1  Eldap User's Guide

The Eldap application provides an api for accessing an LDAP server.
The original code was developed by Torbjörn Törnvist.
2 Reference Manual

The Eldap application provides an api for accessing an LDAP server.

The original code was developed by Torbjörn Törnkvist.
This module provides a client API to the Lightweight Directory Access Protocol (LDAP).

References:

- RFC 4510 - RFC 4519
- RFC 2830

The above publications can be found at IETF.

**DATA TYPES**

Type definitions that are used more than once in this module:

- `handle()`  
  Connection handle
- `attribute()` =
  
  ```
  {Type = string(), Values=[string()]}  
  ```
- `modify_op()`  
  See `mod_add/2, mod_delete/2, mod_replace/2`
- `scope()`  
  See `baseObject/0, singleLevel/0, wholeSubtree/0`
- `dereference()`  
  See `neverDerefAliases/0, derefInSearching/0, derefFindingBaseObj/0, derefAlways/0`
- `filter()`  
  See `present/1, substrings/2, equalityMatch/2, greaterOrEqual/2, lessOrEqual/2, approxMatch/2, extensibleMatch/2, 'and'/1, 'or'/1, 'not'/1`

- `return_value()` =
  
  ```
  ok | {ok, {referral, referrals()}} | {error, Error}  
  ```
- `referrals()` =
  
  ```
  [Address = string()]  
  ```
The contents of Address is server dependent.

**Exports**

- `open([Host]) -> {ok, Handle} | {error, Reason}`  
  Types:
  ```
  Handle = handle()  
  ```
  Setup a connection to an LDAP server, the HOST’s are tried in order.

- `open([Host], [Option]) -> {ok, Handle} | {error, Reason}`  
  Types:
  ```
  Handle = handle()  
  ```
Option = \{port, integer()\} | \{log, function()\} | \{timeout, integer()\} | \\
  \{ssl, boolean()\} | \{sslopts, list()\} | \{tcpopts, list()\}

Setup a connection to an LDAP server, the HOST's are tried in order.

The log function takes three arguments, \(\text{fun}(\text{Level}, \text{FormatString}, [\text{FormatArg}])\) end.

Timeout set the maximum time in milliseconds that each server request may take.

All TCP socket options are accepted except active, binary, deliver, list, mode and packet

close(Handle) -> ok

Types:
  Handle = handle()

Shutdown the connection after sending an unbindRequest to the server. If the connection is tls the connection will be closed with \text{ssl:close/1}, otherwise with \text{gen_tcp:close/1}.

start_tls(Handle, Options) -> return_value()

Same as start_tls(Handle, Options, infinity)

start_tls(Handle, Options, Timeout) -> return_value()

Types:
  Handle = handle()
  Options = \text{ssl:ssl_options()}
  Timeout = infinity | positive_integer()

Upgrade the connection associated with Handle to a tls connection if possible.

The upgrade is done in two phases: first the server is asked for permission to upgrade. Second, if the request is acknowledged, the upgrade to tls is performed.

Error responses from phase one will not affect the current encryption state of the connection. Those responses are:

tls_already_started
  The connection is already encrypted. The connection is not affected.

{response, ResponseFromServer}
  The upgrade was refused by the LDAP server. The ResponseFromServer is an atom delivered by the LDAP server explained in section 2.3 of rfc 2830. The connection is not affected, so it is still un-encrypted.

Errors in the second phase will however end the connection:

Error
  Any error responded from \text{ssl:connect/3}

The Timeout parameter is for the actual tls upgrade (phase 2) while the timeout in \text{eldap:open/2} is used for the initial negotiation about upgrade (phase 1).

simple_bind(Handle, Dn, Password) -> return_value()

Types:
  Handle = handle()
  Dn = string()
  Password = string()

Authenticate the connection using simple authentication.
add(Handle, Dn, [Attribute]) -> return_value()
Types:
   Handle = handle()
   Dn = string()
   Attribute = attribute()
Add an entry. The entry must not exist.

delete(Handle, Dn) -> return_value()
Types:
   Dn = string()
Delete an entry.

mod_add(Type, [Value]) -> modify_op()
Types:
   Type = string()
   Value = string()
Create an add modification operation.

mod_delete(Type, [Value]) -> modify_op()
Types:
   Type = string()
   Value = string()
Create a delete modification operation.

mod_replace(Type, [Value]) -> modify_op()
Types:
   Type = string()
   Value = string()
Create a replace modification operation.

modify(Handle, Dn, [ModifyOp]) -> return_value()
Types:
   Dn = string()
   ModifyOp = modify_op()
Modify an entry.
modify(Handle, "cn=Bill Valentine, ou=people, o=Example Org, dc=example, dc=com",
    [ldap:mod_replace("telephoneNumber", ["555 555 00"]),
     ldap:mod_add("description", ["LDAP Hacker"])]
)

modify_password(Handle, Dn, NewPasswd) -> return_value() | {ok, GenPasswd}
Types:
    Dn = string()
    NewPasswd = string()
Modify the password of a user. See modify_password/4.

modify_password(Handle, Dn, NewPasswd, OldPasswd) -> return_value() | {ok, GenPasswd}
Types:
    Dn = string()
    NewPasswd = string()
    OldPasswd = string()
    GenPasswd = string()
Modify the password of a user.
• Dn. The user to modify. Should be "" if the modify request is for the user of the LDAP session.
• NewPasswd. The new password to set. Should be "" if the server is to generate the password. In this case, the
  result will be {ok, GenPasswd}.
• OldPasswd. Sometimes required by server policy for a user to change their password. If not required, use
  modify_password/3.

modify_dn(Handle, Dn, NewRDN, DeleteOldRDN, NewSupDN) -> return_value()
Types:
    Dn = string()
    NewRDN = string()
    DeleteOldRDN = boolean()
    NewSupDN = string()
Modify the DN of an entry. DeleteOldRDN indicates whether the current RDN should be removed from the attribute
list after the operation. NewSupDN is the new parent that the RDN shall be moved to. If the old parent should remain
as parent, NewSupDN shall be "".

modify_dn(Handle, "cn=Bill Valentine, ou=people, o=Example Org, dc=example, dc=com",
          "cn=Bill Jr Valentine", true, ""
)

search(Handle, SearchOptions) -> {ok, #eldap_search_result{}} | {ok, {referral, referrals()}} | {error, Reason}
Types:
    SearchOptions = #eldap_search{} | [SearchOption]
    SearchOption = {base, string()} | {filter, filter()} | {scope, scope()} |
                   {attributes, [string()]} | {deref, dereference()} | {types_only,
                   boolean()} | {timeout, integer()}

6 | Ericsson AB. All Rights Reserved.: eldap
Search the directory with the supplied the SearchOptions. The base and filter options must be supplied. Default values: scope is `wholeSubtree()`, deref is `derefAlways()`, types_only is `false` and timeout is `0` (meaning infinity).

```erly
Filter = eldap:substrings("cn", [{any,"V"}]),
search(Handle, [{base, "dc=example, dc=com"}, {filter, Filter}, {attributes, ["cn"]}]),
```

The `timeout` option in the `SearchOptions` is for the ldap server, while the timeout in `eldap:open/2` is used for each individual request in the search operation.

- `baseObject() -> scope()`  
  Search baseobject only.

- `singleLevel() -> scope()`  
  Search the specified level only, i.e. do not recurse.

- `wholeSubtree() -> scope()`  
  Search the entire subtree.

- `neverDerefAliases() -> dereference()`  
  Never dereference aliases, treat aliases as entries.

- `derefAlways() -> dereference()`  
  Always dereference aliases.

- `derefInSearching() -> dereference()`  
  Dereference aliases only when searching.

- `derefFindingBaseObj() -> dereference()`  
  Dereference aliases only in finding the base.

- `present(Type) -> filter()`  
  Types:
  ```erly
  Type = string()
  ```
  Create a filter which filters on attribute type presence.

- `substrings(Type, [SubString]) -> filter()`  
  Types:
  ```erly
  Type = string()
  SubString = {StringPart, string()}
  StringPart = initial | any | final
  ```
  Create a filter which filters on substrings.

- `equalityMatch(Type, Value) -> filter()`  
  Types:
  ```erly
  Type = string()
  ```
Value = string()
Create a equality filter.

greaterOrEqual(Type, Value) -> filter()
Types:
  Type = string()
  Value = string()
Create a greater or equal filter.

lessOrEqual(Type, Value) -> filter()
Types:
  Type = string()
  Value = string()
Create a less or equal filter.

approxMatch(Type, Value) -> filter()
Types:
  Type = string()
  Value = string()
Create a approximation match filter.

extensibleMatch(MatchValue, OptionalAttrs) -> filter()
Types:
  MatchValue = string()
  OptionalAttrs = [Attr]
  Attr = {matchingRule,string()} | {type,string()} |
                     {dnAttributes,boolean()}
Creates an extensible match filter. For example,

eldap:extensibleMatch("Bar", [{type,"sn"}, {matchingRule,"caseExactMatch"}])

creates a filter which performs a caseExactMatch on the attribute sn and matches with the value "Bar". The default value of dnAttributes is false.

'and'([Filter]) -> filter()
Types:
  Filter = filter()
Creates a filter where all Filter must be true.

'or'([Filter]) -> filter()
Types:
  Filter = filter()
Create a filter where at least one of the Filter must be true.
'not'(Filter) -> filter()

Types:

\[\text{Filter} = \text{filter()}\]

Negate a filter.

\[\text{paged\_result\_control}(\text{PageSize}) \rightarrow \{\text{control, } "1.2.840.113556.1.4.319", \text{true, binary()}\}\]

Types:

\[\text{PageSize} = \text{positive\_integer()}\]

Paged results is an extension to the LDAP protocol specified by RFC2696

This function creates a control with the specified page size for use in search/3, for example:

\[
\begin{align*}
\text{Control} &= \text{ldap:paged\_result\_control}(50), \\
\{\text{ok, SearchResults}\} &= \text{search(Handle, \{base, "dc=example, dc=com"\}, \{Control\}),}
\end{align*}
\]

\[\text{paged\_result\_control}(\text{PageSize, Cookie}) \rightarrow \{\text{control, } "1.2.840.113556.1.4.319", \text{true, binary()}\}\]

Types:

\[\begin{align*}
\text{PageSize} &= \text{positive\_integer()} \\
\text{Cookie} &= \text{binary()}
\end{align*}\]

Paged results is an extension to the LDAP protocol specified by RFC2696

This function creates a control with the specified page size and cookie for use in search/3 to retrieve the next results page.

For example:

\[
\begin{align*}
\text{PageSize} &= 50, \\
\text{Control1} &= \text{ldap:paged\_result\_control}(\text{PageSize}), \\
\{\text{ok, SearchResults1}\} &= \text{search(Handle, \{base, "dc=example, dc=com"\}, \{Control\}),}
% \text{retrieve the returned cookie from the search results} \\
\text{Cookie1} &= \text{ldap:paged\_result\_cookie(SearchResults1)}, \\
\text{Control2} &= \text{ldap:paged\_result\_control}(\text{PageSize}, \text{Cookie1}), \\
\{\text{ok, SearchResults2}\} &= \text{ldap:search(Handle, \{base, "dc=example, dc=com"\}, \{Control\}),}
% \text{etc}
\end{align*}
\]

\[\text{paged\_result\_cookie(SearchResult)} \rightarrow \text{binary()}\]

Types:

\[\text{SearchResult} = \#\text{ldap\_search\_result}\]

Paged results is an extension to the LDAP protocol specified by RFC2696.

This function extracts the cookie returned from the server as a result of a paged search result.

If the returned cookie is the empty string "", then these search results represent the last in the series.

\[\text{info(Handle)} \rightarrow \text{connection\_info()}\]

Types:

\[\begin{align*}
\text{connection\_info()} &= \#\{\text{socket} := \text{Socket}, \text{socket\_type} := \text{tcp | ssl}\} \\
\text{Socket} &= \text{ssl:sslsocket()} \mid \text{gen\_tcp:socket()}
\end{align*}\]
Currently available information reveals the socket and the transport protocol, TCP or TLS (SSL), used by the LDAP connection.