6

Searching LDAP

In this chapter we will learn:

- Basics of LDAP Search
- LDAP Search using Filters
- Creating custom search filter

Searching LDAP for information is usually the most common operation performed against LDAP. A client application initiates a LDAP search by passing in a search criteria, the information that determines where to search and what to search for. Upon receiving the request, the LDAP server executes the search and returns all the entries that match the criteria.

LDAP Search Criteria

The LDAP search criteria are made up of three mandatory parameters: base, scope and filter and several optional parameters. Let us look at each of these parameters in detail.

Base Parameter

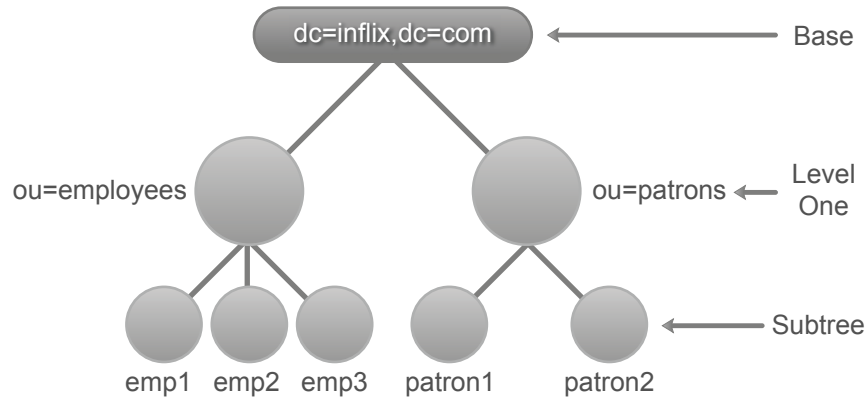
The base portion of the search is a Distinguished Name that identifies the branch of the tree that will be searched. For example, a base "`ou=patrons, dc=inflinx, dc=com`" indicates that the search will start in the Patron branch and moves downwards. It is also possible to specify an empty base which will result in searching the root DSE entry.

Scope Parameter

The scope parameter determines how deep, relative to the base, a LDAP search needs to be performed. LDAP protocol defines three possible search scopes: base, one level and subtree. FIGURE 6-1 illustrates the entries that get evaluated as part of the search with different scopes.

The base scope would restrict the search to the LDAP entry identified by the base parameter. No other entries will be included as part of the search. In our Library application schema, with a base DN `dc=inflinx,dc=com` and scope base, a search would just return the root organization entry.

FIGURE 6-1
Search Scopes



The one level scope indicates searching all the entries one level directly below the base. The base entry itself is not included in the search. So, with base `dc=inflinx,dc=com` and scope one level, a search for all entries would return employees and patrons organizational unit.

Finally, the subtree scope would include the base entry and all of its descendent entries in the search. This is the slowest and most expensive option among the three. In our Library example, a search with this scope and base `dc=inflinx, dc=com` would return all the entries.

Filter Parameter

In our Library application LDAP Server, let's say we want to find all the patrons that live in Midvale area. From the LDAP schema, we know that patron entries have the city attribute that holds the city name they live in. So this requirement essentially boils down to retrieving all entries that have the city attribute with value "Midvale". This is exactly what a search filter does. A search filter defines the characteristics that all the returning entries possess. Logically speaking, the filter gets applied to each entry in the set identified by base and scope. Only the entries that match the filter become part of the returned search results.

LDAP search filter is made up of three components: an attribute type, an operator and a value (or range of values) for the attribute. Depending on the operator, the value part can be optional. These components must always be enclosed inside a parenthesis:

```
Filter = (attributetype operator value)
```

With this information in hand, the search filter to locate all patrons living in Midvale would look like this:

```
(city=Midvale)
```

Now, let's say we want to find all the patrons that live in Midvale area and have an email address so that we can send them occasional Library events. The resulting search filter is essentially a combination of two filter items: an item that identifies patrons in Midvale city and an item that identifies patrons that have an email address. We have already seen the first item of the filter. Here is the other portion of the filter:

```
(mail=*)
```

The =* operator indicates the presence of an attribute. So the expression mail=* will return all entries that have a value in their mail attribute. The LDAP specification defines filter operators that can be used to combine multiple filters and create complex filters. Here is the format for combining the filters:

```
Filter = (operator filter1 filter2)
```

Notice the use of prefix notation where the operator is written before their operands for combining the two filters. Here is the required filter for our use case:

```
(&(city=Midvale)(mail=*))
```

LDAP specification defines a variety of search filter operators. TABLE 6-1 lists some of the commonly used operators.

Optional Parameters

In addition to the above three parameters, it is possible to include several optional parameters to control the search behavior. For example, the timelimit parameter indicates the time allowed to com-

TABLE 6-1

Search Filter Operators

Name	Symbol	Example	Description
Equality Filter	=	(sn=Smith)	Matches all the entries with last name Smith
Substring Filter	=, *	(sn=Smi*)	Matches all entries whose last name begins with Smi
Greater Than or Equals Filter	>=	(sn>=S*)	Matches all entries that are alphabetically greater than or equal to S.
Less Than or Equals Filter	<=	(sn<=S*)	Matches all entries that are alphabetically lower than or equals to S.
Presence Filter	=*	(objectClass=*)	Matches all entries that have the attribute objectClass. This is a popular expression used to retrieve all entries in LDAP
Approximate Filter	~=	(sn~=Smith)	Matches all entries whose last name is a variation of Smith. So this can return Smith and Snith
And Filter	&	(&(sn=Smith)(zip=84121))	Returns all Smith's living in the 84121 area
Or Filter	|	(|(sn=Smith)(sn=Lee))	Returns all entries with last name Smith or Lee
Not Filter	!	(!(sn=Smith))	Returns all entries whose last name is not Smith

plete the search. Similarly, the sizelimit parameter places an upper bound on the number of entries that can be returned as part of the result.

A very commonly used optional parameter involves providing a list of attribute names. By default the LDAP server returns all the attributes associated with entries found in the search. It however does not include operational attributes. Since version 3, LDAP servers can maintain attributes for each entry for purely administrative purposes. These attributes are referred to as operational attributes and are not part of the entry's objectClass. In order to get only a subset of attributes or retrieve operational attributes, we need to provide a list of attributes names in the search criteria.

Examples of Operational attributes include createTimeStamp that holds the time when the entry was created and pwdAccountLocked Time that records the time a user's account got locked.

Spring LDAP Filters

In the previous section we learned that LDAP search filters are very important for narrowing down the search and identifying entries.

LDAP Injection

LDAP injection is a technique where an attacker alters a LDAP query to run arbitrary LDAP statements against directory server. LDAP injection can result in unauthorized data access or modifications to LDAP tree. Applications that don't perform proper input validations or sanitize their input are prone to LDAP injection. This technique is similar to the popular SQL injection attack used against databases.

To better understand LDAP injection, consider a web application that uses LDAP for authentication. Such applications usually provide a web page that allows a user enter his user name and password. In order to verify that username and password match, the application would then construct an LDAP search

query that looks more or less like this:

```
(&(uid=USER_INPUT_UID)(password=USER_INPUT_PWD))
```

Let's assume that the application simply trusts the user input and doesn't perform any validation. Now if we enter the text jdoe)(&))(as user name and any random text as password, the resulting search query would look like this:

```
(&(uid=jdoe)(&))((password=randomstr))
```

If the username jdoe is a valid user id in LDAP, then regardless of the entered password, this query will always evaluate to true. This LDAP injection would allow an attacker by pass the authentication and get into the application. The LDAP Injection & Blind LDAP Injection article available at http://www.blackhat.com/presentations/bh-europe-08/Alonso-Parada/Whitepaper/bh-eu-08-alonso-parada-WP.pdf discusses in great detail about various LDAP injection techniques.

Preventing LDAP injection and any other injection techniques in general begin with proper input validation. It is important to sanitize the entered data and properly encode it before it is used in search filters.

However, creating LDAP filters dynamically can be tedious especially when trying to combine multiple filters. Making sure that all the braces are properly closed can be error prone. It is also important to properly escape special characters.

Spring LDAP provides several filter classes that make it easy to create and encode LDAP filters. All these filters implement the `Filter` interface and are part of the `org.springframework.ldap.filter` package. LISTING 6-1 shows the `Filter` API interface.

```
public interface Filter {
    String encode();
    StringBuffer encode(StringBuffer buf);
    boolean equals(Object o);
    int hashCode();
}
```

LISTING 6-1
Filter API Interface

The first `encode` method in the above interface returns a string representation of the filter. The second `encode` method accepts a `StringBuffer` as its parameter and appends the filter's string representation to it. This modified `StringBuffer` is returned to the caller.

The `Filter` interface hierarchy is shown in FIGURE 6-2. From the hierarch, you can see that `AbstractFilter` implements the `Filter` interface and acts as the root class for all other filter implementa-

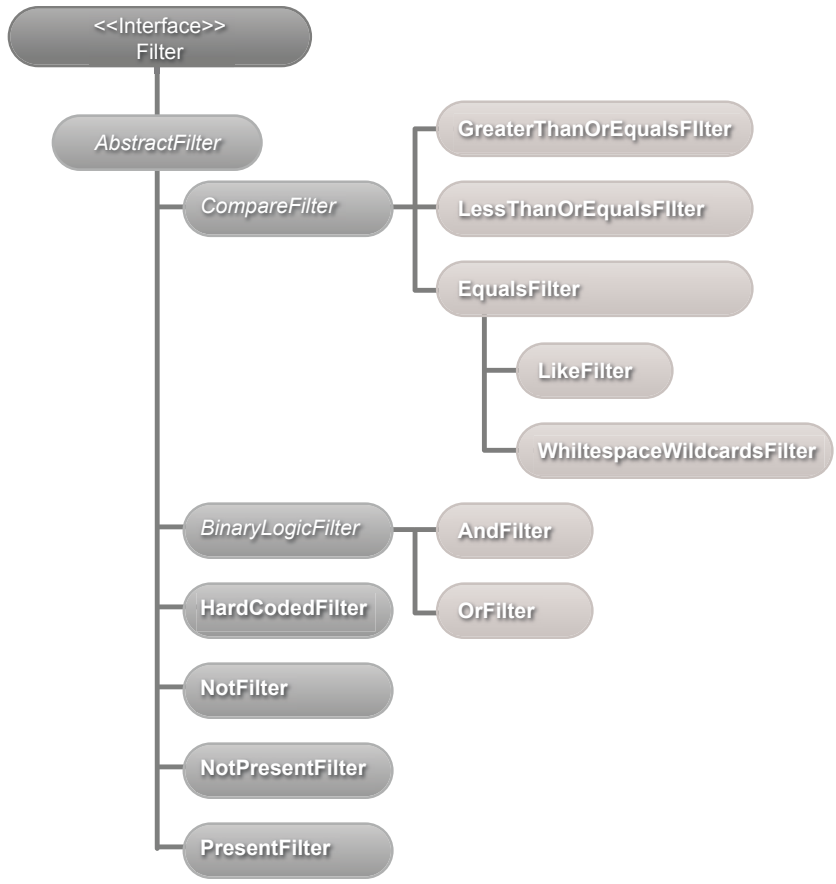


FIGURE 6-2
Filter Hierarchy

Most LDAP attribute values by default are case insensitive for searches.

tions. The `BinaryLogicalFilter` is the super class for binary logical operations such as AND and OR. The `CompareFilter` is super class for filters that compare values such as `EqualsFilter` and `LessThanOrEqualsFilter`.

In the coming sections we will look at each of the filters in FIGURE 6-2. Before we do that, let's create a reusable method that will help us test our filters. LISTING 6-2 shows the `searchAndPrintResults` method that uses the passed in `Filter` parameter and performs search using it. It then outputs the search results to the console. Notice that we will be searching the Patron branch of the LDAP tree.

LISTING 6-2

```
@Component("searchFilterDemo")
public class SearchFilterDemo {

    @Autowired
    @Qualifier("ldapTemplate")
    private SimpleLdapTemplate ldapTemplate;

    public void searchAndPrintResults(Filter filter)
    {
        List<String> results = ldapTemplate.search(
                "ou=patrons,dc=inflinx,dc=com", filter.encode(),
                new AbstractParameterizedContextMapper<String>()
                {
                    @Override
                    protected String doMapFromContext(
                            DirContextOperations context) {
                    return context.getStringAttribute("cn");
                        }
                });
        System.out.println("Results found in search: " +
                results.size());
        for(String commonName: results) {
            System.out.println(commonName);
        }
    }
}
```

Equals Filter

An `EqualsFilter` can be used to retrieve all entries that have the specified attribute and value. Let's say, we want to retrieve all patrons with the first name Jacob. To do this, we create a new instance of `EqualsFilter`.

```
EqualsFilter filter = new EqualsFilter("givenName", "Jacob");
```

The first parameter to the constructor is the attribute name and the second parameter is the attribute value. Invoking the `encode` method on this filter would result in the string (givenName=Jacob).

LISTING 6-3 shows the test case that invokes the `searchAndPrintResults` with the above `EqualsFilter` as parameter. The console output of the method is also shown in the listing. Notice that the results have patrons with first name jacob. That is because, the sn attribute, like most LDAP attributes is defined in the schema as being case insensitive.

LISTING 6-3

```
@Test
public void testEqualsFilter() {
    Filter filter = new EqualsFilter("givenName", "Jacob");
    searchFilterDemo.searchAndPrintResults(filter);
}

Results found in search: 2
Jacob Smith
jacob Brady
```

Like Filter

The Like Filter is useful for searching LDAP when only a partial value of an attribute is known. LDAP specification allows the usage of wildcard * to describe these partial values. Let's say we want

to retrieve all the users whose first name begins with 'Ja'. To do this we create a new instance of `LikeFilter` and pass in the wildcard substring as attribute value.

```
LikeFilter  filter = new LikeFilter("givenName", "Ja*");
```

Invoking the `encode` method on this filter would result in the string (givenName=Ja*). LISTING 6-4 shows the test case and the results of invoking the `searchAndPrintResults` method with the Like Filter.

LISTING 6-4

```
@Test
public void testLikeFilter() {
    Filter filter = new LikeFilter("givenName", "Ja*");
    searchFilterDemo.searchAndPrintResults(filter);
}

Results found in search: 3
Jacob Smith
Jason Brown
jacob Brady
```

The wildcard * in the substring is used to match zero or more characters. However, it is very important to understand that LDAP search filters do not support regular expressions. TABLE 6-2 lists few substring examples.

You might be wondering the necessity of a `LikeFilter` when we can accomplish the same filter expression by simply using the `EqualsFilter` like this:

```
EqualsFilter filter = new EqualsFiler("uid", "Ja*");
```

Using `EqualsFilter` in this scenario will not work because the `encode` method in `EqualsFilter` would consider the wildcard * in the Ja* as a special character and will properly escape it. Thus, the above filter when used for a search would result in all entries that have their first name starting with Ja*.

LDAP Substring	Description
(givenName=*son)	Matches all patrons whose first name ends with son.
(givenName=J*n)	Matches all patrons whose first name starts with J and ends with n.
(givenName=*a*)	Matches all patrons with first name containing the character a.
(givenName=J*s*n)	Matches patrons whose first name starts with J, contains character s and ends with n.

TABLE 6-2

LDAP Substring Examples

Presence Filter

Presence Filters are useful for retrieving LDAP entries that have at least one value in a given attribute. Consider our earlier scenario where we wanted to retrieve all the patrons that have an email address. To do this, we create a `PresentFilter` as shown below:

```
PresentFilter presentFilter = new PresentFilter(“email”);
```

Invoking `encode` method on the presentFilter instance will result in the string (email=*). LISTING 6-5 shows the test code and the result when the `searchAndPrintResults` method is invoked with the above presentFilter.

LISTING 6-5

```
@Test
public void testPresentFilter() {
    Filter filter = new PresentFilter(“mail”);
    searchFilterDemo.searchAndPrintResults(filter);
}

Results found in search: 97
Jacob Smith
Aaren Atp
Aarika Atpco
Aaron Atrc
Aartjan Aalders
Abagael Aasen
Abagail Abadines
……
……
```

Not Present Filter

Not Present Filters are used to retrieve entries that don't have a specified attribute. Attributes that do not have any value in an entry are considered to be Not Present. Now, let's say we want to retrieve all patrons that don't have an email address. To do this we create an instance of `NotPresentFilter` as shown below:

```
NotPresentFilter notPresentFilter = new
NotPresentFilter("email");
```

The encoded version of the notPresentFilter will result in the expression !(email=*). Running the `searchAndPrintResults` will result in the output shown in LISTING 6-6.

LISTING 6-6

```
@Test
public void testNotPresentFilter() {
    Filter filter = new NotPresentFilter("mail");
    searchFilterDemo.searchAndPrintResults(filter);
}

Results found in search: 5
null
Addons Achkar
Adeniyi Adamowicz
Adoree Aderhold
Adorne Adey
```

Not Filter

A `NotFilter` is useful for retrieving entries that do not match a given condition. In the Like Filter section, we looked at retrieving all entries that start with Ja. Now let's say we want to retrieve all entries that don't start with Ja. This is where a `NotFilter` comes into picture. Here is the code for accomplishing this requirement:

```
NotFilter notFilter = new NotFilter(new LikeFilter("givenName",
"Ja*"));
```

Encoding this filter would result in the string !(givenName=Ja*). As you can see, the `NotFilter` simply adds the negation symbol (!) to the filter passed into its constructor. Invoking the `searchAndShowResults` method will result in the output shown in LISTING 6-7.

LISTING 6-7

```
@Test
public void testNotFilter() {
    NotFilter notFilter = new NotFilter(
            new LikeFilter("givenName", "Ja*"));
    searchFilterDemo.searchAndPrintResults(notFilter);
}
Results found in search: 99
Aaren Atp
Aarika Atpco
Aaron Atrc
Aartjan Aalders
Abagael Aasen
Abagail Abadines
```

It is also possible to combine `NotFilter` and `PresentFilter` to create expressions that are equivalent to `NotPresentFilter`. Here is a new implementation that gets all the entries that don't have an email address:

```
NotFilter notFilter = new NotFilter(new PresentFilter("email"));
```

Greater Than Or Equals Filter

The `GreaterThanOrEqualsFilter` is useful for matching all entries that are lexicographically equal or higher than the given attribute value. For example, a search expression (givenName >= James) can be used to retrieve all entries with given name alphabetically after James, in addition to James. LISTING 6-8 shows this implementation along with the output results.

LISTING 6-8

```
@Test
public void testGreaterThanOrEqualsFilter() {
    Filter filter = new GreaterThanOrEqualsFilter("givenName",
                                "Jacob");
        searchFilterDemo.searchAndPrintResults(filter);
}

Results found in search: 3
Jacob Smith
jacob Brady
Jason Brown
```

Less Than Or Equals Filter

The `LessThanOrEqualsFilter` can be used to match entries that are lexicographically equal or lower than the given attribute. So the search expression (givenName<=James) will return all entries with first name alphabetically lower or equal to James. LISTING 6-9 shows the test code that invokes `searchAndPrintResults` implementation of this requirement along with the output.

LISTING 6-9

```
@Test
public void testLessThanOrEqualsFilter() {
    Filter filter = new LessThanOrEqualsFilter("givenName",
                                        "Jacob");
    searchFilterDemo.searchAndPrintResults(filter);
}

Results found in search: 100
Jacob Smith
Aaren Atp
Aarika Atpco
Aaron Atrc
Aartjan Aalders
Abagael Aasen
Abagail Abadines
Abahri Abazari
```

As mentioned before, you will notice that the search includes entries with first name James. The LDAP specification does not provide a less than (<) operator. However, it is possible to combine `Not-`

`Filter` with `GreaterThanOrEqualsFilter` to obtain "Less Than" functionality. Here is an implementation of this idea:

```
NotFilter lessThanFilter = new NotFilter(new GreaterThanOrEqualsF
ilter("givenName", "James"));
```

And Filter

The `AndFilter` is used to combine multiple search filter expressions to create complex search filters. The resulting filter will match entries that meet all the sub filter conditions. For example, the `AndFilter` is suitable for implementing an earlier requirement to get all the patrons that live in the Midvale Area and have an email address. The code below shows this implementation:

```
AndFilter andFilter = new AndFilter();
andFilter.and(new EqualsFilter("postalCode", "84047"));
andFilter.and(new PresentFilter("email"));
```

Invoking the encode method on this filter would result in (&(city=Midvale)(email=*)). LISTING 6-10 shows the test case that creates the `AndFilter` and calls the `searchAndPrintResults` method.

LISTING 6-10

```
@Test
public void testAndFilter() {
    AndFilter andFilter = new AndFilter();
    andFilter.and(new EqualsFilter("postalCode", "84047"));
    andFilter.and(new PresentFilter("mail"));
    searchFilterDemo.searchAndPrintResults(andFilter);
}

Results found in search: 1
Jacob Smith
```

Or Filter

Like the `AndFilter`, an `OrFilter` can be used to combine multiple search filters. However, the resulting filter will match entries that meet any of the sub filter conditions. Here is one implementation

of the `OrFilter`:

```
OrFilter orFilter = new OrFilter();
orFilter.add(new EqualsFilter("postalcode", "84047"));
orFilter.add(new EqualsFilter("postalcode", "84121"));
```

This `OrFilter` will retrieve all patrons that live in the either in 84047 or in 84121 postal codes. The encode method would return the expression (|(postalcode =84047)(postalcode=84121)). The test case for the `OrFilter` is shown in LISTING 6-11.

LISTING 6-11

```
@Test
public void testOrFilter() {
    OrFilter orFilter = new OrFilter();
    orFilter.or(new EqualsFilter("postalCode", "84047"));
    orFilter.or(new EqualsFilter("postalCode", "84121"));
    searchFilterDemo.searchAndPrintResults(orFilter);
}

Results found in search: 2
Jacob Smith
Adriane Admin-mtv
```

Hard Coded Filter

The `HardCodedFilter` is a convenience class that makes it easy to add static filter text while building search filters. Let's say we are writing an admin application that allows the administrator enter search expression in a text box. If we want to use this expression along with other filters for search, we can use `HardCodedFilter` as shown below:

```
AndFilter filter = new AndFilter();
filter.add(new HardCodedFilter(searchExpression));
filter.add(new EqualsFilter("givenName", "smith"));
```

In this code the `searchExpression` variable contains the user entered search expression. `HardCodedFilter` also comes very handy when the static portion of a search filter comes from a properties file

or a configuration file. It is important to remember that this filter does not encode the passed in text. So please use this with caution, especially when dealing with user input directly.

Whitespace Wildcards Filter

The `WhitespaceWildcardsFilter` is another convenience class that makes creation of sub string search filters easier. Like its super class `EqualsFilter`, this class takes an attribute name and a value. However, as the name suggests, it converts all whitespaces in the attribute value to wildcards. Consider the following example:

```
WhitespaceWildcardsFilter filter = new
WhitespaceWildcardsFilter("cn", "John Will");
```

The above filter would result in the following expression: (cn=*John*Will*). This filter can be useful while developing search and white pages applications.

Creating Custom Filters

Even though the filter classes provided by Spring LDAP are sufficient in most cases, there might be scenarios where the current set is not sufficient. Thankfully, Spring LDAP has made it easy to create new filter classes. In this section we will look at creating a custom Approximate Filter.

Approximate Filters are used to retrieve entries with attribute values approximately equal to the specified value. Approximate expressions are created using the ~= operator. So a filter (givenName ~= Adeli) will match entries with first name such as Adel or Adele. The approximate filter is useful in search applications when the actual spelling of the value is not known to the user at the time of the search. The algorithm implementation to find phonetically similar values varies from server to server.

Spring LDAP does not provide any out of the box class to create an Approximate Filter. In LISTING 6-12 we create an implementation of this filter. Notice that the `ApproximateFilter` class extends the `AbstractFilter`. The constructor is defined to accept the attribute type and the attribute value. In the `encode` method we construct the filter expression by concatenating the attribute type, operator and the value.

LISTING 6-12

```
private class ApproximateFilter extends AbstractFilter {
    private static final String APPROXIMATE_SIGN = "~=";

    private String attribute;
    private String value;

    public ApproximateFilter(String attribute, String value)
    {
        this.attribute = attribute;
        this.value = value;
    }

    @Override
    public StringBuffer encode(StringBuffer buff) {
        buff.append('(');
        buff.append(attribute).append(APPROXIMATE_SIGN).
                                    append(value);
        buff.append(')');
            return buff;
        }
}
```

LISTING 6-13 shows the test code for running `searchAndPrintResults` method with `ApproximateFilter` class.

LISTING 6-13

```
@Test
public void testApproximateFilter() {
        ApproximateFilter approx =
            new ApproximateFilter("givenName", "Adeli");
        searchFilterDemo.searchAndPrintResults(approx);
}
```

Here is the output of running the test case.

```
Results found in search: 6
Adel Acker
Adela Acklin
Adele Acres
Adelia Actionteam
Adella Adamczyk
Adelle Adamkowski
```

Handling Special Characters

There will be times where you might need to construct search filters with characters such as '(' or a '*' that have special meaning in LDAP. To successfully execute these filters, it is important to properly escape the special characters. Escaping is done using the format \xx where xx denotes the hexadecimal representation of the character. TABLE 6-3 lists all the special characters along with their escape values.

TABLE 6-3
Special Characters and Escape Values

Special Character	Escape Value
(	\28
)	\29
*	\2a
\	\5c
/	\2f

In addition to the above characters, if any of these characters ",", "=", "+", "<", ">", "#", ";" are used in a DN, those characters also need to be properly escaped.