, the directory server will return an LDAP_UNWILLING_TO_PERFORM error ("Function not implemented") when that back-end function needs to be called.

For example, suppose you have not defined a database function for the back-end add function. When the directory server receives a request for an LDAP add operation, the server returns an LDAP_UNWILLING_TO_PERFORM error to the client.

This provides an easy way to provide custom back-end functions that only cover part of the interface. (For example, you could provide a read-only interface that allows users to search for and compare entries but not to add, modify, or delete entries.)

See Chapter 7, "Defining Functions for LDAP Operations" and Chapter 8, "Defining Functions for Database Operations" for more information on writing functions to handle each LDAP operation and each database operation.

To register your database plug-in functions, you need to write an initialization function and then configure the server to load your plug-in.

For details, follow the procedures outlined in "Writing an Initialization Function" on page 34 and "Configuring the Server" on page 39.

Adding a Back-End to the Server

In the `slapd.conf` configuration file, the `database` directive that specifies the database type. At the end of the file, add a directive for your own database type.

You also need to specify the following directives:

- Use the `suffix` directive to indicate the suffix of the directory tree that is handled by your back-end.
- Use the `plugin database` directive to specify the library containing your plug-in functions and the name of the initialization function that registers your database plug-in.

In addition, you can specify other `plugin` directives to register plug-ins that apply to this back-end.

Other directives that appear within the `ldbm` database section are specific to the `ldbm` back-end. (For example, adding a `dbcachesize` directive to the section for your back-end will not set the database cache size (unless you have registered a configuration function that handles that directive).

Note Do not remove the database section for the default `ldbm` database. The directory server needs to have access to this database, even though you plan to store your directory data in another database.

The following is an example of the section of the configuration file that registers the back-end `mydbtype`. The back-end serves requests for entries that end with the suffix “o=MyCompany.com”. On startup, the server will call the `mydb_init()` function in the `mydb.so` shared object to register the database plug-in functions.

```
database mydbtype

suffix o=MyCompany.com

plugin database /serverroot/plugins/slapd/slapi/examples/mydb.so \
   mydb_init
```

Note that this plugin directive in the example above uses the syntax supported by the Netscape Directory Server 3.x. For the Netscape Directory Server 4.x, the syntax differs slightly:

```
plugin database on mydbplugin \
   /serverroot/plugins/slapd/slapi/examples/mydb.so mydb_init
```

For more information on the plugin directive, see “Editing the slapd.conf File” on page 40.

Chapter

6

Writing Pre/Post-Operation Plug-Ins

This chapter explains how to write functions that the server calls before and after executing an LDAP operation. These functions are called pre-operation and post-operation plug-in functions.

- How Pre/Post-Operation Plug-Ins Work
- Writing Pre/Post-Operation Functions
- Registering Pre/Post-Operation Functions

How Pre/Post-Operation Plug-Ins Work

The Netscape Directory Server can perform the following LDAP operations:

- bind
- unbind
- search
- modify
- add
- delete

- modify RDN
- compare
- abandon

(Note that the Directory Server can also perform extended operations. Extended operations are defined as part of the LDAP v3 protocol. For information on implementing plug-in functions to execute extended operations, see Chapter 11, “Writing Extended Operation Plug-Ins”.)

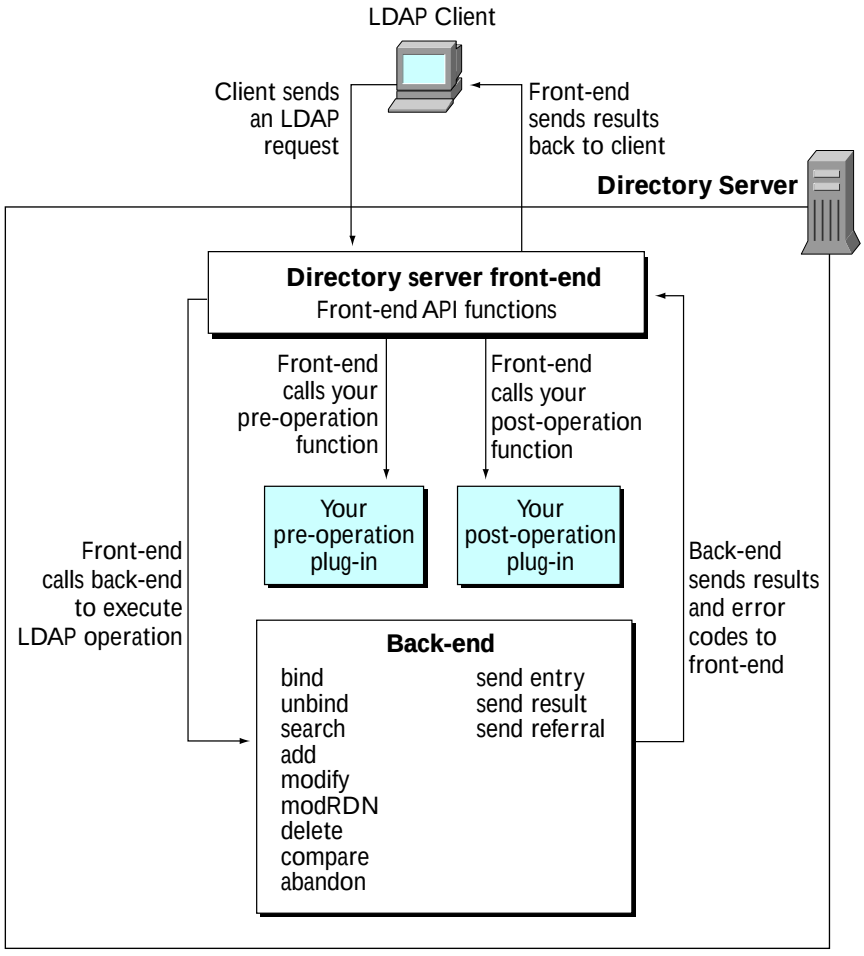
You can set up the Directory Server to call your own plug-in functions before and after executing these LDAP operations. For example, you can write a function that validates an entry before the server performs an LDAP add operation to add it to the directory. Or you can write a function that notifies a user whose entry has just been modified by an LDAP modify operation.

You can also set up the Directory Server to call your own plug-in functions before and after:

- sending an LDAP entry back to the client
- sending an LDAP result code back to the client
- sending an LDAP referral back to the client

Figure 6.1 illustrates how the Directory Server front-end calls pre-operation and post-operation functions before and after executing an LDAP operation.

Figure 6.1 How the server calls pre-operation and post-operation plug-in functions



Writing Pre/Post-Operation Functions

As is the case with other server plug-in functions (see “Working with Parameter Blocks” on page 30), pre-operation functions and post-operation functions are specified in a parameter block that you can set on server startup. Each function corresponds to an ID in the parameter block. In your initialization function, you can call the `slapi_pblock_set()` function to specify the name of your function that corresponds to the pre-operation or post-operation function.

Types of Pre-Operation Functions

Table 6.1 lists the Directory Server pre-operation functions and the purpose of each function.

Table 6.1 Functions called before the Directory Server executes an operation

ID in Parameter Block	Description
SLAPI_PLUGIN_PRE_BIND_FN	Specifies the function called before the Directory Server executes an LDAP bind operation. For information on writing this type of function, see "Processing an LDAP Bind Operation" on page 99.
SLAPI_PLUGIN_PRE_UNBIND_FN	Specifies the function called before the Directory Server executes an LDAP unbind operation. For information on writing this type of function, see "Processing an LDAP Unbind Operation" on page 101.
SLAPI_PLUGIN_PRE_SEARCH_FN	Specifies the function called before the Directory Server executes an LDAP search operation. For information on writing this type of function, see "Processing an LDAP Search Operation" on page 101.
SLAPI_PLUGIN_PRE_COMPARE_FN	Specifies the function called before the Directory Server executes an LDAP compare operation. For information on writing this type of function, see "Processing an LDAP Compare Operation" on page 105.
SLAPI_PLUGIN_PRE_ADD_FN	Specifies the function called before the Directory Server executes an LDAP add operation. For information on writing this type of function, see "Processing an LDAP Add Operation" on page 106.
SLAPI_PLUGIN_PRE_MODIFY_FN	Specifies the function called before the Directory Server executes an LDAP modify operation. For information on writing this type of function, see "Processing an LDAP Modify Operation" on page 108.
SLAPI_PLUGIN_PRE_MODRDN_FN	Specifies the function called before the Directory Server executes an LDAP modify RDN operation. For information on writing this type of function, see "Processing an LDAP Modify RDN Operation" on page 109.
SLAPI_PLUGIN_PRE_DELETE_FN	Specifies the function called before the Directory Server executes an LDAP delete operation. For information on writing this type of function, see "Processing an LDAP Delete Operation" on page 111.

Table 6.1 Functions called before the Directory Server executes an operation

ID in Parameter Block	Description
SLAPI_PLUGIN_PRE_ABANDON_FN	Specifies the function called before the Directory Server executes an LDAP abandon operation. For information on writing this type of function, see "Processing an LDAP Abandon Operation" on page 112.
SLAPI_PLUGIN_PRE_ENTRY_FN	Specifies the function called before the Directory Server sends an entry back to the client (for example, when you call `slapi_send_ldap_search_entry()`, the pre-operation entry function is called before the entry is sent back to the client).
SLAPI_PLUGIN_PRE_REFERRAL_FN	Specifies the function called before the Directory Server sends a referral back to the client (for example, when you call `slapi_send_ldap_referral()`, the pre-operation referral function is called before the referral is sent back to the client).
SLAPI_PLUGIN_PRE_RESULT_FN	Specifies the function called before the Directory Server sends a result code back to the client (for example, when you call `slapi_send_ldap_result()`, the pre-operation result function is called before the result code is sent back to the client).

Types of Post-Operation Functions

Table 6.2 lists the Directory Server post-operation functions and the purpose of each function.

Table 6.2 Functions called after the Directory Server executes an operation

ID in Parameter Block	Description
SLAPI_PLUGIN_POST_BIND_FN	Specifies the function called after the Directory Server executes an LDAP bind operation. For information on writing this type of function, see "Processing an LDAP Bind Operation" on page 99.
SLAPI_PLUGIN_POST_UNBIND_FN	Specifies the function called after the Directory Server executes an LDAP unbind operation. For information on writing this type of function, see "Processing an LDAP Unbind Operation" on page 101.

Table 6.2 Functions called after the Directory Server executes an operation

ID in Parameter Block	Description
SLAPI_PLUGIN_POST_SEARCH_FN	Specifies the function called after the Directory Server executes an LDAP search operation. For information on writing this type of function, see "Processing an LDAP Search Operation" on page 101.
SLAPI_PLUGIN_POST_COMPARE_FN	Specifies the function called after the Directory Server executes an LDAP compare operation. For information on writing this type of function, see "Processing an LDAP Compare Operation" on page 105.
SLAPI_PLUGIN_POST_ADD_FN	Specifies the function called after the Directory Server executes an LDAP add operation. For information on writing this type of function, see "Processing an LDAP Add Operation" on page 106.
SLAPI_PLUGIN_POST_MODIFY_FN	Specifies the function called after the Directory Server executes an LDAP modify operation. For information on writing this type of function, see "Processing an LDAP Modify Operation" on page 108.
SLAPI_PLUGIN_POST_MODRDN_FN	Specifies the function called after the Directory Server executes an LDAP modify RDN operation. For information on writing this type of function, see "Processing an LDAP Modify RDN Operation" on page 109.
SLAPI_PLUGIN_POST_DELETE_FN	Specifies the function called after the Directory Server executes an LDAP delete operation. For information on writing this type of function, see "Processing an LDAP Delete Operation" on page 111.
SLAPI_PLUGIN_POST_ABANDON_FN	Specifies the function called after the Directory Server executes an LDAP abandon operation. For information on writing this type of function, see "Processing an LDAP Abandon Operation" on page 112.
SLAPI_PLUGIN_POST_ENTRY_FN	Specifies the function called after the Directory Server sends an entry back to the client (for example, when you call `slapi_send_ldap_search_entry()`, the post-operation entry function is called after the entry is sent back to the client).

Table 6.2 Functions called after the Directory Server executes an operation

ID in Parameter Block	Description
SLAPI_PLUGIN_POST_REFERRAL_FN	Specifies the function called after the Directory Server sends a referral back to the client (for example, when you call `slapi_send_ldap_referral()`, the pre-operation referral function is called after the referral is sent back to the client).
SLAPI_PLUGIN_POST_RESULT_FN	Specifies the function called after the Directory Server sends a result code back to the client (for example, when you call `slapi_send_ldap_result()`, the pre-operation result function is called after the result code is sent back to the client).

Registering Pre/Post-Operation Functions

To register your pre-operation and post-operation plug-in functions, you need to write an initialization function and then configure the server to load your plug-in.

For details, follow the procedures outlined in “Writing an Initialization Function” on page 34 and “Configuring the Server” on page 39.

Note Each pre-operation and post-operation plug-in is associated with a back-end. Make sure that the `plugin` directive that registers the plug-in is within the database section for that back-end in the `slapd.conf` file. (The `plugin` directive should be after the `database` directive and before the next `database` directive, if any.)

Chapter

7

Defining Functions for LDAP Operations

This chapter explains how to write pre-operation, database, and post-operation functions for specific LDAP operations.

In general, these functions must comply with the prototype specified in “Working with Parameter Blocks” on page 30 (in other words, the functions should pass a single `Slapi_PBlock` argument).

You can define plug-in functions to do the following:

- Specifying Start and Close Functions
- Reading Configuration Files
- Processing an LDAP Bind Operation
- Processing an LDAP Unbind Operation
- Processing an LDAP Search Operation
- Processing an LDAP Compare Operation
- Processing an LDAP Add Operation
- Processing an LDAP Modify Operation
- Processing an LDAP Modify RDN Operation
- Processing an LDAP Delete Operation

- Processing an LDAP Abandon Operation

Specifying Start and Close Functions

For each pre-operation and post-operation plug-in, you can specify the name of a function to be called after the server starts and before the server is shut down.

Use the following parameters to specify these functions:

`SLAPI_PLUGIN_START_FN`	Specifies the function called after the Directory Server starts up.
`SLAPI_PLUGIN_CLOSE_FN`	Specifies the function called before the Directory Server shuts down.

If you register multiple plug-ins with different start and close functions, the functions are called in the order that the plug-ins are registered (in other words, in the order that the `plugin` directives appear in the `slapd.conf` file).

Reading Configuration Files

When the Directory Server starts up, the front-end reads in and parses the `slapd.conf` configuration file. If the front-end encounters an unknown directive, it calls the database plug-in function for handling configuration (the function specified by the `SLAPI_PLUGIN_DB_CONFIG_FN` parameter in the parameter block).

The Directory Server passes the unknown directive to that plug-in function. Because the server is set up in this way, you can add your own directives to the `slapd.conf` configuration file and write a plug-in function to interpret these directives.

The front-end passes the directive to the configuration function in the form of parameters in the parameter block.

Parameter ID	Data Type	Description
SLAPI_CONFIG_FILENAME	char *	Name of the configuration file that is being read (for example, slapd.conf).
SLAPI_CONFIG_LINENO	int	Line number of the current directive in the configuration file.
SLAPI_CONFIG_ARGC	int	Number of arguments in the current directive.
SLAPI_CONFIG_ARGV	char **	Array of the arguments from the current directive.

Processing an LDAP Bind Operation

When the Directory Server receives an LDAP bind request from a client, the front-end determines the DN that the client is attempting to bind as and the authentication method being used. The front-end also gets the credentials used for authentication, and (if SASL is used for authentication), the SASL mechanism used.

Defining Functions for the Bind Operation

In the parameter block, the following parameters specify plug-in functions that are called in the process of executing a bind operation:

- The SLAPI_PLUGIN_PRE_BIND_FN parameter specifies the pre-operation bind function.
- The SLAPI_PLUGIN_DB_BIND_FN parameter specifies the database bind function.
- The SLAPI_PLUGIN_POST_BIND_FN parameter specifies the post-operation bind function.

You register your plug-in functions by calling `slapi_pblock_set()` to set these parameters in your initialization function. (For details, see “Registering Your Plug-In Functions” on page 36.)

Your pre-operation and post-operation bind functions should return 0 if successful. If the pre-operation function returns a non-zero value, the database bind function and the post-operation bind function are never called.

For information on defining a database bind function or a function that handles authentication, see Chapter 9, “Defining Functions for Authentication”.

Getting and Setting Parameters for the Bind Operation

The front-end makes this information available to pre-operation, database, and post-operation plug-in functions in the form of parameters in a parameter block.

Parameter ID	Data Type	Description
`SLAPI_BIND_TARGET`	`char *`	DN of the entry to bind as.
`SLAPI_BIND_METHOD`	`int`	Authentication method used (for example, `LDAP_AUTH_SIMPLE` or `LDAP_AUTH_SASL`).
`SLAPI_BIND_CREDENTIALS`	`struct berval *`	Credentials from the bind request.
`SLAPI_BIND_RET_SASLCREDS`	`struct berval *`	Credentials that you want sent back to the client. (Set this before calling `slapi_send_ldap_result()`.)
`SLAPI_BIND_SASLMECHANISM`	`char *`	SASL mechanism used (for example, `LDAP_SASL_EXTERNAL`).

Note that if the `SLAPI_BIND_SASLMECHANISM` parameter is empty, simple authentication was used and simple credentials were provided.

Processing an LDAP Unbind Operation

When the Directory Server receives an LDAP unbind request from a client, the front-end calls the database unbind function for each back-end. No operation-specific parameters are placed in the parameter block that is passed to the database function.

In the parameter block, the following parameters specify plug-in functions that are called in the process of executing a bind operation:

- The `SLAPI_PLUGIN_PRE_UNBIND_FN` parameter specifies the pre-operation bind function.
- The `SLAPI_PLUGIN_DB_UNBIND_FN` parameter specifies the database bind function.
- The `SLAPI_PLUGIN_POST_UNBIND_FN` parameter specifies the post-operation bind function.

You set these parameters to the names of your functions by calling `slapi_pblock_set()`.

Your plug-in functions should return 0 if successful. If the pre-operation function returns a non-zero value, the database unbind function and the post-operation unbind function are never called.

Processing an LDAP Search Operation

The server processes an LDAP search operation in two stages:

- First, the server gets a list of candidate entries, using an index (if applicable).

  For example, for a search filter that finds entries where “mail=a*”, the server checks the index for the “mail” attribute (if the index exists), finds the keys that start with “a”, and generates a list of matching entries.

  If no applicable index exists, all entries are considered to be candidates.

  To get the list of candidates, the server calls the database search function, which is specified by the `SLAPI_PLUGIN_DB_SEARCH_FN` parameter. For details, see “Getting the List of Candidates” on page 102.

- Next, the server iterates through each candidate in the list and determines if the candidate matches the search criteria.

  If an entry matches the criteria, the server sends the entry to the client.

  To check each candidate, the server calls the database "next candidate" function (specified by the `SLAPI_PLUGIN_DB_NEXT_SEARCH_ENTRY_FN` parameter) for each candidate in the list. For details, see "Iterating through Candidates" on page 104.

The rest of this section explains these stages in more detail.

Getting the List of Candidates

In the parameter block, the `SLAPI_PLUGIN_DB_SEARCH_FN` parameter specifies the database search function.

When the Directory Server receives an LDAP search request, the front-end gets information about the search (such as the scope and base DN). The front-end normalizes the base DN (by calling the `slapi_dn_normalize()` function) and determines if the base DN identifies a DSA-specific entry (DSE). If so, the front-end handles the search request directly and does not pass it to the database search function.

If the base DN is not a DSE, the front-end finds the back-end that services the suffix specified in the base DN. The front-end then passes the search criteria to the search function for that back-end.

The front-end makes this information available to pre-operation, database, and post-operation plug-in functions in the form of parameters in a parameter block. .

Parameter ID	Data Type	Description
SLAPI_SEARCH_TARGET	char *	DN of the base entry in the search operation (the starting point of the search).
SLAPI_SEARCH_SCOPE	int	The scope of the search. The scope can be one of the following values: • LDAP_SCOPE_BASE • LDAP_SCOPE_ONELEVEL • LDAP_SCOPE_SUBTREE
SLAPI_SEARCH_DEREF	int	Method for handling aliases in a search. This method can be one of the following values: • LDAP_DEREF_NEVER • LDAP_DEREF_SEARCHING • LDAP_DEREF_FINDING • LDAP_DEREF_ALWAYS
SLAPI_SEARCH_SIZELIMIT	int	Maximum number of entries to return in the search results.
SLAPI_SEARCH_TIMELIMIT	int	Maximum amount of time (in seconds) allowed for the search operation.
SLAPI_SEARCH_FILTER	Slapi_Filter *	Slapi_Filter struct (an opaque data structure) representing the filter to be used in the search.
SLAPI_SEARCH_STRFILTER	char *	String representation of the filter to be used in the search.
SLAPI_SEARCH_ATTRS	char **	Array of attribute types to be returned in the search results.
SLAPI_SEARCH_ATTRSONLY	int	Specifies whether the search results return attribute types only or attribute types and values. (0 means return both attributes and values; 1 means return attribute types only)

Your search function should return 0 if successful. Call the `slapi_pblock_set()` function to assign the set of search results to the `SLAPI_SEARCH_RESULT_SET` parameter in the parameter block.

The front-end then uses this function in conjunction with the "next entry" function (see "Iterating through Candidates" on page 104) to iterate through the result set. The front-end sends each result back to the client and continues updates the `SLAPI_NENTRIES` parameter with the current number of entries sent back to the client.

If a result is actually a referral, the front-end sends the referral back to the client and updates the `SLAPI_SEARCH_REFERRALS` parameter with the list of referral URLs.

Finally, after sending the last entry to the client, the front-end sends an LDAP result message specifying the number of entries found.

Iterating through Candidates

In the parameter block, the `SLAPI_PLUGIN_DB_NEXT_SEARCH_ENTRY_FN` parameter specifies the database function used to iterate through the list of candidates and determine if each candidate matches the search criteria.

In addition to the parameters specified in "Processing an LDAP Search Operation" on page 101, the "next entry" function has access to the following parameters (which are set by the front-end and the back-end during the course of executing a search operation): .

Parameter ID	Data Type	Description
`SLAPI_SEARCH_RESULT_SET`	`void *`	Set of search results.

Parameter ID	Data Type	Description
SLAPI_SEARCH_RESULT_ENTRY	void *	Entry returned from iterating through the results set. This "next entry" function actually sets this parameter.
SLAPI_SEARCH_RESULT_ENTRY_EXT	void *	(Netscape Directory Server 4.0) Reserved for future use. The context identifying the last result sent in the results set. This "next entry" function actually sets this parameter.
SLAPI_NENTRIES	int	Number of search results found
SLAPI_SEARCH_REFERRALS	struct berval **	Array of the URLs to other LDAP servers that the current server is referring the client to

The "next entry" function should get the next result specified in the set of results in the parameter SLAPI_SEARCH_RESULT_SET. The function should set this next entry as the value of the SLAPI_SEARCH_RESULT_ENTRY parameter in the parameter block, and the "next entry" function should return 0 if successful.

The "next entry" function should set the SLAPI_SEARCH_RESULT_ENTRY parameter to NULL and return -1 if one of the following situations occurs:

- the operation is abandoned (you can check this by calling the slapi_op_abandoned() function)
- the time limit has been exceeded
- the maximum number of entries has been exceeded

If no more entries exist in the set of results, the "next entry" function should set the SLAPI_SEARCH_RESULT_ENTRY parameter to NULL and return 0.

Processing an LDAP Compare Operation

In the parameter block, the SLAPI_PLUGIN_DB_COMPARE_FN parameter specifies the database compare function.

When the Directory Server receives an LDAP compare request from a client, the front-end gets the DN of the entry being compared and the attribute and value being used in the comparison.

The front-end makes this information available to pre-operation, database, and post-operation plug-in functions in the form of parameters in a parameter block.

Parameter ID	Data Type	Description
SLAPI_COMPARE_TARGET	char *	DN of the entry to be compared.
SLAPI_COMPARE_TYPE	char *	Attribute type to use in the comparison.
SLAPI_COMPARE_VALUE	struct berval *	Attribute value to use in the comparison

The compare function should call `slapi_send_ldap_result()` to send LDAP_COMPARE_TRUE if the specified value is equal to the value of the entry's attribute or LDAP_COMPARE_FALSE if the values are not equal.

If successful, the compare function should return 0. If an error occurs (for example, if the specified attribute doesn't exist), the compare function should call `slapi_send_ldap_result()` to send an LDAP error code and should return 1.

Processing an LDAP Add Operation

In the parameter block, the SLAPI_PLUGIN_DB_ADD_FN parameter specifies the database add function.

When the Directory Server receives an LDAP add request from a client, the front-end normalizes the DN of the new entry. The front-end makes this information available to pre-operation, database, and post-operation plug-in functions in the form of parameters in a parameter block.

Parameter ID	Data Type	Description
SLAPI_ADD_TARGET	char *	DN of the entry to be added.
SLAPI_ADD_ENTRY	Slapi_Entry *	The entry to be added (specified as the opaque `Slapi_Entry` datatype).

The add function should check the following:

- If the operation has been abandoned, the function should return -1. (You do not need to call `slapi_send_ldap_result()` to send an LDAP error code to the client. According to the LDAP protocol, the client does not expect a server response after an operation is abandoned.)
- If the entry already exists in the database, the function should call `slapi_send_ldap_result()` to send an LDAP error code `LDAP_ALREADY_EXISTS` and should return -1.
- If the parent entry (or the closest matching entry) is a referral entry (an entry with the object class "ref") and no manageDSAIT control is included with the request, the function should call `slapi_send_ldap_referral()` to send a referral and return -1.

  To determine if a manageDSAIT control is present, call `slapi_pblock_get()` to get the value of the `SLAPI_MANAGEDSAIT` parameter. If the value is 1, the control is included in the request. If 0, the control is not included in the request.
- If the parent entry does not exist, the function should call `slapi_send_ldap_result()` to send an LDAP error code `LDAP_NO_SUCH_OBJECT` and return -1.
- If the entry is not schema-compliant (call `slapi_entry_schema_check()` to determine this), the function should call `slapi_send_ldap_result()` to send the LDAP error code `LDAP_OBJECT_CLASS_VIOLATION` and should return -1.
- If the requestor does not have permission to add the entry (call `slapi_access_allowed()` to determine this), the function should call `slapi_send_ldap_result()` to send the LDAP error code `LDAP_INSUFFICIENT_ACCESS` and should return -1.

  You should also verify that the ACI syntax for the entry is correct (call `slapi_acl_check_mods()` to determine this).

If the add function is successful, the function should call `slapi_send_ldap_result()` to send an `LDAP_SUCCESS` code back to the client and should return 0.

Processing an LDAP Modify Operation

In the parameter block, the SLAPI_PLUGIN_DB_MODIFY_FN parameter specifies the database modify function.

When the Directory Server receives an LDAP modify request from a client, the front-end gets the DN of the entry to be modified and the modifications to be made. The front-end makes this information available to pre-operation, database, and post-operation plug-in functions in the form of parameters in a parameter block.

Parameter ID	Data Type	Description
SLAPI_MODIFY_TARGET	char *	DN of the entry to be modified.
SLAPI_MODIFY_MODS	LDAPMod **	A NULL-terminated array of LDAPMod structures, which represent the modifications to be performed on the entry.

The modify function should check the following:

- If the operation has been abandoned, the function should return -1. (You do not need to call slapi_send_ldap_result() to send an LDAP error code to the client. According to the LDAP protocol, the client does not expect a server response after an operation is abandoned.)
- If the entry is a referral entry (an entry with the object class “ref”) and no manageDSAIT control is included with the request, the function should call slapi_send_ldap_referral() to send a referral and return -1.

  To determine if a manageDSAIT control is present, call slapi_pblock_get() to get the value of the SLAPI_MANAGEDSAIT parameter. If the value is 1, the control is included in the request. If 0, the control is not included in the request.
- If the entry does not exist, check the following:
  - If the closest matching entry is a referral entry and if no manageDSAIT control is included in the request, the function should call slapi_send_ldap_referral() to send a referral and return -1.
  - Otherwise, the function should call slapi_send_ldap_result() to send an LDAP error code LDAP_NO_SUCH_OBJECT and return -1.

- If the entry is not schema-compliant (call `slapi_entry_schema_check()` to determine this), the function should call `slapi_send_ldap_result()` to send the LDAP error code `LDAP_OBJECT_CLASS_VIOLATION` and should return -1.
- If the RDN of the entry contains attribute values that are not part of the entry (for example, if the RDN is “uid=bjensen” but the entry has no uid value or has a different uid value), the function should call `slapi_send_ldap_result()` to send the LDAP error code `LDAP_NOT_ALLOWED_ON_RDN` and should return -1.
- If the requestor does not have permission to modify the entry (call `slapi_access_allowed()` to determine this), the function should call `slapi_send_ldap_result()` to send the LDAP error code `LDAP_INSUFFICIENT_ACCESS` and should return -1.

  You should also verify that the ACI syntax for the entry is correct (call `slapi_acl_check_mods()` to determine this).

If the modify function is successful, the function should call `slapi_send_ldap_result()` to send an `LDAP_SUCCESS` code back to the client and should return 0.

Processing an LDAP Modify RDN Operation

In the parameter block, the `SLAPI_PLUGIN_DB_MODRDN_FN` parameter specifies the database modify RDN function.

When the Directory Server receives an LDAP modify RDN request from a client, the front-end gets the original DN of the entry, the new RDN, and (if the entry is moving to a different location in the directory tree) the DN of the new parent of the entry.

The front-end makes this information available to pre-operation, database, and post-operation plug-in functions in the form of parameters in a parameter block.

Parameter ID	Data Type	Description
SLAPI_MODRDN_TARGET	char *	DN of the entry that you want to rename.
SLAPI_MODRDN_NEWRDN	char *	New RDN to assign to the entry.
SLAPI_MODRDN_DELOLDRDN	int	Specifies whether or not you want to deleted the old RDN. (0 means don't delete the old RDN; 1 means delete the old RDN)
SLAPI_MODRDN_NEWSUPERIOR	char *	DN of the new parent of the entry, if the entry is being moved to a new location in the directory tree.

The modify RDN function should check the following:

- If the operation has been abandoned, the function should return -1. (You do not need to call `slapi_send_ldap_result()` to send an LDAP error code to the client. According to the LDAP protocol, the client does not expect a server response after an operation is abandoned.)
- If the entry is a referral entry (an entry with the object class "ref") and no manageDSAIT control is included with the request, the function should call `slapi_send_ldap_referral()` to send a referral and return -1.

  To determine if a manageDSAIT control is present, call `slapi_pblock_get()` to get the value of the `SLAPI_MANAGEDSAIT` parameter. If the value is 1, the control is included in the request. If 0, the control is not included in the request.
- If the entry does not exist, check the following:
  - If the closest matching entry is a referral entry and if no manageDSAIT control is included in the request, the function should call `slapi_send_ldap_referral()` to send a referral and return -1.
  - Otherwise, the function should call `slapi_send_ldap_result()` to send an LDAP error code `LDAP_NO_SUCH_OBJECT` and return -1.

- If the entry is not schema-compliant (call `slapi_entry_schema_check()` to determine this), the function should call `slapi_send_ldap_result()` to send the LDAP error code `LDAP_OBJECT_CLASS_VIOLATION` and should return -1.
- If the RDN of the entry contains attribute values that are not part of the entry (for example, if the RDN is “uid=bjensen” but the entry has no uid value or has a different uid value), the function should call `slapi_send_ldap_result()` to send the LDAP error code `LDAP_NOT_ALLOWED_ON_RDN` and should return -1.
- If the requestor does not have permission to modify the entry (call `slapi_access_allowed()` to determine this), the function should call `slapi_send_ldap_result()` to send the LDAP error code `LDAP_INSUFFICIENT_ACCESS` and should return -1.

  You should also verify that the ACI syntax for the entry is correct (call `slapi_acl_check_mods()` to determine this).

If the modify RDN function is successful, the function should call `slapi_send_ldap_result()` to send an `LDAP_SUCCESS` code back to the client and should return 0.

Processing an LDAP Delete Operation

In the parameter block, the `SLAPI_PLUGIN_DB_DELETE_FN` parameter specifies the database delete function.

When the Directory Server receives an LDAP delete request from a client, the front-end gets the DN of the entry to be removed from the directory. The front-end makes this information available to pre-operation, database, and post-operation plug-in functions in the form of parameters in a parameter block.

Parameter ID	Data Type	Description
`SLAPI_DELETE_TARGET`	`char *`	DN of the entry to delete.

If the delete function is successful, it should return 0.

Processing an LDAP Abandon Operation

In the parameter block, the SLAPI_PLUGIN_DB_ABANDON_FN parameter specifies the database abandon function.

When the Directory Server receives an LDAP abandon request from a client, the front-end gets the message ID of the operation that should be abandoned. The front-end makes this information available to pre-operation, database, and post-operation plug-in functions in the form of parameters in a parameter block.

Parameter ID	Data Type	Description
SLAPI_ABANDON_MSGID	unsigned long	Message ID of the operation to abandon.

Chapter

8

Defining Functions for Database Operations

This chapter explains how to write functions that perform database operations, such as exporting the directory, importing LDIF files, archiving, and restoring database archives.

In general, these functions must comply with the prototype specified in “Working with Parameter Blocks” on page 30 (in other words, the functions should pass a single `Slapi_PBlock` argument).

You can define plug-in functions to do the following:

- Importing an LDIF File into the Database
- Exporting the Database to an LDIF File
- Saving the Database as an Archive
- Restoring the Database from an Archive
- Reporting the Size of the Database
- Testing the Database
- Generating Indexes for the Database

For information on the database plug-in functions that perform LDAP operations (such as adding entries to the directory database or searching the directory database), see Chapter 7, “Defining Functions for LDAP Operations”.

Importing an LDIF File into the Database

When the Directory Server receives a request to import an LDIF file into the database (for example, from the Administrator Server or from the `ldif2db` command-line utility), the front-end calls the back-end database function for importing LDIF files. The `SLAPI_PLUGIN_DB_LDIF2DB_FN` parameter in the parameter block specifies this function.

The front-end makes the following information available in the form of parameters in a parameter block.

Parameter ID	Data Type	Description
`SLAPI_LDIF2DB_FILE`	`char *`	LDIF file that needs to be imported into the database.
`SLAPI_LDIF2DB_REPLACE_DSE_FN`	`void *`	Function that replaces a DSE (DSA-specific entry) in the front-end. DSA is another term for the Directory Server.
`SLAPI_LDIF2DB_WRITE_DSE_FILE`	`void *`	Function that writes a DSE (DSA-specific entry) in the front-end to a file. DSA is another term for the Directory Server.
`SLAPI_LDIF2DB_NOATTRINDEXES`	`int`	(Netscape Directory Server 4.0 release only) If 1, the database should not be indexed when the database is created. If 0, the import function should automatically generate database indexes.
`SLAPI_LDIF2DB_INCLUDE`	`char **`	(Netscape Directory Server 4.0 release only) An array of the suffixes or DNs identifying the entries in the LDIF file to be included in the database.
`SLAPI_LDIF2DB_EXCLUDE`	`char **`	(Netscape Directory Server 4.0 release only) An array of the suffixes or DNs identifying the entries in the LDIF file to be excluded from the database.

The LDIF import function should parse the LDIF entries in the file specified by the SLAPI_LDIF2DB_FILE parameter and should create corresponding entries in the database.

The array of DNs and suffixes in the SLAPI_LDIF2DB_INCLUDE parameter and the SLAPI_LDIF2DB_EXCLUDE parameter identify entries in the LDIF file that should be included or excluded when creating the database. (This feature is available in the Netscape Directory Server 4.0 release and not in previous releases.)

For example, if the SLAPI_LDIF2DB_INCLUDE parameter includes the suffix "o=Airius.com", the import function should include entries in the LDIF file that have DNs ending with "o=Airius.com". On the other hand, if the SLAPI_LDIF2DB_EXCLUDE parameter includes the suffix "o=AceIndustry.com", the import function should exclude entries in the LDIF file that have DNs ending with "o=AceIndustry.com".

The LDIF import function should also check the value of the SLAPI_LDIF2DB_NOATTRINDEXES parameter. If the value of this parameter is 0, the function should automatically generate indexes when creating the database. If the value is 1, the function should not automatically generate the indexes. (This feature is available in the Netscape Directory Server 4.0 release and not in previous releases.)

In some instances, the LDIF file may contain a DSE. A DSE is a DSA-specific entry (DSA is the X.500 term for Directory Server). In the Netscape Directory Server, DSEs are stored in LDIF format in the DSE file:

```
<server_root>/<server_id>/config/dse.ldif
```

(For example, `/usr/netscape/suitespot/slapd-myhost/config/dse.ldif`).

When the Directory Server starts up, it reads the DSEs from this file and loads them into the front-end.

Because the imported LDIF file may contain DSEs, you should check for DSEs in your import function. If a DSE is specified in the LDIF file, you should call the DSE replacement function (specified by SLAPI_LDIF2DB_REPLACE_DSE_FN in the parameter block) to replace the existing DSEs in the front-end.

At the end of your database import function, you also need to call the DSE file write function (specified by SLAPI_LDIF2DB_WRITE_DSE_FILE in the parameter block). Calling this function will write the DSEs out to the DSE file.

Exporting the Database to an LDIF File

When the Directory Server receives a request to export the database to an LDIF file (for example, from the Administrator Server or from the `db2ldif` command-line utility), the front-end calls the back-end database function for importing LDIF files. The `SLAPI_PLUGIN_DB_DB2LDIF_FN` parameter in the parameter block specifies this function.

The front-end makes the following information available in the form of parameters in a parameter block.

Parameter ID	Data Type	Description
`SLAPI_DB2LDIF_PRINTKEY`	`int`	Specifies whether or not the database keys should be printed out as well. • If 1, you should include the database key for each entry. • If 0, you do not need to include the database key for each entry.
`SLAPI_DB2LDIF_PRINT_DSE_TREE_FN`	`void *`	Function for printing the front-end DSEs in LDIF format.

The LDIF export function should generate LDIF entries corresponding to entries in the database and should store the LDIF entries in the file.

The `SLAPI_DB2LDIF_PRINT_DSE_TREE_FN` parameter specifies the function that prints out the DSEs in the front end. A DSE is a DSA-specific entry (DSA is the X.500 name for the Directory Server). In your database export function, you should call the DSE print function if you want the DSE entries to appear in the LDIF file.

The `SLAPI_DB2LDIF_PRINTKEY` parameter specifies whether or not the database keys should be printed with each entry. If this parameter is 1, you should print out the database keys for each entry. If this parameter is 0, you do not need to print out the keys.

Saving the Database as an Archive

When the Directory Server receives a request to archive the database (for example, from the Administrator Server or from the `db2bak` command-line utility), the front-end calls the back-end database function for archiving the database. The `SLAPI_PLUGIN_DB_DB2ARCHIVE_FN` parameter in the parameter block specifies this function.

The front-end makes the following information available in the form of parameters in a parameter block.

Parameter ID	Data Type	Description
`SLAPI_SEQ_VAL`	`char *`	Specifies the directory in which you want to store the archive.

The database archive function should create an archive of the database in the directory specified by the `SLAPI_SEQ_VAL` parameter.

Restoring the Database from an Archive

When the Directory Server receives a request to restore the database from an archive (for example, from the Administrator Server or from the `bak2db` command-line utility), the front-end calls the back-end database function for restoring the database. The `SLAPI_PLUGIN_DB_ARCHIVE2DB_FN` parameter in the parameter block specifies this function.

The front-end makes the following information available in the form of parameters in a parameter block.

Parameter ID	Data Type	Description
`SLAPI_SEQ_VAL`	`char *`	Specifies the directory containing the archive that you want to restore.

The database restore function should replace the database with the archive in the directory specified by the `SLAPI_SEQ_VAL` parameter.

Reporting the Size of the Database

In some cases, the front-end may need to check the size of the database. The SLAPI_PLUGIN_DB_SIZE_FN parameter in the parameter block specifies the function that calculcates the database size and sets in the SLAPI_DBSIZE parameter.

Parameter ID	Data Type	Description
SLAPI_DBSIZE	int	Specifies the size of the database.

Testing the Database

The command-line options for ns-slapd, the Directory Server executable, includes the dbtest option, which allows you to test the database. If you run the command with this option, the Directory Server executes the function specified by the SLAPI_PLUGIN_DB_TEST_FN parameter in the parameter block.

You can write your own test function for your database and register the function as the database test function. For example, you might write a function that printed status messages to standard output.

Generating Indexes for the Database

This feature is available in the Netscape Directory Server 4.0 release but is not available in earlier releases.

When the Directory Server receives a request to generate indexes for the database (for example, from the Administrator Server or from the db2index command-line utility), the front-end calls the back-end database function for indexing the database. The SLAPI_PLUGIN_DB_DB2INDEX_FN parameter in the parameter block specifies this function.

The front-end makes the following information available in the form of parameters in a parameter block.

Parameter ID	Data Type	Description
`SLAPI_DB2INDEX_ATTRS`	`char **`	Specifies a NULL-terminated array of the attribute types that you want indexed.

The database index function should create indexes of the database for the attributes specified by the `SLAPI_DB2INDEX_ATTRS` parameter.

Chapter

9

Defining Functions for Authentication

This chapter explains how to write a plug-in function to bypass or replace the standard function for authentication with your own function.

Information on authentication with the Netscape Directory Server is organized in the following sections:

Understanding Authentication Methods

Two methods that you can use to authenticate clients are simple authentication and SASL authentication:

- Simple authentication is described in RFC 2251, which you can find at this location:

  ```
  http://ds.internic.net/rfc/rfc2251.txt
  ```

  Simple authentication provides minimal facilities for authentication. In the simple authentication method, clients send a DN and password to the server for authentication. The server compares the password sent by the client against the password stored in the client's directory entry.

- Simple Authentication and Security Layer (SASL) is described in RFC 2222, which you can find at this location:

  ```
  http://ds.internic.net/rfc/rfc2222.txt
  ```

  SASL provides the means to use mechanisms other than simple authentication and SSL to authenticate to the Netscape Directory Server.

How the Directory Server Identifies Clients

The server keeps track of the identity of the LDAP client through the `SLAPI_CONN_DN` and `SLAPI_CONN_AUTHTYPE` parameters.

During an LDAP bind operation, the server authenticates the user and puts the DN and authenticated method in the `SLAPI_CONN_DN` and `SLAPI_CONN_AUTHTYPE` parameters.

When an authenticated client requests the server to perform an LDAP operation, the server checks the DN in the `SLAPI_CONN_DN` parameter to determine if the client has the appropriate access rights.

How the Authentication Process Works

When the Netscape Directory Server receives an LDAP bind request from a client, it processes the request in the following steps:

1. The server parses the LDAP bind request and retrieves the following information:

   - The DN that the client is attempting to authenticate as

- The method of authentication used
- Any credentials (such as a password) included in the request

If the method of authentication is `LDAP_AUTH_SASL` (SASL authentication), the server also retrieves the name of the SASL mechanism used from the LDAP bind request.

2. The server normalizes the DN retrieved from the request. (See the `slapi_dn_normalize()` function for more information on normalized DNs.)

3. The server retrieves any LDAP v3 controls included with the LDAP bind request.

4. If the method of authentication is `LDAP_AUTH_SASL` (SASL authentication), the server determines whether or not the SASL mechanism (specified in the request) is supported.

   If the SASL mechanism is not supported by the server, the server sends an `LDAP_AUTH_METHOD_NOT_SUPPORTED` result code back to the client and ends the processing of the bind request.

5. If the method of authentication is `LDAP_AUTH_SIMPLE` (simple authentication), the server checks if the DN is an empty string or if there are no credentials.

   If the DN is an empty string, if the DN is not specified, or if no credentials are specified, the server assumes that the client is binding anonymously and sends an `LDAP_SUCCESS` result code back to the client.

   The DN and authentication method for the connection (which are used to determine access rights for all operations performed through the connection) are left as NULL and `SLAPD_AUTH_NONE`, respectively.

6. If the DN specified in the request is not served by this directory server (for example, if the DN is `"uid=moxcross, o=kccorp.com"` and the directory root of the server is `"o=Airius.com"`), the server sends one of the following two results back to the client and ends the processing of the bind request:

- If the server is configured with a default referral (an LDAP URL identifying an LDAP server that handles referrals), the server sends an `LDAP_REFERRAL` result code back to the client (`LDAP_PARTIAL_RESULTS` if the client only supports the LDAP v2 protocol).
- If the server is not configured with a default referral, the server sends an `LDAP_NO_SUCH_OBJECT` result code back to the client.

7. The server puts the information from the bind request into the parameter block:
   - `SLAPI_BIND_TARGET` is set to the DN that the client is authenticating as.
   - `SLAPI_BIND_METHOD` is set to the authentication method (for example, `LDAP_AUTH_SIMPLE` or `LDAP_AUTH_SASL`).
   - `SLAPI_BIND_CREDENTIALS` is set to the credentials (for example, the password) included in the request.
   - `SLAPI_BIND_SASLMECHANISM` (if the authentication method is `LDAP_AUTH_SASL`) is set to the name of the SASL mechanism that the client is using for authentication.
8. If the DN is the root DN or the update DN (the DN of the master entity responsible for replicating the directory), the server authenticates the client.
   - If the credentials are correct, the server sets the `SLAPI_CONN_DN` parameter to the DN and the `SLAPI_CONN_AUTHTYPE` parameter to `LDAP_AUTH_SIMPLE`. The server sends an `LDAP_SUCCESS` result code back to the client and ends the processing of the bind request.
   - If the credentials are incorrect, the server sends an `LDAP_INVALID_CREDENTIALS` result code back to the client and ends the processing of the bind request.
9. The server calls any pre-operation bind plug-in functions. If the function returns a non-zero value, the server ends the processing of the bind request.

If you are writing your own plug-in function to handle authentication, you should return a non-zero value so that the server does not attempt to continue processing the bind request.

10. The server calls the database bind function. If the function returns a non-zero value, the server ends the processing of the bind request and sends the appropriate result code back to the client.

11. If the database bind function succeeds, the server sets the `SLAPI_CONN_DN` parameter to the DN and the `SLAPI_CONN_AUTHTYPE` parameter to the authentication method.

12. The server sends an `LDAP_SUCCESS` result code back to the client and ends the processing of the bind request.

    If the client's password has expired or is expiring, the server includes a "password expired" control (with the OID 2.16.840.1.113730.3.4.4) or a "password expiring" control (with the OID 2.16.840.1.113730.3.4.5) with the result sent to the client.

Writing Your Own Authentication Plug-in

There are two types of situations in which you may want to write and implement your own function for authentication:

- You want to replace the standard means of authentication with you own function.

  In this type of situation, you can write a pre-operation bind plug-in function (a function that the server calls before processing an LDAP bind request) that performs the authentication and bypasses the default database bind function.

  For details, see "Writing a Pre-Operation Bind Plug-in" on page 126.

- You are integrating the Netscape Directory Server with your own back-end database and need to implement your own bind function.

  In this type of situation, you want to replace all the default database functions with your own functions. (In other words, you are writing your own back-end to the Directory Server.) You need to write a database bind function to authenticate clients.

  For details, see "Writing a Database Bind Plug-in" on page 137.

Writing a Pre-Operation Bind Plug-in

You can define your own pre-operation bind plug-in function to authenticate LDAP clients. The server will call your function during the authentication process (see Step 9). Your function should return a non-zero value to bypass the default database bind function and the post-operation bind functions.

Note that this means that Step 10 through Step 12 are skipped. Your pre-operation plug-in function is responsible for sending the result code to the client and for setting the DN and authentication method for the connection.

Figure 9.1 summarizes the process of using a pre-operation bind plug-in function to authenticate LDAP clients to the Directory Server.

Figure 9.1 Using a pre-operation bind plug-in function to handle authentication.

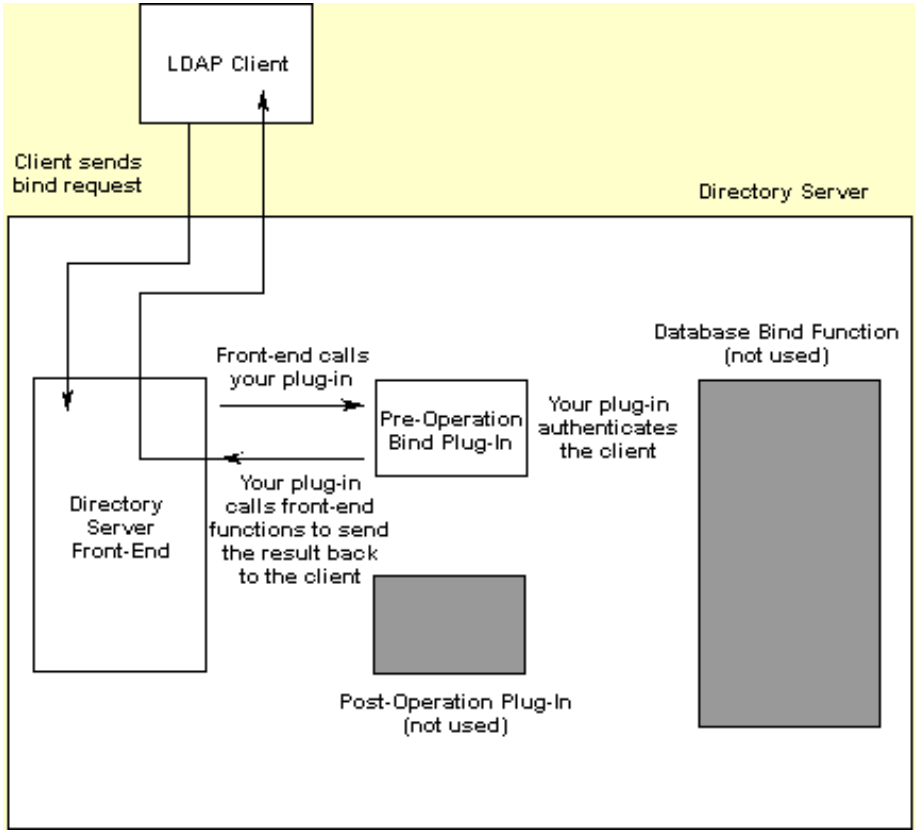
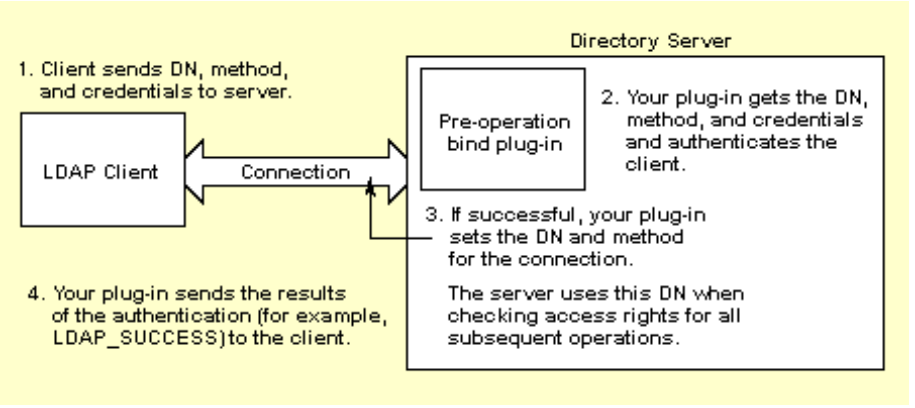


Figure 9.2 illustrates the steps that your pre-operation bind plug-in function must take to authenticate LDAP clients to the Directory Server.

Figure 9.2 How your pre-operation bind plug-in function can authenticate LDAP clients.



Defining Your Authentication Function

The following sections cover guidelines that you can follow when defining your authentication function:

- “Getting and Checking the Bind Parameters” on page 127
- “Getting the Entry and Checking the Credentials” on page 128
- “What to Do If Authentication Fails” on page 129
- “What to Do If Authentication Succeeds” on page 129

Getting and Checking the Bind Parameters

Call the `slapi_pblock_get()` function to get the values of the following parameters:

- `SLAPI_BIND_TARGET` (a string value specifying the DN that the client is attempting to authenticate as)
- `SLAPI_BIND_METHOD` (an integer value specifying the authentication method, such as `LDAP_AUTH_SIMPLE` or `LDAP_AUTH_SASL`)

- SLAPI_BIND_CREDENTIALS (a berval structure containing the credentials sent by the client)

If you plan to support authentication through SASL mechanisms, you should also get the value of the SLAPI_BIND_SASLMECHANISM parameter (a string value specifying the name of the SASL mechanism to use for authentication).

Make sure to check the following:

- Determine if the client is requesting to bind as an anonymous user.

  If the SLAPI_BIND_METHOD parameter is LDAP_AUTH_SIMPLE and the SLAPI_BIND_CREDENTIALS parameter is empty or NULL, the client is attempting to bind anonymously.

  Call slapi_send_ldap_result() to send the LDAP result code LDAP_SUCCESS back to the client.

- If the SLAPI_BIND_METHOD parameter specifies a method that you do not recognize or support, call slapi_send_ldap_result() to send an LDAP_STRONG_AUTH_NOT_SUPPORTED result code back to the client.

In both cases, return a non-zero value to prevent the server from calling the default database function for authentication.

Getting the Entry and Checking the Credentials

Get the entry for the DN specified by the SLAPI_BIND_TARGET parameter and compare the credentials in the SLAPI_BIND_CREDENTIALS parameter against the known credentials for that entry.

By default, the Netscape Directory Server uses the userpassword attribute to store the credentials for an entry. The server encrypts the password using the scheme specified in the passwdhash directive of the slapd.conf configuration file. The scheme can be crypt or sha or "" (for cleartext).

If you need to compare the client's credentials against the value of the userpassword attribute, you can call the slapi_pw_find() function. This function determines which password scheme was used to store the password and uses the appropriate comparison function to compare a given value against the encrypted value of the userpassword attribute.

What to Do If Authentication Fails

If authentication fails, send one of the following result codes back to the client:

- If no entry matches the DN specified by the client, send an `LDAP_NO_SUCH_OBJECT` result code back to the client.

  When calling the `slapi_send_ldap_result()` function to send the result code back to the client, specify the closest matching DN as the `matched` argument.

- If the client fails to provide the necessary credentials or if credentials cannot be found in the entry, send an `LDAP_INAPPROPRIATE_AUTH` result code back to the client.

- If the credentials specified by the client do not match the credentials found in the entry, send an `LDAP_INVALID_CREDENTIALS` result code back to the client.

- If a general error occurs, send an `LDAP_OPERATIONS_ERROR` result code back to the client.

You do not need to set any values for the `SLAPI_CONN_DN` parameter and the `SLAPI_CONN_AUTHTYPE` parameter. By default, these parameters are set to NULL and `LDAP_AUTH_NONE`, which indicate that the client has bound anonymously.

What to Do If Authentication Succeeds

If the authentication is successful, your authentication function should do the following:

- Call `slapi_pblock_set()` to set the values of the `SLAPI_CONN_DN` parameter and the `SLAPI_CONN_AUTHTYPE` parameter to the DN and authentication method.

  This sets the DN and authentication method for the connection to the client. (The server uses this DN and method in subsequent operations when checking access rights.)

  You can set `SLAPI_CONN_AUTHTYPE` to one of the following values:

  - `SLAPD_AUTH_NONE` represents no authentication. (The client is binding anonymously.)

- `SLAPD_AUTH_SIMPLE` represents the simple authentication method.
- `SLAPD_AUTH_SSL` represents authentication through SSL.
- `SLAPD_AUTH_SASL` represents SASL authentication.

Note that these values differ from the values in the `SLAPI_BIND_METHOD` parameter. The values listed above are string values defined in the `slapi-plugin.h` header file, whereas the values of the `SLAPI_BIND_METHOD` parameter (such as `LDAP_AUTH_SIMPLE` and `LDAP_AUTH_SASL`) are integer values defined in the `ldap.h` header file.

- If you want to, specify the credentials that you want sent back to the client.

  If the value of the `SLAPI_BIND_METHOD` parameter is `LDAP_AUTH_SASL` and you want to return a set of credentials to the client, call `slapi_pblock_set()` to set the `SLAPI_BIND_RET_SASLCREDS` parameter to the credentials.

- Send the result of the authentication process back to the client.

  Call `slapi_send_ldap_result()` to send an `LDAP_SUCCESS` return code to the client.

Make sure that your function returns a non-zero value to bypass the default database bind function and any post-operation plug-in functions.

Registering the SASL Mechanism

If you are using SASL as the authentication method, you need to register the SASL mechanisms that you plan to use.

In your initialization function (see “Writing an Initialization Function” on page 34), call the `slapi_register_supported_saslmechanism()` function and specify the name of the SASL mechanism. For example:

```
slapi_register_supported_saslmechanism( "babsmechanism" );
```

Note that if you do not register your SASL mechanism, the Directory Server will send an `LDAP_AUTH_METHOD_NOT_SUPPORTED` result code back to the client and will not call your pre-operation bind function.

Example of a Pre-Operation Bind Plug-In

The following sections document an example of a pre-operation bind plug-in that handles authentication:

Example of a Pre-Operation Bind Function

The following is an example of a pre-operation bind function that authenticates clients and bypasses the default database bind function. In this example, the function just compares the client’s credentials against the value of the `userpassword` attribute for the entry.

```
#include <stdio.h>
#include <string.h>
#include "slapi-plugin.h"
/* Pre-operation plug-in function */
int
test_bind( Slapi_PBlock *pb )
{
   char  *dn;
   int   method, rc = LDAP_SUCCESS;
   struct berval*credentials;
   struct berval**pwvals;
   Slapi_PBlock*searchpb = NULL;
   Slapi_Entry*e = NULL;
   Slapi_Entry**entries = NULL;
   Slapi_Attr*attr = NULL;
   /* Log a message to the server error log. */
   slapi_log_error( SLAPI_LOG_PLUGIN, "test_bind",
      "Pre-operation bind function called.\n" );
```

```
   /* Gets parameters available when processing an LDAP bind
      operation. */
   if ( slapi_pblock_get( pb, SLAPI_BIND_TARGET, &dn ) != 0 ||
       slapi_pblock_get( pb, SLAPI_BIND_METHOD, &method ) != 0 ||
       slapi_pblock_get( pb, SLAPI_BIND_CREDENTIALS, &credentials ) != 0
) {
      slapi_log_error( SLAPI_LOG_PLUGIN, "test_bind",
         "Could not get parameters for bind operation\n" );
      slapi_send_ldap_result( pb, LDAP_OPERATIONS_ERROR,
         NULL, NULL, 0, NULL );
      return( 1 );
   }
   /* Check the authentication method */
   switch( method ) {
   case LDAP_AUTH_SIMPLE:
      /* First, get the entry specified by the DN. */
      searchpb = slapi_search_internal( dn, LDAP_SCOPE_BASE,
         "(objectclass=*)", NULL, NULL, 0 );
      if ( searchpb != NULL ) {
         slapi_pblock_get( pb, SLAPI_PLUGIN_INTOP_RESULT, &rc );
         if ( rc == LDAP_SUCCESS ) {
            slapi_pblock_get( searchpb,
SLAPI_PLUGIN_INTOP_SEARCH_ENTRIES,
               &entries );
            if ( entries != NULL && entries[0] != NULL ) {
               e = entries[0];
            } else {
               slapi_log_error( SLAPI_LOG_PLUGIN, "test_bind",
                  "Could not find entry for %s\n", dn );
               rc = LDAP_NO_SUCH_OBJECT;
               break;
            }
```

```
   } else {
      slapi_log_error( SLAPI_LOG_PLUGIN, "test_bind",
         "Could not find entry for %s (error %d)\n", dn, rc );
      break;
   }
} else {
   slapi_log_error( SLAPI_LOG_PLUGIN, "test_bind",
      "Could not search for entry\n" );
   rc = LDAP_OPERATIONS_ERROR;
   break;;
}
/* Next, check the credentials against the userpassword attribute
   of that entry. */
if ( e != NULL ) {
   if ( slapi_entry_attr_find( e, "userpassword", &attr ) != 0 ) {
      slapi_log_error( SLAPI_LOG_PLUGIN, "test_bind",
         "Entry has no userpassword attribute\n" );
      rc = LDAP_INAPPROPRIATE_AUTH;
      break;
   }
   slapi_attr_get_values( attr, &pwvals );
   if ( slapi_pw_find( pwvals, credentials ) != 0 ) {
      slapi_log_error( SLAPI_LOG_PLUGIN, "test_bind",
         "Credentials are not correct for the entry\n" );
      rc = LDAP_INVALID_CREDENTIALS;
      break;
   }
} else {
   /* This should not happen. The previous section of code
      already checks for this case. */
   slapi_log_error( SLAPI_LOG_PLUGIN, "test_bind",
      "Could find entry for %s\n", dn );
```

```
         rc = LDAP_NO_SUCH_OBJECT;
         break;
      }
      /* Set the DN and the authentication method for the connection. */
      if ( slapi_pblock_set( pb, SLAPI_CONN_DN, slapi_ch_strdup( dn ) )
!= 0 ||
         slapi_pblock_set( pb, SLAPI_CONN_AUTHTYPE, SLAPD_AUTH_SIMPLE )
!= 0 ) {
         slapi_log_error( SLAPI_LOG_PLUGIN, "testbind_init",
            "Failed to set DN and auth method for connection\n" );
         rc = LDAP_OPERATIONS_ERROR;
         break;
      }
      /* Send a "success" result code back to the client. */
      slapi_log_error( SLAPI_LOG_PLUGIN, "test_bind", "Authenticated:
%s\n", dn );
      rc = LDAP_SUCCESS;
      break;
   /* If NONE is specified, the client is requesting to bind
anonymously.
      Normally, this case should be handled by the server's front-end
      before it calls this plug-in function.  Just in case this does
      get through to the plug-in function, you can handle this by
      sending a successful result code back to the client and returning
      1.
    */
   case LDAP_AUTH_NONE:
      slapi_log_error( SLAPI_LOG_PLUGIN, "test_bind", "Authenticating
anonymously\n" );
      rc = LDAP_SUCCESS;
      break;
   /* This plug-in does not support any other method of authentication
*/
   case LDAP_AUTH_SASL:
```

```
   default:
      slapi_log_error( SLAPI_LOG_PLUGIN, "test_bind",
         "Unsupported authentication method requested: %d\n",
         method );
      rc = LDAP_AUTH_METHOD_NOT_SUPPORTED;
      break;
   }
   if ( searchpb ) {
      slapi_free_search_results_internal( searchpb );
      slapi_ch_free( ( void ** )&searchpb );
   }
   slapi_send_ldap_result( pb, rc, NULL, NULL, 0, NULL );
   return( 1 );
}
```

Example of an Initialization Function

To initialize your plug-in, write an initialization function to do the following:

- Call `slapi_pblock_set()` to set the `SLAPI_PLUGIN_PRE_BIND_FN` parameter to the name of your pre-operation bind function. (For details, see “Registering Your Plug-In Functions” on page 36.)
- If you are using SASL as the authentication method, call the `slapi_register_supported_saslmechanism()` function to register your SASL mechanism with the Directory Server.

The following is an example of an initialization function that registers the pre-operation bind function.

```
#include <stdio.h>
#include <string.h>
#include "slapi-plugin.h"
Slapi_PluginDesc bindpdesc = { "test-bind", "Netscape", "0.5",
   "sample bind pre-operation plugin" };
/* Initialization function */
#ifdef _WIN32
```

```
__declspec(dllexport)
#endif
int
testbind_init( Slapi_PBlock *pb )
{
   /* Register the pre-operation bind function and specify
      the server plug-in version. */
   if ( slapi_pblock_set( pb, SLAPI_PLUGIN_VERSION,
      SLAPI_PLUGIN_VERSION_01 ) != 0 ||
      slapi_pblock_set( pb, SLAPI_PLUGIN_DESCRIPTION,
      (void *)&bindpdesc ) != 0 ||
      slapi_pblock_set( pb, SLAPI_PLUGIN_PRE_BIND_FN,
      (void *) test_bind ) != 0 ) {
      slapi_log_error( SLAPI_LOG_PLUGIN, "testbind_init",
         "Failed to set version and function\n" );
      return( -1 );
   }
   return( 0 );
}
```

Editing the slapd.conf File

To register the plug-in, add the `plugin preoperation` directive to your `slapd.conf` file (which is located under the server root directory in `slapd-<server_id>/config`).

For example:

```
plugin preoperation /usr/netscape/suitespot/plugins/slapd/slapi/
examples/libtest-plugin.so testauth_init
```

Note Each pre-operation plug-in is associated with a back-end. Make sure that the `plugin` directive that registers the plug-in is within the database section for that back-end in the `slapd.conf` file. (The `plugin` directive should be after the `database` directive and before the next `database` directive, if any.)

Writing a Database Bind Plug-in

As part of the process of defining a custom back-end to the Netscape Directory Server, you need to define a database bind function to authenticate LDAP clients. The server will call your function during the authentication process (see Step 10).

Unlike the pre-operation bind plug-in function, your function should return 0 if successful. Returning 0 will allow the server to complete Step 11 and Step 12. If authentication is not successful, your function should return the appropriate result code back to the client.

Figure 9.3 summarizes the process of using your database bind plug-in function to authenticate LDAP clients to the Directory Server.

Figure 9.3 Using a database bind plug-in function to handle authentication.

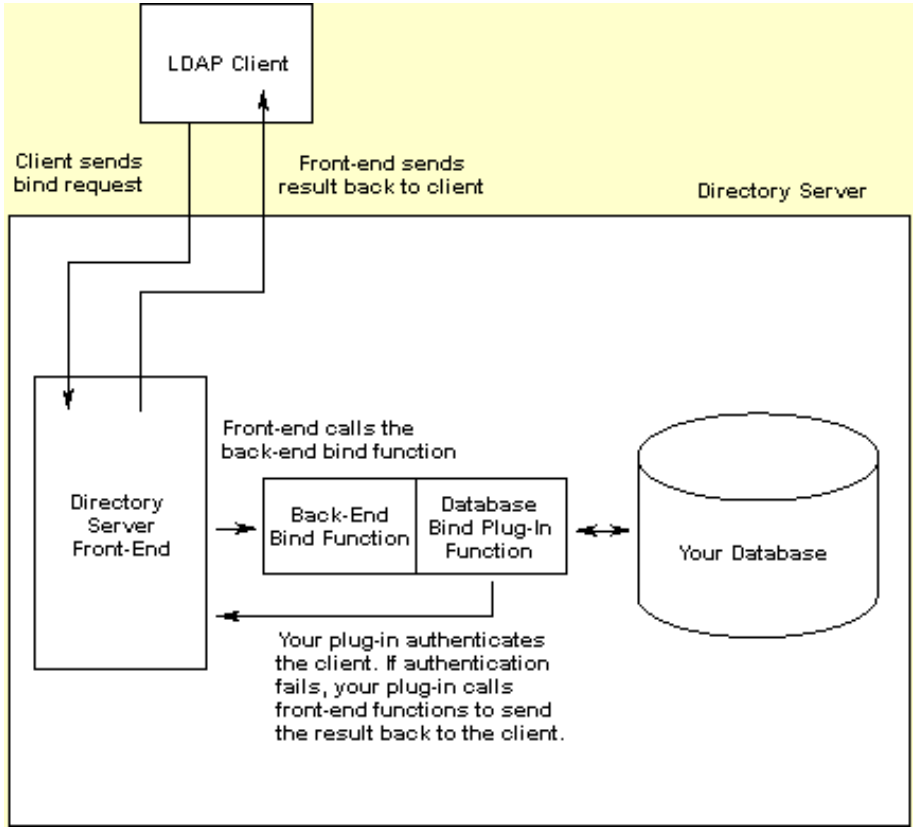
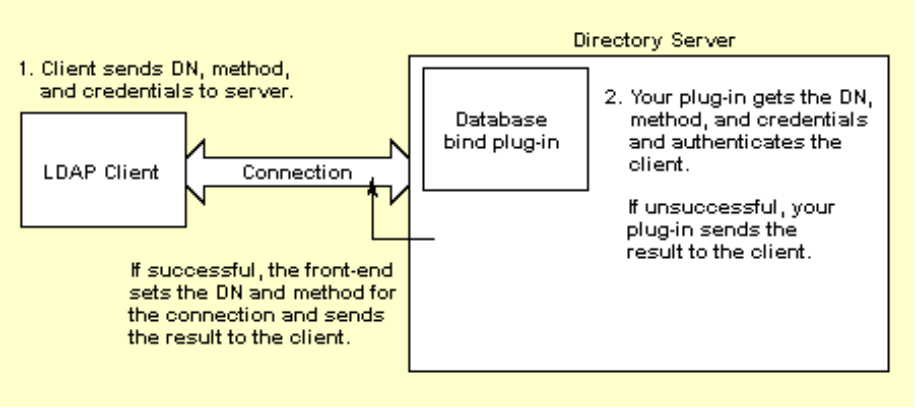


Figure 9.4 illustrates the steps that your server plug-in function must take to authenticate LDAP clients to the Directory Server.

Figure 9.4 How your database bind plug-in function can authenticate LDAP clients.



Defining Your Authentication Function

The following sections cover guidelines that you can follow when defining your authentication function:

- "Getting and Checking the Bind Parameters" on page 127
- "Getting the Entry and Checking the Credentials" on page 128
- "What to Do If Authentication Fails" on page 129
- "What to Do If Authentication Succeeds" on page 129

Getting and Checking the Bind Parameters

Call the `slapi_pblock_get()` function to get the values of the following parameters:

- `SLAPI_BIND_TARGET` (a string value specifying the DN that the client is attempting to authenticate as)
- `SLAPI_BIND_METHOD` (an integer value specifying the authentication method, such as `LDAP_AUTH_SIMPLE` or `LDAP_AUTH_SASL`)

- SLAPI_BIND_CREDENTIALS (a berval structure containing the credentials sent by the client)

If you plan to support authentication through SASL mechanisms, you should also get the value of the SLAPI_BIND_SASLMECHANISM parameter (a string value specifying the name of the SASL mechanism to use for authentication).

Make sure to check the following:

- Determine if the client is requesting to bind as an anonymous user.

  If the SLAPI_BIND_METHOD parameter is LDAP_AUTH_SIMPLE and the SLAPI_BIND_CREDENTIALS parameter is empty or NULL, the client is attempting to bind anonymously.

  Call slapi_send_ldap_result() to send the LDAP result code LDAP_SUCCESS back to the client.

- If the SLAPI_BIND_METHOD parameter specifies a method that you do not recognize or support, call slapi_send_ldap_result() to send an LDAP_AUTH_METHOD_NOT_SUPPORTED result code back to the client.

In both cases, return a non-zero value to indicate to the server that you have already sent a result code to the client. (If you return 0, the server will send an LDAP_SUCCESS result code back to the client.)

Getting the Entry and Checking the Credentials

Get the entry for the DN specified by the SLAPI_BIND_TARGET parameter. If the SLAPI_BIND_METHOD parameter is LDAP_AUTH_SIMPLE, check the credentials in the SLAPI_BIND_CREDENTIALS parameter.

The SLAPI_BIND_CREDENTIALS parameter holds a berval structure that contains the credentials sent by the client.

What to Do If Authentication Fails

If authentication fails, send one of the following result codes back to the client:

- If no entry matches the DN specified by the client, send an LDAP_NO_SUCH_OBJECT result code back to the client.

  When calling the slapi_send_ldap_result() function to send the result code back to the client, specify the closest matching DN as the matched argument.

- If the client fails to provide the necessary credentials or if credentials cannot be found in the entry, send an LDAP_INAPPROPRIATE_AUTH result code back to the client.
- If the credentials specified by the client do not match the credentials found in the entry, send an LDAP_INVALID_CREDENTIALS result code back to the client.
- If a general error occurs, send an LDAP_OPERATIONS_ERROR result code back to the client.

You do not need to set any values for the SLAPI_CONN_DN parameter and the SLAPI_CONN_AUTHTYPE parameter. By default, these parameters are set to NULL and LDAP_AUTH_NONE, which indicate that the client has bound anonymously.

What to Do If Authentication Succeeds

If the authentication is successful, your authentication function should return 0.

You do not need to set the value of the SLAPI_CONN_DN parameter and the SLAPI_CONN_AUTHTYPE parameter; the server does this for you. The server also sends an LDAP_SUCCESS result code to the client.

If the value of the SLAPI_BIND_METHOD parameter is LDAP_AUTH_SASL and you want to return a set of credentials to the client, call slapi_pblock_set() to set the SLAPI_BIND_RET_SASLCREDS parameter to the credentials.

Registering the SASL Mechanism

If you are using SASL as the authentication method, you need to register the SASL mechanisms that you plan to use.

In your initialization function (see “Writing an Initialization Function” on page 34), call the slapi_register_supported_saslmechanism() function and specify the name of the SASL mechanism. For example:

```
slapi_register_supported_saslmechanism( "babsmechanism" );
```

Note that if you do not register your SASL mechanism, the Directory Server will send an `LDAP_AUTH_METHOD_NOT_SUPPORTED` result code back to the client and will not call your pre-operation bind function.

Using SASL with an LDAP Client

If you intend to use SASL as the method for authenticating clients, you need to enable your LDAP clients to use SASL.

In your client, call the `ldap_sasl_bind()` or `ldap_sasl_bind_s()` function to request authentication using SASL. To parse credentials from an asynchronous SASL bind operation, call `ldap_parse_sasl_bind_result()`. These functions are part of the Netscape LDAP C SDK 3.0.

The syntax for these functions are listed below:

```
LDAP_API(int) LDAP_CALL ldap_sasl_bind( LDAP *ld, const char *dn,
   const char *mechanism, struct berval *cred,
   LDAPControl **serverctrls, LDAPControl **clientctrls, int *msgidp );

LDAP_API(int) LDAP_CALL ldap_sasl_bind_s( LDAP *ld, const char *dn,
   const char *mechanism, struct berval *cred,
   LDAPControl **serverctrls, LDAPControl **clientctrls,
   struct berval **servercredp );
```

The parameters are described below:

- `ld` is the connection handle, which is a pointer to the LDAP structure containing information about the connection to the LDAP server.
- `dn` is the distinguished name (DN) that your client is attempting to authenticate as.
- `mechanism` is the name of the SASL mechanism that you want to use for authentication (the mechanism that you register in the initialization function for your server plug-in).
- `cred` is a pointer to the `berval` structure containing the credentials that you want to use for authentication.
- `serverctrls` is a pointer to an array of `LDAPControl` structures representing the LDAP v3 server controls that you want passed to the server for the bind operation.

- `clientctrls` is a pointer to an array of `LDAPControl` structures representing the LDAP v3 client controls applicable to the bind operation.
- `msgidp` is a pointer to the message ID for this bind operation. To check the result of this operation, call the `ldap_result()` function and the `ldap_parse_sasl_bind_result()` function.
- `servercredp` is a pointer to a pointer to the `berval` structure containing any credentials returned by the server. These are the credentials that you set as the value of the `SLAPI_BIND_RET_SASLCREDS` parameter in your pre-operation bind plug-in function.

If you are call the `ldap_sasl_bind()` function, you need to call the `ldap_result()` function to get the result of the authentication attempt:

```
LDAP_API(int) LDAP_CALL ldap_result( LDAP *ld, int msgid, int all,
   struct timeval *timeout, LDAPMessage **result );
```

The parameters are described below:

- `ld` is the connection handle, which is a pointer to the LDAP structure containing information about the connection to the LDAP server.
- `msgid` is the message ID returned as the last argument of the `ldap_sasl_bind()` function.
- `all` applies to LDAP search operations. Since this is not an LDAP search operation, you can pass NULL for this argument.
- `timeout` is the maximum interval to wait for the selection to complete. If `timeout` is a NULL pointer, the select blocks indefinitely. To effect a poll, the `timeout` parameter should be a non-NULL pointer, pointing to a zero-valued `timeval` structure.
- `result` is a pointer to an `LDAPMessage` structure containing the results of the bind operation. You need to pass this to the `ldap_parse_sasl_bind_result()` function.

After calling `ldap_result()`, you need to call the `ldap_parse_sasl_bind_result()` function to parse the results and retrieve the LDAP result code and any credentials returned by the server:

```
LDAP_API(int) LDAP_CALL ldap_parse_sasl_bind_result( LDAP *ld,
   LDAPMessage *res, struct berval **servercredp, int freeit );
```

The parameters are described below:

- `ld` is the connection handle, which is a pointer to the LDAP structure containing information about the connection to the LDAP server.
- `res` is a pointer to the `LDAPMessage` structure containing the results that you want to parse.
- `servercredp` is a pointer to a pointer to the `berval` structure containing any credentials returned by the server. These are the credentials that you set as the value of the `SLAPI_BIND_RET_SASLCREDS` parameter in your pre-operation bind plug-in function.
- `freeit` specifies whether or not the result (the `LDAPMessage` structure) should be freed after the error code is extracted. If 0, the result is not freed. If non-zero, the result is freed.

The following example is an LDAP client that authenticates using the SASL mechanism named `babsmechanism`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <ldap.h>

int
main( int argc, char **argv )
{
    LDAP*ld;
    LDAPModmod0;
    LDAPModmod1;
    LDAPMod*mods[ 3 ];
    char*vals0[ 2 ];
    char*vals1[ 2 ];
    time_tnow;
    charbuf[ 128 ];
    struct berval cred;
    struct berval *servcred;
```

```
    int version;
    /* get a handle to an LDAP connection */
    if ( (ld = ldap_init( "localhost", 389 )) == NULL ) {
   perror( "ldap_init" );
   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.) */
    version = LDAP_VERSION3;
    ldap_set_option( ld, LDAP_OPT_PROTOCOL_VERSION, &version );
    /* authenticate */
    cred.bv_val = "magic";
    cred.bv_len = sizeof( "magic" ) - 1;
    if ( ldap_sasl_bind_s( ld, "uid=bjensen,ou=people,o=airius.com",
"babsmechanism", &cred, NULL, NULL, &servcred ) != LDAP_SUCCESS ) {
   ldap_perror( ld, "ldap_sasl_bind_s" );
   return( 1 );
    }
    /* get and print the credentials returned by the server */
    printf( "Server credentials: %s\n", servcred->bv_val );
    /* construct the list of modifications to make */
    mod0.mod_op = LDAP_MOD_REPLACE;
    mod0.mod_type = "mail";
    vals0[0] = "babs@airius.com";
    vals0[1] = NULL;
    mod0.mod_values = vals0;
    mod1.mod_op = LDAP_MOD_ADD;
    mod1.mod_type = "description";
    time( &now );
    sprintf( buf, "This entry was modified with the modattrs program on
%s",
```

```
        ctime( &now ));
     /* Get rid of \n which ctime put on the end of the time string */
     if ( buf[ strlen( buf ) - 1 ] == '\n' ) {
    buf[ strlen( buf ) - 1 ] = '\0';
     }
     vals1[ 0 ] = buf;
     vals1[ 1 ] = NULL;
     mod1.mod_values = vals1;
     mods[ 0 ] = &mod0;
     mods[ 1 ] = &mod1;
     mods[ 2 ] = NULL;
     /* make the change */
     if ( ldap_modify_s( ld, "uid=bjensen,ou=people,o=airius.com", mods )
        != LDAP_SUCCESS ) {
    ldap_perror( ld, "ldap_modify_s" );
    return( 1 );
     }
     ldap_unbind( ld );
     printf( "modification was successful\n" );
     return( 0 );
}
```

3

Writing Other Types of Plug-Ins

Chapter 10 Writing Entry Store/Fetch Plug-Ins

This chapter describes how to write entry store and entry fetch plug-ins. You can use these types of plug-ins to invoke functions before and after data is read from the default database.

Chapter 11 Writing Extended Operation Plug-Ins

This chapter explains how to write plug-in functions to handle extended operations (which are defined in the LDAP v3 protocol).

Chapter 12 Writing Matching Rule Plug-Ins

This chapter explains how to write plug-in functions that handle matching rules.

Chapter

10

Writing Entry Store/Fetch Plug-Ins

This chapter describes how to write entry store and entry fetch plug-ins. You can use these types of plug-ins to invoke functions before and after data is read from the default database.

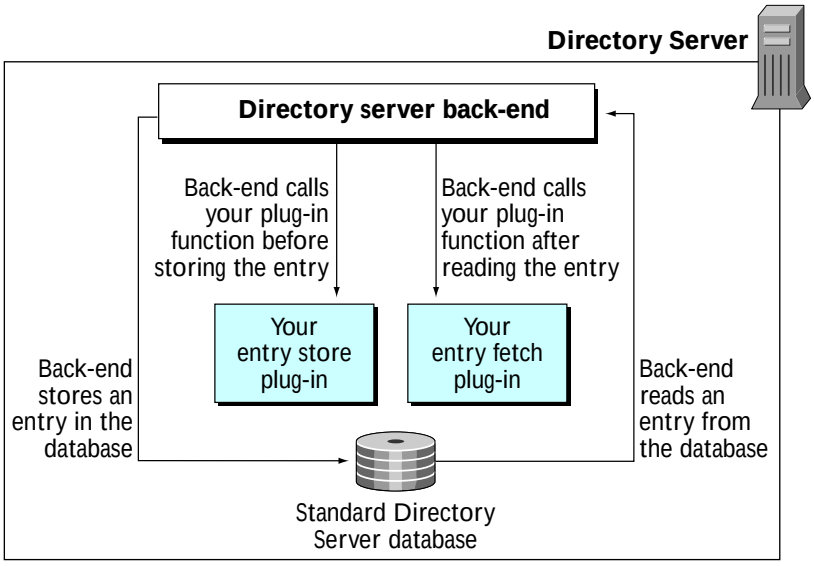
Note Entry store and entry fetch plug-ins work only with the default database. If you are using your own database and have your own database plug-in, entry store and entry fetch plug-ins will not work.

How Entry Store/Fetch Plug-Ins Work

Entry store plug-in functions are called before data is written to the default database. Entry fetch plug-in functions are called after data is read from the default database.

Figure 10.1 illustrates how the Directory Server back-end calls entry store and entry fetch functions before writing and reading data from the database.

Figure 10.1 How the server calls entry store and entry fetch plug-in functions



Writing Entry Store/Fetch Functions

Unlike most other types of plug-in functions, entry store and entry fetch plug-in functions are not passed a parameter block when called.

Instead, entry store and entry fetch plug-in functions must have the following prototype:

```
void function_name( char **entry, unsigned long *len );
```

On Windows NT, use the following prototype and make sure to include the function in a `.def` file:

```
__declspec( dllexport ) void function_name( char **entry, unsigned long
*len );
```

The parameters are described below:

`entry`	Pointer to a string specifying the entry in LDIF format (for details on this format, see “slapi_entry2str()” on page 268).
`len`	Pointer to the length of the `entry` string.

Since the text of the entry is passed in as an argument, you can modify the entry before it gets saved to disk and modify the entry after it is read from disk.

Registering Entry Store/Fetch Functions

Unlike most other types of plug-in functions, you do not register an entry store or entry fetch plug-in function by setting the function name in the parameter block.

Instead, you specify the function name directly in the `slapd.conf` configuration file. Add a directive in the following form to specify the name and location of your plug-in function:

```
plugin entrystore <library_name> <function_name>

plugin entryfetch <library_name> <function_name>
```

`<library_name>` is the name and path to your shared library or dynamic link library, and `<function_name>` is the name of your plug-in function.

For example, the following directives register the function named `my_store()` as the entry store plug-in function and `my_fetch()` as the entry fetch plug-in function. Both functions are defined in the library `/usr/nslib/myentry.so`.

```
plugin entrystore /usr/nslib/myentry.so my_store

plugin entryfetch /usr/nslib/myentry.so my_fetch
```

Note Each entry store and entry fetch plug-in is associated with the default `ldbm` back-end. Make sure that the `plugin` directive that registers the plug-in is within the database section for `ldbm` in the `slapd.conf` file. (The `plugin` directive should be after the `database ldbm` directive and before the next `database` directive, if any.)

Chapter

11

Writing Extended Operation Plug-Ins

This chapter explains how to write plug-in functions to handle extended operations (which are defined in the LDAP v3 protocol).

How Extended Operation Plug-Ins Work

Extended operations are part of the LDAP v3 protocol. You can define your own operation that you want the server to perform, and you can assign an OID to identify that operation.

LDAP clients can request the operation by sending an extended operation request. Within the request, the client specifies:

- the OID of the extended operation that should be performed
- data specific to the extended operation

When the Directory Server receives the request, the server calls the plug-in registered with the specified OID. The plug-in function has access to both the OID and the data in the client's request. The function can send back to the client a response containing:

- an OID
- any additional data

In order to use extended operations, you need to configure both the server and the client to understand the specific extended operation that you want performed.

Writing Extended Operation Functions

Like other plug-in functions, extended operation functions pass a single parameter block and return an integer value:

```
int my_ext_func( Slapi_PBlock *pb );
```

These functions should return 0 if successful and a non-zero value if unsuccessful.

When the Directory Server receives an extended operation request, the front-end calls the extended operation function with the same OID as specified in the request.

The front-end makes the following information available in the form of parameters in a parameter block.

Parameter ID	Data Type	Description
SLAPI_EXT_OP_REQ_OID	char *	Object ID (OID) of the extended operation specified in the request.
SLAPI_EXT_OP_REQ_VALUE	struct berval*	Value specified in the request.
SLAPI_EXT_OP_RET_OID	char *	Object ID (OID) that you want sent back to the client.
SLAPI_EXT_OP_RET_VALUE	struct berval*	Value that you want sent back to the client.

Typically, your function should perform some operation on the value specified in the SLAPI_EXT_OP_REQ_VALUE parameter.

Your function should return one of the following values:

- If your function has sent a result code back to the client, you should return the value SLAPI_PLUGIN_EXTENDED_SENT_RESULT. This indicates that the front-end does not need to send a result code.

- If your function has not sent a result code back to the client (for example, if the result is `LDAP_SUCCESS`), your function should return an LDAP result code. The front-end will send this result code back to the client.
- If your function cannot handle the extended operation with the specified OID, your function should return the value `SLAPI_PLUGIN_EXTENDED_NOT_HANDLED`. The front-end will send an `LDAP_PROTOCOL_ERROR` result code (with an “unsupported extended operation error message”) back to the client.

Registering Extended Operation Functions

As is the case with other server plug-in functions (see “Working with Parameter Blocks” on page 30), extended operation functions are specified in a parameter block that you can set on server startup.

In your initialization function, you can call the `slapi_pblock_set()` function to set the `SLAPI_PLUGIN_EXT_OP_FN` parameter to your function and the `SLAPI_PLUGIN_EXT_OP_OIDLIST` parameter to the list of OIDs of the extended operations supported by your function.

You can write your initialization function so that the OID is passed in from the directive. (See “Passing Extra Arguments to the Plug-Ins” on page 45 for details.) For example, the following initialization function sets the `SLAPI_PLUGIN_EXT_OP_OIDLIST` parameter to the additional parameters specified i

```
int
extended_init( Slapi_PBlock *pb )
{
   int i;
   char **argv;
   char **oids;

/* Get the additional arguments specified in the directive */
   if ( slapi_pblock_get( pb, SLAPI_PLUGIN_ARGV, &argv ) != 0 ) {
      slapi_log_error( SLAPI_LOG_PLUGIN, "extended_init",
      "Server could not get argv.\n" );
```

```
      return( -1 );
   }
   if ( argv == NULL ) {
      slapi_log_error( SLAPI_LOG_PLUGIN, "extended_init",
      "Required argument <oiD> is missing\n" );
      return( -1 );
   }

   /* Get the number of additional arguments and copy them. */
   for ( i = 0; argv[i] != NULL; i++ )
      ;
   oids = (char **) slapi_ch_malloc( (i+1) * sizeof(char *) );
   for ( i = 0; argv[i] != NULL; i++ ) {
      oids[i] = slapi_ch_strdup( argv[i] );
   }
   oids[i] = NULL;
/* Specify the version of the plug-in */
      if ( slapi_pblock_set( pb, SLAPI_PLUGIN_VERSION,
         SLAPI_PLUGIN_VERSION_01 ) != 0  ||

/* Specify the OID of the extended operation */
      slapi_pblock_set( pb, SLAPI_PLUGIN_EXT_OP_OIDLISTs,
         (void*) oids ) != 0  ||

/* Specify the function that the server should call */
      slapi_pblock_set( pb, SLAPI_PLUGIN_EXT_OP_FN,
         (void*) extended_op ) != 0 ) {

         slapi_log_error( SLAPI_LOG_PLUGIN, "extended_init",
         "An error occurred.\n" );
         return( -1 );
      }
```

```
    slapi_log_error( SLAPI_LOG_PLUGIN, "extended_init",
       "Plug-in sucessfully registered.\n" );
    return(0);
}
```

In the `slapd.conf` configuration file. Add a directive in the following form to specify the name and location of your plug-in function and to specify the object identification (OID) of the operation:

```
plugin extendedop <library_name> <function_name> <OID>
```

`<library_name>` is the name and path to your shared library or dynamic link library, `<function_name>` is the name of your plug-in function, and `<OID>` is the object identifier of the extended operation.

For example, the following directive registers the function named `my_ext_op()` as the extended operation plug-in function for the operation with the OID 1.2.3.4.

```
plugin extendedop /usr/nslib/myext.so my_ext_op 1.2.3.4
```

Note Each extended operation plug-in is associated with a back-end. Make sure that the `plugin` directive that registers the plug-in is within the database section for that back-end in the `slapd.conf` file. (The `plugin` directive should be after the `database` directive and before the next `database` directive, if any.)

Specifying Start and Close Functions

For each extended operation plug-in, you can specify the name of a function to be called after the server starts and before the server is shut down.

Use the following parameters to specify these functions:

`SLAPI_PLUGIN_START_FN`	Specifies the function called after the Directory Server starts up.
`SLAPI_PLUGIN_CLOSE_FN`	Specifies the function called before the Directory Server shuts down.

If you register multiple plug-ins with different start and close functions, the functions are called in the order that the plug-ins are registered (in other words, in the order that the `plugin` directives appear in the `slapd.conf` file).

Chapter

12

Writing Matching Rule Plug-Ins

This chapter explains how to write plug-in functions that handle matching rules.

Matching rule plug-in functions are described in the following sections:

Understanding Matching Rules

A matching rule specifies how one or more attributes of a particular syntax should be compared against assertion values. For example, a matching rule that specifies a “sound-alike” comparison attempts to match values that sound like the specified value. Each matching rule is identified by a unique OID (for example, “1.2.3.4”).

LDAP v3 clients can specify a matching rule as part of a search filter in a search request. This type of search filter is called an **extensible match** filter.

Extensible Match Filters

In an “extensible match” filter, the client specifies that it wants to use the matching rule to compare a specified value against the values of entries in the directory. (For example, an extensible match filter might find all entries in which the `sn` attribute “sounds like” `melon`.)

An “extensible match” filter contains the following information:

- the OID of the matching rule or the attribute type that you want to search (or both)
- the value to search for
- a preference indicating whether or not to also search the attributes in the DN

For example, if the OID “1.2.3.4” identifies a matching rule that performs “sounds like” matches, the following extensible match filter attempts to find entries where the `mail` attribute “sounds like” `moxie`.

```
(mail:1.2.3.4:=moxie)
```

In the search filter, the client can specify the OID (that idnetifies a matching rule) and the attribute type. This indicates that the value in the filter should be compared against the attribute using the matching rule.

For example, if the OID 1.2.3.4 specifies a “sound-alike” match and if the string representation of the search filter is:

```
(uid:1.2.3.4:=moxie)
```

the client wants to find entries in which the value of the `uid` attribute sounds like `moxie`.

Note that although the LDAP v3 standard allows clients to omit the OID or the attribute type, at this time, the Netscape Directory Server only supports extensible match filters that specify both the OID and attribute type.

The filter can also specify a preference indicating whether or not to also search the attributes in the DN. For example, if the OID 1.2.3.4 specifies a "sound-alike" match and if the string representation of the search filter is:

```
(sn:dn:1.2.3.4:=moxie)
```

the client wants to find all entries in which the value of the `sn` attribute or the attributes in the DN (for example, `uid`, `cn`, `ou`, or `o`) sound like `moxie`.

Extensible Match Filters in the Directory Server

The Netscape Directory Server 3.0 and more recent versions already include support for certain matching rules (which are used to determine the collation order and operator for searches of international data).

You can enable the Netscape Directory Server to handle your own matching rules for extensible match searches by defining your own matching rules plug-ins and registering them with the server.

You can also build indexes to improve the performance of search operations that use extended match filters.

Understanding Matching Rule Plug-Ins

A matching rule plug-in can create filters that the server can use when handling extensible search filters. A matching rule plug-in can also create indexers to index entries for extensible searches.

Functions Defined in Matching Rule Plug-Ins

The matching rule plug-in consists of the following:

- An indexer function (optional)
- A filter function
- A filter function that uses the index to speed up searches (optional)
- A function to destroy a filter (optional)
- A function to destroy an indexer (optional)
- A factory function to create filters
- A factory function to create indexers (optional)
- A close function to clean up before server shutdown (optional)
- An initialization function to register the factory functions and the close function

When the server starts up and loads the matching rule plug-in, it calls the initialization function. In this function, you pass the server the pointers to the factory functions and the close function. The server calls these functions when needed (see “How Matching Rules Are Identified” on page 162 and “How the Server Associates Plug-Ins with OIDs” on page 163 for details).

How Matching Rules Are Identified

Matching rules are identified by OID. When the server encounters an OID in the following situations, it attempts to find the matching rule plug-in that handles the matching rule with that OID.

The server can encounter a matching rule OID in the following situations:

- When reading in the `slapd.conf` configuration file, the server may encounter an index directive that specifies the OID of the matching rule. For example:

```
index <attribute_name> <filter_type> <matching_rule_oid>
```

If the OID is associated with a matching rule plug-in, the server adds this OID to the list of matching rule OIDs to use for indexing.

For information on setting up the server to index based on matching rule, see “Indexing Based on Matching Rules” on page 166.

- The server may receive an LDAP search request with an “extensible match” filter specifying the OID of the matching rule. For example, a string representation of an “extensible match” filter might be:

  ```
  (sn:dn:1.2.3.4:=Jensen)
  ```

  (The search filter above specifies that the server should use the matching rule identified by the OID 1.2.3.4 to search for the value `Jensen` in the `sn` attribute and in all attributes in the DN.)

  For information on setting up the server to handle extensible match filters, see “Handling Extensible Match Filters” on page 172.

- The server may receive an LDAP search request containing a sorting control, and the sorting control specifies the OID of the matching rule.

  For information on setting up the server to sort based on matching rules, see “Handing Sorting by Matching Rules” on page 181.

In all of these situations, the server uses the matching rule OID to find the plug-in responsible for handling the rule (see “How the Server Associates Plug-Ins with OIDs” on page 163 for details).

How the Server Associates Plug-Ins with OIDs

When the server encounters the OID for a matching rule, it attempts to find the plug-in associated with that matching rule.

If no plug-in is associated with the matching rule, the server calls each matching rule plug-in to find one that handles the specified matching rule.

When the server finds a plug-in that handles the matching rule, the server creates an association between the plug-in and the matching rule OID for future reference.

If no matching rule plug-in supports the specified OID, the server returns an `LDAP_UNAVAILABLE_CRITICAL_EXTENSION` error back to the client.

Finding a Plug-In for Indexing

To determine which matching rule plug-in is responsible for indexing an attribute with a given matching rule (based on its OID), the server does the following for each plug-in:

1. In a new `Slapi_PBlock` parameter block, the server sets the OID in the `SLAPI_PLUGIN_MR_OID` parameter.
2. Next, the server calls the indexer factory function (specified in the `SLAPI_PLUGIN_MR_INDEXER_CREATE_FN` parameter) for the plug-in.
3. The server checks the `SLAPI_PLUGIN_MR_INDEX_FN` parameter.
   - If the parameter is NULL, the plug-in does not handle the matching rule specified by that OID.
   - If the parameter returns an indexer function, this plug-in handles the matching rule specified by that OID.
4. Finally, the server frees the parameter block from memory.

At some point in time, the server may also call the indexer destructor function (specified in the `SLAPI_PLUGIN_MR_DESTROY_FN` parameter) to free the indexer object that was created by the indexer factory function.

Finding a Plug-In for Searching

To determine which matching rule plug-in is responsible for handling an extensible match filter for a given matching rule (based on its OID), the server does the following for each plug-in:

1. In a new `Slapi_PBlock` parameter block, the server sets the following parameters:
   - Sets the OID in the `SLAPI_PLUGIN_MR_OID` parameter.
   - Sets the type (from the filter) in the `SLAPI_PLUGIN_MR_TYPE` parameter.

- Sets the value (from the filter) in the `SLAPI_PLUGIN_MR_VALUE` parameter.

2. Next, the server calls the filter factory function (specified in the `SLAPI_PLUGIN_MR_FILTER_CREATE_FN` parameter) for the plug-in.
3. The server checks the `SLAPI_PLUGIN_MR_FILTER_MATCH_FN` parameter.
   - If the parameter is NULL, the plug-in does not handle the matching rule specified by that OID.
   - If the parameter returns a filter matching function, this plug-in handles the matching rule specified by that OID.
4. Finally, the server gets the following information from the plug-in for future use:
   - The filter index function specified in the `SLAPI_PLUGIN_MR_FILTER_INDEX_FN` parameter
   - The value specified in the `SLAPI_PLUGIN_MR_FILTER_REUSABLE` parameter
   - The filter reset function specified in the `SLAPI_PLUGIN_MR_FILTER_RESET_FN` parameter
   - The filter object specified in the `SLAPI_PLUGIN_OBJECT` parameter
   - The filter destructor function specified in the `SLAPI_PLUGIN_DESTROY_FN` parameter

Information specified in the filter object is used by both the filter index function and the filter matching function.

How the Server Uses Parameter Blocks

The server uses parameter blocks as a means to pass information to and from your plug-in functions.

When calling your matching rule plug-in functions, the server will create a new `Slapi_PBlock` parameter block, set some input parameters, and pass the parameter block to your function. After retrieving output paramters from the block, the server typically frees the parameter block from memory.

In general, you should not expect a parameter block to be passed from plug-in function to plug-in function. For example, the value of a parameter set by one plug-in function may not necessarily be accessible to other plug-in functions, since each function is usually passed a new and different parameter block.

Indexing Based on Matching Rules

This section explains how to set up the server to index entries using a matching rule. The following topics are covered:

Note that you also need to define an initialization function to register your indexer factory function.

How the Server Sets Up the Index

When the server encounters a matching rule OID in an `index` directive in the `slapd.dynamic-ldbm.conf` configuration file, the server determines which plug-in supports the matching rule identified by the OID. See "How the Server Associates Plug-Ins with OIDs" on page 163 for details.

The server gets the OID returned in the `SLAPI_PLUGIN_MR_OID` parameter and associates this OID with the rest of the attribute indexing information (such as the attribute type and the type of index) for future reference.

When adding, modifying, or deleting the values of an attribute, the server will check this information to determine if the attribute is indexed. See “How the Server Updates the Index” on page 167 for information on how attributes are indexed.

How the Server Updates the Index

When a value is added, modified, or removed from an attribute in an entry (or when the RDN of an entry is changed), the server does the following if that attribute has an index that uses matching rules:

1. In a new `Slapi_PBlock` parameter block, the server sets the following parameters:
   - Sets the OID in the `SLAPI_PLUGIN_MR_OID` parameter.
   - Sets the attribute type (of the value being added, modified, or removed) in the `SLAPI_PLUGIN_MR_TYPE` parameter.
2. Next, the server calls the indexer factory function (specified in the `SLAPI_PLUGIN_MR_INDEXER_CREATE_FN` parameter) for the plug-in to create the indexer object.
3. The server generates the index keys for the values to be added or deleted:
   - The server first verifies that the `SLAPI_PLUGIN_MR_INDEX_FN` parameter specifies an indexer function and the `SLAPI_PLUGIN_MR_OID` parameter specifies the official OID of the matching rule.
   - If these are both set, the server sets the `SLAPI_PLUGIN_MR_VALUES` parameter to the array of `berval` structures containing the new or modified values that need to be indexed and calls the indexer function.
   - Next, the server gets the value of the `SLAPI_PLUGIN_MR_KEYS` parameter, which is an array of `berval` structures containing the keys corresponding to the values.
4. The server inserts or deletes the keys and values in the index for that attribute.

5. The server calls the indexer destructor function (specified in the `SLAPI_PLUGIN_MR_DESTROY_FN` parameter) to free the indexer object.

When the server is done, it frees any parameter blocks that were allocated during this process.

Writing the Indexer Factory Function

The indexer factory function takes a single `Slapi_PBlock` argument. This function should be thread-safe. The server may call this function concurrently.

The indexer factory function should do the following:

1. Get the OID from the `SLAPI_PLUGIN_MR_OID` parameter and determine whether or not that OID is supported by your plug-in.

   - If the OID is not supported, you need to return the result code `LDAP_UNAVAILABLE_CRITICAL_EXTENSION`.
   - If the OID is supported, continue with this process.

2. Get the value of the `SLAPI_PLUGIN_MR_USAGE` parameter. This parameter should have one of the following values:

   - If the value is `SLAPI_PLUGIN_MR_USAGE_SORT`, the server is calling your function to sort search results. See "Handing Sorting by Matching Rules" on page 181 for more information.
   - If the value is `SLAPI_PLUGIN_MR_USAGE_INDEX`, the server is calling your function to index an entry.

   You can use this information to set different information in the indexer object or to set a different indexer function, based on whether the function is being called to index or to sort.

3. You can also get any data that you set in the `SLAPI_PLUGIN_PRIVATE` parameter during initialization. (See "Writing an Initialization Function" on page 182.)

4. Create an indexer object containing any information that you want passed to the indexer function.

5. Set the following parameters:

   - Set the `SLAPI_PLUGIN_MR_OID` parameter to the official OID of the matching rule (if the value of that parameter is not the official OID).
   - Set the `SLAPI_PLUGIN_OBJECT` parameter to the indexer object.
   - Set the `SLAPI_PLUGIN_MR_INDEX_FN` parameter to the indexer function. (See "Writing the Indexer Function" on page 171.)
   - Set the `SLAPI_PLUGIN_DESTROY_FN` parameter to the function responsible for freeing any memory allocated by the factory function, such as the indexer object. (See "Writing a Destructor Function" on page 182 for details.)

6. Return 0 (or the result code `LDAP_SUCCESS`) if everything completed successfully.

Getting and Setting Parameters in Indexer Factory Functions

The following table summarizes the different parameters that the indexer factory function should get and set in the parameter block that is passed in:

Table 12.1 Input and output parameters available to a indexer factory function

Parameter Name	Data Type	Description
`SLAPI_PLUGIN_MR_OID`	`char *`	(Input parameter) Matching rule OID (if any) specified in the `index` directive.
`SLAPI_PLUGIN_MR_TYPE`	`char *`	(Input parameter) Attribute type (if any) specified in the `index` directive.

Table 12.1 Input and output parameters available to a indexer factory function

Parameter Name	Data Type	Description
SLAPI_PLUGIN_MR_USAGE	unsigned int	(Input parameter) Specifies the intended use of the indexer object. This parameter can have one of the following values: • SLAPI_PLUGIN_MR_USAGE_INDEX specifies that the indexer object should be used to index entries. • SLAPI_PLUGIN_MR_USAGE_SORT specifies that the indexer object should be used to sort entries. You can use this to specify different information in the indexer object or different indexer functions, based on whether the plug-in is used for indexing or sorting. For information on sorting search results, see "Handing Sorting by Matching Rules" on page 181.
SLAPI_PLUGIN_PRIVATE	void *	(Input parameter) Pointer to any private data originally specified in the initialization function. (See "Writing an Initialization Function" on page 182 for details.)
SLAPI_PLUGIN_MR_OID	char *	(Output parameter) Official matching rule OID of the index.
SLAPI_PLUGIN_MR_INDEX_FN	void * (function pointer)	(Output parameter) Name of the function called by the server to generate a list of keys used for indexing a set of values.
SLAPI_PLUGIN_DESTROY_FN	void * (function pointer)	(Output parameter) Name of the function to be called to free the indexer object.
SLAPI_PLUGIN_OBJECT	void *	(Output parameter) Pointer to the indexer object created by your factory function.

Writing the Indexer Function

The indexer function takes a single `Slapi_PBlock` argument. This function will never be called for the same indexer object concurrently. (If you plan to manipulate global variables, keep in mind that the server can call this function concurrently for different indexer objects.)

The indexer function should do the following:

1. Get the values of the following parameters:
   - Get the indexer object from the `SLAPI_PLUGIN_OBJECT` parameter (if the parameter is set).
   - Get the array of values that you want indexed from the `SLAPI_PLUGIN_MR_VALUES` parameter.
2. Generate index keys for these values, and set the `SLAPI_PLUGIN_MR_KEYS` parameter to the array of these keys.
3. Return 0 (or the result code `LDAP_SUCCESS`) if everything completed successfully.

The server adds or removes the keys and the corresponding values from the appropriate indexes.

Getting and Setting Parameters in Indexer Functions

The following table summarizes the different parameters that the indexer function should get and set in the parameter block that is passed in:

Table 12.2 Input and output parameters available to an indexer function

Parameter Name	Data Type	Description
SLAPI_PLUGIN_MR_VALUES	struct berval **	(Input parameter) Pointer to an array of berval structures containing the values of the entry's attributes that need to be indexed.
SLAPI_PLUGIN_OBJECT	void *	(Input parameter) Pointer to the indexer object created by the indexer factory function. (See "Writing the Indexer Factory Function" on page 168 for details.)
SLAPI_PLUGIN_MR_KEYS	struct berval **	(Output parameter) Keys generated for the values specified in the SLAPI_PLUGIN_MR_VALUES parameter. The server creates indexes using these keys.

Handling Extensible Match Filters

This section explains how to set up the server to process searches that use extensible match filters (matching rules). The following topics are covered:

Note that you also need to define an initialization function to register your filter factory function.

How the Server Handles the Filter

When the server processes a search request that has an extensible match filter, the server does the following:

1. First, the server finds the plug-in associated with this OID, if an association between the OID and plug-in has already been made.

   If no association has been made yet, the server attempts to find a matching rule plug-in that handles the OID. (See “How the Server Associates Plug-Ins with OIDs” on page 163 for details.)

2. Next, the server attempts to generate a list of search result candidate from the indexes. In a new `Slapi_PBlock` parameter block, the server does the following:

   - The server puts the filter object in the `SLAPI_PLUGIN_OBJECT` parameter, and calls the filter index function (specified in the `SLAPI_PLUGIN_MR_FILTER_INDEX_FN` parameter).
   - The server checks the value of the `SLAPI_PLUGIN_MR_QUERY_OPERATOR` parameter. If the operator is an known type (such as `SLAPI_OP_EQUAL`), the server will use the operator when searching the index for candidates. (For details, see “Query Operators in Matching Rules” on page 174).
   - The server sets the `SLAPI_PLUGIN_MR_VALUES` parameter to each of the values specified in the filter and calls the indexer function (which is specified in the `SLAPI_PLUGIN_MR_INDEX_FN` parameter) to generate the key (specified in the `SLAPI_PLUGIN_MR_KEYS` parameter).
   - The server uses the keys and the query operator to find potential candidates in the indexes.

   The server considers all entries to be potential candidates if at least one of the following is true:

   - The matching rule plug-in has no indexer function (specified in the `SLAPI_PLUGIN_MR_INDEX_FN` parameter).
   - No index applies to the search (for example, if the query operator does not correspond to an index).

- No keys are generated for the specified values.

3. For each candidate entry, the server checks to see if the entry matches the search filter by doing the following:

   - The server calls the filter matching function (which is specified in the `SLAPI_PLUGIN_MR_FILTER_MATCH_FN` parameter), passing in the filter object, the entry, and the attributes of the entry.
   - If the entry does not match but the search request also specifies that the attributes in the DN should be searched, the server calls the filter matching function again, passing in the filter object, the entry, and the attributes in the DN.

   The server checks the value returned by the filter matching function:

   - If the function returns 0, the entry matched the search filter.
   - If the function returns -1, the entry did not match the search filter.
   - If the function returns an LDAP error code (a positive value), an error occurred.

4. If the entry matches the filter, the server verifies that the entry is in the scope of the search before returning the entry to the LDAP client as a search result.

Query Operators in Matching Rules

As mentioned in “How the Server Handles the Filter” on page 173, the server uses a query operator when searching the index for possible candidates.

Note that this applies to the `ldbm` default back-end database. If you are using your own back-end or if you have not set up indexing by matching rules, the server does not make use of the query operator.

The server checks the value of the SLAPI_PLUGIN_MR_QUERY_OPERATOR parameter to determine which operator is specified. The following table lists the possible values for this parameter.

Table 12.3 Query operators in extensible match filters

Operator	Description
SLAPI_OP_LESS	<
SLAPI_OP_LESS_OR_EQUAL	<=
SLAPI_OP_EQUAL	=
SLAPI_OP_GREATER_OR_EQUAL	>=
SLAPI_OP_GREATER	>

If the query operator is SLAPI_OP_EQUAL, the server attempts to find the keys in the index that match the value specified in the search filter. In the case of the other query operators, the server attempts to find ranges of keys that match the value.

Writing a Filter Factory Function

The filter factory function takes a single Slapi_PBlock argument. This function should be thread-safe. The server may call this function concurrently. (Each incoming LDAP request is handled by a separate thread. Multiple threads may call this function if processing multiple requests that have extensible match filters.)

The filter factory function should do the following:

1. Get the OID from the SLAPI_PLUGIN_MR_OID parameter and determine whether or not that OID is supported by your plug-in.
   - If the OID is not supported, you need to return the result code LDAP_UNAVAILABLE_CRITICAL_EXTENSION. The server will send this back to the client.
   - If the OID is supported, continue with this process.
2. Get and check the values of the SLAPI_PLUGIN_MR_TYPE and SLAPI_PLUGIN_MR_VALUE parameters.

The values of these parameters are the attribute type and value specified in the extensible match filter.

3. You can also get any data that you set in the `SLAPI_PLUGIN_PRIVATE` parameter during initialization. (See "Writing an Initialization Function" on page 182.)

4. Create a filter object, putting the following information in the object:

   - The official OID of the matching rule (optional)
   - The attribute type specified in the filter
   - The value specified in the filter
   - Any additional data that you want made available to the filter index function (for example, the query operator, if specified in the filter)

   The server will call your filter index function at a later time to extract this information from the filter object.

5. Set the following parameters:

   - Set the `SLAPI_PLUGIN_MR_OID` parameter to the official OID of the matching rule, if the value of that parameter is not the official OID (optional).
   - Set the `SLAPI_PLUGIN_OBJECT` parameter to the filter object.
   - Set the `SLAPI_PLUGIN_MR_FILTER_INDEX_FN` parameter to the filter index function (see "Writing a Filter Index Function" on page 178), if you have set up indexes based on this matching rule (optional).
   - Set the `SLAPI_PLUGIN_MR_FILTER_MATCH_FN` parameter to the filter matching function. (See "Writing a Filter Matching Function" on page 180.)
   - Set the `SLAPI_PLUGIN_DESTROY_FN` parameter to the function responsible for freeing the filter object, if you have defined this function (optional). See "Writing a Destructor Function" on page 182 for details.

6. Return 0 (or the result code `LDAP_SUCCESS`) if everything completed successfully.

Getting and Setting Parameters in Filter Factory Functions

The following table summarizes the different parameters that the filter factory function should get and set in the parameter block that is passed in:

Table 12.4 Input and output parameters available to a filter factory function

Parameter Name	Data Type	Description
SLAPI_PLUGIN_MR_OID	char *	(Input parameter) Matching rule OID (if any) specified in the extensible match filter.
SLAPI_PLUGIN_MR_TYPE	char *	(Input parameter) Attribute type (if any) specified in the extensible match filter.
SLAPI_PLUGIN_MR_VALUE	struct berval *	(Input parameter) Value specified in the extensible match filter.
SLAPI_PLUGIN_PRIVATE	void *	(Input parameter) Pointer to any private data originally specified in the initialization function. (See “Writing an Initialization Function” on page 182 for details.)
SLAPI_PLUGIN_MR_FILTER_MATCH_FN	mrFilterMatchFn (function pointer)	(Output parameter) Name of the function called by the server to match an entry’s attribute values against the value in the extensible search filter.
SLAPI_PLUGIN_MR_FILTER_INDEX_FN	void * (function pointer)	(Output parameter) Name of the function called by the server to generate a list of keys used for indexing a set of values.
SLAPI_PLUGIN_DESTROY_FN	void * (function pointer)	(Output parameter) Name of the function to be called to free the filter object.
SLAPI_PLUGIN_OBJECT	void *	(Output parameter) Pointer to the filter object created by your factory function.

Writing a Filter Index Function

The filter index function takes a single `Slapi_PBlock` argument. This function will never be called for the same filter object concurrently. (If you plan to manipulate global variables, keep in mind that the server can call this function concurrently for different filter objects.)

The filter index function should do the following:

7. Get the filter object from the `SLAPI_PLUGIN_OBJECT` parameter (if the parameter is set).

8. Using data from the object, determine and set the values of the following parameters:

   - Set the `SLAPI_PLUGIN_MR_OID` parameter to the official OID of the matching rule.
   - Set the `SLAPI_PLUGIN_MR_TYPE` parameter to the attribute type in the filter object.
   - Set the `SLAPI_PLUGIN_MR_VALUES` parameter to the values in the filter object.
   - Set the `SLAPI_PLUGIN_MR_QUERY_OPERATOR` parameter to the query operator that corresponds to this search filter. (See “Query Operators in Matching Rules” on page 174 for possible values for this parameter.)
   - Set the `SLAPI_PLUGIN_OBJECT` parameter to the filter object.
   - Set the `SLAPI_PLUGIN_MR_INDEX_FN` parameter to the indexer function. (See “Writing the Indexer Function” on page 171.)

9. Return 0 (or the result code `LDAP_SUCCESS`) if everything completed successfully.

Getting and Setting Parameters in Filter Index Functions

The following table summarizes the different parameters that the filter index function should get and set in the parameter block that is passed in:

Table 12.5 Input and output parameters available to a filter index function

Parameter Name	Data Type	Description
SLAPI_PLUGIN_OBJECT	void *	(Input and output parameter) Pointer to the filter object created by the factory function. (For details, see “Writing a Filter Factory Function” on page 175.)
SLAPI_PLUGIN_MR_QUERY_OPERATOR	int	(Output parameter) Query operator used by the server to determine how to compare the keys generated from SLAPI_PLUGIN_MR_VALUES and SLAPI_PLUGIN_MR_INDEX_FN against keys in the index. For a list of possible values for this parameter, see “Query Operators in Matching Rules” on page 174.
SLAPI_PLUGIN_MR_OID	char *	(Output parameter) Official matching rule OID (if any) specified in the extensible match filter.
SLAPI_PLUGIN_MR_TYPE	char *	(Output parameter) Attribute type (if any) specified in the extensible match filter.
SLAPI_PLUGIN_MR_VALUES	struct berval **	(Output parameter) Pointer to an array of berval structures containing the values specified in the extensible match filter.
SLAPI_PLUGIN_MR_INDEX_FN	void * (function pointer)	(Output parameter) Name of the function called by the server to generate a list of keys used for indexing a set of values.

Writing a Filter Matching Function

The filter matching function has the following prototype:

```
#include "slapi-plugin.h"
typedef int (*mrFilterMatchFn) (void* filter,
   Slapi_Entry* entry, Slapi_Attr* attrs);
```

This function passes the following arguments:

- `filter` is a pointer to the filter object.
- `entry` is a pointer to the `Slapi_Entry` entry that should be compared against the filter.
- `attrs` is the first `Slapi_Attr` attribute in the entry or in the set of DN attributes. (The extensible match filter might specify that the attributes in the DN of an entry should also be included in the search.)

This function will never be called for the same filter object concurrently. (If you plan to manipulate global variables, keep in mind that the server can call this function concurrently for different filter objects.)

The filter matching function should do the following:

10. From the filter object, get the attribute type, the values, and the query operator.
11. Find the corresponding attribute in the attributes passed into the function. Make sure to check for subtypes of an attribute (for example, “cn;lang-ja”) in the filter and in the attributes specified by `attrs`.

    You can call the `slapi_attr_type_cmp()` function to compare the attribute in the filter against the attributes passed in as arguments.
12. Using the query operator to determine how the values should be compared, compare the values from the filter against the values from the values in the attribute.
13. Return one of the following values:
    - 0 if the values of the attribute match the value specified in the filter.
    - -1 if the values don’t match.

- An LDAP error code (a positive number) if an error occurred.

Handing Sorting by Matching Rules

If you have set up indexing by a matching rule, you can also sort search results by that matching rule. The server can use the keys in the index to sort the search results.

When processing a request to sort by a matching rule, the server does the following:

14. In a new `Slapi_PBlock` parameter block, the server sets the following parameters:
    - Sets the OID in the `SLAPI_PLUGIN_MR_OID` parameter.
    - Sets the attribute type (of the value being added, modified, or removed) in the `SLAPI_PLUGIN_MR_TYPE` parameter.
    - Sets the `SLAPI_PLUGIN_MR_USAGE` parameter to `SLAPI_PLUGIN_MR_USAGE_SORT`. (This indicates that the created indexer object will be used for sorting, not indexing.)
15. Next, the server calls the indexer factory function (specified in the `SLAPI_PLUGIN_MR_INDEXER_CREATE_FN` parameter) for the plug-in.
16. The server generates the index keys for the values to be sorted:
    - The server sets the `SLAPI_PLUGIN_MR_VALUES` parameter to the array of `berval` structures containing the values to be sorted.
    - The server calls the indexer function (specified by the `SLAPI_PLUGIN_MR_INDEXER_FN` parameter).
    - Next, the server gets the value of the `SLAPI_PLUGIN_MR_KEYS` parameter, which is an array of `berval` structures containing the keys corresponding to the values.
17. The server compares the keys to sort the results.

Writing a Destructor Function

The server calls the destructor function to free any memory that you've allocated (for example, the indexer object or the filter object).

The destructor function takes a single `Slapi_PBlock` argument. The following table summarizes the different parameters that the destructor function should get and set in the parameter block that is passed in:

Table 12.6 Input and output parameters available to a destructor function

Parameter Name	Data Type	Description
`SLAPI_PLUGIN_OBJECT`	`void *`	(Input parameter) Pointer to the filter object or indexer object created by your factory function.

For example, your destructor function can get the indexer object from the `SLAPI_PLUGIN_OBJECT` parameter and free the object from memory.

This function will never be called for the same indexer or filter object concurrently. (If you plan to manipulate global variables, keep in mind that the server can call this function concurrently for different filter or indexer objects.)

Writing an Initialization Function

Internally, the server keeps a list of matching rule plug-ins. When dealing with matching rules, the server attempts to find the matching rule plug-in to handle the given matching rule. (See "How the Server Associates Plug-Ins with OIDs" on page 163 for details.)

In order to add your plug-in to that internal list, you need to write an initialization function. The initialization function takes a single `Slapi_PBlock` argument. The function should set the following parameters:

- The `SLAPI_PLUGIN_MR_FILTER_CREATE_FN` parameter should be set to the filter factory function. See "How the Server Handles the Filter" on page 173 and "Writing a Filter Factory Function" on page 175 for details.

- The SLAPI_PLUGIN_MR_INDEXER_CREATE_FN parameter should be set to the indexer factory function, if you have defined one (optional). See “How the Server Sets Up the Index” on page 166 and “Writing the Indexer Factory Function” on page 168 for details.
- The SLAPI_PLUGIN_CLOSE_FN parameter should be set to the “close” function, if you have defined one (optional). See “Specifying Start and Close Functions” on page 184 for details.
- The SLAPI_PLUGIN_PRIVATE parameter should be set to any private data you want made accessible to the plug-in functions (optional).

You need to register the initialization function (see “Registering Matching Rule Functions” on page 184) so that the server runs the function when starting up.

The following table summarizes the different parameters that the initialization function should get and set in the parameter block that is passed in:

Table 12.7 Input and output parameters available to a matching rule plug-in initialization function

Parameter Name	Data Type	Description
SLAPI_PLUGIN_ARGC	int	(Input parameter) Number of arguments in the plugin directive (not including the library name and initialization function name).
SLAPI_PLUGIN_ARGV	char **	(Input parameter) Array of string arguments in the plugin directive (not including the library name and initialization function name).
SLAPI_PLUGIN_MR_FILTER_CREATE_FN	void * (function pointer)	(Output parameter) The factory function used for creating filters.
SLAPI_PLUGIN_MR_INDEXER_CREATE_FN	void * (function pointer)	(Output parameter) The factory function used for creating indexers.
SLAPI_PLUGIN_CLOSE_FN	void * (function pointer)	(Output parameter) The “close” function, which the server calls before shutting down.
SLAPI_PLUGIN_PRIVATE	void *	(Output parameter) Pointer to any private data you want passed to your plug-in functions.

Registering Matching Rule Functions

To register your matching rule plug-in, add the following directive to the `slapd.conf` configuration file:

```
plugin matchingrule <library_name> <init_fn_name> [<arguments>...]
```

`<init_fn_name>` is the name of your initialization function and `<library_name>` is the name and path of the library where the function is defined. For example, the following directive registers the initialization function named `my_init_fn()`, which is defined in the library `/usr/ns/myplugin.so`:

```
plugin matchingrule /usr/ns/myplugin.so my_init_fn
```

Example of a Matching Rule Plug-In

[??? I need to put one together. ???]

Specifying Start and Close Functions

For each matching rule operation plug-in, you can specify the name of a function to be called after the server starts and before the server is shut down. These functions take a single `Slapi_PBlock` argument.

The following table summarizes the different parameters that the initialization function should get and set in the parameter block that is passed in:

Table 12.8 Input and output parameters available to a matching rule plug-in initialization function

Parameter Name	Data Type	Description
`SLAPI_PLUGIN_START_FN`	`void *` `(function pointer)`	(Output parameter) The function called after the Directory Server starts up.
`SLAPI_PLUGIN_CLOSE_FN`	`void *` `(function pointer)`	(Output parameter) The function called before the Directory Server shuts down.

If you register multiple plug-ins with different start and close functions, the functions are called in the order that the plug-ins are registered (in other words, in the order that the `plugin` directives appear in the `slapd.conf` file).

4

Reference

Chapter 13 Data Type and Structure Reference

This chapter summarizes the data types and structures that you can use when writing server plug-in functions.

Chapter 14 Function Reference

This chapter summarizes the front-end functions available in the API. You can call these functions in your own server plug-in functions.

Chapter 15 Parameter Reference

This chapter describes the parameters available in the Slapi_PBlock parameter block, the type of data associated with each parameter, and the plug-in functions in which those parameters are accessible.

Chapter

13

Data Type and Structure Reference

This chapter summarizes the data types and structures that you can use when writing server plug-in functions.

Summary of Data Types and Structures

The functions in the server plug-in API use the following data types and structures:

- `berval`
- `computed_attr_context`
- `LDAPControl`
- `LDAPMod`
- `mrFilterMatchFn`
- `plugin_referral_entry_callback`
- `plugin_result_callback`
- `plugin_search_entry_callback`
- `send_ldap_referral_fn_ptr_t`

- send_ldap_result_fn_ptr_t
- send_ldap_search_entry_fn_ptr_t
- Slapi_Attr
- slapi_compute_callback_t
- slapi_compute_output_t
- Slapi_CondVar
- Slapi_Entry
- Slapi_Filter
- Slapi_MatchingRuleEntry
- Slapi_Mutex
- Slapi_PBlock
- Slapi_PluginDesc

berval

`berval` represents binary data that is encoded using simplified Basic Encoding Rules (BER). The data and size of the data are included in a `berval` structure.

`berval` is defined as follows:

```
struct berval {
   unsigned long bv_len;
   char *bv_val;
};
```

The fields in this structure are described below:

`bv_len`	The length of the data.
`bv_val`	The binary data.

Use a `berval` structure when working with attributes that contain binary data (such as a JPEG or audio file).

computed_attr_context

Reserved for future use.

Represents information used for a computed attribute.

Syntax

```
typedef struct _computed_attr_context computed_attr_context;
```

Description

This type is reserved for future use.

`computed_attr_context` is the data type for an opaque structure that represents information about a computed attribute.

Before the Netscape Directory Server sends an entry back to a client, it determines if any of the attributes are computed, generates the attributes, and includes the generated attributes in the entry.

As part of this process, the server creates a `computed_attr_context` structure to pass relevant information to the functions generating the attribute values. (Relevant information might include the attribute type, the BER-encoded request so far, and the parameter block).

LDAPControl

`LDAPControl` represents a client or server control associated with an LDAP operation. Controls are part of the LDAP v3 protocol. You can use a client or server control to extend the functionality of an LDAP control.

For example, you can use a server control to specify that you want the server to sort search results in an LDAP search operation.

`LDAPControl` is defined as follows:

```
typedef struct ldapcontrol {
   char *ldctl_oid;
   struct berval ldctl_value;
   char ldctl_iscritical;
} LDAPControl, *PLDAPControl;
```

The fields in this structure are described below:

`ldctl_oid`	Object ID (OID) of the control.
`ldctl_value`	`berval` structure containing the value used by the control for the operation.
`ldctl_iscritical`	Specifies whether or not the control is critical to the operation. This field can have one of the following values: • `LDAP_OPT_ON` specifies that the control is critical to the operation. • `LDAP_OPT_OFF` specifies that the control is not critical to the operation.

LDAPMod

`LDAPMod` is a type of structure that you use to specify changes to an attribute in an directory entry. Before you call the `slapi_add_internal()` and `slapi_modify_internal()` routines to add or modify an entry in the directory, you need to fill `LDAPMod` structures with the attribute values that you intend to add or change.

`LDAPMod` is defined as follows:

```
typedef struct ldapmod {
   int mod_op;
   char *mod_type;
   union {
      char **modv_strvals;
      struct berval **modv_bvals;
   } mod_vals;
} LDAPMod;
#define mod_values mod_vals.modv_strvals
#define mod_bvalues mod_vals.modv_bvals
```

The fields in this structure are described below:

`mod_op`	The operation to be performed on the attribute and the type of data specified as the attribute values. This field can have one of the following values: • `LDAP_MOD_ADD` specifies that you want to add the attribute values to the entry. • `LDAP_MOD_DELETE` specifies that you want to remove the attribute values from the entry. • `LDAP_MOD_REPLACE` specifies that you want to replace the existing value of the attribute with the value(s) in `mod_values` or `mod_bvalues`. In addition, if you are specifying binary values (as opposed to strings), you should OR (|) `LDAP_MOD_BVALUES` with the operation type. For example: `mod->mod_op = LDAP_MOD_ADD | LDAP_MOD_BVALUES`
`mod_type`	The attribute type that you want to add, delete, or replace.
`mod_values`	Pointer to a NULL-terminated array of string values for the attribute.
`mod_bvalues`	Pointer to a NULL-terminated array of `berval` structures for the attribute.

The following section of code sets up an `LDAPMod` structure to change the email address of a user's entry to "bab@ace.com":

```
Slapi_PBlock *rcpb;
LDAPMod attribute1;
LDAPMod *list_of_attrs[2];
char *mail_values[] = { "bab@ace.com", NULL };
char *dn;
...
/* Identify the entry that you want changed */
dn = "cn=Barbara Jensen, ou=Product Development, o=Ace Industry, c=US";
/* Specify that you want to replace the value of an attribute */
attribute1.mod_op = LDAP_MOD_REPLACE;
```

```
/* Specify that you want to change the value of the mail attribute */
attribute1.mod_type = "mail";

/* Specify the new value of the mail attribute */
attribute1.mod_values = mail_values;

/* Add the change to the list of attributes that you want changed */
list_of_attrs[0] = &attribute_change;
list_of_attrs[1] = NULL;

/* Update the entry with the change */
rcpb = slapi_modify_internal( dn, list_of_attrs, NULL, 1 );
...
```

mrFilterMatchFn

`mrFilterMatchFn` specifies the prototype for a “filter matching” callback function. This function is called by the server when processing a search operation. The server calls this function for each potential candidate entry that might match a search filter.

Syntax

```
#include “slapi-plugin.h”
typedef int (*mrFilterMatchFn) (void* filter,
   Slapi_Entry* entry, Slapi_Attr* attrs);
```

Parameters

The function has the following parameters:

`filter`	Pointer to the filter structure created by your filter factory function. (For details, see “Writing a Filter Factory Function” on page 175.)
`entry`	Pointer to the `Slapi_Entry` structure representing the candidate entry being checked by the server.
`attrs`	Pointer to the `Slapi_Attr` structure representing the first attribute in the entry. To iterate through the rest of the attributes in the entry, call `slapi_entry_next_attr()`.

Description

`mrFilterMatchFn` specifies the prototype for a “filter matching” function that is called by the server when processing an “extensible match” filter.

An “extensible match” filter specifies either the OID of a matching rule or an attribute type (or both) that indicates how matching entries are found. For example, a “sound-alike” matching rule might find all entries that sound like a given value.

To handle an “extensible match” filter for a matching rule, you can write a matching rule plug-in. (For more information, see Chapter 12, “Writing Matching Rule Plug-Ins”.)

One of the functions that you need to define is the “filter matching” function (the function with the prototype specified by `mrFilterMatchFn`). The server calls this function for each potential matching candidate entry. The server passes pointers to a filter structure (that you create in your filter factory function -- see “Writing a Filter Factory Function” on page 175 for details), the candidate entry, and the entry’s attributes.

In your filter matching function, you need to retrieve information about the filter (such as the attribute type and value specified in the filter) from the filter structure. You use this information to compare the value in the filter against the attribute values in the candidate entry.

For more information on writing a filter matching function, see “Writing a Filter Matching Function” on page 180.

Example [To be added]

See Also

plugin_referral_entry_callback

`plugin_referral_entry_callback` specifies the prototype for a callback function. This function is called by the server when an internal search finds an entry that contains LDAP v3 referrals (search result references, which are normally sent back to the client as results found by the search) and if the client supports the LDAP v3 protocol (see “slapi_search_internal_callback()” on page 336 for details).

Syntax

```
#include “slapi-plugin.h”
typedef int (*plugin_referral_entry_callback)(char *referral,
   void *callback_data);
```

Parameters

The function has the following parameters:

`referral`	Referral (in the form of an LDAP URL) found by the search.
`callback_data`	Pointer to the data specified by the `callback_data` argument in the `slapi_search_internal_callback()` function call.

Returns

0 if successful or -1 if an error occurred.

Example

[To be added]

See Also

plugin_result_callback

`plugin_result_callback` specifies the prototype for a callback function. This function is called by the server when the internal search function needs to send a result code (see “slapi_search_internal_callback()” on page 336 for details).

Syntax

```
#include “slapi-plugin.h”
typedef void (*plugin_result_callback)(int rc,
   void *callback_data);
```

Parameters

The function has the following parameters:

`rc`	Result code to be sent
`callback_data`	Pointer to the data specified by the `callback_data` argument in the `slapi_search_internal_callback()` function call.

Example

[To be added]

See Also

plugin_search_entry_callback

`plugin_search_entry_callback` specifies the prototype for a callback function. This function is called by the server when an internal search finds a matching entry (see “slapi_search_internal_callback()” on page 336 for details).

Syntax

```
#include “slapi-plugin.h”
typedef int (*plugin_search_entry_callback)(Slapi_Entry *e,
   void *callback_data);
```

Parameters

The function has the following parameters:

`e`	Pointer to the `Slapi_Entry` structure representing an entry found by the search.
`callback_data`	Pointer to the data specified by the `callback_data` argument in the `slapi_search_internal_callback()` function call.

Returns

0 if successful or -1 if an error occurred.

Example

[To be added]

See Also

send_ldap_referral_fn_ptr_t

`send_ldap_referral_fn_ptr_t` specifies the prototype for a callback function that you can write to send LDAP v3 referrals (search result references) back to the client. You can register your function so that it is called whenever the `slapi_send_ldap_referral()` function is called.

Syntax

```
#include "slapi-plugin.h"
typedef int (*send_ldap_referral_fn_ptr_t)( Slapi_PBlock *pb,
   Slapi_Entry *e, struct berval **refs, struct berval ***urls);
```

Parameters

The function has the following parameters:

`pb`	Parameter block.
`e`	Pointer to the `Slapi_Entry` structure representing the entry that you are working with.
`refs`	Pointer to the NULL-terminated array of `berval` structures containing the LDAP v3 referrals (search result references) found in the entry.
`urls`	Pointer to the array of `berval` structures used to collect LDAP referrals for LDAP v2 clients.

Returns

0 if successful, or -1 if an error occurs.

Description

The `slapi_send_ldap_referral()` function is responsible for sending LDAP v3 referrals (search result references) back to the client. You can replace the function that sends LDAP v3 referrals to the client with your own function. To do this:

1. Write a function with the prototype specified by `send_ldap_result_fn_ptr_t`.
2. In your plug-in initialization function, register your function by setting the `SLAPI_PLUGIN_DB_REFERRAL_FN` parameter in the parameter block to the name of your function.

See "slapi_send_ldap_referral()" on page 339 for information on the default function that sends LDAP v3 referrals to clients.

Example

[To be added]

See Also

`slapi_send_ldap_referral()`.

send_ldap_result_fn_ptr_t

`send_ldap_result_fn_ptr_t` specifies the prototype for a callback function that you can write to send LDAP result codes back to the client. You can register your function so that it is called whenever the `slapi_send_ldap_result()` function is called.

Syntax

```
#include "slapi-plugin.h"
typedef void (*send_ldap_result_fn_ptr_t)( Slapi_PBlock *pb,
   int err, char *matched, char *text, int nentries,
   struct berval **urls );
```

Parameters

The function has the following parameters:

`pb`	Parameter block.
`err`	LDAP result code that you want sent back to the client (for example, `LDAP_SUCCESS`).
`matched`	When sending back an `LDAP_NO_SUCH_OBJECT` result code, use this argument to specify the portion of the target DN that could be matched.
`text`	Error message that you want sent back to the client. Use NULL if you do not want an error message sent back.
`nentries`	When sending back the result code for an LDAP search operation, use this argument to specify the number of matching entries found.
`urls`	When sending back an `LDAP_PARTIAL_RESULTS` result code to an LDAP v2 client or an `LDAP_REFERRAL` result code to an LDAP v3 client, use this argument to specify the array of `berval` structures containing the referral URLs.

Description

The `slapi_send_ldap_result()` function is responsible for sending LDAP result codes back to the client. You can replace the function that sends LDAP result codes to the client with your own function. To do this:

1. Write a function with the prototype specified by `send_ldap_result_fn_ptr_t`.
2. In your plug-in initialization function, register your function by setting the `SLAPI_PLUGIN_DB_RESULT_FN` parameter in the parameter block to the name of your function.

See “slapi_send_ldap_result()” on page 341 for information on the default function that sends LDAP result codes to clients.

Example [To be added]

See Also `slapi_send_ldap_result()`.

send_ldap_search_entry_fn_ptr_t

`send_ldap_result_fn_ptr_t` specifies the prototype for a callback function that you can write to send search results (entries found by a search) back to the client. You can register your function so that it is called whenever the `slapi_send_ldap_search_entry()` function is called.

Syntax

```
#include "slapi-plugin.h"
typedef int (*send_ldap_search_entry_fn_ptr_t)
   ( Slapi_PBlock *pb, Slapi_Entry *e, LDAPControl **ectrls,
   char **attrs, int attrsonly );
```

Description

The `slapi_send_ldap_search_entry()` function is responsible for sending entries found by a search back to the client. You can replace the function that sends entries to the client with your own function. To do this:

1. Write a function with the prototype specified by `send_ldap_search_entry_fn_ptr_t`.
2. In your plug-in initialization function, register your function by setting the `SLAPI_PLUGIN_DB_ENTRY_FN` parameter in the parameter block to the name of your function.

See "slapi_send_ldap_search_entry()" on page 343 for information on the default function that sends entries to clients.

Example [To be added]

See Also `slapi_send_ldap_search_entry()`.

Slapi_Attr

Represents an attribute in an entry.

Syntax `typedef struct slapi_attr Slapi_Attr;`

Description `Slapi_Attr` is the data type for an opaque structure that represents an attribute in a directory entry. In certain cases, your server plug-in may need to work with an entry's attributes.

The following table summarizes the front-end API functions that you can call to work with attributes.

To do this...	Call this function
Iterate through the attributes in an entry	`slapi_entry_first_attr()`, `slapi_entry_next_attr()`
Determine if an entry contains a specific attribute	`slapi_entry_attr_find()`
Get the type of an attribute	`slapi_attr_get_type()`
Get the object identification (OID) of an attribute	`slapi_attr_get_oid()`
Get the values of an attribute	`slapi_attr_get_values()`
Determine if an attribute has the specified value	`slapi_attr_value_find()`
Compare two values of an attribute	`slapi_attr_value_cmp()`
Get the flags set for an attribute	`slapi_attr_get_flags()`
Determine if a specified flag has been set for an attribute	`slapi_attr_flag_is_set()`

See Also `Slapi_Entry`.

slapi_compute_callback_t

Reserved for future use.

slapi_compute_callback_t specifies the prototype for a callback function that is called by the server when generating a computed attribute. If you want to use computed attributes, you should write a function of this type.

Syntax

```
#include “slapi-plugin.h”
typedef int (*slapi_compute_callback_t)
   (computed_attr_context *c, char* type,
   Slapi_Entry *e, slapi_compute_output_t outputfn);
```

Parameters

The function has the following parameters:

c	Pointer to the computed_attr_context structure containing information relevant to the computed attribute.
type	Attribute type of the attribute to be generated.
e	Pointer to the Slapi_Entry structure representing the entry to be sent back to the client.
outputfn	Pointer to the slapi_compute_output_t function responsible for BER-encoding the computed attribute and for adding it to the BER element to be sent to the client.

Returns

One of the following values:

- -1 if the function is not responsible for generating the computed attribute.
- 0 if the function successfully generates the computed attribute.
- An LDAP error code if an error occurred.

Example

[To be added]

See Also

slapi_compute_output_t

Reserved for future use.

slapi_compute_output_t specifies the prototype for a callback function. This function is passed to the slapi_compute_callback_t function.

Syntax

```
#include "slapi-plugin.h"
typedef int (*slapi_compute_output_t)
   (computed_attr_context *c, Slapi_Attr *a, Slapi_Entry *e);
```

Parameters

The function has the following parameters:

c	Pointer to the computed_attr_context structure containing information relevant to the computed attribute.
a	Pointer to the Slapi_Attr structure representing the generated attribute.
e	Pointer to the Slapi_Entry structure representing the entry to be sent back to the client.

Returns

One of the following values:

- 0 if the function successfully BER-encodes the computed attribute and adds it to the BER element to be sent to the client.
- An LDAP error code if an error occurred.

Description

slapi_compute_output_t specifies the prototype for a callback function that BER-encodes a computed attribute and appends it to the BER element to be sent to the client.

You do not need to define a function of this type. The server will pass a function of this type your slapi_compute_callback_t function. In your slapi_compute_callback_t function, you need to call this slapi_compute_output_t function.

For example:

```
static int
my_compute_callback(computed_attr_context *c, char* type,
   Slapi_Entry *e, slapi_compute_output_t outputfn)
{
   ...
```

```
    int rc;
    Slapi_Attr my_computed_attr;
    ...
    /* Call the output function after created the computed
       attribute and setting its values. */
    rc = (*outputfn) (c, &my_computed_attr, e);
    ...
}
```

In the example above, the `slapi_compute_output_t` function `outputfn` is passed in as an argument to `my_compute_callback` function. After generating the computed attribute, you need to call `outputfn`, passing it the context, the newly created attribute, and the entry. `outputfn` BER-encodes the attribute and appends it to the BER element to be sent to the client.

Note that you do not need to define `outputfn` yourself. You just need to call the function passed in as the last argument.

Example [To be added]

See Also

Slapi_CondVar

Represents a condition variable in the server plug-in.

Syntax

```
#include slapi-plugin.h
typedef struct slapi_condvar Slapi_CondVar;
```

Description

`Slapi_CondVar` is the data type for an opaque structure that represents a condition variable in the server plug-in.

See Also

`slapi_new_condvar()`, `slapi_wait_condvar()`, `slapi_notify_condvar()`, `slapi_destroy_condvar()`.

Slapi_Entry

Represents an entry in the directory.

Syntax `typedef struct slapi_entry Slapi_Entry;`

Description `Slapi_Entry` is the data type for an opaque structure that represents an entry in the directory. In certain cases, your server plug-in may need to work with an entry in the directory.

The following table summarizes the front-end API functions that you can call to work with attributes.

To do this...	Call this function
Allocate memory for a entry structure	`slapi_entry_alloc()`
Copy an existing entry structure	`slapi_entry_dup()`
Free an entry structure from memory	`slapi_entry_free()`
Create an entry structure from an LDIF description of the entry	`slapi_str2entry()`
Generate an LDIF description of an entry from its entry structure	`slapi_entry2str()`
Get the DN of an entry	`slapi_entry_get_dn()`
Set the DN of an entry	`slapi_entry_set_dn()`
Determine if an entry complies with the schema	`slapi_entry_schema_check()`
Iterate through the attributes in an entry	`slapi_entry_first_attr()`, `slapi_entry_next_attr()`
Determine if an entry contains a specified attribute	`slapi_entry_attr_find()`
Add new values to an attribute in an entry	`slapi_entry_add_values()`
Add values to an attribute in an entry without replacing the existing attribute values	`slapi_entry_attr_merge()`
Remove values from an attriubte in an entry	`slapi_entry_delete_values()`
Determine if any values from the entry's RDN are present as attributes	`slapi_entry_rdn_values_present()`

See Also `Slapi_Attr`.

Slapi_Filter

Represents a search filter.

Syntax

```
typedef struct slapi_filter Slapi_Filter;
```

Description

`Slapi_Filter` is the data type for an opaque structure that represents an search filter. (For more information on working with search filters, see “Working with Search Filters” on page 56.)

The following table summarizes the front-end API functions that you can call to work with attributes.

To do this...	Call this function
Determine if an entry matches a filter's criteria	`slapi_filter_test()`
Get the filter type	`slapi_filter_get_choice()`
Get the attribute type and value used for comparison in a filter (only applicable to `LDAP_FILTER_EQUALITY`, `LDAP_FILTER_GE`, `LDAP_FILTER_LE`, and `LDAP_FILTER_APPROX` searches)	`slapi_filter_get_ava()`
Get the type of attribute that the filter is searching for (only applicable to `LDAP_FILTER_PRESENT` searches)	`slapi_filter_get_type()`
Get the substring pattern used for the filter (applicable only to `LDAP_FILTER_SUBSTRING` searches)	`slapi_filter_get_subfilt()`
Convert a string representation of a filter to a filter of the datatype `Slapi_Filter`	`slapi_str2filter()`
Construct a new `LDAP_FILTER_AND`, `LDAP_FILTER_OR`, or `LDAP_FILTER_NOT` filter from other filters	`slapi_filter_join()`
Get the components of a filter (only applicable to `LDAP_FILTER_AND`, `LDAP_FILTER_OR`, and `LDAP_FILTER_NOT` searches)	`slapi_filter_list_first()`, `slapi_filter_list_next()`
Free a filter from memory	`slapi_filter_free()`

Slapi_MatchingRuleEntry

Represents a matching rule.

Syntax

```
#include slapi-plugin.h
typedef struct slapi_matchingRuleEntry Slapi_MatchingRuleEntry;
```

Description

`Slapi_MatchingRuleEntry` is the data type for an opaque structure that represents a matching rule.

See Also

`slapi_matchingrule_free()`, `slapi_matchingrule_get()`, `slapi_matchingrule_new()`, `slapi_matchingrule_register()`, `slapi_matchingrule_set()`.

Slapi_Mutex

Represents a mutex in the server plug-in.

Syntax

```
#include slapi-plugin.h
typedef struct slapi_mutex Slapi_Mutex;
```

Description

`Slapi_Mutex` is the data type for an opaque structure that represents a mutual exclusive lock in the server plug-in.

See Also

`slapi_new_mutex()`, `slapi_lock_mutex()`, `slapi_unlock_mutex()`, `slapi_destroy_mutex()`.

Slapi_PBlock

Contains name-value pairs that you can get or set for each LDAP operation.

Syntax

```
#include slapi-plugin.h
typedef struct slapi_pblock Slapi_PBlock;
```

Description

`Slapi_PBlock` contains name-value pairs that you can use to retrieve information from the server and set information to be used by the server.

For most types of plug-in functions, the server passes in a `Slapi_PBlock` structure that typically includes data relevant to the operation being processed. You can get the value of a parameter by calling the `slapi_pblock_get()` function.

For example, when the plug-in function for an LDAP bind operation is called, the server puts the DN and credentials in the `SLAPI_BIND_TARGET` and `SLAPI_BIND_CREDENTIALS` parameters of the `Slapi_PBlock` structure. You can call `slapi_pblock_get()` to get the DN and credentials of the client requesting authentication.

For plug-in initialization functions, you can use the `Slapi_PBlock` structure to pass information to the server, such as the description of your plug-in and the names of your plug-in functions. You can set the value of a parameter by calling the `slapi_pblock_set()` function.

For example, in order to register a pre-operation bind plug-in function, you need to call `slapi_pblock_set()` to set the version number, description, and name of the plug-in function as the `SLAPI_PLUGIN_VERSION`, `SLAPI_PLUGIN_DESCRIPTION`, and `SLAPI_PLUGIN_PRE_BIND_FN` parameters.

The available parameters that you can use depends on the type of plug-in function you are writing. For information about the available parameters, see Chapter 15, “Parameter Reference.”

See Also

`slapi_pblock_new()`, `slapi_pblock_get()`, `slapi_pblock_set()`, `slapi_pblock_destroy()`.

Slapi_PluginDesc

`Slapi_PluginDesc` represents information about a server plug-in. In your initialization function, you specify information about your plug-in in this structure and call `slapi_pblock_set()` to put the structure in the `SLAPI_PLUGIN_DESCRIPTION` parameter.

`Slapi_PluginDesc` is defined as follows:

```
typedef struct slapi_plugindesc {
   char *spd_id;
   char *spd_vendor;
   char *spd_version;
   char *spd_description;
} Slapi_PluginDesc;
```

The fields in this structure are described below:

`spd_id`	Unique identifier for the server plug-in.
`spd_vendor`	Name of the vendor supplying the server plug-in (for example, `Airius.com`).
`spd_version`	Version of the server plug-in used for your own tracking purposes (for example, `0.5`). Note that this is different from the value of the `SLAPI_PLUGIN_VERSION`, which specifies the general version of plug-in technology; the Netscape Directory Server uses that version to determine if it supports a plug-in.
`spd_description`	Description of the server plug-in.

For more information on using Slapi_PluginDesc to specify plug-in information, see “Specifying Information about the Plug-In” on page 35.

Chapter

14

Function Reference

This chapter summarizes the front-end functions available in the API. You can call these functions in your own server plug-in functions.

Summary of Front-End Functions

The server plug-in API includes the following functions:

- `slapi_access_allowed()`
- `slapi_acl_check_mods()`
- `slapi_add_entry_internal()`
- `slapi_add_internal()`
- `slapi_attr_basetype()`
- `slapi_attr_flag_is_set()`
- `slapi_attr_get_flags()`
- `slapi_attr_get_oid()`
- `slapi_attr_get_type()`

- slapi_attr_get_values()
- slapi_attr_type2plugin()
- slapi_attr_type_cmp()
- slapi_attr_value_cmp()
- slapi_attr_value_find()
- slapi_berval_cmp()
- slapi_call_syntax_assertion2keys_ava()
- slapi_call_syntax_assertion2keys_sub()
- slapi_call_syntax_values2keys()
- slapi_ch_bvdup()
- slapi_ch_bvecdup()
- slapi_ch_calloc()
- slapi_ch_free()
- slapi_ch_malloc()
- slapi_ch_realloc()
- slapi_ch_strdup()
- slapi_compute_add_evaluator()
- slapi_control_present()
- slapi_delete_internal()
- slapi_destroy_condvar()
- slapi_destroy_mutex()
- slapi_dn_beparent()
- slapi_dn_ignore_case()
- slapi_dn_isbesuffix()

- slapi_dn_isparent()
- slapi_dn_isroot()
- slapi_dn_issuffix()
- slapi_dn_normalize()
- slapi_dn_normalize_case()
- slapi_dn_parent()
- slapi_entry2str()
- slapi_entry_add_rdn_values()
- slapi_entry_add_values()
- slapi_entry_alloc()
- slapi_entry_attr_delete()
- slapi_entry_attr_find()
- slapi_entry_attr_get_charptr()
- slapi_entry_attr_get_int()
- slapi_entry_attr_hasvalue()
- slapi_entry_attr_merge()
- slapi_entry_attr_replace()
- slapi_entry_attr_set_charptr()
- slapi_entry_attr_set_long()
- slapi_entry_delete_values()
- slapi_entry_dup()
- slapi_entry_first_attr()
- slapi_entry_free()
- slapi_entry_get_dn()

- slapi_entry_next_attr()
- slapi_entry_rdn_values_present()
- slapi_entry_schema_check()
- slapi_entry_set_dn()
- slapi_filter_free()
- slapi_filter_get_ava()
- slapi_filter_get_choice()
- slapi_filter_get_subfilt()
- slapi_filter_get_type()
- slapi_filter_join()
- slapi_filter_list_first()
- slapi_filter_list_next()
- slapi_filter_test()
- slapi_get_supported_controls()
- slapi_get_supported_extended_ops()
- slapi_get_supported_saslmechanisms()
- slapi_ldap_init()
- slapi_ldap_unbind()
- slapi_lock_mutex()
- slapi_log_error()
- slapi_matchingrule_free()
- slapi_matchingrule_get()
- slapi_matchingrule_new()
- slapi_matchingrule_register()

- slapi_matchingrule_set()
- slapi_matchingrule_unregister()
- slapi_modify_internal()
- slapi_modrdn_internal()
- slapi_mr_filter_index()
- slapi_mr_indexer_create()
- slapi_new_condvar()
- slapi_new_mutex()
- slapi_notify_condvar()
- slapi_op_abandoned()
- slapi_pblock_destroy()
- slapi_pblock_get()
- slapi_pblock_new()
- slapi_pblock_set()
- slapi_pw_find()
- slapi_register_supported_control()
- slapi_register_supported_saslmechanism()
- slapi_search_internal()
- slapi_search_internal_callback()
- slapi_send_ldap_referral()
- slapi_send_ldap_result()
- slapi_send_ldap_search_entry()
- slapi_str2entry()
- slapi_str2filter()

- slapi_unlock_mutex()
- slapi_value_find()
- slapi_verify_aci_syntax()
- slapi_wait_condvar()

The rest of this chapter describes each function in more detail.

slapi_access_allowed()

Determines if a user (who is requesting the current operation) has the access rights to perform an operation on a given entry, attribute, or value.

Syntax

```
#include "slapi-plugin.h"
int slapi_access_allowed( Slapi_PBlock *pb, Slapi_Entry *e,
   char *attr, struct berval *val, int access );
```

Parameters

The function has the following parameters:

`pb`	Parameter block passed into this function.
`e`	Entry that you want to check the access rights for.
`attr`	Attribute that you want to check the access rights for.
`val`	Pointer to the `berval` structure containing the value that you want to check the access rights for.
`access`	Type of access rights that you want to check for (for example, to check for write access, pass `SLAPI_ACL_WRITE` as the value of this argument).

The value of the `access` argument can be one of the following:

`SLAPI_ACL_ADD`	Permission to add a specified entry.
`SLAPI_ACL_COMPARE`	Permission to compare the specified values of an attribute in an entry.
`SLAPI_ACL_DELETE`	Permission to delete a specified entry.
`SLAPI_ACL_READ`	Permission to read a specified attribute.
`SLAPI_ACL_SEARCH`	Permission to search on a specified attribute or value.
`SLAPI_ACL_WRITE`	Permission to write a specified attribute or value or to rename a specified entry.

Returns

One of the following values:

- `LDAP_SUCCESS`, if the user has the specified rights to the entry, attribute, or value.
- `LDAP_INSUFFICIENT_ACCESS`, if the user does not have the specified rights to the entry, attribute, or value.

- One of the following error codes, if a problem occurs:

`LDAP_OPERATIONS_ERROR`	An error occured while executing the operation. This error can occur if, for example, the type of access rights you've specified are not recognized by the server (in other words, you did not pass a value from the previous table).
`LDAP_INVALID_SYNTAX`	Invalid syntax was specified. This error can occur if the ACLassociated with an entry, attribute, or value uses the wrong syntax.
`LDAP_UNWILLING_TO_PERFORM`	The DSA (this directory server) is unable to perform the specified operation. This error can occur if, for example, you are requesting write access to a read-only database.

Description Call `slapi_access_allowed()` to determine if a user has access rights to a specified entry, attribute, or value. The function performs this check for users who request the operation that invokes this plug-in.

For example, suppose you are writing a preoperation plug-in for the add operation. You can call this function to determine if users have the proper access rights before they can add an entry to the directory.

As part of the process of determining if the user has access rights, the `slapi_access_allowed()` function does the following:

- Checks to see if the user requesting the operation is the root DN.

  If so, the function returns `LDAP_SUCCESS`. (The root DN has permission to perform any operation.)

- Gets information about the operation being requested, the connection to the client, and the back-end database where directory information is stored.

  - If (for some reason) the function cannot determine which operation is being requested, the function returns `LDAP_OPERATIONS_ERROR`.

  - If no connection to a client exists (in other words, if the request for the operation was made by the server or its back-end), the function returns `LDAP_SUCCESS`. (The server and its back-end are not restricted by access control lists.)

- If the back-end database is read-only and the request is checking for write access (`SLAPI_ACL_WRITE`), the function returns `LDAP_UNWILLING_TO_PERFORM`.

- Determines if the user requesting the operation is attempting to modify his or her own entry.

  ACLs can be set up to allow users the rights to modify their own entries. The `slapi_access_allowed()` function checks for this condition.

Example [To be added]

See Also

slapi_acl_check_mods()

Determines if a user has the rights to perform the specified modifications on an entry.

Syntax

```
#include "slapi-plugin.h"
int slapi_acl_check_mods( Slapi_PBlock *pb, Slapi_Entry *e,
   LDAPMod **mods, char **errbuf );
```

Parameters

The function has the following parameters:

`pb`	Parameter block passed into this function.
`e`	Entry that you want to check the access for.
`mods`	Array of `LDAPMod` structures, representing the modifications to be made to the entry.
`errbuf`	Pointer to a string containing an error message, if an error occurs when the function executes.

Returns

One of the following values:

- `LDAP_SUCCESS`, if the user has write permission to the values in the specified attributes.
- `LDAP_INSUFFICIENT_ACCESS`, if the user does not have write permission to the values of the specified attribute.
- One of the following error codes, if a problem occurs:

`LDAP_OPERATIONS_ERROR`	An error occured while executing the operation.
`LDAP_INVALID_SYNTAX`	Invalid syntax was specified. This error can occur if the ACLassociated with an entry, attribute, or value uses the wrong syntax.
`LDAP_UNWILLING_TO_PERFORM`	The DSA (this directory server) is unable to perform the specified operation. This error can occur if, for example, you are requesting write access to a read-only database.

Description

Call `slapi_acl_check_mods()` to determine if a user has access rights to modify the specified entry. The function performs this check for users who request the operation that invokes this plug-in.

For example, suppose you are writing a database plug-in. You can call this function to determine if users have the proper access rights before they can add, modify, or delete entries from the database.

As part of the process of determining if the user has access rights, the `slapi_access_allowed()` function does the following:

- Checks to access control for the directory is disabled (for example, if the `slapd.conf` file contains the directive `accesscontrol off`).

  If access control is disabled, the function returns `LDAP_SUCCESS`.

- For each value in each attribute specified in the `LDAPMod` array, the function determines if the user has permissions to write to that value. (Essentially, the function calls `slapi_access_allowed()` with `SLAPI_ACL_WRITE` as the access right to check.)

  - If (for some reason) the function cannot determine which operation is being requested, the function returns `LDAP_OPERATIONS_ERROR`.

  - If no connection to a client exists (in other words, if the request for the operation was made by the server or its back-end), the function returns `LDAP_SUCCESS`. (The server and its back-end are not restricted by access control lists.)

  - If the back-end database is read-only and the request is checking for write access (`SLAPI_ACL_WRITE`), the function returns `LDAP_UNWILLING_TO_PERFORM`.

Example [To be added]

See Also

slapi_add_entry_internal()

Performs an LDAP add operation to add a new directory entry (specified by an `Slapi_Entry` structure) from your plug-in. (To add an entry specified a string DN and an array of `LDAPMod` structures, call `slapi_add_internal()` instead.)

Syntax

```
#include "slapi-plugin.h"
Slapi_PBlock *slapi_add_entry_internal( Slapi_Entry * e,
   LDAPControl **controls, int log_change );
```

Parameters

The function has the following parameters:

`mods`	Pointer to an `Slapi_Entry` structure representing the new entry that you want to add.
`controls`	A NULL-terminated array of LDAP controls (see "LDAPControl" on page 193) that you want applied to the add operation.
`log_change`	Specifies whether or not you want to log the changes to the replication log. • If 1, write the changes to the replication log. • If 0, do not write the changes to the replication log.

Returns

A new parameter block (see "Slapi_PBlock" on page 215) with the following parameter set:

- `SLAPI_PLUGIN_INTOP_RESULT` specifies the LDAP result code for the internal LDAP operation (for example, `LDAP_SUCCESS` if the operation is successful or `LDAP_PARAM_ERROR` if an invalid parameter is used).

  If the DN of the new entry has a suffix that is not served by the Directory Server, `SLAPI_PLUGIN_INTOP_RESULT` is set to `LDAP_REFERRAL`.

Description

This function allows you to add an entry to the directory from a plug-in function.

Unlike the standard LDAP add operation, no LDAP result code to a client. Instead, the result code is placed in a parameter block that is returned by the function.

Example

[To be added]

See Also

slapi_add_internal()

Performs an LDAP add operation to add a new directory entry (specified by a DN and a set of attributes) from your plug-in. (To add an entry specified in an `Slapi_Entry` structure, call `slapi_add_entry_internal()` instead.)

Syntax

```
#include "slapi-plugin.h"
Slapi_PBlock *slapi_add_internal( char * dn, LDAPMod **attrs,
   LDAPControl **controls, int log_change );
```

Parameters

The function has the following parameters:

`dn`	Distinguished name (DN) of the entry that you want to add.
`mods`	Pointer to a NULL-terminated array of pointers to `LDAPMod` structures representing the attributes of the new entry that you want to add.
`controls`	A NULL-terminated array of LDAP controls (see "LDAPControl" on page 193) that you want applied to the add operation.
`log_change`	Specifies whether or not you want to log the changes to the replication log. • If 1, write the changes to the replication log. • If 0, do not write the changes to the replication log.

Returns

A new parameter block (see "Slapi_PBlock" on page 215) with the following parameter set:

- `SLAPI_PLUGIN_INTOP_RESULT` specifies the LDAP result code for the internal LDAP operation (for example, `LDAP_SUCCESS` if the operation is successful or `LDAP_PARAM_ERROR` if an invalid parameter is used).

  If the DN of the new entry has a suffix that is not served by the Directory Server, `SLAPI_PLUGIN_INTOP_RESULT` is set to `LDAP_REFERRAL`.

Description

This function allows you to add an entry to the directory from a plug-in function. The arguments for this function are similar to the arguments for the standard `ldap_add()` function in the Netscape LDAP C SDK.

Unlike the standard LDAP add operation, no LDAP result code to a client. Instead, the result code is placed in a parameter block that is returned by the function.

Example

[To be added]

See Also

slapi_attr_basetype()

Returns the base type of an attribute (for example, if given `cn;lang-jp`, returns `cn`).

Syntax

```
#include “slapi-plugin.h”
char *slapi_attr_basetype( char *type, char *buf,
   size_t bufsiz );
```

Parameters

The function has the following parameters:

`type`	Attribute type that you want to get the base type from.
`buf`	Buffer to hold the returned base type.
`bufsiz`	Size of the buffer.

Returns

NULL if the base type fits in the buffer. If the base type is longer than the buffer, the function allocates memory for the base type and returns a pointer to it. You should free the returned base type when done by calling `slapi_ch_free()`.

Example

[To be added]

See Also

slapi_attr_flag_is_set()

Determines if certain flags are set for a particular attribute. These flags can identify an attribute as a single-valued attribute, an operational attribute, or as a read-only attribute.

Syntax

```
#include "slapi-plugin.h"
int slapi_attr_flag_is_set( Slapi_Attr *attr,
   unsigned long flag );
```

Parameters

The function has the following parameters:

`attr`	Attribute that you want to check.
`flag`	Flag that you want to check in the attribute.

The value of the `flag` argument can be one of the following:

`SLAPI_ATTR_FLAG_SINGLE`	Flag that determines if the attribute is single-valued.
`SLAPI_ATTR_FLAG_OPATTR`	Flag that determines if the attribute is an operational attribute.
`SLAPI_ATTR_FLAG_READONLY`	Flag that determines if the attribute is read-only.

Returns

1 if the specified flag is set, or 0 if the specified flag is not set.

Example

[To be added]

See Also

slapi_attr_get_flags()

Gets the flags associated with the specified attribute. These flags can identify an attribute as a single-valued attribute, an operational attribute, or as a read-only attribute.

Syntax

```
#include “slapi-plugin.h”
int slapi_attr_get_flags( Slapi_Attr *attr, unsigned long *flags
);
```

Parameters

The function has the following parameters:

`attr`	Attribute that you want to get the flags for.
`flags`	When you call `slapi_attr_get_flags()`, this parameter is set to a pointer to the flags of the specified attribute. Do not free the flags; the flags are part of the actual data in the attribute, not a copy of the data.

To determine which flags have been set, you can bitwise AND the value of the `flags` argument with one or more of the following:

`SLAPI_ATTR_FLAG_SINGLE`	Flag that determines if the attribute is single-valued.
`SLAPI_ATTR_FLAG_OPATTR`	Flag that determines if the attribute is an operational attribute.
`SLAPI_ATTR_FLAG_READONLY`	Flag that determines if the attribute is read-only.

Returns

0 if successful.

Example

[To be added]

See Also

slapi_attr_get_oid()

Gets the object identifier (OID) for the specified attribute.

Syntax

```
#include "slapi-plugin.h"
int slapi_attr_get_oid( Slapi_Attr *attr, char **oid );
```

Parameters

The function has the following parameters:

`attr`	Attribute that you want to get the OID for.
`oid`	When you call `slapi_attr_get_oid()`, this parameter is set to a pointer to the OID of the specified attribute. Do not free this OID; the OID is part of the actual data in the attribute, not a copy of the data.

Returns

0 if successful, or -1 if the attribute's OID could not be determined.

Example

[To be added]

See Also

slapi_attr_get_type()

Gets the name of the attribute type from a specified attribute.

Syntax

```
#include "slapi-plugin.h"
int slapi_attr_get_type( Slapi_Attr *attr, char **type );
```

Parameters

The function has the following parameters:

`attr`	Attribute that you want to get the type for.
`type`	When you call `slapi_attr_get_type()`, this parameter is set to a pointer to the type of the specified attribute. Do not free this attribute type; the type is part of the actual data in the attribute, not a copy of the data.

Returns

0 if successful.

Example

[To be added]

See Also

slapi_attr_get_values()

Gets the value of the specified attribute.

Syntax

```
#include “slapi-plugin.h”
int slapi_attr_get_values( Slapi_Attr *attr, struct berval
***vals );
```

Parameters

The function has the following parameters:

`attr`	Attribute that you want to get the flags for.
`vals`	When you call `slapi_attr_get_values()`, this parameter is set to a pointer to a NULL-terminated array of `berval` structures representing the values of the attribute. Do not free the array; the array is part of the actual data in the attribute, not a copy of the data.

Returns

0 if successful.

Description

Call `slapi_attr_get_values()` to get the values of a specified attribute. Because an attribute can have multiple values, this function gets the value as an array of `berval` structures.

Example

[To be added]

See Also

slapi_attr_type2plugin()

Gets a pointer to information about the syntax plug-in responsible for handling the specified attribute type.

Syntax

```
#include “slapi-plugin.h”
int slapi_attr_type2plugin( char *type, void **pi );
```

Parameters

The function has the following parameters:

`type`	Type of attribute that you want to get the plug-in for.
`pi`	Pointer to the plug-in structure.

Returns

0 if successful, or -1 if the corresponding plug-in is not found.

Description

Syntax plug-ins are plug-ins that you can write to index and search for specific attribute types.

Example

[To be added]

See Also

slapi_attr_type_cmp()

Compares two attribute types to determine if they are the same.

Syntax

```
#include "slapi-plugin.h"
int slapi_attr_type_cmp( char *t1, char *t2, int opt );
```

Parameters

The function has the following parameters:

`t1` Name of the first attribute type that you want to compare.

`t2` Name of the second attribute type that you want to compare.

`opt` One of the following values:

- `0`
  Compare the types as is.
- `1`
  Compare only the base names of the types (for example, if the type is `cn;lang-en`, the function compares only the `cn` part of the type).
- `2`
  Ignore any options in the second type that are not in the first type. For example, if the first type is `cn` and the second type is `cn;lang-en`, the `lang-en` option in the second type is not part of the first type. In this case, the function considers the two types to be the same.

Returns

0 if the type names are equal, or a non-zero value if they are not equal.

Example

[To be added]

See Also

slapi_attr_value_cmp()

Compares two values for a given attribute to determine if they are equal.

Syntax

```
#include “slapi-plugin.h”
int slapi_attr_value_cmp( Slapi_Attr *attr, struct berval *v1,
struct berval *v2 );
```

Parameters The function has the following parameters:

`attr`	Attribute used to determine how these values are compared (for example, if the attribute contains case-insensitive strings, the strings are compared without regard to case).
`v1`	Pointer to the `berval` structure containing the first value that you want to compare.
`v2`	Pointer to the `berval` structure containing the second value that you want to compare.

Returns 0 if the values are equal, or -1 if they are not equal.

Example [To be added]

See Also

slapi_attr_value_find()

Determines is an attribute contains a given value.

Syntax

```
#include "slapi-plugin.h"
int slapi_attr_value_find( Slapi_Attr *a, struct berval *v );
```

Parameters The function has the following parameters:

a	Attribute that you want to check.
v	Pointer to the `berval` structure containing the value that you want to search for.

Returns 0 if the attribute contains the specified value, or -1 if the attribute does not contain the specified value.

Example [To be added]

See Also

slapi_berval_cmp()

Compares two `berval` structures to determine if they are equal.

Syntax

```
#include "slapi-plugin.h"
int slapi_berval_cmp (const struct berval *L,
   const struct berval *R);
```

Parameters

The function has the following parameters:

`L`	Pointer to the first `berval` structure that you want to compare.
`R`	Pointer to the second `berval` structure that you want to compare.

Returns

One of the following values:

- a negative value if L is less than R
- 0 if L is equal to R
- a positive value if L is greater than R

Example

[To be added]

See Also

slapi_call_syntax_assertion2keys_ava()

[?? Still working on the doc for this function. Not done yet. But any help would be appreciated. ??]

When processing a search, calls the function (defined in the specified syntax plug-in) responsible for returning an array of values (specified by the search filter) to compare against the entries in the directory.

This function applies to searches that use the filter types `LDAP_FILTER_EQUALITY` and `LDAP_FILTER_APPROX`.

Syntax

```
#include "slapi-plugin.h"
int slapi_call_syntax_assertion2keys_ava( void *vpi, struct
berval *val, struct berval ***ivals, int ftype );
```

Parameters

The function has the following parameters:

`vpi`	Handle to plug-in for this attribute type
`val`	Pointer to the `berval` structure containing the value from the search filter (for example, if the filter is `ou=Accounting`, the argument `val` is `Accounting`)
`ivals`	Pointer to an array of `berval` structures containing the values returned by the plug-in function (these values can now be compared against entries in the directory)
`ftype`	Type of filter (for example, `LDAP_FILTER_EQUALITY`)

Returns

0 if successful, or -1 if an error occurs (for example, if the corresponding function for the specified plug-in is not found).

Description

When processing a search that uses an attribute-value assertion (AVA) filter (for example, `ou=Accounting` or `ou=~Accounting`), the back-end needs to compare the value specified in the search filter against the value of the specified attribute in each entry.

The function invokes the syntax plug-in specified by the `vpi` argument. (This is the plug-in associated with the type of attribute used in the search. You can get this handle by calling the `slapi_attr_type2plugin()` function.)

The syntax plug-in function invoked by this function is responsible for comparing the value specified by `val` against the actual values of the attributes in the directory entries. The syntax plug-in function returns a list of matching entry keys (the `ivals` argument) to the backend.

Example [To be added]

See Also

slapi_call_syntax_assertion2keys_sub()

[?? Still working on the doc for this function. Not done yet. But any help would be appreciated. ??]

When processing a search, calls the function (defined in the specified syntax plug-in) responsible for returning an array of values (specified by the search filter) to compare against the entries in the directory.

This function applies to searches that use the filter type `LDAP_FILTER_SUBSTRINGS`.

Syntax

```
#include "slapi-plugin.h"
int slapi_call_syntax_assertion2keys_sub( void *vpi, char
*initial, char **any, char *final, struct berval ***ivals );
```

Parameters

The function has the following parameters:

`vpi`	Handle to plug-in for this attribute type
`initial`	"Starts with" value from the search filter (for example, if the filter is `ou=Sales*`, the argument `initial` is `Sales`)
`any`	Array of "contains" values from the search filter (for example, if the filter is `ou=*Corporate*Sales*`, the argument `any` is an array containing `Corporate` and `Sales`)
`final`	"Ends with" value from the search filter (for example, if the filter is `ou=*Sales`, the argument `final` is `Sales`)
`ivals`	Pointer to an array of `berval` structures containing the values returned by the plug-in function (these values can now be compared against entries in the directory)

Returns

0 if successful, or -1 if an error occurs (for example, if the corresponding function for the specified plug-in is not found).

Description

The `ldbm` backend (the default backend database) calls this function when processing searches in which the filter type is `LDAP_FILTER_SUBSTRINGS`.

The function invokes the syntax plug-in specified by the `vpi` argument. (This is the plug-in associated with the type of attribute used in the search. You can get this handle by calling the `slapi_attr_type2plugin()` function.)

The syntax plug-in function invoked by this function is responsible for comparing the values (specified by `initial`, `any`, and `final`) against the actual values of the attributes in the directory entries. The syntax plug-in function returns a list of matching entry keys (the `ivals` argument) to the backend.

Example [To be added]

See Also

slapi_call_syntax_values2keys()

[?? Still working on the doc for this function. Not done yet. But any help would be appreciated. ??]

When adding or removing values from an index, calls the function (defined in the specified syntax plug-in) responsible for returning an array of keys matching the specified values.

Syntax

```
#include "slapi-plugin.h"
int slapi_call_syntax_values2keys( void *vpi,
   struct berval **vals, struct berval ***ivals, int ftype );
```

Parameters

The function has the following parameters:

`vpi`	Handle to plug-in for this attribute type
`vals`	Pointer to the `berval` structure containing the value to add or delete
`ivals`	Pointer to an array of `berval` structures containing the values returned by the plug-in function (these values can now be compared against entries in the directory)
`ftype`	Type of filter (for example, `LDAP_FILTER_EQUALITY`)

Returns

0 if successful, or -1 if an error occurs (for example, if the corresponding function for the specified plug-in is not found).

Description

Example

[To be added]

See Also

slapi_ch_bvdup()

Makes a copy of an existing `berval` structure.

Syntax

```
#include "slapi-plugin.h"
extern struct berval* slapi_ch_bvdup( const struct berval *v );
```

Parameters

The function has the following parameters:

v — Pointer to the `berval` structure that you want to copy.

Returns

Pointer to the new copy of the `berval` structure. If the structure cannot be duplicated (for example, if no more virtual memory exists), the `slapd` program terminates.

Example

[To be added]

See Also

slapi_ch_bvecdup()

Makes a copy of an array of existing `berval` structures.

Syntax

```
#include “slapi-plugin.h”
extern
struct berval** slapi_ch_bvecdup (const struct berval **v);
```

Parameters

The function has the following parameters:

v	Pointer to the array of `berval` structures that you want to copy.

Returns

Pointer to an array of the new copy of the `berval` structures. If the structures cannot be duplicated (for example, if no more virtual memory exists), the `slapd` program terminates.

Example

[To be added]

See Also

slapi_ch_calloc()

Allocates space for an array of a number of elements of a specified size. This function calls the standard `calloc()` C function and terminates the `slapd` server with an “out of memory” error message if memory cannot be allocated.

Syntax

```
#include “slapi-plugin.h”
char * slapi_ch_calloc( unsigned long nelem,
   unsigned long size );
```

Parameters

The function has the following parameters:

`nelem`	Number of elements that you want to allocate memory for.
`size`	Size of the element that you want to allocate memory for.

Returns

Pointer to the newly allocated space of memory. If space cannot be allocated (for example, if no more virtual memory exists), the `slapd` program terminates.

Example

[To be added]

See Also

slapi_ch_free()

Frees space allocated by the `slapi_ch_malloc()`, `slapi_ch_realloc()`, and `slapi_ch_calloc()` functions and sets the pointer to NULL. Call this function instead of the standard `free()` C function.

Syntax

```
#include "slapi-plugin.h"
void slapi_ch_free( void **ptr );
```

Parameters

The function has the following parameters:

`ptr`	Address of the pointer to the block of memory that you want to free. If NULL, no action occurs.

Example

[To be added]

See Also

slapi_ch_malloc()

Allocates space in memory. This function calls the standard `malloc()` C function and terminates the `slapd` server with an “out of memory” error message if memory cannot be allocated.

Syntax

```
#include “slapi-plugin.h”
char * slapi_ch_malloc( unsigned long size );
```

Parameters

The function has the following parameters:

`size`	Size of the space that you want to allocate memory for.

Returns

Pointer to the newly allocated space of memory. If space cannot be allocated (for example, if no more virtual memory exists), the `slapd` program terminates.

Example

[To be added]

See Also

slapi_ch_realloc()

Changes the size of a block of allocated memory. This function calls the standard `realloc()` C function and terminates the `slapd` server with an "out of memory" error message if memory cannot be allocated.

Syntax

```
#include "slapi-plugin.h"
char * slapi_ch_realloc( char *block, unsigned long size );
```

Parameters

The function has the following parameters:

`block`	Pointer to an existing block of allocated memory.
`size`	New size of the block of memory you want allocated.

Returns

Pointer to the reallocated space of memory. If space cannot be allocated (for example, if no more virtual memory exists), the `slapd` program terminates.

Example

[To be added]

See Also

slapi_ch_strdup()

Makes a copy of an existing string. This function calls the standard `strdup()` C function and terminates the `slapd` server with an “out of memory” error message if memory cannot be allocated.

Syntax

```
#include “slapi-plugin.h”
char * slapi_ch_strdup( char *s );
```

Parameters

The function has the following parameters:

s	Pointer to the string you want to copy.

Returns

Pointer to a copy of the string. If space cannot be allocated (for example, if no more virtual memory exists), the `slapd` program terminates.

Example

[To be added]

See Also

slapi_compute_add_evaluator()

Reserved for future use.

Registers the specified function as an evaluator that the server will call to generate a computed attribute.

Syntax

```
#include “slapi-plugin.h”
int slapi_compute_add_evaluator( slapi_compute_callback_t
function);
```

Parameters

The function has the following parameters:

`function`	Function that you want to register.

Returns

One of the following values:

- 0 if the function is successfully registered.
- `ENOMEM` if memory cannot be allocated to register this function.

Description

The `slapi_compute_add_evaluator()` function registers a function of the `slapi_compute_callback_t` type as an evaluator of computed attributes.

Before the server sends an entry as a search result back to the client, the server determines if any of the requested attributes are computed attributes and generates the values for those attributes.

To do this, the server calls each registered evaluator function for each individually requested attribute. An evaluator function has the type `slapi_compute_callback_t`. This function returns -1 if it is not responsible for generating the specified attribute, 0 if it successfully generates the attribute, or an LDAP error code (greater than zero) if an error occurs.

If you want to set up the server to generate the value of a computed attribute and send the attribute back with each entry, you can define an evaluator function and register the function with the server by calling the `slapi_compute_add_evaluator()` function.

Example

[To be added]

See Also

slapi_control_present()

Determines whether or not the specified object identification (OID) identifies a control that is present in a list of controls.

Syntax

```
#include "slapi-plugin.h"
int slapi_control_present( LDAPControl **controls, char *oid,
   struct berval **val, int *iscritical );
```

Parameters

The function has the following parameters:

`controls`	List of controls that you want to check.
`oid`	OID of the control that you want to find.
`val`	If the control is present in the list of controls, specifies the pointer to the `berval` structure containing the value of the control.
`iscritical`	If the control is present in the list of controls, specifies whether or not the control is critical to the operation of the server. • 0 means that the control is not critical to the operation. • 1 means that the control is critical to the operation.

Returns

1 if the specified control is present in the list of controls, or 0 if the control is not present in the list of controls.

Example

[To be added]

See Also

slapi_delete_internal()

Performs an LDAP delete operation to remove a directory entry when called from your plug-in.

Syntax

```
#include "slapi-plugin.h"
Slapi_PBlock *slapi_delete_internal( char * dn,
   LDAPControl **controls, int log_change );
```

Parameters

The function has the following parameters:

`dn`	Distinguished name (DN) of the entry that you want to delete.
`controls`	A NULL-terminated array of LDAP controls (see "LDAPControl" on page 193) that you want applied to the delete operation.
`log_change`	Specifies whether or not you want to log the changes to the replication log. • If 1, write the changes to the replication log. • If 0, do not write the changes to the replication log.

Returns

A new parameter block (see "Slapi_PBlock" on page 215) with the following parameter set:

- `SLAPI_PLUGIN_INTOP_RESULT` specifies the LDAP result code for the internal LDAP operation (for example, `LDAP_SUCCESS` if the operation is successful or `LDAP_PARAM_ERROR` if an invalid parameter is used).

  If the DN of the entry has a suffix that is not served by the Directory Server, `SLAPI_PLUGIN_INTOP_RESULT` is set to `LDAP_REFERRAL`.

Description

This function allows you to delete an entry from the directory from a plug-in function.

Unlike the standard LDAP delete operation, no LDAP result code to a client. Instead, the result code is placed in a parameter block that is returned by the function.

Example

[To be added]

See Also

slapi_destroy_condvar()

Frees a `Slapi_CondVar` structure from memory.

Syntax

```
#include "slapi-plugin.h"
void slapi_destroy_condvar( Slapi_CondVar *cvar );
```

Parameters

The function has the following parameters:

`cvar`	Pointer to the `Slapi_CondVar` structure that you want to free from memory.

Description

This function frees a `Slapi_CondVar` structure from memory. Before calling this function, you should make sure that this condition variable is no longer in use.

Example

[To be added]

See Also

slapi_destroy_mutex()

Frees a `Slapi_Mutex` structure from memory.

Syntax

```
#include "slapi-plugin.h"
void slapi_destroy_mutex( Slapi_Mutex *mutex );
```

Parameters

The function has the following parameters:

`mutex`	Pointer to the `Slapi_Mutex` structure that you want to free from memory.

Description

This function frees a `Slapi_Mutex` structure from memory. The calling function must ensure that no thread is currently in a lock-specific function. Locks do not provide self-referential protection against deletion.

Example

[To be added]

See Also

slapi_dn_beparent()

Gets a copy of the distinguished name (DN) of the parent of an entry, unless the specified entry's DN is the suffix of the local database.

If you don't want to check if the entry's DN is the suffix of the local database, call the `slapi_dn_beparent()` function instead.

[??? Do we also want to document the ability to use this with DNS-style addresses? e.g., binky@groening.hell.com or bongo.hell.com? ???]

Syntax

```
#include "slapi-plugin.h"
char *slapi_dn_beparent( Slapi_PBlock *pb, char *dn );
```

Parameters

The function has the following parameters:

`pb`	Parameter block
`dn`	DN of the entry that you want to find the parent for

Returns

DN of the parent entry, or NULL if the specified DN is NULL, if the DN is an empty string, if the DN has no parent (for example, `o=Airius.com`), or if the specified DN is the suffix of the local database.

Example

[To be added]

See Also

slapi_dn_ignore_case()

Convert all characters in a distinguished name (DN) to lowercase.

Syntax

```
#include “slapi-plugin.h”
char *slapi_dn_ignore_case( char *dn );
```

Parameters The function has the following parameters:

dn	DN that you want to convert to lowercase.

Returns The DN with lowercase characters. Note that variable passed in as the dn argument is also converted in-place.

Example [To be added]

See Also

slapi_dn_isbesuffix()

Determines whether or not the specified distinguished name (DN) is the suffix of the local database. Before calling this function, you should call `slapi_dn_normalize_case()` to normalize the DN and convert all characters to lowercase.

Syntax

```
#include "slapi-plugin.h"
int slapi_dn_isbesuffix( Slapi_PBlock *pb, char *dn );
```

Parameters

The function has the following parameters:

`pb`	Parameter block
`dn`	DN that you want to check.

Returns

1 if the specified DN is the suffix for the local database, or 0 if the DN is not the suffix.

Example

[To be added]

See Also

slapi_dn_isparent()

Determines whether or not a particular DN is the parent of another specified DN. Before calling this function, you should call `slapi_dn_normalize_case()` to normalize the DNs and convert all characters to lowercase.

Syntax

```
#include "slapi-plugin.h"
int slapi_dn_isparent( const char *parentdn, char *childdn );
```

Note that in the Netscape Directory Server 3.x releases, the `parentdn` argument was not declared with `const`:

```
int slapi_dn_isparent( char *parentdn, char *childdn );
```

Parameters

The function has the following parameters:

`parentdn`	Determine if this DN is the parent of `childdn`.
`childdn`	Determine if this DN is the child of `parentdn`.

Returns

A non-zero value if `parentdn` is the parent of `childdn`, or 0 if the `parentdn` is not the parent of `childdn`.

Example

[To be added]

See Also

slapi_dn_isroot()

Determines whether or not the specified DN is the root DN for this local database. Before calling this function, you should call `slapi_dn_normalize_case()` to normalize the DN and convert all characters to lowercase.

Syntax

```
#include "slapi-plugin.h"
int slapi_dn_isroot( Slapi_PBlock *pb, char *dn );
```

Parameters

The function has the following parameters:

`pb`	Parameter block
`dn`	DN that you want to check.

Returns

1 if the specified DN is the root DN of the local database, or 0 if the DN is not the root DN.

Example

[To be added]

See Also

slapi_dn_issuffix()

Determines whether or not a DN is equal to the specified suffix. Before calling this function, you should call `slapi_dn_normalize_case()` to normalize the DN and convert all characters to lowercase.

If you want to determine if a DN is the same as the suffix for the local database, call the `slapi_dn_isbesuffix()` function instead.

Syntax

```
#include "slapi-plugin.h"
int slapi_dn_issuffix( const char *dn, const char *suffix );
```

Note that in the Netscape Directory Server 3.x releases, the arguments were not declared with `const`:

```
int slapi_dn_issuffix( char *dn, char *suffix );
```

Parameters

The function has the following parameters:

`dn`	DN that you want to check.
`suffix`	Suffix that you want compared against the DN.

Returns

1 if the specified DN is the same as the specified suffix, or 0 if the DN is not the same as the suffix.

Example

[To be added]

See Also

slapi_dn_normalize()

Convert a distinguished name (DN) to canonical format (no leading or trailing spaces, no spaces between components, and no spaces around the equals sign). For example, given the following DN:

```
cn = Moxie Cross , ou = Engineering , o = Airius
```

the function returns:

```
cn=Moxie Cross,ou=Engineering,o=Airius
```

Syntax

```
#include "slapi-plugin.h"
char *slapi_dn_normalize( char *dn );
```

Parameters

The function has the following parameters:

`dn`	DN that you want to normalize.

Returns

The normalized DN. Note that variable passed in as the `dn` argument is also converted in-place.

Example

[To be added]

See Also

slapi_dn_normalize_case()

Converts a distinguished name (DN) to canonical format and converts all characters to lowercase. Calling this function has the same effect as calling the `slapi_dn_normalize()` function followed by the `slapi_dn_ignore_case()` function.

Syntax

```
#include “slapi-plugin.h”
char *slapi_dn_normalize_case( char *dn );
```

Parameters

The function has the following parameters:

`dn`	DN that you want to normalize and convert to lowercase.

Returns

The normalized DN with all lowercase characters. Note that variable passed in as the `dn` argument is also converted in-place.

Example

[To be added]

See Also

slapi_dn_parent()

Gets a copy of the distinguished name (DN) of the parent of an entry. Before calling this function, you should call `slapi_dn_normalize_case()` to normalize the DN and convert all characters to lowercase.

If you want to check if the DN is the suffix of the local database, call the `slapi_dn_beparent()` function instead.

[??? Do we also want to document the ability to use this with DNS-style addresses? e.g., binky@groening.hell.com or bongo.hell.com? ???]

Syntax

```
#include "slapi-plugin.h"
char *slapi_dn_parent( char *dn );
```

Parameters

The function has the following parameters:

`dn`	DN of the entry that you want to find the parent for

Returns

DN of the parent entry, or NULL if the specified DN is NULL, if the DN is an empty string, or if the DN has no parent (for example, `o=Airius.com`).

Example

[To be added]

See Also

slapi_entry2str()

Generates a description of an entry as a string.

Syntax

```
#include “slapi-plugin.h”
char *slapi_entry2str( Slapi_Entry *e, int *len );
```

Parameters

The function has the following parameters:

`e`	Entry that you want to generate a description for.
`len`	Length of the string returned by this function.

Returns

The description of the entry as a string.

Description

This function returns the LDIF string representation of the entry or NULL if an error occurs. The string has the following format:

```
dn: <dn>\n
[<attr>: <value>\n]*
```

For example:

```
dn: uid=jrent2, ou=People, o=airius.com
cn: Judy Rentz
sn: Rentz
...
```

When you no longer need to use the string, you should free it from memory by calling the `slapi_ch_free()` function.

To convert a string description in this format to an entry of the `Slapi_Entry` data type, call the `slapi_str2entry()` function.

Example

[To be added]

See Also

`slapi_str2entry()`.

slapi_entry_add_rdn_values()

Adds the components in an entry's relative distinguished name (RDN) to the entry as attribute values. (For example, if the entry's RDN is `uid=bjensen`, the function adds `uid=bjensen` to the entry as an attribute value.)

Syntax

```
#include "slapi-plugin.h"
int slapi_entry_add_rdn_values( Slapi_Entry *e );
```

Parameters

The function has the following parameters:

e	Entry that you want to add the attributes to

Returns

One of the following values:

- `LDAP_SUCCESS`, if the values were successfully added to the entry. The function also returns `LDAP_SUCCESS` if the entry is NULL, if the entry's DN is NULL, or if the entry's RDN is NULL.
- `LDAP_INVALID_DN_SYNTAX`, if the entry's DN cannot be parsed.

Example

[To be added]

See Also

slapi_entry_add_values()

Adds the specified values to an entry. If any of the values already exist, the function returns with an error. If you do not want errors returned for duplicate values, call the `slapi_entry_attr_merge()` function instead.

Syntax

```
#include "slapi-plugin.h"
int slapi_entry_add_values( Slapi_Entry *e, char *type, struct
berval **vals );
```

Parameters

The function has the following parameters:

`e`	Entry that you want to add the values to
`type`	Attribute type that you want to add the values to
`vals`	Array of the `berval` structures containing the values that you want to add

Returns

One of the following values:

- `LDAP_SUCCESS`, if the values were successfully added to the entry.
- `LDAP_TYPE_OR_VALUE_EXISTS`, if any of the specified values already exists in the entry.
- `LDAP_CONSTRAINT_VIOLATION`, if the values could not be added to the entry.

Example

[To be added]

See Also

slapi_entry_alloc()

Allocates memory for a new entry of the data type `Slapi_Entry`. When you are done with the entry, you can free the entry by calling the `slapi_entry_free()` function.

Syntax

```
#include "slapi-plugin.h"
Slapi_Entry *slapi_entry_alloc();
```

Returns

A pointer to the newly allocated entry of the data type `Slapi_Entry`. If space cannot be allocated (for example, if no more virtual memory exists), the `slapd` program terminates.

Description

This function returns an empty `Slapi_Entry` structure. You can call other front-end functions to set the DN and attributes of this entry.

When you are no longer using the entry, you should free it from memory by calling the `slapi_entry_free()` function.

Example

[To be added]

See Also

`slapi_entry_dup()`, `slapi_entry_free()`.

slapi_entry_attr_delete()

Deletes an attribute from an entry.

Syntax

```
#include “slapi-plugin.h”
int slapi_entry_attr_delete( Slapi_Entry *e, char *type );
```

Parameters The function has the following parameters:

`e`	Entry that you want to delete the attribute from.
`type`	Attribute type that you want to delete.

Returns One of the following values:

- 0 if successful.
- 1 if the specified attribute is not part of the entry.
- -1 if an error occurred.

Example [To be added]

See Also

slapi_entry_attr_find()

Determines if an entry has the specified attribute. If the entry contains the attribute, the function returns that attribute.

Syntax

```
#include “slapi-plugin.h”
int slapi_entry_attr_find( Slapi_Entry *e, char *type,
Slapi_Attr **attr );
```

Parameters

The function has the following parameters:

`e`	Entry that you want to check
`type`	Name of the attribute that you want to check
`attr`	Pointer to the attribute, if the attribute is in the entry.

Returns

0 if the entry contains the specified attribute, or -1 if the entry does not contain the attribute.

Example

[To be added]

See Also

slapi_entry_attr_get_charptr()

Gets the first value of an attribute in an entry as a string.

Syntax

```
#include “slapi-plugin.h”
char *slapi_entry_attr_get_charptr(const Slapi_Entry* e, char
*type);
```

Parameters

The function has the following parameters:

`e`	Entry that you want to get the value from.
`type`	Attribute type that you want to get the value from.

Returns

A copy of the first value in the attribute, or NULL if the entry does not contain the attribute. When you are done working with this value, you should free it from memory by calling the `slapi_ch_free()` function.

Example

[To be added]

See Also

slapi_entry_attr_get_int()

Gets the first value of an attribute in an entry as an integer.

Syntax

```
#include "slapi-plugin.h"
int slapi_entry_attr_get_int(const Slapi_Entry* e, char *type);
```

Parameters

The function has the following parameters:

`e`	Entry that you want to get the value from.
`type`	Attribute type that you want to get the value from.

Returns

The first value in the attribute, converted to an integer, or 0 if the entry does not contain the attribute.

Example

[To be added]

See Also

slapi_entry_attr_hasvalue()

Determines if an attribute in an entry contains a specified value.

Syntax

```
#include “slapi-plugin.h”
int slapi_entry_attr_hasvalue(Slapi_Entry *e, char *type, const
char *value);
```

Parameters The function has the following parameters:

`e`	Entry that you want to check.
`type`	Attribute type that you want to check the values of.
`value`	Value that you want to find in the attribute.

Returns One of the following values:

- 1 if the attribute contains the specified value.
- 0 if the attribute does not contain the specified value.

Example [To be added]

See Also

slapi_entry_attr_merge()

Adds the specified values to an attribute in an entry. Unlike the `slapi_entry_add_values()` function, this function does not return with an error if any of the values already exist in the entry.

Syntax

```
#include “slapi-plugin.h”
int slapi_entry_attr_merge( Slapi_Entry *e, char *type, struct
berval **vals );
```

Parameters

The function has the following parameters:

`e`	Entry that you want to add the values to
`type`	Attribute type that you want to add the values to
`vals`	Array of the `berval` structures containing the values that you want to add

Returns

One of the following values:

- 0 if successful.
- -1 if an error occurred.

Example

[To be added]

See Also

slapi_entry_attr_replace()

Replaces the values of an attribute in an entry with the specified values.

Syntax

```
#include “slapi-plugin.h”
int slapi_entry_attr_replace( Slapi_Entry *e, char *type, struct
berval **vals );
```

Parameters

The function has the following parameters:

`e`	Entry that you want to replace values in.
`type`	Attribute type that you want to replace the values of.
`vals`	Array of the `berval` structures containing the values that should replace the existing values of the attribute.

Returns

One of the following values:

- 0 if successful.
- -1 if an error occurred.

Example

[To be added]

See Also

slapi_entry_attr_set_charptr()

Replaces the values of an attribute in an entry with a specified string.

Syntax

```
#include “slapi-plugin.h”
void slapi_entry_attr_set_charptr(Slapi_Entry* e, char *type,
char *value);
```

Parameters

The function has the following parameters:

`e`	Entry in which you want to set the value.
`type`	Attribute type in which you want to set the value.
`value`	Value that you want to assign to the attribute.

Example

[To be added]

See Also

slapi_entry_attr_set_long()

Replaces the values of an attribute in an entry with a specified integer.

Syntax

```
#include “slapi-plugin.h”
void slapi_entry_attr_set_long(Slapi_Entry* e, char *type,
unsigned long l);
```

Parameters The function has the following parameters:

`e`	Entry in which you want to set the value.
`type`	Attribute type in which you want to set the value.
`l`	Integer value that you want assigned to the attribute.

Example [To be added]

See Also

slapi_entry_delete_values()

Deletes the specified values from an entry.

Syntax

```
#include “slapi-plugin.h”
int slapi_entry_delete_values( Slapi_Entry *e, char *type,
struct berval **vals );
```

Parameters

The function has the following parameters:

`e`	Entry that you want to remove values from
`type`	Attribute type that you want to remove the values from
`vals`	Array of the `berval` structures containing the values that you want to remove. If NULL, removes the entire attribute specified by `type` from the entry.

Returns

One of the following values:

- `LDAP_SUCCESS`, if the values were successfully removed the entry.
- `LDAP_NO_SUCH_ATTRIBUTE`, if the specified attribute type or values cannot be found in the entry.

Example

[To be added]

See Also

slapi_entry_dup()

Makes a copy of an entry, its DN, and its attributes.

Syntax

```
#include "slapi-plugin.h"
Slapi_Entry *slapi_entry_dup( const Slapi_Entry *e );
```

Parameters

The function has the following parameters:

e	Entry that you want to copy.

Returns

The new copy of the entry. If the structure cannot be duplicated (for example, if no more virtual memory exists), the `slapd` program terminates.

Description

This function returns a copy of an existing `Slapi_Entry` structure. You can call other front-end functions to change the DN and attributes of this entry.

When you are no longer using the entry, you should free it from memory by calling the `slapi_entry_free()` function.

Note that in the Netscape Directory Server 3.x releases, the `e` argument was not declared with `const`:

```
Slapi_Entry *slapi_entry_dup( Slapi_Entry *e );
```

Example

[To be added]

See Also

`slapi_entry_alloc()`, `slapi_entry_free()`.

slapi_entry_first_attr()

Finds the first attribute in an entry. If you want to iterate through the attributes in an entry, use this function in conjunction with the `slapi_entry_next_attr()` function.

Syntax

```
#include “slapi-plugin.h”
int slapi_entry_first_attr( Slapi_Entry *e, Slapi_Attr **attr );
```

Parameters

The function has the following parameters:

`e`	Entry that you want to get the attribute from
`attr`	Pointer to the first attribute in the entry

Returns

0 if successful.

Example

[To be added]

See Also

slapi_entry_free()

Frees an entry, its DN, and its attributes from memory.

Syntax

```
#include "slapi-plugin.h"
void slapi_entry_free( Slapi_Entry *e );
```

Parameters

The function has the following parameters:

e	Entry that you want to free. If NULL, no action occurs.

Description

Call this function to free an entry that you have allocated by using the `slapi_entry_alloc()` function or the `slapi_entry_dup()` function.

Example

[To be added]

See Also

`slapi_entry_alloc()`, `slapi_entry_dup()`.

slapi_entry_get_dn()

Gets the distinguished name (DN) of the specified entry.

Syntax

```
#include "slapi-plugin.h"
char *slapi_entry_get_dn( Slapi_Entry *e );
```

Parameters

The function has the following parameters:

e	Entry that you want to get the DN of

Returns

The DN of the entry. Note that this returns a pointer to the actual DN in the entry, not a copy of the DN. You should not free the DN unless you plan to replace it by calling `slapi_entry_set_dn()`.

Example

[To be added]

See Also

slapi_entry_next_attr()

Finds the next attribute after `prevattr` in an entry. If you want to iterate through the attributes in an entry, use this function in conjunction with the `slapi_entry_first_attr()` function.

Syntax

```
#include "slapi-plugin.h"
int slapi_entry_next_attr( Slapi_Entry *e, Slapi_Attr *prevattr,
Slapi_Attr **attr );
```

Parameters

The function has the following parameters:

`e`	Entry that you want to get the attribute from
`prevattr`	Previous attribute in the entry.
`attr`	Pointer to the next attribute after `prevattr` in the entry

Returns

0 if successful, or -1 if `prevattr` was the last attribute in the entry.

Example

[To be added]

See Also

slapi_entry_rdn_values_present()

Determines whether or not the values in an entry's relative distinguished name (RDN) are also present as attribute values. (For example, if the entry's RDN is `cn=Barbara Jensen`, the function determines if the entry has the `cn` attribute with the value `Barbara Jensen`.)

Syntax

```
#include "slapi-plugin.h"
int slapi_entry_rdn_values_present( Slapi_Entry *e );
```

Parameters

The function has the following parameters:

e	Entry that you want to get the attribute from

Returns

1 if the values in the RDN are present in attributes of the entry, or 0 if the values are not present.

Example

[To be added]

See Also

slapi_entry_schema_check()

Determines whether or not the specified entry complies with the schema for its object class.

Syntax

```
#include “slapi-plugin.h”
int slapi_entry_schema_check( Slapi_PBlock *pb, Slapi_Entry *e
);
```

Parameters The function has the following parameters:

`pb`	Parameter block
`e`	Entry that you want to check.

Returns One of the following values:

- 0 if the entry complies with the schema or if schema checking is turned off. The function also returns 0 if the entry has additional attributes not allowed by the schema and has the object class `extensibleObject`.
- 1 if the entry is missing the `objectclass` attribute, is missing any required attributes, has any attributes not allowed by the schema (but does not have the object class `extensibleObject`), or if the entry has multiple values for a single-valued attribute.

Example [To be added]

See Also

slapi_entry_set_dn()

Sets the distinguished name (DN) of an entry.

Syntax

```
#include "slapi-plugin.h"
void slapi_entry_set_dn( Slapi_Entry *e, char *dn );
```

Parameters

The function has the following parameters:

e	Entry that you want to assign the DN to.
dn	Distinguished name you want assigned to the entry.

Description

This function sets the pointer to the DN to the DN that you specify. Note that the old DN is not overwritten and is still in memory. Because of this, you should first call `slapi_entry_get_dn()` to get the pointer to the current DN, free the DN, then call `slapi_entry_set_dn()` to set the pointer to your new DN.

Example

[To be added]

See Also

slapi_filter_free()

Frees the specified filter and (optionally) the set of filters that comprise it (for example, the set of filters in an `LDAP_FILTER_AND` type filter).

Syntax

```
#include "slapi-plugin.h"
void slapi_filter_free( Slapi_Filter *f, int recurse );
```

Parameters

The function has the following parameters:

`f`	Filter that you want to free.
`recurse`	If 1, recursively frees all filters that comprise this filter. If 0, only frees the filter specified by `f`.

Example

[To be added]

See Also

slapi_filter_get_ava()

(Applies only to filters of the types LDAP_FILTER_EQUALITY, LDAP_FILTER_GE, LDAP_FILTER_LE, LDAP_FILTER_APPROX) Gets the attribute type and the value from the filter.

Syntax

```
#include "slapi-plugin.h"
int slapi_filter_get_ava( Slapi_Filter *f, char **type, struct
berval **bval );
```

Parameters

The function has the following parameters:

f	Filter that you want to get the attribute and value from.
type	Pointer to the attribute type of the filter.
bval	Pointer to the address of the berval structure containing the value of the filter.

Returns

0 if successful, or -1 if the filter is not one of the types listed above.

Description

Filters of the type LDAP_FILTER_EQUALITY, LDAP_FILTER_GE, LDAP_FILTER_LE, and LDAP_FILTER_APPROX generally compare a value against an attribute. For example:

```
(cn=Barbara Jensen)
```

This filter finds entries in which the value of the cn attribute is equal to Barbara Jensen.

Call the slapi_filter_get_ava() function to get the attribute type and value from this filter. In the case of the example above, calling the slapi_filter_get_ava() function gets the attribute type cn and the value Barbara Jensen.

Example

[To be added]

See Also

slapi_filter_get_choice()

Gets the type of the specified filter (for example, LDAP_FILTER_EQUALITY).

Syntax

```
#include "slapi-plugin.h"
int slapi_filter_get_choice( Slapi_Filter *f );
```

Parameters

The function has the following parameters:

f	Filter that you want to get type of.

Returns

One of the following values:

- LDAP_FILTER_AND (AND filter)
  For example: (&(ou=Accounting)(l=Sunnyvale))
- LDAP_FILTER_OR (OR filter)
  For example: (|(ou=Accounting)(l=Sunnyvale))
- LDAP_FILTER_NOT (NOT filter)
  For example: (!(l=Sunnyvale))
- LDAP_FILTER_EQUALITY (equals filter)
  For example: (ou=Accounting)
- LDAP_FILTER_SUBSTRINGS (substring filter)
  For example: (ou=Account*Department)
- LDAP_FILTER_GE ("greater than or equal to" filter)
  For example: (supportedLDAPVersion>=3)
- LDAP_FILTER_LE ("less than or equal to" filter)
  For example: (supportedLDAPVersion<=2)
- LDAP_FILTER_PRESENT (presence filter)
  For example: (mail=*)
- LDAP_FILTER_APPROX (approximation filter)
  For example: (ou~=Sales)
- LDAP_FILTER_EXTENDED (extensible filter)

For example: `(o:dn:=Airius)`

Example [To be added]

See Also

slapi_filter_get_subfilt()

(Applies only to filters of the type LDAP_FILTER_SUBSTRINGS) Gets the substring values from the filter.

Syntax

```
#include "slapi-plugin.h"
int slapi_filter_get_subfilt( Slapi_Filter *f, char **type, char
**initial, char ***any, char **final );
```

Parameters

The function has the following parameters:

f	Filter that you want to get the substring values from.
type	Pointer to the attribute type of the filter.
initial	Pointer to the initial substring ("starts with") of the filter.
any	Pointer to an array of the substrings ("contains") for the filter.
final	Pointer to the final substring ("ends with") of the filter.

Returns

0 if successful, or -1 if the filter is not one of the types listed above.

Description

Filters of the type LDAP_FILTER_SUBSTRINGS generally compare a set of substrings against an attribute. For example:

```
(cn=John*Q*Public)
```

This filter finds entries in which the value of the cn attribute starts with John, contains Q, and ends with Public.

Call the slapi_filter_get_subfilt() function to get these substring values as well as the attribute type from this filter. In the case of the example above, calling the slapi_filter_get_subfilt() function gets the initial substring John, the any substring Q, and the final substring Public in addition to the attribute type cn.

Example

[To be added]

See Also

slapi_filter_get_type()

(Applies only to filters of the type LDAP_FILTER_PRESENT) Gets the attribute type specified in the filter.

Syntax

```
#include "slapi-plugin.h"
int slapi_filter_get_type( Slapi_Filter *f, char **type );
```

Parameters

The function has the following parameters:

f	Filter that you want to get the substring values from.
type	Pointer to the attribute type of the filter.

Returns

0 if successful, or -1 if the filter is not one of the types listed above.

Description

Filters of the type LDAP_FILTER_PRESENT generally determine if a specified attribute is assigned a value. For example:

```
(mail=*)
```

This filter finds entries that have a value assigned to the mail attribute.

Call the slapi_filter_get_type() function to get the attribute type from this filter. In the case of the example above, calling the slapi_filter_get_type() function gets the attribute type mail.

Example

[To be added]

See Also

slapi_filter_join()

Joins the two specified filters using one of the following filter types: LDAP_FILTER_AND, LDAP_FILTER_OR, or LDAP_FILTER_NOT. (When specifying the filter type LDAP_FILTER_NOT, the second filter should be NULL.)

Syntax

```
#include "slapi-plugin.h"
Slapi_Filter *slapi_filter_join( int ftype, Slapi_Filter *f1,
Slapi_Filter *f2 );
```

Parameters

The function has the following parameters:

ftype	Type of composite filter you want to create.
f1	First filter that you want to join.
f2	Second filter that you want to join. If ftype is LDAP_FILTER_NOT, specify NULL for this argument.

Returns

The new filter constructed from the other two filters.

Description

Filters of the type LDAP_FILTER_AND, LDAP_FILTER_OR, and LDAP_FILTER_NOT generally consist of one or more other filters. For example:

```
(&(ou=Accounting)(l=Sunnyvale))
(|(ou=Accounting)(l=Sunnyvale))
(!(l=Sunnyvale))
```

Each of these examples contain one or more LDAP_FILTER_EQUALITY filters.

Call the slapi_filter_join() function to create a new filter of the type LDAP_FILTER_AND, LDAP_FILTER_OR, or LDAP_FILTER_NOT.

Example

[To be added]

See Also

slapi_filter_list_first()

(Applies only to filters of the types LDAP_FILTER_EQUALITY, LDAP_FILTER_GE, LDAP_FILTER_LE, LDAP_FILTER_APPROX) Gets the first filter that makes up the specified filter.

To iterate through all filters that make up a specified filter, use this function in conjunction with the slapi_filter_list_next() function.

Syntax

```
#include "slapi-plugin.h"
Slapi_Filter *slapi_filter_list_first( Slapi_Filter *f );
```

Parameters

The function has the following parameters:

f	Filter that you want to get the first component of.

Returns

The first filter that makes up the specified filter f.

Description

Filters of the type LDAP_FILTER_AND, LDAP_FILTER_OR, and LDAP_FILTER_NOT generally consist of one or more other filters. For example, if the filter is:

```
(&(ou=Accounting)(l=Sunnyvale))
```

the first filter in this list is:

```
(ou=Accounting)
```

Call the slapi_filter_list_first() function to get the first filter in the list.

Example

[To be added]

See Also

slapi_filter_list_next()

(Applies only to filters of the types LDAP_FILTER_EQUALITY, LDAP_FILTER_GE, LDAP_FILTER_LE, LDAP_FILTER_APPROX) Gets the next filter (following fprev) that makes up the specified filter f.

To iterate through all filters that make up a specified filter, use this function in conjunction with the slapi_filter_list_first() function.

Syntax

```
#include "slapi-plugin.h"
Slapi_Filter *slapi_filter_list_next( Slapi_Filter *f,
Slapi_Filter *fprev );
```

Parameters

The function has the following parameters:

f	Filter from which you want to get the next component (after fprev).
fprev	Filter within the specified filter f.

Returns

The next filter (after fprev) that makes up the specified filter f.

Description

Filters of the type LDAP_FILTER_AND, LDAP_FILTER_OR, and LDAP_FILTER_NOT generally consist of one or more other filters. For example, if the filter is:

```
(&(ou=Accounting)(l=Sunnyvale))
```

the next filter after (ou=Accounting) in this list is:

```
(l=Sunnyvale)
```

Call the slapi_filter_list_next() function to get the filters from this list.

Example

[To be added]

See Also

slapi_filter_test()

Determines if the specified entry matches a particular filter.

Syntax

```
#include “slapi-plugin.h”
int slapi_filter_test( Slapi_PBlock *pb, Slapi_Entry *e,
Slapi_Filter *f, int verify_access );
```

Parameters The function has the following parameters:

`pb`	Parameter block.
`e`	Entry that you want to test.
`f`	Filter that you want to test the entry against.
`verify_access`	If 1, verifies that the current user has access rights to search the specified entry. If 0, bypasses any access control.

Returns One of the following values:

- 0 if the entry matched the filter or if the specified filter is NULL.
- -1 if the filter type is unknown.
- A positive value (an LDAP error code) if an error occurred.

Example [To be added]

See Also

slapi_free_search_results_internal()

Frees search results returned by the `slapi_search_internal()` and `slapi_search_internal_callback()` functions.

Syntax

```
#include "slapi-plugin.h"
void slapi_free_search_results_internal(Slapi_PBlock *pb);
```

Parameters

The function has the following parameters:

`pb`	Parameter block returned by the `slapi_search_internal()` and `slapi_search_internal_callback()` functions.

Example [To be added]

See Also

slapi_get_supported_controls()

Gets an array of object identifications (OIDs) representing the controls supported by the directory server. You can register new controls by calling the `slapi_register_supported_control()` function.

Syntax

```
#include "slapi-plugin.h"
int slapi_get_supported_controls( char ***ctrloidsp,
   unsigned long **ctrlopsp );
```

Parameters

The function has the following parameters:

`ctrloidsp`	Pointer to an array of the OIDs of the controls supported by the server.
`ctrlopsp`	Pointer to an array of IDs specifying the LDAP operations that support each control. For a list of the possible IDs and their values, see `slapi_register_supported_control()`.

Returns

0 if successful.

Description

When you call the `slapi_register_supported_control()` function to register a control, you specify the OID of the control and the IDs of the operations that support the control.

The server records this information in two arrays: an array of control OIDs and an array of operations that support the control. You can get these arrays by calling the `slapi_get_supported_controls()` function.

For each OID specified in the `ctrloidsp` array, the corresponding array element (with the same index) in the `ctrlopsp` array identifies the operations that support the control.

For a list of the possible IDs for the operations, see `slapi_register_supported_control()`.

Example

[To be added]

See Also

slapi_get_supported_extended_ops()

Gets an array of the object IDs (OIDs) of the extended operations supported by the server. You can register new extended operations by putting the OID in the `SLAPI_PLUGIN_EXT_OP_OIDLIST` parameter and calling the `slapi_pblock_set()` function.

Syntax

```
#include “slapi-plugin.h”
char **slapi_get_supported_extended_ops( void );
```

Returns

Pointer to an array of the OIDs of the extended operations supported by the server.

Example

[To be added]

See Also

slapi_get_supported_saslmechanisms()

Gets an array of the names of the supported Simple Authentication and Security Layer (SASL) mechanisms. You can register new SASL mechanisms by calling the `slapi_register_supported_saslmechanism()` function.

Syntax

```
#include “slapi-plugin.h”
char ** slapi_get_supported_saslmechanisms( void );
```

Returns Pointer to an array of the names of SASL mechanisms supported by the server.

Example [To be added]

See Also

slapi_ldap_init()

Initializes an LDAP session with another LDAP server.

Syntax

```
#include “slapi-plugin.h”
LDAP *slapi_ldap_init( char *ldaphost, int ldapport, int secure,
int shared );
```

Parameters

The function has the following parameters:

`ldaphost`	Space-delimited list of one or more host names (or IP address in dotted notation, such as "141.211.83.36") of the LDAP servers that you want to connect to. The names can be in *`hostname`*:*`portnumber`* format (in which case, *`portnumber`* overrides the port number specified by the `ldapport` argument).
`ldapport`	Port number of the LDAP server.
`secure`	Determines whether or not to establish the connection over SSL. Set this to a non-zero value to establish the connection over SSL.
`shared`	Determines whether or not the LDAP session (the `LDAP` structure) can be shared by different threads. Set this to a non-zero value to allow multiple threads to share the session.

Returns

One of the following values:

- If successful, returns a pointer to an `LDAP` structure, which should be passed to subsequent calls to other LDAP API functions.
- If unsuccessful, returns NULL.

Description

This function initializes an LDAP session with another LDAP server. If you want to connect to another LDAP server over SSL or if you want to allow multiple threads to use the same connection, call this function instead of the `ldap_init()` function provided with the Netscape Directory SDK.

This function allocates an `LDAP` structure containing information about the session, including the host name and port of the LDAP server, preferences for the session (such as the maximum number of entries to return in a search), and the error code of the last LDAP operation performed.

You can specify a list of LDAP servers that you want to attempt to connect to. Your client will attempt to connect to the first LDAP server in the list. If the attempt fails, your client will attempt to connect to the next LDAP server in the list.

If you specify a non-zero value for the `secure` argument, this function initializes the plug-in for SSL and installs the I/O routines for SSL.

If you specify a non-zero value for the `shared` argument, this function installs the server's threading functions and allows multiple threads to share this session (the returned `LDAP` structure). Note that the Netscape Directory Server processes each request in a separate thread. When handling multiple requests, it is possible for the server to call your plug-in function concurrently for different threads.

If you initialize a session by calling this function, make sure to call the `slapi_ldap_unbind()` function (not the `ldap_unbind()` or `ldap_unbind_s()` functions provided with the Netscape Directory SDK) when you are done with the session.

Example [To be added]

See Also

slapi_ldap_unbind()

Unbinds from another LDAP server and frees the resources contained in the LDAP structure.

Syntax

```
#include "slapi-plugin.h"
void slapi_ldap_unbind( LDAP *ld );
```

Parameters

The function has the following parameters:

`ld`	Connection handle, which is a pointer to an `LDAP` structure containing information about the connection to the LDAP server.

Description

This function unbinds from another LDAP server. Call this function if you initialized the LDAP session with the `slapi_ldap_init()` function. (Do not call the `ldap_unbind()` or `ldap_unbind_s()` functions provided with the Netscape Directory SDK.)

Example

[To be added]

See Also

slapi_lock_mutex()

Locks the specified mutex.

Syntax

```
#include "slapi-plugin.h"
void slapi_lock_mutex( Slapi_Mutex *mutex );
```

Parameters

The function has the following parameters:

`mutex`	Pointer to an `Slapi_Mutex` structure representing the mutex that you want to lock.

Description

This function locks the mutex specified by the `Slapi_Mutex` structure. After this function returns, any other thread that attempts to acquire the same lock blocks until the holder of the lock releases the lock. Acquiring the lock is not an interruptible operation, nor is there any timeout mechanism.

Example

[To be added]

See Also

slapi_log_error()

Writes a message to the error log for the directory server. (By default, the error log file is <server_root>/<server_id>/logs/errors.)

Syntax

```
#include "slapi-plugin.h"
int slapi_log_error( int severity, char *subsystem, char *fmt,
... );
```

Parameters The function has the following parameters:

severity	Level of severity of the message. In combination with the severity level specified by the administrator, this determines whether or not the message is written to the log.
subsystem	Name of the subsystem in which this function is called. The string that you specify here appears in the error log in the following format: <subsystem_name>: <message>
fmt, ...	Message that you want written. This message can be in printf()-style format. For example: ..., "%s\n", myString);

The severity argument corresponds to the Log Level setting selected by in the Server Manager under Server Preferences | LDAP. If a Log Level setting is selected, messages with that severity level are written to the error log. The severity argument can have one of the following values:

SLAPI_LOG_FATAL	Always written to the error log. This severity level indicates that a fatal error has occured in the server.
SLAPI_LOG_TRACE	Written to the error log if the Log Level setting "Trace function calls" is selected. This severity level is typically used to indicate what function is being called.
SLAPI_LOG_PACKETS	Written to the error log if the Log Level setting "Packet handling" is selected.
SLAPI_LOG_ARGS	Written to the error log if the Log Level setting "Heavy trace output" is selected.
SLAPI_LOG_CONNS	Written to the error log if the Log Level setting "Connection management" is selected.
SLAPI_LOG_BER	Written to the error log if the Log Level setting "Packets sent/received" is selected.

SLAPI_LOG_FILTER	Written to the error log if the Log Level setting “Search filter processing” is selected.
SLAPI_LOG_CONFIG	Written to the error log if the Log Level setting “Config file processing” is selected.
SLAPI_LOG_ACL	Written to the error log if the Log Level setting “Access control list processing” is selected.
SLAPI_LOG_SHELL	Written to the error log if the Log Level setting “Log communications with shell back-ends” is selected.
SLAPI_LOG_PARSE	Written to the error log if the Log Level setting “Log entry parsing” is selected.
SLAPI_LOG_HOUSE	Written to the error log if the Log Level setting “Housekeeping” is selected.
SLAPI_LOG_REPL	Written to the error log if the Log Level setting “Replication” is selected.
SLAPI_LOG_CACHE	Written to the error log if the Log Level setting “Entry cache” is selected.
SLAPI_LOG_PLUGIN	Written to the error log if the Log Level setting “Plug-ins” is selected. This severity level is typically used to identify messages from server plug-ins.

Returns One of the following values:

- 0 if successful.
- -1 if an unknown severity level is specified.

Example [To be added]

See Also

slapi_matchingrule_free()

Frees the specified matching rule structure (and optionally, its members) from memory.

Syntax

```
#include "slapi-plugin.h"
void slapi_matchingrule_free(Slapi_MatchingRuleEntry **mrEntry,
int freeMembers);
```

Parameters

The function has the following parameters:

`mrEntry`	The `Slapi_MatchingRuleEntry` structure that you want to free from memory.
`freeMembers`	If 1, the function also frees the members of the structure from memory.

Description

This function frees an `Slapi_MatchingRuleEntry` structure (and, optionally, its members) from memory. Call this function when you are done working with the structure.

Example

[To be added]

See Also

slapi_matchingrule_get()

Gets information about a matching rule.

Syntax

```
#include “slapi-plugin.h”
int slapi_matchingrule_get(Slapi_MatchingRuleEntry *mr, int arg,
void *value);
```

Parameters

The function has the following parameters:

`mr`	`Slapi_MatchingRuleEntry` structure that you want to get data from.
`arg`	ID specifying the type of information you want to get.
`value`	Pointer to a variable to hold the retrieved data.

The `arg` argument can have one of the following values:

ID	Data Type of the value Argument	Description
`SLAPI_MATCHINGRULE_NAME`	`char *`	Name of the matching rule.
`SLAPI_MATCHINGRULE_OID`	`char *`	OID of the matching rule.
`SLAPI_MATCHINGRULE_DESC`	`char *`	Description of the matching rule.
`SLAPI_MATCHINGRULE_SYNTAX`	`char *`	Syntax supported by the matching rule.
`SLAPI_MATCHINGRULE_OBSOLETE`	`int`	If 1, the matching rule is obsolete.

Returns

One of the following values:

- 0 if the information was successfully retrieved.
- -1 if an error occurred (for example, if an invalid argument was specified).

Description

This function gets information about a matching rule from the `Slapi_MatchingRuleEntry` structure. To set information in this structure, call the `slapi_matchingrule_set()` function.

Example

[To be added]

See Also

slapi_matchingrule_new()

Allocates memory for a new `Slapi_MatchingRuleEntry` structure.

Syntax

```
#include "slapi-plugin.h"
Slapi_MatchingRuleEntry *slapi_matchingrule_new();
```

Returns

A new `Slapi_MatchingRuleEntry` structure, or NULL if memory could not be allocated.

Description

This function allocates memory for a new `Slapi_MatchingRuleEntry` structure, which represents a matching rule. After you call this function, you can call the `slapi_matchingrule_set()` function to set the values in this structure and call the `slapi_matchingrule_register()` function to register the matching rule.

Example

[To be added]

See Also

slapi_matchingrule_register()

Registers the specified matching rule with the server.

Syntax

```
#include “slapi-plugin.h”
int slapi_matchingrule_register(Slapi_MatchingRuleEntry
*mrEntry);
```

Parameters

The function has the following parameters:

`mrEntry`	`Slapi_MatchingRuleEntry` structure representing the matching rule that you want to register.

Returns

One of the following values:

- 0 if the matching rule was successfully registered.
- -1 if an error occurred.

Description

This function registers the specified matching rule with the server. To create the matching rule and set its values, call the `slapi_matchingrule_new()` function and the `slapi_matchingrule_set()` function.

Example

[To be added]

See Also

slapi_matchingrule_set()

Sets information about the matching rule.

Syntax

```
#include "slapi-plugin.h"
int slapi_matchingrule_set(Slapi_MatchingRuleEntry *mr, int arg,
void *value);
```

Parameters

The function has the following parameters:

`mr`	`Slapi_MatchingRuleEntry` structure that you want to set data in.
`arg`	ID specifying the type of information you want to set.
`value`	The value that you want to set.

The `arg` argument can have one of the following values:

ID	Data Type of the value Argument	Description
`SLAPI_MATCHINGRULE_NAME`	`char *`	Name of the matching rule.
`SLAPI_MATCHINGRULE_OID`	`char *`	OID of the matching rule.
`SLAPI_MATCHINGRULE_DESC`	`char *`	Description of the matching rule.
`SLAPI_MATCHINGRULE_SYNTAX`	`char *`	Syntax supported by the matching rule.
`SLAPI_MATCHINGRULE_OBSOLETE`	`int`	If 1, the matching rule is obsolete.

Returns

One of the following values:

- 0 if the information was successfully set.
- -1 if an error occurred (for example, if an invalid argument was specified).

Description

This function sets information in an `Slapi_MatchingRuleEntry` structure. To get information from this structure, call the `slapi_matchingrule_get()` function.

Example

[To be added]

See Also

slapi_matchingrule_unregister()

Placeholder for future function. (Currently, this function does nothing.)

Syntax

```
#include "slapi-plugin.h"
int slapi_matchingrule_unregister(char *oid);
```

Parameters

The function has the following parameters:

`oid`	OID of the matching rule that you want to unregister.

Returns

One of the following values:

- 0 if the matching rule was successfully unregistered.
- -1 if an error occurred (for example, if an invalid argument was specified).

Description

At this time, this function is a placeholder and does not do anything.

Example

[To be added]

See Also

slapi_modify_internal()

Performs an LDAP modify operation to modify an entry in the directory from your plug-in.

Syntax

```
#include "slapi-plugin.h"
Slapi_PBlock *slapi_modify_internal( char *dn, LDAPMod **mods,
   LDAPControl **controls, int log_change );
```

Parameters

The function has the following parameters:

`dn`	Distinguished name (DN) of the entry that you want to modify.
`mods`	Pointer to a NULL-terminated array of pointers to `LDAPMod` structures representing the attributes that you want to modify.
`controls`	A NULL-terminated array of LDAP controls (see "LDAPControl" on page 193) that you want applied to the modify operation.
`log_change`	Specifies whether or not you want to log the changes to the replication log. • If 1, write the changes to the replication log. • If 0, do not write the changes to the replication log.

Returns

A new parameter block (see "Slapi_PBlock" on page 215) with the following parameter set:

- `SLAPI_PLUGIN_INTOP_RESULT` specifies the LDAP result code for the internal LDAP operation (for example, `LDAP_SUCCESS` if the operation is successful or `LDAP_PARAM_ERROR` if an invalid parameter is used).

  If the DN of the entry has a suffix that is not served by the Directory Server, `SLAPI_PLUGIN_INTOP_RESULT` is set to `LDAP_REFERRAL`.

Description

This function allows you to modify an entry in the directory from a plug-in function. The arguments for this function are similar to the arguments for the standard `ldap_modify()` function in the Netscape LDAP C SDK.

Unlike the standard LDAP modify operation, no LDAP result code to a client. Instead, the result code is placed in a parameter block that is returned by the function.

Example

[To be added]

See Also

slapi_modrdn_internal()

Performs an LDAP modify RDN operation to rename a directory entry from your plug-in.

Syntax

```
#include "slapi-plugin.h"
Slapi_PBlock *slapi_modrdn_internal( char * olddn,
   char * newrdn, int deloldrdn, LDAPControl **controls,
   int log_change);
```

Parameters

The function has the following parameters:

`olddn`	Distinguished name (DN) of the entry that you want to rename.
`newdn`	New relative distinguished name (RDN) of the entry.
`deloldrdn`	Specifies whether or not you want to remove the old RDN from the entry. • If 1, removes the old RDN. • If 0, leaves the old RDN as an attribute of the entry.
`controls`	A NULL-terminated array of LDAP controls (see "LDAPControl" on page 193) that you want applied to the modify RDN operation.
`log_change`	Specifies whether or not you want to log the changes to the replication log. • If 1, write the changes to the replication log. • If 0, do not write the changes to the replication log.

Returns

A new parameter block (see "Slapi_PBlock" on page 215) with the following parameter set:

- `SLAPI_PLUGIN_INTOP_RESULT` specifies the LDAP result code for the internal LDAP operation (for example, `LDAP_SUCCESS` if the operation is successful or `LDAP_PARAM_ERROR` if an invalid parameter is used).

  If the DN of the entry has a suffix that is not served by the Directory Server, `SLAPI_PLUGIN_INTOP_RESULT` is set to `LDAP_REFERRAL`.

Description

This function allows you to rename an entry to the directory from a plug-in function. The arguments for this function are similar to the arguments for the standard `ldap_modrdn2()` function in the Netscape LDAP C SDK.

Unlike the standard LDAP modify RDN operation, no LDAP result code to a client. Instead, the result code is placed in a parameter block that is returned by the function.

Example [To be added]

See Also

slapi_mr_filter_index()

Calls the indexer function associated with an extensible match filter.

Syntax

```
#include "slapi-plugin.h"
int slapi_mr_filter_index (Slapi_Filter *f, Slapi_PBlock *pb);
```

Parameters

The function has the following parameters:

`f`	Pointer to a `Slapi_Filter` structure, representing the extensible match filter for which you want to find the indexer function.
`pb`	Parameter block containing information about the extensible match filter.

Returns

The result code returned by the indexer function.

Description

If the filter specified by the `f` argument is an extensible match filter, this function calls the indexer function associated with the filter.

Before calling this function, make sure that the parameter block `pb` contains the information needed by the indexer function. You can pass information to the indexer function by using the following parameters:

- `SLAPI_PLUGIN_MR_VALUES` should contain a NULL-terminated list of values from the extensible match filter.
- `SLAPI_PLUGIN_OBJECT` should contain information that you want to pass to the indexer function.

The indexer function should set the `SLAPI_PLUGIN_MR_KEYS` parameter of the parameter block pb to an array of the keys that correspond to the values in the `SLAPI_PLUGIN_MR_VALUES` parameter.

For more information on filter index functions and indexer functions, see Chapter 12, "Writing Matching Rule Plug-Ins".

Example

[To be added]

See Also

slapi_mr_indexer_create()

Calls the indexer factory function for the plug-in responsible for a specified matching rule.

Syntax

```
#include "slapi-plugin.h"
int slapi_mr_indexer_create (Slapi_PBlock *opb);
```

Parameters

The function has the following parameters:

`pb`	Parameter block containing information about the matching rule and attribute type to be used in indexing or sorting.

Returns

The result code returned by the indexer factory function.

Description

This function calls the indexer factory function for the plug-in responsible for handing a specified matching rule. The matching rule is identified by the OID in the `SLAPI_PLUGIN_MR_OID` parameter.

If no plug-ins are associated with this matching rule, the function calls the indexer factory function for each matching rule plug-in until the `SLAPI_PLUGIN_MR_INDEX_FN` parameter is set to an indexer function.

Before calling this function, make sure that the parameter block `pb` contains the information needed by the indexer factory function. You can pass information to the indexer factory function by using the following parameters:

- `SLAPI_PLUGIN_MR_OID` should contain the OID of the matching rule that you want used for indexing or sorting.
- `SLAPI_PLUGIN_MR_TYPE` should contain the attribute type that you want used for indexing or sorting.
- `SLAPI_PLUGIN_MR_USAGE` should specify if the indexer will be used for indexing (`SLAPI_PLUGIN_MR_USAGE_INDEX`) or for sorting (`SLAPI_PLUGIN_MR_USAGE_SORT`).

The indexer factory function should set the following parameters:

- `SLAPI_PLUGIN_MR_OID` should contain the official OID of the matching rule that you want used for indexing or sorting.
- `SLAPI_PLUGIN_MR_INDEX_FN` should specify the name of the indexer function responsible for indexing or sorting, based on the matching rule OID or attribute type.

- `SLAPI_PLUGIN_OBJECT` should contain any information that you want passed to the indexer function.
- `SLAPI_PLUGIN_DESTROY_FN` should specify the name of the function responsible for freeing any memory allocated by this indexer factory function (for example, memory allocated for a structure that you pass to the indexer function using `SLAPI_PLUGIN_OBJECT`).

For more information on filter index functions and indexer functions, see Chapter 12, “Writing Matching Rule Plug-Ins”.

Example [To be added]

See Also

slapi_new_condvar()

Creates a new condition variable and returns a pointer to the corresponding `Slapi_CondVar` structure.

Syntax

```
#include "slapi-plugin.h"
Slapi_CondVar *slapi_new_condvar( Slapi_Mutex *mutex );
```

Parameters

The function has the following parameters:

`mutex`	Pointer to an `Slapi_Mutex` structure representing the mutex that you want used to protect this condition variable.

Returns

A pointer to the new `Slapi_CondVar` structure, or NULL if memory cannot be allocated.

Description

This function creates a new condition variable and returns a pointer to the `Slapi_CondVar` structure. You can create the `Slapi_Mutex` structure by calling the `slapi_new_mutex()` function.

To wait on the condition variable, call the `slapi_wait_condvar()` function. To notify waiting threads, call the `slapi_notify_condvar()` function.

When you are done working with this `Slapi_CondVar` structure, call the `slapi_destroy_condvar()` function to free the structure from memory.

Example

[To be added]

See Also

slapi_new_mutex()

Creates a new mutex and returns a pointer to the corresponding `Slapi_Mutex` structure.

Syntax

```
#include "slapi-plugin.h"
Slapi_Mutex *slapi_new_mutex();
```

Returns

A pointer to the new `Slapi_Mutex` structure, or NULL if memory cannot be allocated.

Description

This function creates a new mutex and returns a pointer to the `Slapi_Mutex` structure. You can lock this mutex by calling the `slapi_lock_mutex()` function and unlock the mutex by calling the `slapi_unlock_mutex()` function.

When you are done working with the mutex, you can free the `Slapi_Mutex` structure by calling the `slapi_destroy_mutex()` function.

Example

[To be added]

See Also

slapi_notify_condvar()

Notifies a thread that is waiting on the specified condition variable.

Syntax

```
#include "slapi-plugin.h"
int slapi_notify_condvar( Slapi_CondVar *cvar, int notify_all );
```

Parameters

The function has the following parameters:

`cvar`	Pointer to an `Slapi_CondVar` structure representing the condition variable.
`notify_all`	If 1, notifies all threads that are waiting on the condition variable.

Returns

One of the following values:

- A non-zero value if the thread (or threads) are successfully notified.
- 0 if an error occurs (for example, if the condition variable is NULL or if the mutex associated with the condition variable is not locked).

Description

This function notifies one or all threads that are waiting on the condition variable (see the `slapi_wait_condvar()` function). Before calling this function, the calling thread must lock the mutex associated with this condition variable.

Example

[To be added]

See Also

slapi_op_abandoned()

Determines whether or not the client has abandoned the current operation (the operation that passes in the parameter block).

Syntax

```
#include “slapi-plugin.h”
int slapi_op_abandoned( Slapi_PBlock *pb );
```

Parameters

The function has the following parameters:

pb	Parameter block passed in from the current operation.

Returns

1 if the operation has been abandoned, or 0 if the operation has not been abandoned.

Example

[To be added]

See Also

slapi_pblock_destroy()

Frees the specified parameter block from memory.

Syntax

```
#include “slapi-plugin.h”
void slapi_pblock_destroy( Slapi_PBlock *pb );
```

Parameters

The function has the following parameters:

pb	Parameter block that you want to free.

Example

[To be added]

See Also

slapi_pblock_get()

Gets the value of a name-value pair from a parameter block.

Syntax

```
#include "slapi-plugin.h"
int slapi_pblock_get( Slapi_PBlock *pb, int arg, void *value );
```

Parameters

The function has the following parameters:

`pb`	Parameter block.
`arg`	ID of the name-value pair that you want to get. For a list of IDs that you can specify, see Chapter 15, "Parameter Reference".
`value`	Pointer to the value retrieved from the parameter block.

Returns

0 if successful, or -1 if an error occurs (for example, if an invalid ID is specified).

Example

[To be added]

See Also

slapi_pblock_new()

Creates a new parameter block.

Syntax

```
#include “slapi-plugin.h”
Slapi_PBlock *slapi_pblock_new();
```

Returns Pointer to the new parameter block.

Example [To be added]

See Also

slapi_pblock_set()

Sets the value of a name-value pair in a parameter block.

Syntax

```
#include "slapi-plugin.h"
int slapi_pblock_set( Slapi_PBlock *pb, int arg, void *value );
```

Parameters

The function has the following parameters:

`pb`	Parameter block.
`arg`	ID of the name-value pair that you want to set. For a list of IDs that you can specify, see Chapter 15, "Parameter Reference".
`value`	Pointer to the value that you want to set in the parameter block.

Returns

0 if successful, or -1 if an error occurs (for example, if an invalid ID is specified).

Example

[To be added]

See Also

slapi_pw_find()

Determines whether or not a specified password matches one of the encrypted values of an attribute. For example, you can call this function to determine if a given password matches a value in the `userpassword` attribute.

Syntax

```
#include "slapi-plugin.h"
int slapi_pw_find( struct berval **vals, struct berval *v );
```

Parameters

The function has the following parameters:

`vals`	Pointer to the array of `berval` structures containing the values of an attribute that stores passwords (for example, the `userpassword` attribute). To get this value of the attribute, call the `slapi_entry_attr_find()` function.
`v`	Pointer to the `berval` structure containing the password that you want to check (for example, you can get this value from the `SLAPI_BIND_CREDENTIALS` parameter in the parameter block).

Returns

0 if the password specified by `v` was found in `vals`, or a non-zero value if the password `v` was not found in `vals`.

Description

When the Netscape Directory Server stores the password for an entry in the `userpassword` attribute, it encrypts the password using the scheme specified in the `passwdhash` directive of the `slapd.conf` configuration file. The scheme can be `crypt` or `sha` or `""` (for cleartext).

If you need to determine if a given password is one of the values of the `userpassword` attribute, you can call the `slapi_pw_find()` function. This function determines which password scheme was used to store the password and uses the appropriate comparison function to compare a given value against the encrypted values of the `userpassword` attribute.

Example

[To be added]

See Also

slapi_register_supported_control()

Registers the specified control with the server. This function associates the control with an object identification (OID). When the server receives a request that specifies this OID, the server makes use of this information to determine if the control is supported by the server or its plug-ins.

Syntax

```
#include "slapi-plugin.h"
void slapi_register_supported_control( char *controloid,
unsigned long controlops );
```

Parameters

The function has the following parameters:

`controloid`	OID of the control you want to register.
`controlops`	Operation that the control is applicable to.

The `controlops` argument can have one or more of the following values:

ID	Description
`SLAPI_OPERATION_BIND`	The specified control applies to the LDAP bind operation.
`SLAPI_OPERATION_UNBIND`	The specified control applies to the LDAP unbind operation.
`SLAPI_OPERATION_SEARCH`	The specified control applies to the LDAP search operation.
`SLAPI_OPERATION_MODIFY`	The specified control applies to the LDAP modify operation.
`SLAPI_OPERATION_ADD`	The specified control applies to the LDAP add operation.
`SLAPI_OPERATION_DELETE`	The specified control applies to the LDAP delete operation.
`SLAPI_OPERATION_MODDN`	The specified control applies to the LDAP modify DN operation.
`SLAPI_OPERATION_MODRDN`	The specified control applies to the LDAPv3 modify RDN operation.
`SLAPI_OPERATION_COMPARE`	The specified control applies to the LDAP compare operation.

ID	Description
SLAPI_OPERATION_ABANDON	The specified control applies to the LDAP abandon operation.
SLAPI_OPERATION_EXTENDED	The specified control applies to the LDAPv3 extended operation.
SLAPI_OPERATION_ANY	The specified control applies to any LDAP operation.
SLAPI_OPERATION_NONE	The specified control applies to none of the LDAP operations.

You can specify a combination of values by bitwise ORing the values together (for example, SLAPI_OPERATION_ADD | SLAPI_OPERATION_DELETE).

Example [To be added]

See Also

slapi_register_supported_saslmechanism()

Registers the specified Simple Authentication and Security Layer (SASL) mechanism with the server.

Syntax

```
#include “slapi-plugin.h”
void slapi_register_supported_saslmechanism( char *mechanism );
```

Parameters

The function has the following parameters:

`mechanism`	Name of the SASL mechanism

Example

[To be added]

See Also

slapi_search_internal()

Performs an LDAP search operation to search the directory from your plug-in.

Syntax

```
#include “slapi-plugin.h”
Slapi_PBlock *slapi_search_internal( char *base, int scope,
   char *filter, LDAPControl **controls, char **attrs,
   int attrsonly );
```

Parameters The function has the following parameters:

`base`	Distinguished name (DN) of the entry that serves as the starting point for the search. For example, setting `base` to `"o=Ace Industry, c=US"` restricts the search to entries at Ace Industry in the United States.
`scope`	Scope of the search, which can be one of the following values: • `LDAP_SCOPE_BASE` searches the entry specified by `base`. • `LDAP_SCOPE_ONELEVEL` searches all entries one level beneath the entry specified by `base`. • `LDAP_SCOPE_SUBTREE` searches the entry specified by `base` and all entries at all levels beneath the entry specified by `base`.
`filter`	String representation of the filter to apply in the search.
`controls`	A NULL-terminated array of LDAP controls (see “LDAPControl” on page 193) that you want applied to the search operation.
`attrs`	A NULL-terminated array of attribute types to return from entries that match filter. If you specify a NULL, all attributes will be returned.
`attrsonly`	Specifies whether or not attribute values are returned along with the attribute types. This parameter can have the following values: • 0 specifies that both attribute types and attribute values are returned. • 1 specifies that only attribute types are returned.

Returns A new parameter block (see “Slapi_PBlock” on page 215) with the following parameters set:

- `SLAPI_PLUGIN_INTOP_RESULT` specifies the LDAP result code for the internal LDAP operation (for example, `LDAP_SUCCESS` if the operation is successful or `LDAP_PARAM_ERROR` if an invalid parameter is used).
- `SLAPI_PLUGIN_INTOP_SEARCH_ENTRIES` specifies an array of entries (pointers to `Slapi_Entry` structures) found by the search.
- `SLAPI_PLUGIN_INTOP_SEARCH_REFERRALS` specifies an array of referrals (strings) found by the search. Referrals are in the form of LDAP URLs.

When you are done working with the search results, you should free the results in the parameter block by calling the `slapi_free_search_results_internal()` function.

Description This function allows you to search the directory from a plug-in function. The arguments for this function are similar to the arguments for the standard `ldap_search()` function in the Netscape LDAP C SDK.

Unlike the standard LDAP search operation, no data is sent to a client:

- The internal search operation does not send an LDAP result code to a client.
- When the internal search operation finds a matching entry, no entry is sent to a client.
- If the internal search operation finds an entry that contains LDAP v3 referrals, no referrals are sent to a client.

Although these tasks are not performed during an internal search operation, you can write your own callback functions that are invoked when these events occur. For details, see “slapi_search_internal_callback()” on page 336.

Example [To be added]

See Also

slapi_search_internal_callback()

Performs an LDAP search operation to search the directory from your plug-in. Unlike the `slapi_search_internal()` function, this function allows you to specify callback functions that are invoked when the search operation finds matching entries or entries with referrals.

Syntax

```
#include “slapi-plugin.h”
int slapi_search_internal_callback( char *ibase, int scope,
   char *ifstr, char **attrs, int attrsonly,
   void *callback_data, LDAPControl **controls,
   plugin_result_callback prc,
   plugin_search_entry_callback psec,
   plugin_referral_entry_callback prec);
```

Parameters

The function has the following parameters:

`ibase` — Distinguished name (DN) of the entry that serves as the starting point for the search. For example, setting `base` to `"o=Ace Industry, c=US"` restricts the search to entries at Ace Industry in the United States.

`scope` — Scope of the search, which can be one of the following values:

- `LDAP_SCOPE_BASE` searches the entry specified by `base`.
- `LDAP_SCOPE_ONELEVEL` searches all entries one level beneath the entry specified by `base`.
- `LDAP_SCOPE_SUBTREE` searches the entry specified by `base` and all entries at all levels beneath the entry specified by `base`.

`ifstr` — String representation of the filter to apply in the search.

`attrs` — A NULL-terminated array of attribute types to return from entries that match filter. If you specify a NULL, all attributes will be returned.

`attrsonly` — Specifies whether or not attribute values are returned along with the attribute types. This parameter can have the following values:

- 0 specifies that both attribute types and attribute values are returned.
- 1 specifies that only attribute types are returned.

`callback_data` — A pointer to data that you want passed to the callback functions specified by the `prc`, `psec`, and `prec` arguments.

`controls`	A NULL-terminated array of LDAP controls (see “LDAPControl” on page 193) that you want applied to the search operation.
`prc`	Callback function that the server calls to send result codes. The function must have the prototype specified by `plugin_result_callback`.
`psec`	Callback function that the server calls when finding a matching entry in the directory. The function must have the prototype specified by `plugin_search_entry_callback`.
`prec`	Callback function that the server calls when finding an entry that contains LDAP v3 referrals. The function must have the prototype specified by `plugin_referral_entry_callback`.

Returns A new parameter block (see “Slapi_PBlock” on page 215) with the following parameters set:

- `SLAPI_PLUGIN_INTOP_RESULT` specifies the LDAP result code for the internal LDAP operation (for example, `LDAP_SUCCESS` if the operation is successful or `LDAP_PARAM_ERROR` if an invalid parameter is used).
- `SLAPI_PLUGIN_INTOP_SEARCH_ENTRIES` specifies an array of entries (pointers to `Slapi_Entry` structures) found by the search.
- `SLAPI_PLUGIN_INTOP_SEARCH_REFERRALS` specifies an array of referrals (strings) found by the search. Referrals are in the form of LDAP URLs.

When you are done working with the search results, you should free the results in the parameter block by calling the `slapi_free_search_results_internal()` function.

Description Like the `slapi_search_internal()` function, the `slapi_search_internal_callback()` function allows you to search the directory from a plug-in function.

Unlike a search operation requested by a client, no result code, search entries, or referrals are sent to a client by `slapi_search_internal_callback()`. However, you can write your own callback functions that are invoked when these events occur:

- You can write a callback function that is invoked when the search operation normally sends a result code to the client.

This function must have the prototype specified by `plugin_result_callback`. You specify this function in the `prc` argument of `slapi_search_internal_callback()`.

- You can write a callback function that is invoked when the search operation normally sends a search entry to the client.

  This function must have the prototype specified by `plugin_search_entry_callback`. You specify this function in the `psec` argument of `slapi_search_internal_callback()`.

- You can write a callback function that is invoked when the search operation normally sends LDAP v3 referrals (search result references, which are normally sent to the client as results found by the search) to an LDAP v3 client.

  Note he function is not called for LDAP v2 clients, since version 2 of the LDAP protocol does not support the mechanism for sending search result references to clients. (In the LDAP v2 protocol, the server just sends an `LDAP_PARTIAL_RESULT` result code back to the client.)

  This function must have the prototype specified by `plugin_referral_entry_callback`. You specify this function in the `prec` argument of `slapi_search_internal_callback()`.

You can also pass data to these callback functions. Specify the data that you want to pass as the `callback_data` argument of `slapi_search_internal_callback()`.

The rest of the arguments for this function are similar to the arguments for the `slapi_search_internal()` function.

Example [To be added]

See Also

slapi_send_ldap_referral()

Processes an entry's LDAP v3 referrals (which are found in the entry's `ref` attribute). For LDAP v3 clients, this function sends the LDAP referrals back to the client. For LDAP v2 clients, this function copies the referrals to an array of `berval` structures that you can pass to `slapi_send_ldap_result()` function at a later time.

Syntax

```
#include "slapi-plugin.h"
int slapi_send_ldap_referral( Slapi_PBlock *pb, Slapi_Entry *e,
struct berval **refs, struct berval ***urls );
```

Parameters

The function has the following parameters:

`pb`	Parameter block.
`e`	Pointer to the `Slapi_Entry` structure representing the entry that you are working with.
`refs`	Pointer to the NULL-terminated array of `berval` structures containing the LDAP v3 referrals (search result references) found in the entry.
`urls`	Pointer to the array of `berval` structures used to collect LDAP referrals for LDAP v2 clients.

Returns

0 if successful, or -1 if an error occurs.

Description

When you call this function, the server processes the LDAP referrals specified in the `refs` argument. The server processes referrals in different ways, depending on the version of the LDAP protocol supported by the client:

- In the LDAP v3 protocol, references to other LDAP servers (search result references) can be sent to clients as search results. (For example, a server can send a mixture of entries found by the search and references to other LDAP servers as the results of a search.)

  When you call the `slapi_send_ldap_referral()` function for LDAP v3 clients, the server sends the referrals specified in the `refs` argument back to the client as search result references. (The `urls` argument is not used in this case.)

- In the LDAP v2 protocol, servers can send the LDAP result code `LDAP_PARTIAL_RESULTS` to refer the client to other LDAP server.

When you call the `slapi_send_ldap_referral()` function for LDAP v2 clients, the server collects the referrals specified in `refs` in the `urls` argument. No data is sent to the LDAP v2 client.

To get the referrals to an LDAP v2 client, you need to pass the `urls` argument (along with an `LDAP_PARTIAL_RESULTS` result code) to the `slapi_send_ldap_result()` function. `slapi_send_ldap_result()` concatenates the referrals specified in the `urls` argument and sends the resulting string to the client as part of the error message.

If you want to define your own function for sending referrals, write a function complies with the type definition `send_ldap_referral_fn_ptr_t` and set the `SLAPI_PLUGIN_DB_REFERRAL_FN` parameter in the parameter block to the name of your function.

Example [To be added]

See Also `slapi_send_ldap_result()`, `slapi_send_ldap_search_entry()`.

slapi_send_ldap_result()

Sends an LDAP result code back to the client.

Syntax

```
#include "slapi-plugin.h"
void slapi_send_ldap_result( Slapi_PBlock *pb, int err,
   char *matched, char *text, int nentries,
   struct berval **urls );
```

Parameters

The function has the following parameters:

`pb`	Parameter block.
`err`	LDAP result code that you want sent back to the client (for example, `LDAP_SUCCESS`).
`matched`	When sending back an `LDAP_NO_SUCH_OBJECT` result code, use this argument to specify the portion of the target DN that could be matched. (Pass NULL in other situations.)
`text`	Error message that you want sent back to the client. (Pass NULL if you do not want an error message sent back.)
`nentries`	When sending back the result code for an LDAP search operation, use this argument to specify the number of matching entries found.
`urls`	When sending back an `LDAP_PARTIAL_RESULTS` result code to an LDAP v2 client or an `LDAP_REFERRAL` result code to an LDAP v3 client, use this argument to specify the array of `berval` structures containing the referral URLs. (Pass NULL in other situations.)

Description

Call `slapi_send_ldap_result()` to send an LDAP result code (such as `LDAP_SUCCESS`) back to the client.

The following arguments are intended for use only in certain situations:

- `matched`

  When sending an `LDAP_NO_SUCH_OBJECT` result code back to a client, use `matched` to specify how much of the target DN could be found in the database. For example, if the client was attempting to find the DN:

  ```
  cn=Babs Jensen, ou=Product Division, o=Ace Industries, c=US
  ```

  and the database contains entries for "c=US" and "o=Ace Industries, c=US", but no entry for "ou=Product Division, o=Ace Industries, c=US", you should set the `matched` parameter to:

```
o=Ace Industries, c=US
```

- `urls`

  When sending an `LDAP_PARTIAL_RESULTS` result code back to an LDAP v2 client or an `LDAP_REFERRAL` result code back to an LDAP v3 client, use `urls` to specify the referral URLs.

  For LDAP v3 referrals, you can call the `slapi_send_ldap_referral()` to send referrals to LDAP v3 clients and collect them for LDAP v2 clients. You can pass the array of collected referrals to the urls argument of `slapi_send_ldap_results()`. For example:

```
struct berval **urls;

...

slapi_send_ldap_referral( ld, e, &refs, &urls );

slapi_send_ldap_result( ld, LDAP_PARTIAL_RESULTS, NULL, NULL, 0,
   urls );
```

If you want to define your own function for sending result codes, write a function complies with the type definition `send_ldap_result_fn_ptr_t` and set the `SLAPI_PLUGIN_DB_RESULT_FN` parameter in the parameter block to the name of your function.

Example [To be added]

See Also `slapi_send_ldap_referral()`, `slapi_send_ldap_search_entry()`.

slapi_send_ldap_search_entry()

Sends an entry found by a search back to the client.

Syntax

```
#include "slapi-plugin.h"
int slapi_send_ldap_search_entry( Slapi_PBlock *pb,
   Slapi_Entry *e, LDAPControl **ectrls, char **attrs,
   int attrsonly );
```

Parameters

The function has the following parameters:

`pb`	Parameter block.
`e`	Pointer to the `Slapi_Entry` structure representing the entry that you want to send back to the client.
`ectrls`	Pointer to the array of `LDAPControl` structures representing the controls associated with the search request.
`attrs`	Attribute types specified in the LDAP search request
`attrsonly`	Specifies whether or not the attribute values should be sent back with the result. • If 0, the values are included. • If 1, the values are not included.

Description

Call `slapi_send_ldap_search_entry()` to send an entry found by a search back to the client.

`attrs` is the array of attribute types that you want to send from the entry. This value is equivalent to the `SLAPI_SEARCH_ATTRS` parameter in the parameter block.

`attrsonly` specifies whether you want to send only the attribute types or the attribute types and their values:

- Pass 0 for this parameter if you want to send both the attribute types and values to the client.
- Pass 1 for this parameter if you want to send only the attribute types (not the attribute values) to the client.

This value is equivalent to the `SLAPI_SEARCH_ATTRSONLY` parameter in the parameter block.

If you want to define your own function for sending entries, write a function complies with the type definition `send_ldap_search_entry_fn_ptr_t` and set the `SLAPI_PLUGIN_DB_ENTRY_FN` parameter in the parameter block to the name of your function.

Example [To be added]

See Also `slapi_send_ldap_referral()`, `slapi_send_ldap_search_entry()`.

slapi_str2entry()

Converts a LDIF description of a directory entry (a string value) into an entry of the `Slapi_Entry` type.

Syntax

```
#include "slapi-plugin.h"
Slapi_Entry *slapi_str2entry( char *s, int flags );
```

Parameters

The function has the following parameters:

`s`	Description of an entry that you want to convert to `Slapi_Entry`.
`flags`	One or more flags specifying how the entry should be generated

The value of the `flags` argument can be one of the following values:

`SLAPI_STR2ENTRY_REMOVEDUPVALS`	Removes any duplicate values in the attributes of the entry.
`SLAPI_STR2ENTRY_ADDRDNVALS`	Adds the relative distinguished name (RDN) components (for example, `uid=bjensen`) as attributes of the entry.

Returns

Pointer to the `Slapi_Entry` structure representing the entry, or NULL if the string cannot be converted (for example, if no DN is specified in the string).

Description

A directory entry can be described by a string in LDIF format (for details, see "Converting Between Entries and Strings" on page 53).

Calling the `slapi_str2entry()` function converts a string description in this format to a `Slapi_Entry` structure, which you can pass to other API functions.

Note

This function modifies the string argument `s`. If you still need to use this string value, you should make a copy of this string before calling `slapi_str2entry()`.

If an error occurred during the conversion process, the function returns NULL instead of the entry.

When you are done working with the entry, you should call the `slapi_entry_free()` function.

To convert an entry to a string description, call the `slapi_entry2str()` function.

Example [To be added]

See Also `slapi_entry2str()`.

slapi_str2filter()

Converts a string description of a search filter into into a filter of the `Slapi_Filter` type.

Syntax

```
#include "slapi-plugin.h"
Slapi_Filter *slapi_str2filter( char *str );
```

Parameters

The function has the following parameters:

`str`	String description of a search filter.

Returns

Pointer to the `Slapi_Filter` structure representing the search filter, or NULL if the string cannot be converted (for example, if an empty string is specified or if the filter syntax is incorrect). When you are done working with this filter, you should free the `Slapi_Filter` structure by calling `slapi_filter_free()`.

Example

[To be added]

See Also

slapi_unlock_mutex()

Unlocks the specified mutex.

Syntax

```
#include "slapi-plugin.h"
int slapi_unlock_mutex( Slapi_Mutex *mutex );
```

Parameters The function has the following parameters:

`mutex`	Pointer to an `Slapi_Mutex` structure representing the mutex that you want to unlock.

Returns One of the following values:

- A non-zero value if the mutex was successfully unlocked.
- 0 if the mutex was NULL or was not locked by the calling thread.

Description This function unlocks the mutex specified by the `Slapi_Mutex` structure.

Example [To be added]

See Also

slapi_value_find()

[??? Does this exist? It doesn't seem to be defined anywhere. ???]

Syntax

```
#include "slapi-plugin.h"
int slapi_value_find( void *plugin, struct berval **vals, struct
berval *v );
```

slapi_verify_aci_syntax()

Determines whether or not the access control items (ACIs) on an entry are valid.

Syntax

```
#include "slapi-plugin.h"
int slapi_acl_verify_aci_syntax (Slapi_Entry *e, char **errbuf);
```

Parameters

The function has the following parameters:

`e`	Entry that you want to check
`errbuf`	Pointer to the error message returned if the ACI syntax is invalid.

Returns

0 if successful, or -1 if an error occurs.

Example

[To be added]

See Also

slapi_wait_condvar()

Waits on a condition variable.

Syntax

```
#include “slapi-plugin.h”
int slapi_wait_condvar( Slapi_CondVar *cvar, struct timeval
*timeout );
```

Parameters

The function has the following parameters:

`cvar`	Pointer to an `Slapi_CondVar` structure representing the condition variable that you want to wait on.
`timeout`	Timeout period to wait for notification on the condition variable. If NULL, the calling function blocks indefinitely.

Returns

One of the following values:

- A non-zero value if successful.
- 0 if an error occurs (for example, if the condition variable is NULL or if the mutex associated with the condition variable is not locked).

Description

This function waits on the condition variable until it receives notification (see the `slapi_notify_condvar()` function). Before calling this function, the calling thread must lock the mutex associated with this condition variable.

Example

[To be added]

See Also

[??? Where are these used? ???]

#defineSLAPI_SEQ_FIRST1

#defineSLAPI_SEQ_LAST2

#defineSLAPI_SEQ_PREV3

#defineSLAPI_SEQ_NEXT4

SLAPI_BERVAL_EQ()

Compares the values in two `berval` structures to determine if they are equal.

Syntax `SLAPI_BERVAL_EQ(L,R)`

Parameters The macro has the following parameters:

`L`	Pointer to the first `berval` structure that you want to compare.
`R`	Pointer to the second `berval` structure that you want to compare.

Returns 1 if the two values are equal, or 0 if they are not equal.

Example [To be added]

See Also

Chapter

15

Parameter Reference

This chapter describes the parameters available in the `Slapi_PBlock` parameter block, the type of data associated with each parameter, and the plug-in functions in which those parameters are accessible.

To get the values of these parameters, call the `slapi_pblock_get()` function. To set the values of these parameters, call the `slapi_pblock_set()` function. Using these parameters, you can get and set the following information:

- Parameters for Registering Plug-In Functions
- Parameters Accessible to All Plug-Ins
- Parameters for the Configuration Function
- Parameters for the Bind Function
- Parameters for the Search Function
- Parameters for the Add Function
- Parameters for the Compare Function
- Parameters for the Delete Function
- Parameters for the Modify Function
- Parameters for the Modify RDN Function

Parameters for Registering Plug-In Functions

The parameters listed in this section identify plug-in functions recognized by the server. To register your plug-in function, set the value of the appropriate parameter to the name of your function.

Note With the exception of the parameters for matching rule plug-in functions, you do not need to get the value of any of these parameters.

The parameters for registering plug-in functions are organized in the following sections:

- Database Plug-Ins
- Pre-Operation/Data Validation Plug-Ins
- Post-Operation/Data Notification Plug-Ins
- Extended Operation Plug-Ins
- Matching Rule Plug-Ins

Database Plug-Ins

The parameters listed in this section are used to register database plug-in functions.

Each parameter corresponds to an operation performed by the back-end database. When integrating your own database with the Directory Server, you need to write and register your own plug-in functions that handle these operations.

To register your plug-in function, write an initialization function that sets the values of the following parameters to your functions.

Parameter ID	Description
`SLAPI_PLUGIN_DB_BIND_FN`	This function is called in response to an LDAP bind request.
`SLAPI_PLUGIN_DB_UNBIND_FN`	This function is called in response to an LDAP unbind request.
`SLAPI_PLUGIN_DB_SEARCH_FN`	This function is called in response to an LDAP search request. The function should collect a set of candidates for the search results.
`SLAPI_PLUGIN_DB_COMPARE_FN`	This function is called in response to an LDAP compare request.
`SLAPI_PLUGIN_DB_MODIFY_FN`	This function is called in response to an LDAP modify request.
`SLAPI_PLUGIN_DB_MODRDN_FN`	This function is called in response to an LDAP modify RDN request.
`SLAPI_PLUGIN_DB_ADD_FN`	This function is called in response to an LDAP add request.
`SLAPI_PLUGIN_DB_DELETE_FN`	This function is called in response to an LDAP delete request.
`SLAPI_PLUGIN_DB_ABANDON_FN`	This function is called in response to an LDAP abandon operation request.
`SLAPI_PLUGIN_DB_CONFIG_FN`	This function is called when the server is parsing the `slapd.conf` configuration file. If the Directory Server encounters a directive that it does not recognize, it passes the directive to the back-end by using this routine.

Parameter ID	Description
SLAPI_PLUGIN_CLOSE_FN	This function is called when the server is shutting down. You can use this function to prepare your back-end database for shutdown.
SLAPI_PLUGIN_DB_FLUSH_FN	The front-end periodically calls this function. The function is intended to flush any open caches and to save information to disk.
SLAPI_PLUGIN_START_FN	This function is called on start-up and is intended to initialize the database and prepare it for use.
SLAPI_PLUGIN_DB_SEQ_FN	
SLAPI_PLUGIN_DB_ENTRY_FN	This function is called when the server is sending an entry back to the client.
SLAPI_PLUGIN_DB_REFERRAL_FN	This function is called when the server is sending a set of referrals to the client.
SLAPI_PLUGIN_DB_RESULT_FN	This function is called when the server is sending a set of search results to the client.
SLAPI_PLUGIN_DB_LDIFDB_FN	This function is called when the server reads an LDIF file into the database.
SLAPI_PLUGIN_DB_DBLDIF_FN	This function is called when the server exports the datbase to an LDIF file.
SLAPI_PLUGIN_DB_ARCHIVEDB_FN	This function is called when the server restores an archive to the database.
SLAPI_PLUGIN_DB_DBARCHIVE_FN	This function is called when the server backs up the database to an archive.
SLAPI_PLUGIN_DB_NEXT_SEARCH_ENTRY_FN	This function is called for each candidate in the search result set. The function should test each candidate against the search filter and send matching entries back to the client.

Parameter ID	Description
SLAPI_PLUGIN_DB_NEXT_SEARCH_ENTRY_EXT_FN	(Netscape Directory Server 4.0) Reserved for future use. This function is a new version of the "next search entry function". This version of the function can include the LDAP result code with the last search entry and can pass the context of the result set (the last result sent) between function calls.
SLAPI_PLUGIN_DB_ENTRY_RELEASE_FN	(Netscape Directory Server 4.0) Reserved for future use. This function is called after the server sends the last entry back to the client. This function is intended to free the context of the result set (the last result sent).
SLAPI_PLUGIN_DB_SIZE_FN	Specifies the function called to get the size of the database and set as SLAPI_DBSIZE in the parameter block.
SLAPI_PLUGIN_DB_TEST_FN	Specifies the function called to test the back-end and the database.
SLAPI_PLUGIN_DB_DB2INDEX_FN	(Netscape Directory Server 4.0 only) Specifies the function called to generate indexes for the existing database.

Pre-Operation/Data Validation Plug-Ins

The parameters listed in this section are used to register pre-operation/data validation plug-in functions.

To register your plug-in function, write an initialization function that sets the values of the following parameters to your functions.

Parameter ID	Description
SLAPI_PLUGIN_PRE_BIND_FN	This function is called before an LDAP bind operation is completed.
SLAPI_PLUGIN_PRE_UNBIND_FN	This function is called before an LDAP unbind operation is completed.
SLAPI_PLUGIN_PRE_SEARCH_FN	This function is called before an LDAP search operation is completed.
SLAPI_PLUGIN_PRE_COMPARE_FN	This function is called before an LDAP compare operation is completed.
SLAPI_PLUGIN_PRE_MODIFY_FN	This function is called before an LDAP modify operation is completed.
SLAPI_PLUGIN_PRE_MODRDN_FN	This function is called before an LDAP modify RDN operation is completed.
SLAPI_PLUGIN_PRE_ADD_FN	This function is called before an LDAP add operation is completed.
SLAPI_PLUGIN_PRE_DELETE_FN	This function is called before an LDAP delete operation is completed.
SLAPI_PLUGIN_PRE_ABANDON_FN	This function is called before an LDAP abandon operation is completed.
SLAPI_PLUGIN_PRE_ENTRY_FN	This function is called before an entry is sent back to the client.
SLAPI_PLUGIN_PRE_REFERRAL_FN	This function is called before a set of referrals is sent back to the client.
SLAPI_PLUGIN_PRE_RESULT_FN	This function is called before a set of search results is sent back to the client.
SLAPI_PLUGIN_START_FN	This function is called after the server starts up. You can specify a start function for each pre-operation plug-in.
SLAPI_PLUGIN_CLOSE_FN	This function is called before the server shuts down. You can specify a close function for each pre-operation plug-in.

Post-Operation/Data Notification Plug-Ins

The parameters listed in this section are used to register post-operation/data notification plug-in functions.

Parameter ID	Description
SLAPI_PLUGIN_POST_BIND_FN	This function is called after an LDAP bind operation is completed.
SLAPI_PLUGIN_POST_UNBIND_FN	This function is called after an LDAP unbind operation is completed.
SLAPI_PLUGIN_POST_SEARCH_FN	This function is called after an LDAP search operation is completed.
SLAPI_PLUGIN_POST_COMPARE_FN	This function is called after an LDAP compare operation is completed.
SLAPI_PLUGIN_POST_MODIFY_FN	This function is called after an LDAP modify operation is completed.
SLAPI_PLUGIN_POST_MODRDN_FN	This function is called after an LDAP modify RDN operation is completed.
SLAPI_PLUGIN_POST_ADD_FN	This function is called after an LDAP add operation is completed.
SLAPI_PLUGIN_POST_DELETE_FN	This function is called after an LDAP delete operation is completed.
SLAPI_PLUGIN_POST_ABANDON_FN	This function is called after an LDAP abandon operation is completed.
SLAPI_PLUGIN_POST_ENTRY_FN	This function is called after an entry is sent back to the client.
SLAPI_PLUGIN_POST_REFERRAL_FN	This function is called after a set of referrals is sent back to the client.
SLAPI_PLUGIN_POST_RESULT_FN	This function is called after a set of search results is sent back to the client.
SLAPI_PLUGIN_START_FN	This function is called after the server starts up. You can specify a start function for each post-operation plug-in.
SLAPI_PLUGIN_CLOSE_FN	This function is called before the server shuts down. You can specify a close function for each post-operation plug-in.

Extended Operation Plug-Ins

The parameters listed below are used to register extended operation plug-in functions.

Parameter ID	Data Type	Description
SLAPI_PLUGIN_EXT_OP_FN	void *	Your plug-in function for handling an extended operation.
SLAPI_PLUGIN_EXT_OP_OIDLIST	char **	NULL-terminated array of OIDs identifying the extended operations handled by the plug-in function.
SLAPI_PLUGIN_START_FN	void *	This function is called after the server starts up. You can specify a start function for each extended operation plug-in.
SLAPI_PLUGIN_CLOSE_FN	void *	This function is called before the server shuts down. You can specify a close function for each extended operation plug-in.

Matching Rule Plug-Ins

The parameters listed in below are used to register matching rule plug-in functions.

Parameter ID	Description
SLAPI_PLUGIN_MR_FILTER_CREATE_FN	Factory function for creating filter functions. This function must be thread-safe, since the server may call it concurrently with other functions.
SLAPI_PLUGIN_MR_INDEXER_CREATE_FN	Factory function for creating indexer functions. This function must be thread-safe, since the server may call it concurrently with other functions.
SLAPI_PLUGIN_MR_FILTER_MATCH_FN	Filter function.
SLAPI_PLUGIN_MR_FILTER_INDEX_FN	Filter function that uses an index to accelerate the processing of a search request.

Parameter ID	Description
SLAPI_PLUGIN_MR_FILTER_RESET_FN	Function for resetting the filter function.
SLAPI_PLUGIN_MR_INDEX_FN	Indexer function.
SLAPI_PLUGIN_DESTROY_FN	Function for freeing a filter function or indexer function.
SLAPI_PLUGIN_CLOSE_FN	Function called before server shutdown (use this function to clean up before shutdown).

Parameters Accessible to All Plug-Ins

The parameters listed in this section are accessible to all types of plug-ins. The parameters in this section are organized in the following sections:

- Information About the Database
- Information About the Connection
- Information About the Operation
- Notes in the Access Log
- Information About the Plug-In

Information About the Database

The parameters listed below specify information about the back-end database. These parameters are available for all types of plug-ins. Note that these specific parameters can not be set by calling `slapi_pblock_set()`. You can, however, get these parameters by calling `slapi_pblock_get()`.

Parameter ID	Data Type	Description
SLAPI_BACKEND		
SLAPI_BE_MONITORDN	char *	(Netscape Directory Server 3.x releases only) DN used to monitor the back-end database. Note that this is no longer supported in the Netscape Directory Server 4.0 release.
SLAPI_BE_TYPE	char *	Type of back-end database (this is the type specified by the `database` directive in the `slapd.conf` file).
SLAPI_BE_READONLY	int	Specifies whether or not the back-end database is read-only (this is determined by the `readonly` directive in the `slapd.conf` file). • 1 means that the database back-end is read-only. • 0 means that the database back-end is writable.
SLAPI_DBSIZE	int	Specifies the size of the back-end database. If you are using your own database instead of the default database, your `SLAPI_DB_SIZE_FN` function should set the value of this parameter.
SLAPI_BE_LASTMOD		

Information About the Connection

The parameters listed below specify information about the connection. These parameters are available for all types of plug-ins.

Parameter ID	Data Type	Description
SLAPI_CONNECTION		
SLAPI_CONN_ID	int	ID identifying the current connection.
SLAPI_CONN_DN	char *	DN of the user authenticated on the current connection. If you call slapi_pblock_get() to get this DN, you should call slapi_ch_free() to free the resulting DN when done.
SLAPI_CONN_AUTHTYPE	char *	Method used to authenticate the current user. This parameter can have one of the following values: • SLAPD_AUTH_NONE specifies that no authentication mechanism was used (for example, in cases of anonymous authentication). • SLAPD_AUTH_SIMPLE specifies that simple authentication (user name and password) was used to authenticate the current user. • SLAPD_AUTH_SSL specifies that SSL (certificate-based authentication) was used to authenticate the current user. • SLAPD_AUTH_SASL specifies that a SASL (simple authentication and security layer) mechanism was used to authenticate the current user.

Information About the Operation

The parameters listed below specify information about the current operation. These parameters are available for all types of plug-ins.

Parameter ID	Data Type	Description
SLAPI_OPERATION		
SLAPI_OPINITIATED_TIME	time_t	Time when the server began processing the operation
SLAPI_REQUESTOR_ISROOT	int	Specifies whether or not the user requesting the operation is the "root DN". • 1 means that the "root DN" is requesting the operation. • 0 means that the user requesting the operation is not the "root DN." The "root DN" is the "superuser" of the directory. This DN is specified by the `rootdn` directive in the `slapd.conf` configuration file.

Parameter ID	Data Type	Description
SLAPI_REQUESTOR_ISUPDATEDN	int	Specifies whether or not the user requesting the operation is the "update DN". • 1 means that the "update DN" is requesting the operation. • 0 means that the user requesting the operation is not the "update DN." The "update DN" is the master entity responsible for updating the directory during replication. This DN is specified by the `updatedn` directive in the `slapd.conf` configuration file.
SLAPI_REQUESTOR_DN	char *	Specifies the DN of the user requesting the operation.
SLAPI_TARGET_DN	char *	Specifies the DN to which the operation applies (for example, the DN of the entry being added or removed)
SLAPI_REQCONTROLS	LDAPControl **	Array of the controls specified in the request.

Notes in the Access Log

This feature is available in the Netscape Directory Server 4.0 release and is not available in earlier releases.

The parameters listed below specify notes that can be appended to access log entries. These parameters are available for all types of plug-ins.

Parameter ID	Data Type	Description
SLAPI_OPERATION_NOTES	unsigned int	Flags specifying the notes that you want appended to access log entries. You can set this parameter to the following value: • SLAPI_OP_NOTE_UNINDEXED specifies that you want the string "Notes=U" appended to access log entries. You can use this to indicate that a search operation could not use indexes to generate a smaller list of candidates. If no flags are set, no notes are appended to access log entries.

For more information, see "Adding Notes to Access Log Entries" on page 50.

Information About the Plug-In

The parameters listed below specify information about the plug-in that is available to all plug-in functions defined in the current library. These parameters are available for all types of plug-ins.

Parameter ID	Data Type	Description
SLAPI_PLUGIN		
SLAPI_PLUGIN_PRIVATE	void *	Private data that you want passed to your plug-in functions.
SLAPI_PLUGIN_TYPE	int	Specifies the type of plug-in function (see "Types of Plug-Ins" on page 367)
SLAPI_PLUGIN_ARGV	char **	NULL-terminated array of command-line arguments specified for the plugin directive in the slapd.conf file.
SLAPI_PLUGIN_ARGC	int	Number of command-line arguments specified for the plugin directive in the slapd.conf file.

Parameter ID	Data Type	Description
SLAPI_PLUGIN_VERSION	char *	Specifies the version of the plug-in function (see “Version Information” on page 367).
SLAPI_PLUGIN_OPRETURN	int	Specifies the return value of the LDAP operation that has just been processed.
SLAPI_PLUGIN_OBJECT		
SLAPI_PLUGIN_DESTROY_FN		

Types of Plug-Ins

The SLAPI_PLUGIN_TYPE parameter can have one of the following values, which identifies the type of the current plug-in (see Table 2.1 for a more detailed description of the types of server plug-ins):

Defined Constant	Description
SLAPI_PLUGIN_DATABASE	Database plug-in
SLAPI_PLUGIN_EXTENDEDOP	Extended operation plug-in
SLAPI_PLUGIN_PREOPERATION	Pre-operation/data validation plug-in
SLAPI_PLUGIN_POSTOPERATION	Post-operation/data notification plug-in
SLAPI_PLUGIN_MATCHINGRULE	Matching rule plug-in
SLAPI_PLUGIN_SYNTAX	Syntax plug-in

Version Information

To set the value of the SLAPI_PLUGIN_VERSION parameter, you can specify one of the following values:

Defined Constant	Description
SLAPI_PLUGIN_CURRENT_VERSION	The current version of the Netscape Directory Server plug-in
SLAPI_PLUGIN_VERSION_01	Version 1 of the plug-in interface, which is supported by the Netscape Directory Server 3.x and 4.x releases
SLAPI_PLUGIN_VERSION_02	Version 2 of the plug-in interface, which is supported by the Netscape Directory Server 4.x release

Parameters for the Configuration Function

The following table lists the parameters in the parameter block passed to the database configuration function. If you are writing a pre-operation, database, or post-operation configuration function, you can get these values by calling the `slapi_pblock_get()` function.

Parameter ID	Data Type	Description
`SLAPI_CONFIG_FILENAME`	`char *`	Name of the configuration file that is being read (for example, `slapd.conf`)
`SLAPI_CONFIG_LINENO`	`int`	Line number of the current directive in the configuration file.
`SLAPI_CONFIG_ARGC`	`int`	Number of arguments in the current directive.
`SLAPI_CONFIG_ARGV`	`char **`	Array of the arguments from the current directive.

See “Reading Configuration Files” on page 98 for more information on these parameters.

Parameters for the Bind Function

The following table lists the parameters in the parameter block passed to the database bind function. If you are writing a pre-operation, database, or post-operation bind function, you can get these values by calling the `slapi_pblock_get()` function.

Parameter ID	Data Type	Description
`SLAPI_BIND_TARGET`	`char *`	DN of the entry to bind as.
`SLAPI_BIND_METHOD`	`int`	Authentication method used (for example, `LDAP_AUTH_SIMPLE` or `LDAP_AUTH_SASL`).
`SLAPI_BIND_CREDENTIALS`	`struct berval *`	Credentials from the bind request.
`SLAPI_BIND_RET_SASLCREDS`	`struct berval *`	Credentials that you want sent back to the client. (Set this before calling `slapi_send_ldap_result()`.)
`SLAPI_BIND_SASLMECHANISM`	`char *`	SASL mechanism used (for example, `LDAP_SASL_EXTERNAL`).

See “Processing an LDAP Bind Operation” on page 99 for more information on these parameters.

Parameters for the Search Function

The following table lists the parameters in the parameter block passed to the database search function. If you are writing a pre-operation, database, or post-operation search function, you can get these values by calling the `slapi_pblock_get()` function.

Parameter ID	Data Type	Description
`SLAPI_SEARCH_TARGET`	`char *`	DN of the base entry in the search operation (the starting point of the search).
`SLAPI_SEARCH_SCOPE`	`int`	The scope of the search. The scope can be one of the following values: • `LDAP_SCOPE_BASE` • `LDAP_SCOPE_ONELEVEL` • `LDAP_SCOPE_SUBTREE`
`SLAPI_SEARCH_DEREF`	`int`	Method for handling aliases in a search. This method can be one of the following values: • `LDAP_DEREF_NEVER` • `LDAP_DEREF_SEARCHING` • `LDAP_DEREF_FINDING` • `LDAP_DEREF_ALWAYS`
`SLAPI_SEARCH_SIZELIMIT`	`int`	Maximum number of entries to return in the search results.
`SLAPI_SEARCH_TIMELIMIT`	`int`	Maximum amount of time (in seconds) allowed for the search operation.

Parameter ID	Data Type	Description
SLAPI_SEARCH_FILTER	Slapi_Filter *	Slapi_Filter struct (an opaque data structure) representing the filter to be used in the search.
SLAPI_SEARCH_STRFILTER	char *	String representation of the filter to be used in the search.
SLAPI_SEARCH_ATTRS	char **	Array of attribute types to be returned in the search results.
SLAPI_SEARCH_ATTRSONLY	int	Specifies whether the search results return attribute types only or attribute types and values. (0 means return both attributes and values; 1 means return attribute types only)

The following parameters are set by the front-end and back-end as part of the process of executing the search. .

Parameter ID	Data Type	Description
SLAPI_SEARCH_RESULT_SET	void *	Set of search results.
SLAPI_SEARCH_RESULT_ENTRY	void *	Entry returned from iterating through the results set.
SLAPI_SEARCH_RESULT_ENTRY_EXT	void *	(Netscape Directory Server 4.0) Reserved for future use. The context identifying the last result sent in the results set. This "next entry" function actually sets this parameter.
SLAPI_NENTRIES	int	Number of search results found
SLAPI_SEARCH_REFERRALS	struct berval **	Array of the URLs to other LDAP servers that the current server is referring the client to

See "Processing an LDAP Search Operation" on page 101 and "Iterating through Candidates" on page 104 for more information on these parameters.

Parameters for the Add Function

The following table lists the parameters in the parameter block passed to the database add function. If you are writing a pre-operation, database, or post-operation add function, you can get these values by calling the `slapi_pblock_get()` function.

Parameter ID	Data Type	Description
`SLAPI_ADD_TARGET`	`char *`	DN of the entry to be added.
`SLAPI_ADD_ENTRY`	`Slapi_Entry *`	The entry to be added (specified as the opaque `Slapi_Entry` datatype).

See “Processing an LDAP Add Operation” on page 106 for more information on these parameters.

Parameters for the Compare Function

The following table lists the parameters in the parameter block passed to the database compare function. If you are writing a pre-operation, database, or post-operation compare function, you can get these values by calling the `slapi_pblock_get()` function.

Parameter ID	Data Type	Description
`SLAPI_COMPARE_TARGET`	`char *`	DN of the entry to be compared.
`SLAPI_COMPARE_TYPE`	`char *`	Attribute type to use in the comparison.
`SLAPI_COMPARE_VALUE`	`struct berval *`	Attribute value to use in the comparison

See “Processing an LDAP Compare Operation” on page 105 for more information on these parameters.

Parameters for the Delete Function

The following table lists the parameters in the parameter block passed to the database delete function. If you are writing a pre-operation, database, or post-operation delete function, you can get these values by calling the `slapi_pblock_get()` function.

Parameter ID	Data Type	Description
`SLAPI_DELETE_TARGET`	`char *`	DN of the entry to delete.

See “Processing an LDAP Delete Operation” on page 111 for more information on these parameters.

Parameters for the Modify Function

The following table lists the parameters in the parameter block passed to the database modify function. If you are writing a pre-operation, database, or post-operation modify function, you can get these values by calling the `slapi_pblock_get()` function.

Parameter ID	Data Type	Description
`SLAPI_MODIFY_TARGET`	`char *`	DN of the entry to be modified.
`SLAPI_MODIFY_MODS`	`LDAPMod **`	A NULL-terminated array of `LDAPMod` structures, which represent the modifications to be performed on the entry.

See “Processing an LDAP Modify Operation” on page 108 for more information on these parameters.

Parameters for the Modify RDN Function

The following table lists the parameters in the parameter block passed to the database modify RDN function. If you are writing a pre-operation, database, or post-operation modify RDN function, you can get these values by calling the `slapi_pblock_get()` function.

Parameter ID	Data Type	Description
`SLAPI_MODRDN_TARGET`	`char *`	DN of the entry that you want to rename.
`SLAPI_MODRDN_NEWRDN`	`char *`	New RDN to assign to the entry.
`SLAPI_MODRDN_DELOLDRDN`	`int`	Specifies whether or not you want to deleted the old RDN. (0 means don't delete the old RDN; 1 means delete the old RDN)
`SLAPI_MODRDN_NEWSUPERIOR`	`char *`	DN of the new parent of the entry, if the entry is being moved to a new location in the directory tree.

See "Processing an LDAP Modify RDN Operation" on page 109 for more information on these parameters.

Parameters for the Abandon Function

The following table lists the parameters in the parameter block passed to the database abandon function. If you are writing a pre-operation, database, or post-operation abandon function, you can get these values by calling the `slapi_pblock_get()` function.

Parameter ID	Data Type	Description
`SLAPI_ABANDON_MSGID`	`unsigned long`	Message ID of the operation to abandon.

See "Processing an LDAP Abandon Operation" on page 112 for more information on these parameters.

Parameters for Database Import

The following table lists the parameters in the parameter block passed to the database import function, which is responsible for importing LDIF files into the database. If you are writing your own plug-in function for performing this work, you can get these values by calling the `slapi_pblock_get()` function.

Parameter ID	Data Type	Description
`SLAPI_LDIF2DB_FILE`	`char *`	LDIF file that needs to be imported into the database.
`SLAPI_LDIF2DB_REMOVEDUPVALS`	`int`	Specifies whether or not the duplicate values of attributes should be removed. • If 1, you should remove any duplicate attribute values when creating an entry. • If 0, you do not need to remove any duplicate attribute values when creating an entry.
`SLAPI_LDIF2DB_NOATTRINDEXES`	`int`	(Netscape Directory Server 4.0 release only) If 1, the database should not be indexed when the database is created. If 0, the import function should automatically generate database indexes.
`SLAPI_LDIF2DB_INCLUDE`	`char **`	(Netscape Directory Server 4.0 release only) An array of the suffixes or DNs identifying the entries in the LDIF file to be included in the database.
`SLAPI_LDIF2DB_EXCLUDE`	`char **`	(Netscape Directory Server 4.0 release only) An array of the suffixes or DNs identifying the entries in the LDIF file to be excluded from the database.

See “Importing an LDIF File into the Database” on page 114 for more information on these parameters.

Parameters for Database Export

The following table lists the parameters in the parameter block passed to the database import function, which is responsible for importing LDIF files into the database. If you are writing your own plug-in function for performing this work, you can get these values by calling the `slapi_pblock_get()` function.

Parameter ID	Data Type	Description
`SLAPI_DB2LDIF_PRINTKEY`	`int`	Specifies whether or not the database keys should be printed out as well. • If 1, you should include the database key for each entry. • If 0, you do not need to include the database key for each entry.
`SLAPI_DB2LDIF_PRINT_DSE_TREE_FN`	`void *`	Function for printing the front-end DSEs in LDIF format.

Se e “Importing an LDIF File into the Database” on page 114 for more information on these parameters.

Parameters for Database Archive

The following table lists the parameters in the parameter block passed to the database archive function, which is responsible for archiving the database. If you are writing your own plug-in function for performing this work, you can get these values by calling the `slapi_pblock_get()` function.

Parameter ID	Data Type	Description
`SLAPI_SEQ_VAL`	`char *`	Specifies the directory in which you want to store the archive.

Se e “Saving the Database as an Archive” on page 117 for more information on these parameters.

Parameters for Database Restore

The following table lists the parameters in the parameter block passed to the database restore function, which is responsible for restoring the database from an archive. If you are writing your own plug-in function for performing this work, you can get these values by calling the `slapi_pblock_get()` function.

Parameter ID	Data Type	Description
`SLAPI_SEQ_VAL`	`char *`	Specifies the directory containing the archive.

Se e “Restoring the Database from an Archive” on page 117 for more information on these parameters.

Parameters for Database Indexing

This feature is available in the Netscape Directory Server 4.0 release but is not available in earlier releases.

The following table lists the parameters in the parameter block passed to the database indexing function, which is responsible for generating indexes for the database. If you are writing your own plug-in function for performing this work, you can get these values by calling the `slapi_pblock_get()` function.

Parameter ID	Data Type	Description
`SLAPI_DB2INDEX_ATTRS`	`char **`	Specifies a NULL-terminated array of the attribute types that you want indexed.

Se e “Generating Indexes for the Database” on page 118 for more information on these parameters.

Parameters for Extended Operations

The following table lists the parameters in the parameter block passed to extended operation functions. If you are writing your own plug-in function for performing this work, you can get these values by calling the `slapi_pblock_get()` function.

Parameter ID	Data Type	Description
`SLAPI_EXT_OP_REQ_OID`	`char *`	Object ID (OID) of the extended operation specified in the request.
`SLAPI_EXT_OP_REQ_VALUE`	`struct berval*`	Value specified in the request.
`SLAPI_EXT_OP_RET_OID`	`char *`	Object ID (OID) that you want sent back to the client.
`SLAPI_EXT_OP_RET_VALUE`	`struct berval*`	Value that you want sent back to the client.

Parameters for Internal LDAP Operations

The parameters listed below are used in conjunction with functions that you can call to perform LDAP operations from a plug-in (these internal operations do not return any data to a client).

Parameter ID	Data Type	Description
SLAPI_PLUGIN_INTOP_RESULT	int	Result code of the internal LDAP operation
SLAPI_PLUGIN_INTOP_SEARCH_ENTRIES	Slapi_Entry **	Array of entries found by an internal LDAP search operation (see “slapi_search_internal() ” on page 334 for details)
SLAPI_PLUGIN_INTOP_SEARCH_REFERRALS	char **	Array of referrals (in the form of LDAP URLs) found by an internal LDAP search operation (see “slapi_search_internal() ” on page 334 for details)

The following functions set all three parameters:

- `slapi_search_internal()`
- `slapi_search_internal_callback()`

The following functions set only the `SLAPI_PLUGIN_INTOP_RESULT` parameter:

- `slapi_add_internal()`
- `slapi_add_entry_internal()`
- `slapi_delete_internal()`
- `slapi_modify_internal()`
- `slapi_modrdn_internal()`

Parameters for Matching Rule Plug-Ins

The parameters listed below are used in conjunction with matching rule plug-ins.

Parameter ID	Data Type	Description
SLAPI_PLUGIN_MR_OID		
SLAPI_PLUGIN_MR_TYPE		
SLAPI_PLUGIN_MR_VALUE		
SLAPI_PLUGIN_MR_VALUES		
SLAPI_PLUGIN_MR_KEYS		
SLAPI_PLUGIN_MR_FILTER_REUSABLE		
SLAPI_PLUGIN_MR_QUERY_OPERATOR		

The following parameters are listed in the slapi-plugin.h header file but are not currently used:

- SLAPI_MR_FILTER_ENTRY
- SLAPI_MR_FILTER_TYPE
- SLAPI_MR_FILTER_VALUE
- SLAPI_MR_FILTER_OID
- SLAPI_MR_FILTER_DNATTRS

[Still Working On These]

[?? Still working on the doc for this function. Not done yet.??]

[?? THE REST OF THIS SECTION IS WIP ??]

Parameter ID	Data Type	Description
SLAPI_PLUGIN_DB_NO_ACL		

Parameter ID	Data Type	Description
SLAPI_OP_LESS		
SLAPI_OP_LESS_OR_EQUAL		
SLAPI_OP_EQUAL		
SLAPI_OP_GREATER_OR_EQUAL		
SLAPI_OP_GREATER		
SLAPI_OP_SUBSTRING		

Parameter ID	Data Type	Description
SLAPI_PLUGIN_SYNTAX_FILTER_AVA		
SLAPI_PLUGIN_SYNTAX_FILTER_SUB		
SLAPI_PLUGIN_SYNTAX_VALUESKEYS		
SLAPI_PLUGIN_SYNTAX_ASSERTIONKEYS_AVA		
SLAPI_PLUGIN_SYNTAX_ASSERTIONKEYS_SUB		
SLAPI_PLUGIN_SYNTAX_NAMES		
SLAPI_PLUGIN_SYNTAX_OID		
SLAPI_PLUGIN_SYNTAX_FLAGS		
SLAPI_PLUGIN_SYNTAX_COMPARE		

Parameter ID	Data Type	Description
SLAPI_PLUGIN_SYNTAX_FLAG_ORKEYS		
SLAPI_PLUGIN_SYNTAX_FLAG_ORDERING		

Parameter ID	Data Type	Description
SLAPI_MANAGEDSAIT		

[?? Might take this out. Don't know if all this is necessary. ??]

The front-end checks for inconsistencies in this information (for example, if an LDAP v2 client is trying to use the SASL authentication method).

The front-end also checks for LDAP v3 controls in the request. (In LDAP v3, a request can include a control that provides additional information to extend the current operation.) If the control is critical and is not supported by the Directory Server or is not supported for this operation, the front-end returns an `LDAP_UNAVAILABLE_CRITICAL_EXTENSION` error to the client.

If the request contains controls supported by the Directory Server, the front-end sets the controls as the value of the `SLAPI_REQCONTROLS` parameter in the parameter block.

Parameter ID	Data Type	Description
`SLAPI_ENTRY_PRE_OP`		
`SLAPI_ENTRY_POST_OP`		

Parameter ID	Data Type	Description
`SLAPI_RESCONTROLS`		
`SLAPI_ADD_RESCONTROL`		

Parameter ID	Data Type	Description
`SLAPI_SEQ_TYPE`		
`SLAPI_SEQ_ATTRNAME`		
`SLAPI_SEQ_VAL`		

Parameter ID	Data Type	Description
`SLAPI_CHANGENUMBER`		
`SLAPI_LOG_OPERATION`		

[Still Working On These]

Glossary

This glossary defines terms commonly used when working with LDAP.

base DN The distinguished name (DN) that identifies the starting point of a search.

For example, if you want to search all of the entries that under the "ou=People,o=Airius.com" subtree of the directory, "ou=People,o=Airius.com" is the base DN.

computed attribute An attribute with values that are generated by the Netscape Directory Server front-end (these values are not stored in the back-end database).

continuation reference See search reference.

control LDAP controls are specified as part of the LDAP v3 protocol. A control provides the means to specify additional information for an operation. Clients and servers can send controls as part of the requests and responses for an operation.

DIT The hierarchical organization of entries that make up a directory. DIT stands for "Directory Information Tree."

DSA An X.500 term for a directory server. DSA stands for "Directory System Agent."

DSE An entry containing server-specific information. DSE stands for "DSA-specific entry." Each server has different attribute values for the DSE.

extended operation An extension mechanism in the LDAP v3 protocol. You can define extended operations to perform services not covered by the protocol. The extended operation mechanism specifies the means for an LDAP client to request a custom operation (not specified in the LDAP protocol) from an LDAP server.

operational attributes Attributes that are used by servers for administering the directory. For example, creatorsName is an operational attribute that specifies the DN of the user who added the entry. Operational attributes are not returned in any search results unless you specify the attribute by name in the search request.

referral Refers an LDAP client to another LDAP server. An LDAP server can be configured to send your client a referral if your client requests a DN with a suffix that is not in the server's directory tree (for example, if the directory includes entries under "o=Airius.com" and your client requests an entry under "o=AiriusWest.com").

Referrals contain LDAP URLs that specify the host, port, and base DN of another LDAP server.

Note that referrals are not the same as (but are similar to) search references. A search reference is returned as part of the results of a search; a referral is returned when the base DN of a search (or the target DN of any other LDAP operation) is not part of the LDAP server's directory tree.

referral hop limit The maximum number of referrals that your client should follow in a row. For example, suppose your client receives a referral from LDAP server A to LDAP server B. After your client follows the referral to LDAP server B, that server sends you a referral to LDAP server C, which in turn refers you to LDAP server D. Your client has been referred 3 times in a row. If the referral hop limit is 2, the referral hop limit has been exceeded.

root DSE An entry (a DSE) that is located at the root of the DIT.

search reference Also known as continuation references, search result references, or smart referrals. A search reference is an entry in the directory that refers to another LDAP server (the reference is in the form of an LDAP URL).

Search references are returned in search results along with entries found in the search. (A referral, on the other hand, is returned before searching through any entries. A referral is returned if the base DN does not have a suffix that is handled by the server.)

search result reference See search reference.

server plug-in The Netscape Directory Server 3.0 supports a plug-in interface that allows you to extend the functionality of the server. You can write plug-ins that handle extended operations or SASL authentication requests.

smart referral See search reference.

subschema entry Entry containing all the schema definitions (definitions of object classes, attributes, matching rules, and so on) used by entries in part of a directory tree.

Netscape Directory Server Programmer's Guide