
LPI exam 301 prep, Topic 301: Concepts, architecture, and design

Senior Level Linux Professional (LPIC-3)

Skill Level: Intermediate

[Sean A. Walberg](mailto:sean@ertw.com) (sean@ertw.com)
Senior Network Engineer

23 Oct 2007

In this tutorial, Sean Walberg helps you prepare to take the Linux Professional Institute® Senior Level Linux Professional (LPIC-3) exam. In this first in a series of six tutorials, Sean introduces you to Lightweight Directory Access Protocol (LDAP) concepts, architecture, and design. By the end of this tutorial, you will know about LDAP concepts and architecture, directory design, and schemas.

Section 1. Before you start

Learn what these tutorials can teach you and how you can get the most from them.

About this series

The [Linux Professional Institute](#) (LPI) certifies Linux system administrators at three levels: *junior level* (also called "certification level 1"), *advanced level* (also called "certification level 2"), and *senior level* (also called "certification level 3"). To attain certification level 1, you must pass exams 101 and 102. To attain certification level 2, you must pass exams 201 and 202. To attain certification level 3, you must have an active advanced-level certification and pass exam 301 ("core"). You may also pass additional specialty exams at the senior level.

developerWorks offers tutorials to help you prepare for the five junior, advanced, and senior certification exams. Each exam covers several topics, and each topic has a corresponding self-study tutorial on developerWorks. Table 1 lists the six topics and corresponding developerWorks tutorials for LPI exam 301.

Table 1. LPI exam 301: Tutorials and topics

LPI exam 301 topic	developerWorks tutorial	Tutorial summary
Topic 301	LPI exam 301 prep: Concepts, architecture, and design	(This tutorial.) Learn about LDAP concepts and architecture, learn how to design and implement an LDAP directory, and learn about schemas. See the detailed objectives below.
Topic 302	LPI exam 301 prep: Installation and development	Coming soon.
Topic 303	LPI exam 301 prep: Configuration	Coming soon.
Topic 304	LPI exam 301 prep: Usage	Coming soon.
Topic 305	LPI exam 301 prep: Integration and migration	Coming soon.
Topic 306	LPI exam 301 prep: Capacity planning	Coming soon.

To pass exam 301 (and attain certification level 3), you should:

- Have several years experience with installing and maintaining Linux® on a number of computers for various purposes.
- Have integration experience with diverse technologies and operating systems.
- Have professional experience as, or training for, an enterprise-level Linux professional (including having experience as a part of another role).
- Know advanced and enterprise levels of Linux administration including installation, management, security, troubleshooting, and maintenance.
- Be able to use open source tools to measure capacity planning and troubleshoot resource problems.
- Have professional experience using LDAP to integrate with UNIX® and Microsoft® Windows® services, including Samba, Pluggable Authentication Modules (PAM), e-mail, and Microsoft Active Directory directory service.
- Be able to plan, architect, design, build, and implement a full environment using Samba and LDAP as well as measure the capacity planning and security of the services.
- Be able to create scripts in Bash or Perl or have knowledge of at least one system-programming language (such as C).

The Linux Professional Institute does not endorse any third-party exam preparation material or techniques in particular.

About this tutorial

Welcome to "Concepts, architecture, and design," the first of six tutorials designed to prepare you for LPI exam 301. In this tutorial, you learn about LDAP concepts and architecture, how to design and implement an LDAP directory, and about schemas.

This tutorial is organized according to the [LPI objectives for this topic](#). Very roughly, expect more questions on the exam for objectives with higher weights.

Objectives

Table 2 provides the detailed objectives for this tutorial.

Table 2. Concepts, architecture, and design: Exam objectives covered in this tutorial

LPI exam objective	Objective weight	Objective summary
301.1 Concepts and architecture	3	Be familiar with LDAP and X.500 concepts.
301.2 Directory design	2	Design and implement an LDAP directory while planning an appropriate Directory Information Tree to avoid redundancy. You should have an understanding of the types of data that are appropriate for storage in an LDAP directory.
301.3 Schemas	3	Be familiar with schema concepts and the base schema files included with an OpenLDAP installation.

Prerequisites

To get the most from this tutorial, you should have an advanced knowledge of Linux and a working Linux system on which to practice the commands covered.

If your fundamental Linux skills are a bit rusty, you may want to first review the [tutorials for the LPIC-1 and LPIC-2 exams](#).

Different versions of a program may format output differently, so your results may not look exactly like the listings and figures in this tutorial.

System requirements

To follow along with the examples in these tutorials, you need a Linux workstation with the OpenLDAP package and support for PAM. Most modern distributions meet

these requirements.

Section 2. Concepts and architecture

This section covers material for topic 301.1 for the Senior Level Linux Professional (LPIC-3) exam 301. This topic has a weight of 3.

In this section, learn about:

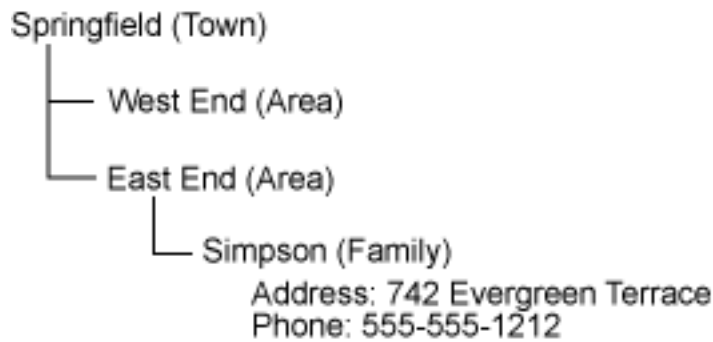
- LDAP and X.500 technical specification
- Attribute definitions
- Directory namespaces
- Distinguished names
- LDAP Data Interchange Format
- Meta-directories
- Changetype operations

Most of the LPIC-3 exam focuses on the use of the Lightweight Directory Access Protocol (LDAP). Accordingly, the first objective involves understanding what LDAP is, what it does, and some of the basic terminology behind the concept. When you understand this, you will be able to move on to designing your directory and integrating your applications with the directory.

LDAP, what is it?

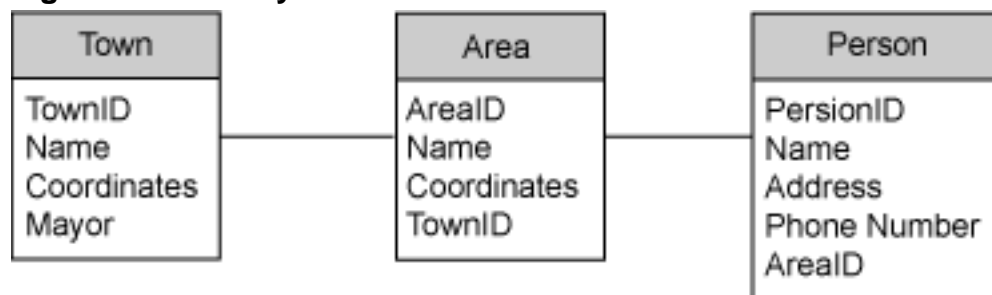
Before talking about LDAP, let's review the concept of directories. The classic example of a directory is the phone book, where people are listed in alphabetical order along with their phone numbers and addresses. Each person (or family) represents an object, and the phone number and address are attributes of that object. Though not always obvious at a glance, some objects are businesses instead of people, and these may include fax numbers or hours of operation.

Unlike its printed counterpart, a computer directory is hierarchical in nature, allowing for objects to be placed under other objects to indicate a parent-child relationship. For instance, the phone directory could be extended to have objects representing areas of the city, each with the people and business objects falling into their respective area objects. These area objects would then fall under a city object, which might further fall under a state or province object, and so forth, much like Figure 1. This would make a printed copy much harder to use because you would need to know the name and geographical location, but computers are made to sort information and search various parts of the directory, so this is not a problem.

Figure 1. A sample directory

Looking at Figure 1, knowing where the Simpson's record is tells you more than just the address and phone number. You also know they are in the East end in the town of Springfield. This structure is called a tree. Here, the root of the tree is the Springfield object, and the various objects represent further levels of branching.

This directory-based approach to storing data is quite different than the relational databases that you may be familiar with. To compare the two models, Figure 2 shows what the telephone data might look like if modeled as a relational database.

Figure 2. Directory data modeled in relational form

In the relational model, each type of data is a separate table that allows different types of information to be held. Each table also has a link to its parent table so that the relationships between the objects can be held. Note that the tables would have to be altered to add more information fields.

Remember that nothing about the directory model places any restrictions on how the data may be stored on disk. In fact, OpenLDAP supports many back ends including flat files and Structured Query Language (SQL) databases. The mechanics of laying out the tables on disk are largely hidden from you. For instance, Active Directory provides an LDAP interface to its proprietary back end.

The history of LDAP

LDAP was conceived in Request for Comments (RFC) 1487 as a lightweight way to access an X.500 directory instead of the more complex Directory Access Protocol. (See the [Resources](#) section for links to this and related RFCs.) X.500 is a standard (and a family of standards) from the International Telecommunication Union (ITU, formerly the CCITT) that specifies how directories are to be implemented. You may be familiar with the X.509 standard that forms the core of most Public Key Infrastructure (PKI) and Secure Sockets Layer (SSL) certificates. LDAP has since

evolved to version 3 and is defined in RFC 4511.

Connecting to an X.500 database initially required the use of the Open Systems Interconnection (OSI) suite of protocols and, in true ITU fashion, required understanding of thick stacks of protocol documentation. LDAP allowed Internet Protocol (IP)-based networks to connect to the same directory with far fewer development cycles than using OSI protocols. Eventually the popularity of IP networks led to the creation of LDAP servers that support only as many X.500 concepts as necessary.

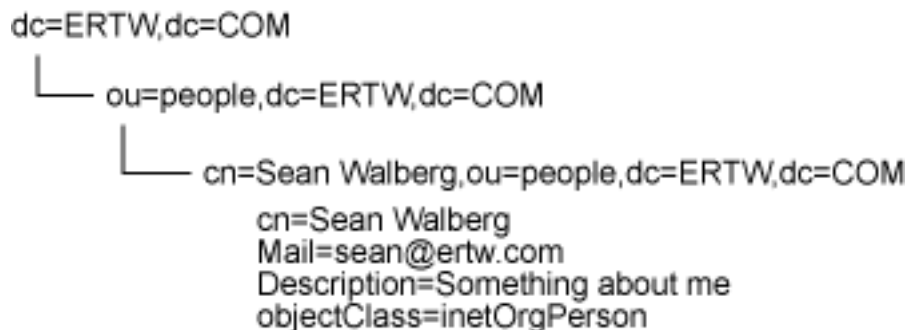
Despite the triumph of LDAP and IP over X.500 and OSI, the underlying organization of the directory data is still X.500-ish. Concepts that you will learn over the course of this tutorial, such as Distinguished Names and Object Identifiers, are brought up from X.500.

X.500 was intended as a way to create a global directory system, mostly to assist with the X.400 series of standards for e-mail. LDAP can be used as a global directory with some effort, but it is mostly used within an enterprise.

A closer look at naming and attributes

In the LDAP world, names are important. Names let you access and search records, and often the name gives an indication of where the record is within the LDAP tree. Figure 3 shows a typical LDAP tree.

Figure 3. A typical LDAP tree showing a user



At the top, or *root*, of the tree is an entity called `dc=ertw,dc=com`. The `dc` is short for *domain component*. Because `ertw` is under the `.com` top-level domain, the two are separated into two different units. Components of a name are concatenated with a comma when using the X.500 nomenclature, with the new components being added to the left. Nothing technically prevents you from referring to the root as `dc=ertw.com`, though in the interest of future interoperability it is best to have the domain components separate (in fact, RFC 2247 recommends the separate domain components).

`dc=ertw,dc=com` is a way to uniquely identify that entity in the tree. In X.500 parlance, this is called the *distinguished name*, or the *DN*. The DN is much like a primary key in the relational-database world because there can be only one entity with a given DN within the tree. The DN of the topmost entry is called the *Root DN*.

Under the root DN is an object with the DN of `ou=people,dc=ertw,dc=com`. `ou` means *organizational unit*, and you can be sure it falls under the root DN because the `ou` appears immediately to the left of the root DN. You can also call `ou=people` the *relative distinguished name*, or *RDN*, because it is unique within its level. Put in recursive terms, the DN of an entity is the entity's RDN plus the DN of the parent. Most LDAP browsers show only the RDN because it eliminates redundancy.

Moving down the tree to `cn=Sean Walberg,ou=people,dc=ertw,dc=com`, you find the record for a person. `cn` means *common name*. For the first time, though, a record has some additional information in the form of *attributes*. Attributes provide additional information about the entity. In fact, you'll see the leftmost component of the DN is duplicated; in this case, it's the `cn` attribute. Put another way, the RDN of an entity is composed of one (or more) attributes of the entity.

While `mail` and `description` are easy enough to understand, `objectClass` is not as obvious. An object class is a group of attributes that correspond to a particular entity type. One object class may contain attributes for people and another for UNIX accounts. By applying the two object classes to an entity, both sets of attributes are available to be stored.

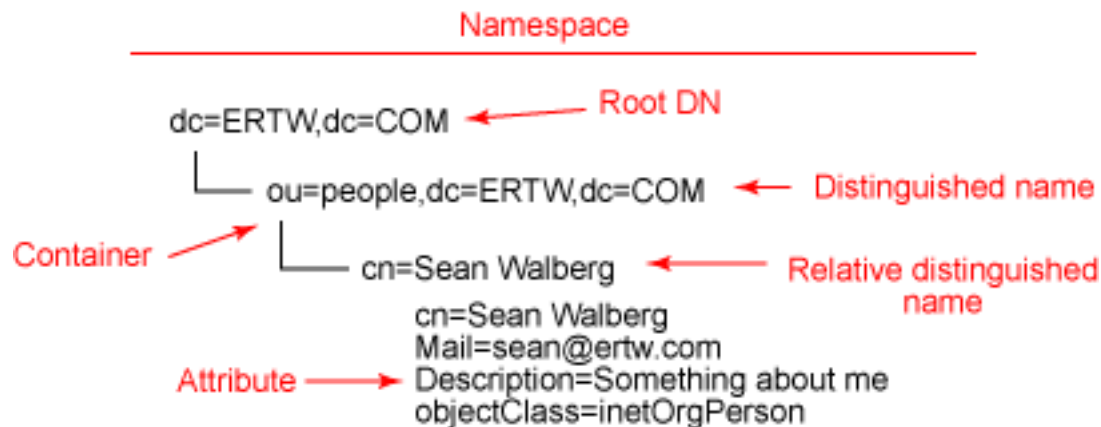
Each object class is assigned an object identifier (OID) that uniquely identifies it. The object class also specifies the attributes, and which ones are mandatory and which are optional. Mandatory attributes must have some data for the entity to be saved. The object class also identifies the type of data held and whether multiple attributes of the same name are allowed. For instance, a person might have only one employee number but multiple first names (for example, Bob, Robert, and Rob).

The bottom-level objects are not the only ones to have object classes associated with them. These objects, called *containers*, also have object classes and attributes. The `people` `ou` is of type `organizationalUnit` and has a `description` attribute along with `ou=people` to create the RDN. The root of the tree is of type `dcObject` and `organization`. Knowing which object classes to assign an object depends on what is being held in the object and under it. Refer to the [Schemas](#) section for more details.

The root DN also defines the *namespace* of the tree or, to be more technical, the *Directory Information Tree* (DIT). Something ending in `dc=ibm,dc=com` would fall outside of the namespace from [Figure 3](#), whereas the record for Sean Walberg falls within the namespace. With that in mind, though, it is possible that one LDAP server contains multiple namespaces. A somewhat abstract item called the *Root DSE* contains the information about all the namespaces available on the server. DSE means the DSA-Specific Entry, and DSA means Directory System Agent (that is, the LDAP server).

[Figure 4](#) summarizes the terminology associated with the LDAP tree.

Figure 4. Summary of LDAP terminology



Finally, an LDAP tree can be synchronized with other trees or data sources. For instance, one branch of the tree could come from a security system, another from a customer database, and the rest could be stored in the LDAP server. This is called a *meta-directory* and is intended to be a single source of data for applications such as single sign-on.

The LDIF file

Data can get into an LDAP server in one of two ways. Either it can be loaded in over the network, using the LDAP protocol, or it can be imported from the server through a file in the *LDAP Data Interchange Format* (LDIF). LDIF can be used at any time, such as to create the initial tree, and to perform a bulk add or modify of the data some time later. The output of a search can also be in LDIF for easy parsing or import to another server. The full specification for LDIF is in RFC 2849 (see [Resources](#) for a link).

Adding records

The LDIF that generated the tree from Figure 3 is shown in Listing 1.

Listing 1. A simple LDIF file to populate a tree

```
# This is a comment
dn: dc=ertw,dc=com
dc: ertw
description: This is my company
  the description continues on the next line
  indented by one space
objectClass: dcObject
objectClass: organization
o: ERTW.COM

dn: ou=people,dc=ertw,dc=com
ou: people
description: Container for users
objectclass: organizationalunit

dn: cn=Sean Walberg,ou=people,dc=ertw,dc=com
objectclass: inetOrgPerson
cn: Sean Walberg
cn: Sean A. Walberg
sn: Walberg
homephone: 555-111-2222
```



```
mail: sean@ertw.com
description: Watch out for this guy
ou: Engineering
```

Before delving into the details of the LDIF file, note that the attribute names are case insensitive. That is, `objectclass` is the same as both `objectClass` and `OBJECTCLASS`. Many people choose to capitalize the first letter of each word except the first, such as `objectClass`, `homePhone`, and `thisIsAreallyLongAttribute`.

The first line of the LDIF shows a UNIX-style comment, which is prefixed by a hash sign (#), otherwise known as a pound sign or an octothorpe. LDIF is a standard ASCII file and can be edited by humans, so comments can be helpful. Comments are ignored by the LDAP server, though.

Records in the LDIF file are separated by a blank line and contain a list of attributes and values separated by a colon (:). Records begin with the `dn` attribute, which identifies the distinguished name of the record. [Figure 1](#), therefore, shows three records: the `dc=ertw, ou=people, and cn=Sean Walberg` RDNs, respectively.

Choosing attributes

The attribute names may be confusing at this point. How do you choose which object class to assign a record? How do you find out which attributes are available? How do you know that `o` stands for organization?

To put it very simply, the answers to all of these questions lie in the schema, which is covered in [Schemas](#). The schema provides a description of which attributes mean what. The schema also maps attributes into object classes. Adding an object class to a record allows you to use the attributes that fall within it.

The final piece of the puzzle is to understand how the LDAP tree is to be used. If you're going to be authenticating UNIX accounts against the tree, your users had better have an object class that gives them the same `userid` attribute that your system is looking for.

Looking back at [Figure 1](#), you can see the first record defined is the root of the tree. The distinguished name comes first. Next comes a list of all the attributes and values, separated by a colon. Colons within the value do not need any special treatment. The LDAP tools understand that the first colon separates the attribute from the value. If you need to define two values for an attribute, then simply list them as two separate lines. For example, the root object defines two object classes.

Each record must define at least one object class. The object class, in turn, may require that certain attributes be present. In the case of the root object, the `dcObject` object class requires that a *domain component*, or `dc`, be defined, and the `organization` object class requires that an *organization* attribute, or `o`, be defined. Because an object must have an attribute and value corresponding to the RDN, the `dcObject` object class is required to import the `dc` attribute. Defining an `o` attribute is not required to create a valid record.

A `description` is also used on the root object to describe the company. The purpose here is to demonstrate the comment format. If your value needs to span multiple lines, start each new line with a leading space instead of a value. Remember that specifying multiple `attribute: value` pairs defines multiple instances of the attribute.

The second record in [Figure 1](#) defines an `organizationalUnit`, which is a container for people objects in this case. The third defines a user of type `inetOrgPerson`, which provides common attributes for defining people within an organization. Note that two `cn` attributes are defined; one is also used in the DN of the record. The second, with the middle initial, will help for searching, but it is the first that is required to satisfy the condition that the RDN be defined.

In the user record there is also an `ou` that does not correspond to the `organizationalUnit` the user is in. The container the user object belongs to can always be found by parsing the DN. This `ou` attribute refers to something defined by the user, in this case a department. No referential integrity is imposed by the server, though the application may be looking for a valid DN such as `ou=Engineering,ou=Groups,dc=ertw,dc=com`.

The only other restriction placed on LDIF files that add records is that the tree must be built in order, from the root. [Figure 1](#) shows the root object being built, then an `ou`, then a user within that `ou`. Now that the structure is built, users can be added directly to the `people` container, but if a new container is to be used, it must be created first.

The LDIF behind adding objects is quite easy. The format gets more complex when objects must be changed or deleted. LDIF defines a `changetype`, which can be one of the following:

- `add` adds an item (default).
- `delete` deletes the item specified by the DN.
- `modrdn` renames the specified object within the current container, or moves the object to another part of the tree.
- `moddn` is synonymous with `modrdn`.
- `modify` makes changes to attributes within the current DN.

Deleting users

Deleting an item is the simplest case, only requiring the `dn` and `changetype`. Listing 2 shows a user being deleted.

Listing 2. Deleting a user with LDIF

```
dn: cn=Fred Smith,ou=people,dc=ertw,dc=com
changetype: delete
```

Manipulating the DN

Manipulating the DN of the object is slightly more complex. Despite the fact that there are two commands, `moddn` and `modrdn`, they do the same thing! The operation consists of three separate parts:

1. Specify the new RDN (leftmost component of the DN).
2. Determine if the old RDN should be replaced by the new RDN within the record, or if it should be left.
3. Optionally, move the record to a new part of the tree by specifying a new parent DN.

Consider Jane Smith, who changes her name to Jane Doe. The first thing to do is change her `cn` attribute to reflect the name change. Because the new name is the primary way she wishes to be referred to, and the common name forms part of the DN, the `moddn` operation is appropriate. (If the common name weren't part of the DN, this would be an attribute change, which is covered in the next section.) The second choice is to determine if the `cn: Jane Smith` should stay in addition to `cn: Jane Doe`, which allows people to search for her under either name. Listing 3 shows the LDIF that performs the change.

Listing 3. LDIF to change a user's RDN

```
# Specify the record to operate on
dn: cn=Jane Smith,ou=people,dc=ertw,dc=com
changetype: moddn
# Specify the new RDN, including the attribute
newrdn: cn=Jane Doe
# Should the old RDN (cn=Jane Smith) be deleted? 1/0, Default = 1 (yes)
deleteoldrdn: 0
```

Listing 3 begins by identifying Jane's record, then the `moddn` operator. The new RDN is specified, continuing to use a common name type but with the new name. Finally, `deleteoldrdn` directs the server to keep the old name. Note that while `newrdn` is the only necessary option to the `moddn` `changetype`, if you omit `deleteoldrdn`, the action is to delete the old RDN. According to RFC 2849, `deleteoldrdn` is a required element.

Should the new Mrs. Jane Doe be sent to a new part of the tree, such as a move to `ou=managers,dc=ertw,dc=com`, the LDIF must specify the new part of the tree somehow, such as in Listing 4.

Listing 4. Moving a record to a new part of the tree

```
dn: cn=Jane Doe,ou=people,dc=ertw,dc=com
changetype: modrdn
newrdn: cn=Jane Doe
deleteoldrdn: 0
newsuperior: ou=managers,dc=ertw,dc=com
```

Curiously, a new RDN must be specified even though it is identical to the old one, and the OpenLDAP parser now requires that `deleteoldrdn` is present despite it being meaningless when the RDN stays the same. `newsuperior` follows, which is the DN of the new parent in the tree.

One final note on the `modrdn` operation is that the order matters, unlike most other LDIF formats. After the `changetype` comes the `newrdn`, followed by `deleteoldrdn`, and, optionally, `newsuperior`.

Modifying attributes

The final `changetype` is `modify`, which is used to modify attributes of a record. Based on the earlier discussion of `moddn`, it should be clear that `modify` does not apply to the DN or the RDN of a record.

Listing 5 shows several modifications made to a single record.

Listing 5. Modifying a record through LDIF

```
dn: cn=Sean Walberg,dc=ertw,dc=com
changetype: modify
replace: homePhone
homePhone: 555-222-3333
-
changetype: modify
add: title
title: network guy
-
changetype: modify
delete: mail
-
```

The LDIF for the `modify` operation looks similar to the others. It begins with the DN of the record, then the `changetype`. After that comes either `replace:`, `add:`, or `delete:`, followed by the attribute. For `delete`, this is enough information. The others require the attribute:value pair. Each change is followed by a dash (-) on a blank line, including the final change.

LDIF has an easy-to-read format, both for humans and computers. For bulk import and export of data, LDIF is a useful tool.

Section 3. Directory design

This section covers material for topic 301.2 for the Senior Level Linux Professional (LPIC-3) exam 301. The topic has a weight of 2.

In this section, learn about:

- Defining LDAP directory content
- Directory organization
- How to plan appropriate Directory Information Trees

Determining if LDAP is appropriate

Like any other tool, LDAP is not appropriate for every solution. Before choosing LDAP, you must ask yourself some questions:

- How often will changes be made, and what kind of changes are they?
- What will be using the data?
- What kind of queries will be made against the data?
- Is the information hierarchical in nature?

LDAP databases are geared toward read-intensive operations. People may only change personal information a few times a year, but we can expect to look up attributes far more than that, such as resolving the `UserID` owning a file to a printable name when looking through a directory. LDAP data tends to be heavily indexed, which means every change to the underlying data requires multiple changes to the indexes that help the server find the data later. LDAP also doesn't offer a means to make mass updates to the tree, other than performing a search and then modifying each individual DN.

The LDAP specifications do not define transactions, which are common in relational databases. Transactions ensure that all the operations within the transaction succeed, or else the data is rolled back to the pre-transaction state (individual servers may implement this, but it is not a requirement). These limitations on updating data and the lack of transactions make LDAP a poor choice for bank transactions.

Determining the user, or the consumer, of the data is also important. If none of the consumers speak LDAP, then LDAP may not be a good fit. To LDAP's credit, it is an extremely simple protocol to implement and is available for most languages on most platforms. With a mere handful of available operations defined, it can be integrated into existing applications with ease.

LDAP provides search functionality, but with nowhere near the level of a relational database. The LDAP server may store the underlying data using SQL, but you as the user are abstracted from this and cannot make use of it. Thus, you are limited to the search filters that are supported by your LDAP server. These filters will be investigated in more detail in later articles in this series, but for now understand that the filters are just that -- filters. You can perform some powerful searches, such as "show me all the employees who live in Washington and those that live in Texas who are over 40." Statements equivalent to the SQL `GROUP BY` are not available, though. LDAP is best suited for look-up style operations, such as "show me the

username of the person with UID 4131," and "give me the names of everyone working for Jim Smith."

Finally, the information you are trying to store should lend itself to hierarchical storage. You could store a flat list of data in an LDAP server, but it would probably be a waste of resources.

When you can answer these questions and have determined that LDAP is the correct solution, it is time to design the directory tree.

Organizing your tree

The foremost idea to keep in your mind is that reorganization of the tree is undesirable. The goal is to break down your objects such that each branch of the tree holds objects of a similar type, but the chances of an object having to be moved is low. The reasons for this are twofold. One is that it's just a hassle to do. The second is that a move changes the DN of the object, and then you have to update all the objects that reference the moved object.

The root DN

The root of your tree should be something that represents your company. RFC 2247 calls for the use of the `dc` attribute and the familiar `dc=example,dc=com` format to map the company's primary domain into a DN. Your company probably has many Internet domains but has one that is preferred. It is also common to choose something under the `local` top-level domain (TLD), which doesn't currently exist on the Internet. Microsoft has long suggested using this TLD for their Active Directory implementations despite it not being a reserved TLD.

If you don't want to use a domain name in your root DN, you can simply have an object with `objectclass: organization`, which gives a root DN of `o=My Corporation`.

Filling in the structure

Deciding what goes at the next level of the DIT is difficult. It is often tempting to describe the company's organizational structure as a series of nested `organizationalUnit` objects, but companies are constantly reorganizing, which breaks the first rule. Instead, consider using attributes to store this information instead.

Depending on your use of the LDAP server and how you choose to name objects, you may want to keep all users in one tree or separate them according to role. For instance, you can create one container for employees and one for customers, or you can group them all into a single container. The choice depends on your applications and how you plan on managing the tree. If the sales department takes care of managing the customer list, and the systems administrators take care of the staff, it might be better to create two containers. Figure 5 shows a DIT that has been broken down into customers and users.

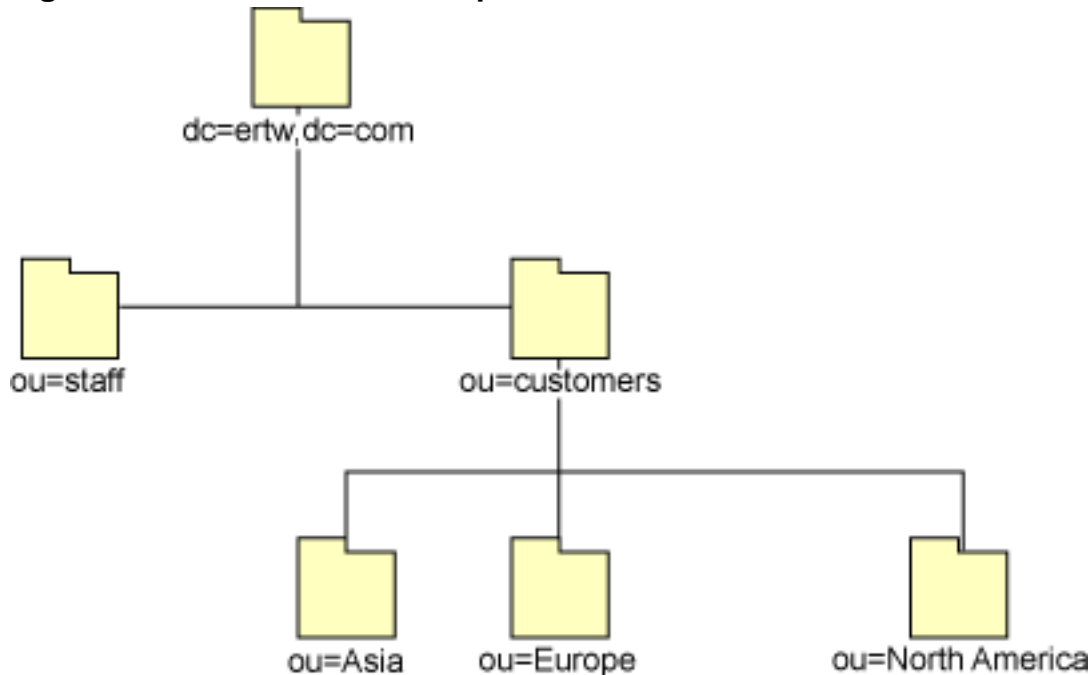
Figure 5. An LDAP tree with separate branches for users and customers

Figure 5 shows one grouping for staff and one for customers. All staff reside in the same organizational unit (OU), but customers are broken up by region. In this situation, this allows different people to manage their own region's customers.

If you plan on using LDAP for authentication of UNIX resources, you must then decide where to store that information. User accounts have already been taken care of earlier, but you will need to store groups, and possibly other maps such as hosts, services, networks, and aliases. Which of these you store in LDAP, and which you leave as local files depends on you and if you need to be able to update them centrally.

The simplest case is to store each map in a separate `organizationalUnit`. You will find that when configuring your UNIX system to read this information, you need to specify the DN of the container and any filters. If you store groups in the user OU, you will have to write some filters. You also may have to adjust the filters if you ever make any structural changes to the staff OU.

Determining the object classes

So far the discussion has centered on the layout of the LDAP tree with the basic goal being to avoid having to rename objects in the future. After you decide upon the tree, you must then decide which object classes to use. It is certainly possible to assign different object classes to objects under the same branch of the tree, but this will almost certainly lead to maintenance problems in the future.

To add more stress to the situation, it's not always possible to add more object classes to an object if you make a mistake. LDAP schemas define two types of object classes: *structural* and *auxiliary*. Structural object classes usually inherit properties from other object classes in a chain that ends up at an object class called `top`. Structural object classes can be said to define the object's identity, while

auxiliary object classes are there to add attributes. `organizationalUnit` is a structural object class, as is `inetOrgPerson`. Going back to [Listing 1](#), the top-level entry had two object classes: `dcObject` and `organization`. `organization` is the structural object class. `dcObject` plays the auxiliary role by defining the `dc` attribute

The part that can cause problems is that an entry can only have one structural object class. Sometimes you see `inetOrgPerson`, `organizationalPerson`, `person`, and `top` in the same record, but they are all part of the same inheritance tree. `inetOrgPerson` and `account` are both structural, are not in the same inheritance tree, and therefore can't be used together. Some LDAP servers permit this, but eventually this behavior may change and cause problems.

There is a third type of object class called **abstract**. It is much like structural except that it must be inherited to be used. `top` is such a class.

Without getting into the specifics of each application there are some general structural object classes that are useful:

- `inetOrgPerson`: Defines a generic person, along with some contact information.
- `organizationalRole`: Much like a person, but it defines a generic role such as IT Helpdesk or Fire Warden.
- `organizationalUnit`: A generic container, may describe a department within a container or may be used to separate various parts of the LDAP tree such as groups, people, and servers
- `organization`: A company or other organization.
- `groupOfNames`: Stores one or more DN's referring to members of a group. Not necessarily useful for UNIX groups, but helpful for meeting invitations or other simple things.

Stick to these for your people and organizations and you will be safe. Most extensions, such as authentication, use auxiliary attributes. When in doubt, consult the schema.

The final design consideration is the choice of DN's. Most branches are fairly easy because there is no chance of duplication. A UNIX group's name and ID is unique, so using `cn` as the RDN of a group is possible. What happens when you have two employees called Fred Smith? Because the DN must be unique, `cn=Fred Smith,ou=People,dc=example,dc=com` could be either of them. Either something else must be used, such as `employeeNumber`, or the RDN will have to be made from two different attributes separated by a plus sign (+). For example, `cn=Fred Smith+userid=123,ou=people,dc=example,dc=com` has an RDN made from two different attributes. Whatever you do, do it consistently!

Section 4. Schemas

This section covers material for topic 301.3 for the Senior Level Linux Professional (LPIC-3) exam 301. The topic has a weight of 3.

In this section, learn about:

- LDAP schema concepts
- How to create and modify schemas
- Attribute and object class syntax

Up until now, the schema has been mentioned several times but not fully explained. The schema is a collection of object classes and their attributes. A schema file contains one or more object classes and attributes in a text format the LDAP server can understand. You import the schema file into your LDAP server's configuration, and then use the object classes and attributes in your objects. If the available schemas don't fit your needs, you can create your own or extend an existing one.

LDAP schema concepts

Technically, a schema is a packaging mechanism for object classes and attributes. However, the grouping of object classes is not random. Schemas are generally organized along an application, such as a core, X.500 compatibility, UNIX network services, sendmail, and so on. If you have a need to integrate an application with LDAP, you generally have to add a schema to your LDAP server.

A more in-depth look at OpenLDAP configuration will be in a later tutorial in this series, but the way to add a schema is with `include /path/to/file.schema`. After restarting the server, the new schema will be available.

When the schema is loaded, you then apply the new object classes to the relevant objects. This can be done through an LDIF file or through the LDAP application program interface (API). Applying the object classes gives you more attributes to use.

Creating and modifying schemas

Schemas have a fairly simple format. Listing 6 shows the schema for the `inetOrgPerson` object class along with some of its attributes.

Listing 6. Part of the `inetOrgPerson` definition

```
attributetype ( 2.16.840.1.113730.3.1.241
```

```

NAME 'displayName'
DESC 'RFC2798: preferred name to be used when displaying entries'
EQUALITY caseIgnoreMatch
SUBSTR caseIgnoreSubstringsMatch
SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
SINGLE-VALUE )

attributetype ( 0.9.2342.19200300.100.1.60
  NAME 'jpegPhoto'
  DESC 'RFC2798: a JPEG image'
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.28 )

objectclass ( 2.16.840.1.113730.3.2.2
  NAME 'inetOrgPerson'
  DESC 'RFC2798: Internet Organizational Person'
  SUP organizationalPerson
  STRUCTURAL
  MAY (
    audio $ businessCategory $ carLicense $ departmentNumber $
    displayName $ employeeNumber $ employeeType $ givenName $
    homePhone $ homePostalAddress $ initials $ jpegPhoto $
    labeledURI $ mail $ manager $ mobile $ o $ pager $
    photo $ roomNumber $ secretary $ uid $ userCertificate $
    x500uniqueIdentifier $ preferredLanguage $
    userSMIMECertificate $ userPKCS12 )
  )

```

Line spacing is not important in schema files -- it is mostly there for human readability. The first definition is an `attributetype`, which means an attribute. The parentheses enclose the definition of the attribute. First comes a series of numbers, separated by periods, called the *object ID*, or *OID*, which is a globally unique number. The OID is also hierarchical, such that if you're assigned the 1.1 tree, you can create anything like 1.1.1 or 1.1.2.3.4.5.6 without having to register it.

Registering OIDs

You can't just pick any series of numbers for your own OID because you don't know if the value you choose is, or will be, in use elsewhere. OIDs must be unique. Some servers let you specify textual OIDs (mycompany.1.2) which would be unique, just not necessarily compatible. Anything under the 1.1 namespace can be used locally but is not guaranteed to be unique.

The best solution is to register your own OID. There are many ways to do this, depending on if you are a country, a telephone carrier, or fall under other categories. Luckily, the Internet Assigned Numbers Authority (IANA) will freely give you your own branch under .1.3.6.4.1 if you ask.

Following the OID is a series of keywords, each of which may have a value after it. First the `NAME` of the attribute defines the name that humans will use, such as in the LDIF file or when retrieving the information through the LDAP API. Sometimes you might see the name in the form of `NAME ('foo' 'bar')`, which means that either `foo` or `bar` are acceptable. The server, however, considers the first to be the primary name of the attribute.

`DESC` provides a description of the attribute. This helps you understand the attribute if you're browsing the schema file. `EQUALITY`, `SUBSTR`, and `ORDERING` (not shown) require a *matching rule*. This defines how strings are compared, searched, and sorted, respectively. `caseIgnoreMatch` is a case-insensitive match, and

`caseIgnoreSubstringsMatch` is also case insensitive. See the [Resources](#) section for Web sites that define all the standard matching rules. Like most things in LDAP, a server can define its own matching methods for its own attributes, so there are no comprehensive lists of matching rules.

The `SYNTAX` of the attribute defines the format of the data by referencing an OID. RFC 2252 lists the standard syntaxes and their OIDs. If the OID is immediately followed by a number in curly braces (`{}`), this represents the maximum length of the data. `1.3.6.1.4.1.1466.115.121.1.15` represents a `DirectoryString` that is a UTF-8 string.

Finally, the `SINGLE-VALUE` keyword has no arguments and specifies that only one instance of `displayName` is allowed.

The `jpegPhoto` attribute has a very short definition: just the OID, the name and description, and a syntax meaning a JPEG object, which is an encoded string of the binary data. It is not practical to search or sort a picture, and multiple photos can exist in a single record.

Defining an object class follows a similar method. The `objectclass` keyword starts the definition, followed by parentheses, the OID of the object class, and then the definitions. `NAME` and `DESC` are the same as before. `SUP` defines a superior object class, which is another way of saying that the object class being defined inherits from the object class specified by the `SUP` keyword. Thus, an `inetOrgPerson` carries the attributes of an `organizationalPerson`.

The `STRUCTURAL` keyword defines this as a structural object class, which can be considered the primary type of the object. Other options are `AUXILIARY`, which adds attributes to an existing object, and `ABSTRACT`, which is the same as structural but cannot be used directly. Abstract object classes must be inherited by another object class, which can then be used. The `top` object class is abstract. It is inherited by most other structural object classes, including `person`, which is the parent of `organizationalPerson`, which, in turn, is inherited by `inetOrgPerson`.

Two keywords, `MAY` and `MUST`, define the attributes that are allowed and mandatory, respectively, for records using that particular object class. For mandatory items, you may not save a record without all the items being defined. Each attribute is separated by a dollar sign (`$`), even if the line continues on the next line.

It is not a good idea to modify structural object classes, or even existing, well-known, auxiliary object classes. Because these are well known, you may cause incompatibility issues in the future if your server is different. Usually the best solution is to define your own auxiliary object class, create a local schema, and apply it to your records. For instance, if you are a university and want to store student attributes, you might consider creating a student object class that is inherited from either `organizationalPerson` or `inetOrgPerson` and adding your own attributes. You could then create auxiliary object classes to add more attributes such as class schedules.

Understanding which schemas to use

After learning about how schemas are created, it is tempting to start fresh -- to create your own schema based on your environment. This would certainly take care of your present needs, but it could quite possibly make things more difficult in the long run as you add more functionality to your LDAP tree and integrate other systems. The best approach is to stick with standard object classes and attributes when you can and extend when you must.

OpenLDAP usually stores its schema files in `/etc/openldap/schema`, in files with a `.schema` extension. Table 3 lists the default schemas along with their purposes.

Table 3. Schemas that ship with OpenLDAP

File name	Purpose
corba.schema	Defines some object classes and attributes for handling Common Object Request Broker Architecture (CORBA) object references across multiple machines.
core.schema	Defines many common attributes and object classes. This schema is where you will find the <code>organizationalUnit</code> , <code>top</code> , <code>dcObject</code> , and <code>organizationalRole</code> . <code>core.schema</code> is the first place you should look if you need to find something.
cosine.schema	Attributes and object classes from the X.500 specifications. While there are some useful things in here, there are often better alternatives in others such as <code>core</code> and <code>inetorgperson</code> .
dyngroup.schema	An experimental set of objects used with Netscape Enterprise Server.
inetorgperson.schema	Defines the <code>inetOrgPerson</code> object (which extends objects from <code>core.schema</code>).
java.schema	Like <code>corba.schema</code> , this schema defines a series of object classes and attributes to handle the lookup of Java™ classes within an LDAP tree.
misc.schema	Implements some objects to handle mail lookups within the tree. It is best to consult your e-mail server's documentation to see which schema it uses.
nis.schema	This is the schema you use if you move authentication to LDAP. <code>nis.schema</code> defines <code>posixAccount</code> , which provides attributes for storing authentication data within the user's object. It also has the various map types to handle groups,

	networks, services, and other files that go into network-based authentication mechanisms such as the Network Information System (NIS).
openldap.schema	This is more for example purposes and shows some basic objects.
ppolicy.schema	A set of objects to implement password policies in LDAP, such as aging. Note that some of these are handled by the traditional UNIX shadow mechanisms and are already handled in nis.schema.

In addition, RFC 4519 explains common attributes. After finding the attributes you want, you can then look through the schema files to determine which files need to be included in your LDAP configuration and which object classes you must use for your records.

Section 5. Summary

In this tutorial you learned about LDAP concepts, architecture, and design. LDAP grew out of a need to connect to X.500 directories over IP in a simplified way. A directory presents data to you in a hierarchical manner, much like a tree. Within this tree are records that are identified by a distinguished name and have many attribute-value pairs, including one or more object classes that determine what data can be stored in the record.

LDAP itself refers to the protocol used to search and modify the tree. Practically though, the term LDAP is used for all components, such as LDAP server, LDAP data, or just "It's in LDAP".

Data in LDAP is often imported and exported with LDIF, which is a textual representation of the data. An LDIF file specifies a `changetype`, such as `add`, `delete`, `modrdn`, `moddn`, and `modify`. These operations let you add entries, delete entries, move data around in the tree, and change attributes of the data.

Designing the tree correctly is crucial to long-term viability of the LDAP server. A correct design means fewer change operations are needed, which leads to consistent data that can easily be found by other applications. By choosing common attributes, you ensure that other consumers of the LDAP data understand the meaning of the attributes and that fewer translations are required.

The LDAP schema dictates which attributes can be used in your server. Within the schema are definitions of the attributes, including OIDs to uniquely identify them, instructions on how to store and sort the data, and textual descriptions of what the attributes do. Object classes group attributes together and can be defined as structural, auxiliary, or abstract.

Structural object classes define the record, so a record may only have one structural object class. Auxiliary object classes add more attributes for specific purposes and can be added to any record. An abstract object class must be inherited and cannot be used directly.

Resources

Learn

- Review the entire [LPI exam prep tutorial series](#) on developerWorks to learn Linux fundamentals and prepare for system administrator certification.
- At the [LPIC Program](#), find task lists, sample questions, and detailed objectives for the three levels of the Linux Professional Institute's Linux system administration certification.
- [RFC 1487](#), *X.500 Lightweight Directory Access Protocol*, gives you some insight into the development of LDAP and the history of X.500.
- [RFC 2247](#), *Using Domains in LDAP/X.500 Distinguished Names*, is a brief description of the `domainComponent` attribute and how to use it properly in your LDAP tree.
- [RFC 2252](#), *Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions*, lists the standard syntaxes for attributes, which can help you figure out what format a certain attribute is expecting.
- [RFC 2849](#), *The LDAP Data Interchange Format (LDIF) - Technical Specification*, describes the LDIF language. It uses Backus-Naur Form (BNF) to specify the language, which can be tricky to understand. [RFC 2234](#), *Augmented BNF for Syntax Specifications: ABNF*, might help you understand the various operators.
- [RFC 4511](#), *Lightweight Directory Access Protocol (LDAP): The Protocol*, is the latest draft of the LDAP protocol.
- [RFC 4519](#), *Lightweight Directory Access Protocol (LDAP): Schema for User Applications*, is an updated list of the commonly-used attributes; this list helps ensure you're using the same attributes everyone else is to describe the same data.
- The [OID descriptions for 2.5.13](#) link to detailed descriptions of how each matching rule (string comparison, substrings, and ordering) works.
- This [FAQ entry on object classes](#) gives details on some of trickier rules of dealing with object classes. Some of OpenLDAP's error messages are terse, especially when dealing with LDIF imports.
- The [overlay](#) framework in OpenLDAP is key because the meta-directory concept can be carried further than just tying together multiple LDAP servers. A request for a particular tree or OID can be directed to custom code that can call a script, read a database, or call an API. Another description is on the [Symas Corporation](#) Web site.
- "[Demystifying LDAP Data](#)" (O'Reilly, November 2006) explains object class inheritance. It looks at the `inetOrgPerson` object class and describes structural and auxiliary object classes.
- [LDAP for Rocket Scientists](#) is an excellent open source guide, despite being a

work in progress.

- In the [developerWorks Linux zone](#), find more resources for Linux developers, and scan our [most popular articles and tutorials](#).
- See all [Linux tips](#) and [Linux tutorials](#) on developerWorks.
- Stay current with [developerWorks technical events and Webcasts](#).

Get products and technologies

- [OpenLDAP](#) is a great choice if you're looking for an LDAP server.
- [phpLDAPadmin](#) is a Web-based LDAP administration tool.
- [Luma](#) is a good GUI to look at if that's more your style.
- [Order the SEK for Linux](#), a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- With [IBM trial software](#), available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the [developerWorks community](#) through blogs, forums, podcasts, and community topics in our [new developerWorks spaces](#).

About the author

Sean A. Walberg

Sean Walberg has been working with Linux and UNIX since 1994 in academic, corporate, and Internet service provider environments. He has written extensively about systems administration over the past several years.

Trademarks

DB2, Lotus, Rational, Tivoli, and WebSphere are trademarks of IBM Corporation in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.