

52

Directory Services

WHAT'S IN THIS CHAPTER?

- The architecture and concepts of Active Directory
- Tools for accessing the Active Directory
- How to read and modify data in Active Directory
- Searching for objects in Active Directory
- User and group management programmatically
- Using DSML (Directory Service Markup Language) to access the Active Directory

Microsoft's Active Directory is a *directory service* that provides a central, hierarchical store for user information, network resources, services, and so on. The information in this directory service can be extended to also store custom data that is of interest for the enterprise. For example, Microsoft Exchange Server and Microsoft Dynamics use Active Directory extensively to store public folders and other items.

Before the release of Active Directory, Exchange Server used its own private store for its objects. It was necessary for a system administrator to configure two user IDs for a single person: a user account in the Windows NT domain to enable a logon and a user in Exchange Directory. This was necessary because of the additional information required by users (such as e-mail addresses, phone numbers, and so on), and the user information for the NT domain was not extensible to add the required information.

Now, the system administrator has to configure just a single user for a person in Active Directory; the information for a `user` object can be extended so that it fits the requirements of Exchange Server. You can also extend this information. For example, you can extend user information in Active Directory with a skills list. Then, it would easily be possible to track down a C# developer by searching for the required C# skill.

This chapter shows how you can use the .NET Framework to access and manipulate the data in a directory service using classes from the `System.DirectoryServices`, `System.DirectoryServices.AccountManagement`, and `System.DirectoryServices.Protocols` namespaces.



This chapter uses Windows Server 2008 R2 with Active Directory configured. You can also use Windows 2003 Server or other directory services.

After discussing the architecture and how to program Active Directory, you create a Windows application in which you can specify properties and a filter to search for `user` objects. Similar to other chapters, you can also download the code for the examples in this chapter from the Wrox web site at www.wrox.com.

THE ARCHITECTURE OF ACTIVE DIRECTORY

Before starting to program Active Directory, you need to know how it works, what it is used for, and what data can be stored there.

Active Directory Features

The features of Active Directory can be summarized as follows:

- **Hierarchical grouping of data.** Objects can be stored inside other container objects. Instead of having a single, large list of users, you can group users inside organizational units. An organizational unit can contain other organizational units, so you can build a tree.
- **Multimaster replication.** With Active Directory, every *domain controller* (DC) is a master. With multiple masters, updates can be applied to any DC. This model is much more scalable than a single-master model because updates can be made to different servers concurrently. The disadvantage of this model is more complex replication, which is discussed later in this chapter.
- **Flexible replication topology.** This supports replications across slow links in WANs. How often data should be replicated is configurable by the domain administrators.
- **Open standards.** Active Directory supports open standards. The *Lightweight Directory Access Protocol* (LDAP) is an Internet standard that can be used to access many different directory services, including the data in Active Directory. With LDAP, a programming interface, LDAP API, is also defined. The LDAP API can be used to access Active Directory with the C language. Another standard used within Active Directory is *Kerberos*, which is used for authentication. The Windows Server Kerberos service can also be used to authenticate UNIX clients.
- **Active Directory Service Interface (ADSI).** ADSI defines COM interfaces to access directory services. ADSI makes it possible to access all features of Active Directory. Classes from the namespace `System.DirectoryServices` wrap ADSI COM objects to make directory services accessible from .NET applications.
- **Directory Service Markup Language (DSML).** DSML is another standard to access directory services. It is a platform-independent approach and is supported by the OASIS group.
- **Fine-grained security.** With Active Directory, fine-grained security is available. Every object stored in Active Directory can have an associated access control list that defines who can do what with that object.

The objects in the directory are *strongly typed*, which means that the type of an object is exactly defined; no attributes that are not specified may be added to an object. In the *schema*, the object types as well as the parts of an object (attributes) are defined. Attributes can be mandatory or optional.

Active Directory Concepts

Before programming Active Directory, you need to know some basic terms and definitions.

Objects

Active Directory stores objects. An object refers to something concrete such as a user, a printer, or a network share. Objects have mandatory and optional attributes that describe them. Some examples of the attributes of a `user` object are the first name, last name, e-mail address, phone number, and so on.

Figure 52-1 shows a container object called Wrox Press that contains some other objects: two user objects, a contact object, a printer object, and a user group object.

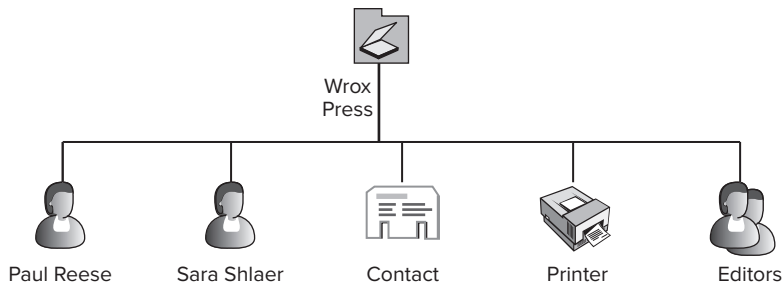


FIGURE 52-1

Schema

Every object is an instance of a class defined in the *schema*. The schema *defines the types* and is itself stored in objects in Active Directory. You must differentiate between `classSchema` and `attributeSchema`: `classSchema` defines the types of objects and details what mandatory and optional attributes an object has. `attributeSchema` defines what an attribute looks like and the allowed syntax for a specific attribute.

You can define custom types and attributes and add these to the schema. Be aware, however, that a new schema type cannot be removed from Active Directory. You can mark it as inactive so that new objects cannot be created, but there can be existing objects of that type, so it is not possible to remove classes or attributes defined in the schema.

The user group Administrator doesn't have enough rights to create new schema entries; the group Enterprise Admins is needed for that.

Configuration

In addition to objects and class definitions stored as objects, the configuration of Active Directory itself is stored in Active Directory. It stores the information about all sites, such as the replication interval, which is set up by the system administrator. Because the configuration itself is stored in Active Directory, you can access the configuration information like all other objects in Active Directory.

The Active Directory Domain

A domain is a security boundary of a Windows network. In the Active Directory domain, the objects are stored in a hierarchical order. Active Directory itself is made up of one or more domains. Figure 52-2 shows the hierarchical order of objects in a domain; the domain is represented by a triangle. Container objects such as `Users`, `Computers`, and `Books` can store other objects. Each oval in the picture represents an object, with the lines between the objects representing parent-child relationships. For example, `Books` is the parent of `.NET` and `Java`, and `Pro C#`, `Begin C#`, and `ASP.NET` are child objects of the `.NET` object.

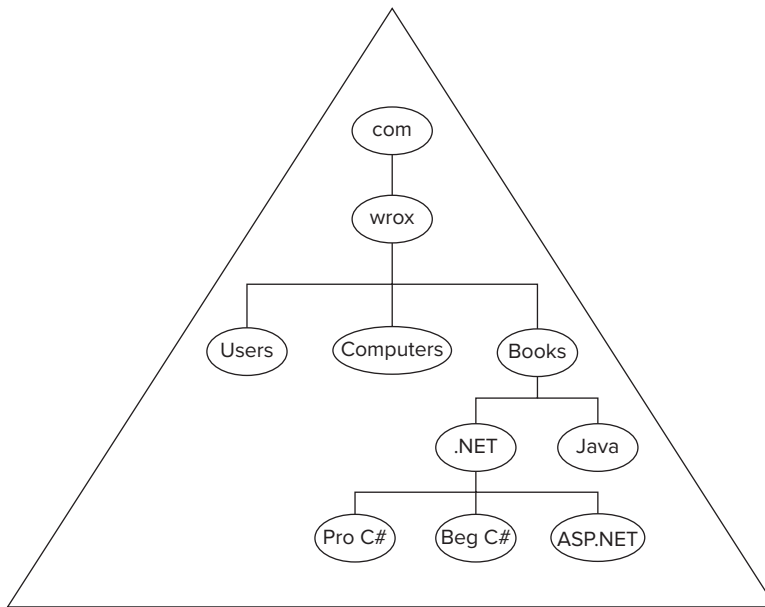


FIGURE 52-2

Domain Controller

A single domain can have multiple domain controllers, each of which stores all of the objects in the domain. There is no master server, and all DCs are treated equally; you have a multimaster model. The objects are replicated across the servers inside the domain.

Site

A *site* is a location in the network that holds at least one DC. If you have multiple locations in the enterprise, which are connected with slow network links, you can use multiple sites for a single domain. For backup or scalability reasons, each site can have one or more DCs running. Replication between servers in a site can happen at shorter intervals because of the faster network connection. Replication is configured to occur at larger time intervals between servers across sites, depending on the speed of the network. Of course, replication intervals can be configured by the domain administrator.

Domain Tree

Multiple domains can be connected by trust relationships. These domains share a *common schema*, a *common configuration*, and a *global catalog* (more on global catalogs shortly). A common schema and a common configuration imply that this data is replicated across domains. Domain trees share the same class and attribute schema. The objects themselves are not replicated across domains.

Domains connected in such a way form a domain tree. Domains in a domain tree have a contiguous, hierarchical namespace. This means that the domain name of the child domain is the name of that child domain appended to the name of the parent domain. Between domains, trusts using the Kerberos protocol are established.

For example, you have the root domain `wrox.com`, which is the *parent domain* of the child domains `india.wrox.com` and `uk.wrox.com`. A trust is set up between the parent and the child domains, so that accounts from one domain can be authenticated by another domain.

Forest

Multiple domain trees that are connected by using a common schema, a common configuration, and a global catalog without a contiguous namespace are called a *forest*. A forest is a set of domain trees; it can be used if the company has a subcompany for which a different domain name should be used. Here is one example: `wrox.com` should be relatively independent of the domain `wiley.com`, but it should be possible to have a common management, and be possible for users from `wrox.com` to access resources from the `wiley.com` domain and vice versa. With a forest, you can have trusts between multiple domain trees.

Global Catalog

A search for an object can span multiple domains. If you look for a specific `user` object with some attributes, you must search every domain. Starting with `wrox.com`, the search continues to `uk.wrox.com` and `india.wrox.com`; across slow links, such a search could take a while.

To make searches faster, all objects are copied to the *global catalog* (GC). The GC is replicated in every domain of a forest. There is at least one server in every domain holding a GC. For performance and scalability reasons, you can have more than one GC server in a domain. Using a GC, a search through all the objects can happen on a single server.

The GC is a *read-only cache* of all the objects that can be used only for searches; the domain controllers must be used to do updates.

Not all attributes of an object are stored in the GC. You can define whether an attribute should be stored with an object. The decision whether to store an attribute in the GC depends on how the attribute is used. If the attribute is frequently used in searches, putting it into the GC makes the search faster. A picture of a user isn't useful in the GC because you would never search for a picture. Conversely, a phone number would be a useful addition to the store. You can also define that an attribute should be indexed so that a query for it is faster.

Replication

As a programmer, you are unlikely ever to configure replication, but because it affects the data you store in Active Directory, you need to know how it works. Active Directory uses a *multimaster* server architecture. Updates happen to every domain controller in the domain. The *replication latency* defines how long it takes until an update starts:

- The configurable *change notification* happens, by default, every 5 minutes inside a site if some attributes change. The DC where a change occurred informs one server after the other with 30-second intervals, so the fourth DC can get the change notification after 7 minutes. The default change notification across sites is set to 180 minutes. Intra- and intersite replication can each be configured to other values.
- If no changes have occurred, the *scheduled replication* occurs every 60 minutes inside a site. This is to ensure that a change notification wasn't missed.
- For security-sensitive information, such as account lockout, *immediate notification* can occur.

With a replication, only the changes are copied to the DCs. With every change of an attribute, a version number (update sequence number or USN) and a time stamp are recorded. These are used to help resolve conflicts if updates have been made to the same attribute on different servers.

Here's an example. The mobile phone attribute of the user John Doe has the USN number 47. This value is already replicated to all DCs. One system administrator changes the phone number. The change occurs on the server DC1; the new USN of this attribute on the server DC1 is now 48, whereas the other DCs still have the USN 47. For someone still reading the attribute, the old value can be read until the replication to all domain controllers has occurred.

The rare case can happen that another administrator changes the phone number attribute, and a different DC is selected because this administrator received a faster response from the server DC2. The USN of this attribute on the server DC2 is also changed to 48.

At the notification intervals, notification happens because the USN for the attribute changed, and the last time replication occurred was with a USN value of 47. The replication mechanism now detects that the servers DC1 and DC2 both have a USN of 48 for the phone number attribute. Which server is the winner is not really important, but one server must definitely win. To resolve this conflict, the time stamp of the change is used. Because the change happened later on DC2, the value stored in the DC2 domain controller is replicated.

When reading objects, you must be aware that the data is not necessarily current. The currency of the data depends on replication latencies. When updating objects, another user can still read some old values after the update. It's also possible that different updates can happen at the same time.

Characteristics of Active Directory Data

Active Directory doesn't replace a relational database or the registry, so what kind of data would you store in it?

- With Active Directory you get **hierarchical data**. You can have containers that store further containers and objects, too. Containers themselves are objects as well.
- The data should be used for **read-mostly**. Because of replication occurring at certain time intervals, you cannot be sure that you will read up-to-date data. You must be aware that, in applications, the information you read is possibly not the current up-to-date information.
- Data should be of **global interest** to the enterprise, because adding a new data type to the schema replicates it to all the servers in the enterprise. For data types of interest to only a small number of users, the domain enterprise administrator normally wouldn't install new schema types.
- The data stored should be of **reasonable size** because of replication issues. It is fine to store data with a size of 100K in the directory, if the data changes only once a week. However, if the data changes every hour, data of this size is too large. Always think about replicating the data to different servers: where the data gets transferred to and at what intervals. If you have larger data, it's possible to put a link into Active Directory and store the data itself in a different place.

To summarize, the data you store in Active Directory should be hierarchically organized, of reasonable size, and of importance to the enterprise.

Specifying Schema

Active Directory objects are strongly typed. The schema defines the types of the objects, mandatory and optional attributes, and the syntax and constraints of these attributes. As mentioned earlier, in the schema, it is necessary to differentiate between class-schema and attribute-schema objects.

A class is a collection of attributes. With the classes, single inheritance is supported. As you can see in Figure 52-3, the user class derives from the organizationalPerson class, organizationalPerson is a subclass of person, and the base class is top. The classSchema that defines a class describes the attributes with the systemMayContain attribute.

Figure 52-3 shows only a few of all the systemMayContain values. Using the ADSI Edit tool, you can easily see all the values; you look at this tool in the next section, "ADSI Edit." In the root class top, you can see that every object can have common name (cn), displayName, objectGUID, whenChanged, and whenCreated attributes. The person class derives from top. A person object also has a userPassword and a

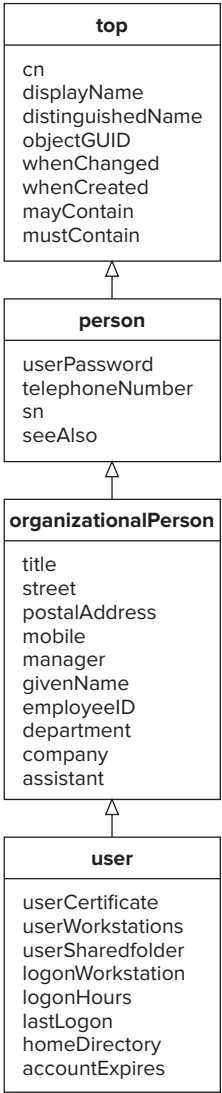


FIGURE 52-3

telephoneNumber. organizationalPerson is derived from person. In addition to the attributes of person, it has a manager, department, and company, and a user has extra attributes needed to log on to a system.

ADMINISTRATION TOOLS FOR ACTIVE DIRECTORY

Looking into some of the Active Directory administration tools can help to give you an idea of Active Directory, what data is in there, and what can be done programmatically.

The system administrator has many tools to enter new data, update data, and configure Active Directory:

- The *Active Directory Users and Computers* MMC snap-in is used to enter new users and update user data.
- The *Active Directory Sites and Services* MMC snap-in is used to configure sites in a domain and for replication between these sites.
- The *Active Directory Domains and Trusts* MMC snap-in can be used to build up a trust relationship between domains in a tree.
- *ADSI Edit* is the editor for Active Directory, where every object can be viewed and edited.



To run these tools on Windows 7, you need to install Windows 7 Remote Administration Tools. For other Windows versions, there are also separate downloads for these system management tools.

The following sections get into the functionality of the tools Active Directory Users and Computers and ADSI Edit because these tools are important in regard to creating applications using Active Directory.

Active Directory Users and Computers

The Active Directory Users and Computers snap-in is the tool that system administrators use to manage users. Select Start ➤ Administrative ➤ Tools ➤ Active Directory Users and Computers to start this program (see Figure 52-4).

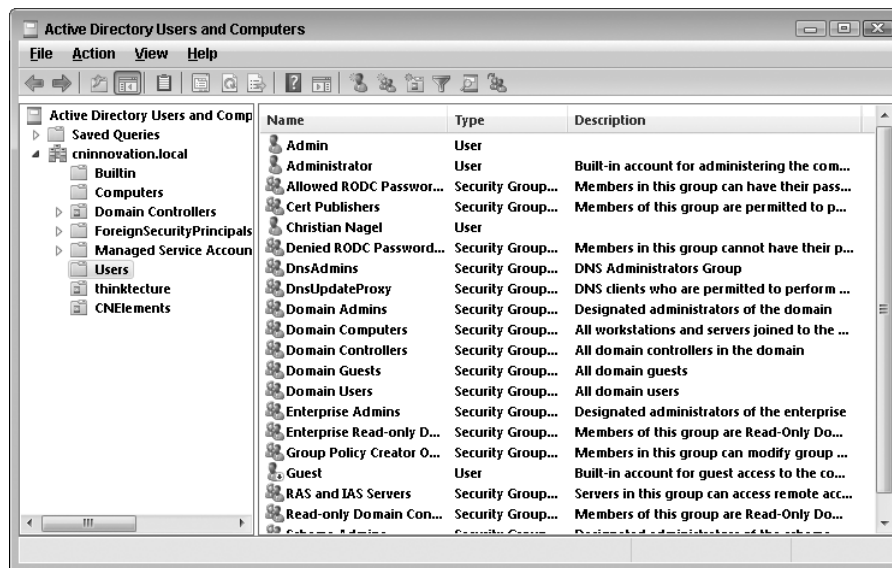


FIGURE 52-4

With this tool you can add new users, groups, contacts, organizational units, printers, shared folders, or computers, and modify existing ones. Figure 52-5 shows the attributes that can be entered for a user object: office, phone numbers, e-mail addresses, web pages, organization information, addresses, groups, and so on.

Active Directory Users and Computers can also be used in big enterprises with millions of objects. It's not necessary to look through a list with a thousand objects, because you can select a custom filter to display only some of the objects. You can also perform an LDAP query to search for the objects in the enterprise. You explore these possibilities later in this chapter.

ADSI Edit

ADSI Edit is the editor of Active Directory. ADSI Edit offers greater control than the Active Directory Users and Computers tool (see Figure 52-6); with ADSI Edit, everything can be configured, and you can also look at the schema and the configuration. This tool is not very intuitive to use, however, and it is very easy to enter wrong data.

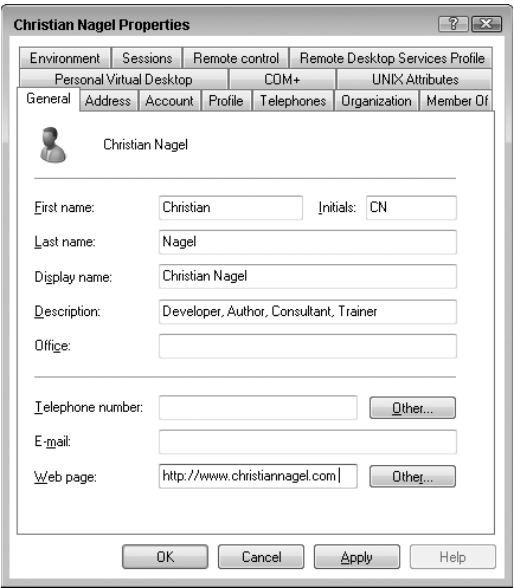


FIGURE 52-5

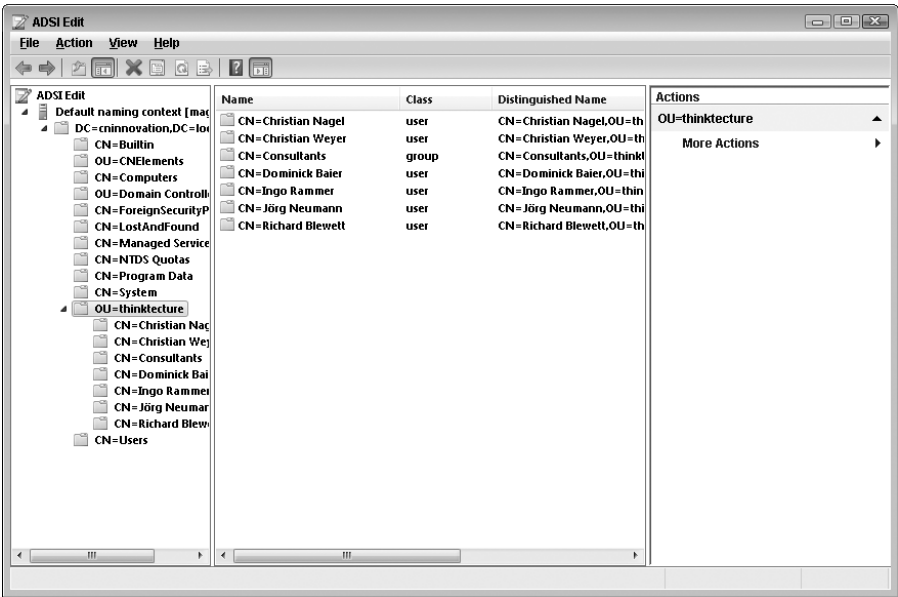


FIGURE 52-6

By opening the properties window of an object, you can view and change every attribute of an object in Active Directory. With this tool, you can see mandatory and optional attributes, with their types and values (see Figure 52-7).

PROGRAMMING ACTIVE DIRECTORY

To develop programs for Active Directory, you can use the classes from either the `System.DirectoryServices` or the `System.DirectoryServices.Protocols` namespaces. In the namespace `System.DirectoryServices`, you can find classes that wrap Active Directory Service Interfaces (ADSI) COM objects to access Active Directory.

ADSI is a programmatic interface to directory services. It defines some COM interfaces that are implemented by ADSI providers. This means that the client can use different directory services with the same programmatic interfaces. The .NET Framework classes in the `System.DirectoryServices` namespace make use of ADSI.

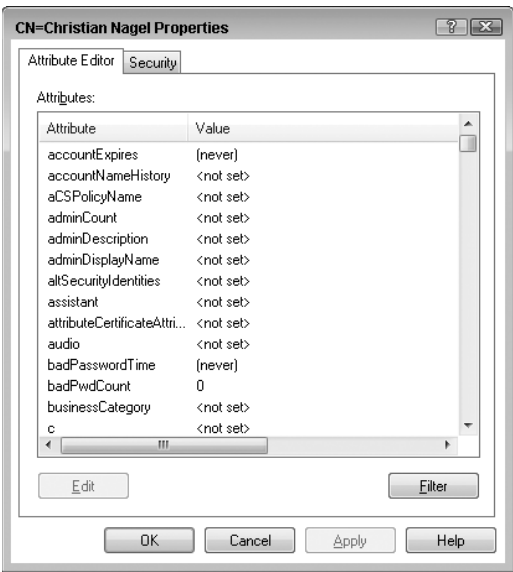


FIGURE 52-7

Figure 52-8 shows some ADSI Providers (LDAP, IIS, and NDS) that implement COM interfaces such as IADs and IUnknown. The assembly `System.DirectoryServices` makes use of the ADSI providers.

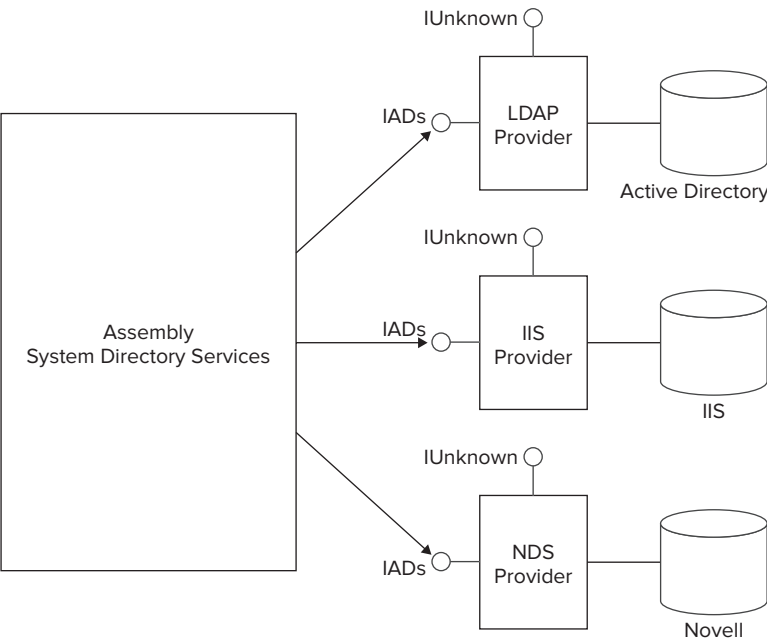


FIGURE 52-8

Classes from the namespace `System.DirectoryServices.Protocols` make use of Directory Services Markup Language (DSML) Services for Windows. With DSML, standardized Web service interfaces are defined by the OASIS group (www.oasis-open.org/committees/dsml).

To use the classes from the `System.DirectoryServices` namespace, you need to reference the `System.DirectoryServices` assembly. With the classes in this assembly, you can query objects, view and update properties, search for objects, and move objects to other container objects. In the code segments that follow later in this section, you use a simple C# console application that demonstrates the functionality of the classes in the `System.DirectoryServices` namespace.

This section covers the following:

- Classes in the `System.DirectoryServices` namespace
- The process of connecting to Active Directory (binding)
- Getting directory entries, creating new objects, and updating existing entries
- Searching Active Directory

Classes in `System.DirectoryServices`

The following table shows the major classes in the `System.DirectoryServices` namespace.

CLASS	DESCRIPTION
<code>DirectoryEntry</code>	This is the main class of the <code>System.DirectoryServices</code> namespace. An object of this class represents an object in the Active Directory store. This class is used to bind to an object and to view and update properties. The properties of the object are represented in a <code>PropertyCollection</code> . Every item in the <code>PropertyCollection</code> has a <code>PropertyValueCollection</code> .
<code>DirectoryEntries</code>	<code>DirectoryEntries</code> is a collection of <code>DirectoryEntry</code> objects. The <code>Children</code> property of a <code>DirectoryEntry</code> object returns a list of objects in a <code>DirectoryEntries</code> collection.
<code>DirectorySearcher</code>	This is the main class used for searching for objects with specific attributes. To define the search, the <code>SortOption</code> class and the enumerations <code>SearchScope</code> , <code>SortDirection</code> , and <code>ReferralChasingOption</code> can be used. The search results in a <code>SearchResult</code> or a <code>SearchResultCollection</code> . You also get <code>ResultPropertyCollection</code> and <code>ResultPropertyValueCollection</code> objects.

Binding to Directory Services

To get the values of an object in Active Directory, you need to connect to the Active Directory service. This connecting process is called *binding*. The binding path can look like this:

```
LDAP://dc01.thinktecture.com/OU=Development, DC=thinktecture, DC=Com
```

With the binding process, you can specify these items:

- The *protocol* specifies the provider to be used
- The *server name* of the domain controller
- The *port number* of the server process
- The *distinguished name* of the object; this identifies the object you want to access
- The *username and password*, if the user who is allowed to access Active Directory is different from the current logged-on user
- An *authentication* type, if encryption is needed

The following subsections discuss these options in more detail.

Protocol

The first part of a binding path, the protocol, specifies the ADSI provider. The provider is implemented as a COM server; for identification, a `progID` can be found in the registry directly under `HKEY_CLASSES_ROOT`. The providers that are available with Windows 7 are listed in the following table.

PROVIDER	DESCRIPTION
LDAP	LDAP Server, such as the Exchange directory and Active Directory Servers since Windows 2000.
GC	GC is used to access the global catalog in Active Directory. It can be used for fast queries.
IIS	With the ADSI provider for IIS, it's possible to create new web sites and to administer them in the IIS catalog.
NDS	This progID is used to communicate with Novell Directory Services.

Server Name

The server name follows the protocol in the binding path. The server name is optional if you are logged on to an Active Directory domain. Without a server name, *serverless binding* occurs; this means that Windows Server 2008 tries to get the “best” domain controller in the domain that’s associated with the user doing the bind. If there is no server inside a site, the first domain controller that can be found will be used.

A serverless binding might look like this: LDAP:

```
//OU=Sales, DC=Thinktecture, DC=Local
```

Port Number

After the server name, you can specify the port number of the server process by using the syntax `xxx`. The default port number for the LDAP server is 389:

```
LDAP://dc01.sentinel.net:389.
```

The Exchange server uses the same port number as the LDAP server. If the Exchange server is installed on the same system — for example, as a domain controller of Active Directory — a different port can be configured.

Distinguished Name

The fourth part that you can specify in the path is the distinguished name (DN). The DN is a unique name that identifies the object you want to access. With Active Directory, you can use LDAP syntax that is based on X.500 to specify the name of the object.

Here’s an example of a distinguished name:

```
CN=Christian Nagel, OU=Consultants, DC=thinktecture, DC=local
```

This DN specifies the common name (CN) of Christian Nagel in the organizational unit (OU) called Consultants in the domain component (DC) called thinktecture of the domain thinktecture.local. The part specified at the right (DC) is the root object of the domain. The name must follow the hierarchy in the object tree.

You can find the LDAP specification for the string representation of distinguished names in RFC 2253 at www.ietf.org/rfc/rfc2253.txt.

Relative Distinguished Name

A *relative distinguished name* (RDN) is used to reference objects within a container object. With an RDN, the specification of OU and DC is not needed because a common name is enough. `CN=Christian Nagel` is the relative distinguished name inside the organizational unit. A relative distinguished name can be used if you already have a reference to a container object and if you want to access child objects.

Default Naming Context

If a distinguished name is not specified in the path, the binding process will be made to the default naming context. You can read the default naming context with the help of `rootDSE`. LDAP 3.0 defines `rootDSE` as the root of a directory tree on a directory server. For example:

```
LDAP://rootDSE
```

or

```
LDAP://servername/rootDSE
```

By enumerating all properties of the `rootDSE`, you can get the information about the `defaultNamingContext` that will be used when no name is specified. `schemaNamingContext` and `configurationNamingContext` specify the required names to be used to access the schema and the configuration in the Active Directory store.

The following code is used to get all properties of `rootDSE`:



```
try
{
    using (var de = new DirectoryEntry())
    {
        de.Path = "LDAP://magellan/rootDSE";
        de.Username = @"cninnovation\christian";
        de.Password = "Pa$$w0rd";

        PropertyCollection props = de.Properties;
        foreach (string prop in props.PropertyNames)
        {
            PropertyValueCollection values = props[prop];
            foreach (string val in values)
            {
                Console.WriteLine("{0}: ", prop);
                Console.WriteLine(val);
            }
        }
    }
}
catch (COMException ex)
{
    Console.WriteLine(ex.Message);
}
```

code snippet DirectoryServicesSamples/Program.cs



To have this code running on your machine, you must change the path to the object to access, including the server name.

This program shows the default naming context (`defaultNamingContext DC=cninnovation, DC=local`), the context that can be used to access the schema (`CN=Schema, CN=Configuration, DC=cninnovation, DC=local`), and the naming context of the configuration (`CN=Configuration, DC=cninnovation, DC=local`), as you can see here:

```
currentTime: 20090925131508.0Z
subschemaSubentry: CN=Aggregate,CN=Schema,CN=Configuration,DC=cninnovation,
DC=local
dsServiceName: CN=NTDS Settings,CN=MAGELLAN,CN=Servers,
CN=Default-First-Site-Name,CN=Sites,CN=Configuration,DC=cninnovation,DC=local
namingContexts: DC=cninnovation,DC=local
```

```

namingContexts: CN=Configuration,DC=cninnovation,DC=local
namingContexts: CN=Schema,CN=Configuration,DC=cninnovation,DC=local
namingContexts: DC=DomainDnsZones,DC=cninnovation,DC=local
namingContexts: DC=ForestDnsZones,DC=cninnovation,DC=local
defaultNamingContext: DC=cninnovation,DC=local
schemaNamingContext: CN=Schema,CN=Configuration,DC=cninnovation,DC=local
configurationNamingContext: CN=Configuration,DC=cninnovation,DC=local
rootDomainNamingContext: DC=cninnovation,DC=local
supportedControl: 1.2.840.113556.1.4.319
supportedControl: 1.2.840.113556.1.4.801

```

Object Identifier

Every object has a *globally unique identifier* (GUID). A GUID is a unique 128-bit number as you may already know from COM development. You can bind to an object using the GUID. This way, you always get to the same object, regardless of whether the object was moved to a different container. The GUID is generated at object creation and always remains the same.

You can get to a GUID string representation with `DirectoryEntry.NativeGuid`. This string representation can then be used to bind to the object.

This example shows the pathname for a serverless binding to bind to a specific object represented by a GUID:

```
LDAP:///<GUID=14abbd652aae1a47abc60782dcfc78ea>
```

Username

If a different user must be used for accessing the directory (maybe the current user doesn't have the required permissions to access Active Directory), explicit *user credentials* must be specified for the binding process. Active Directory has multiple ways to specify the username.

Downlevel Logon

With a downlevel logon, the username can be specified with the pre-Windows 2000 domain name:

```
domain\username
```

Distinguished Name

The user can also be specified by a distinguished name of a user object, for example:

```
CN=Administrator, CN=Users, DC=thinktecture, DC=local
```

User Principal Name

The *user principal name* (UPN) of an object is defined with the `userPrincipalName` attribute. The system administrator specifies this with the logon information in the Account tab of the User properties with the Active Directory Users and Computers tool. Note that this is not the e-mail address of the user.

This information also uniquely identifies a user and can be used for a logon:

```
Nagel@thinktecture.local
```

Authentication

For secure encrypted authentication, the *authentication* type can also be specified. The authentication can be set with the `AuthenticationType` property of the `DirectoryEntry` class. The value that can be assigned is one of the `AuthenticationTypes` enumeration values. Because the enumeration is marked with the `[Flags]` attribute, multiple values can be specified. Some of the possible values are where the data sent is encrypted; `ReadOnlyServer`, where you specify that you need only read access; and `Secure` for secure authentication.

Binding with the DirectoryEntry Class

The `System.DirectoryServices.DirectoryEntry` class can be used to specify all the binding information. You can use the default constructor and define the binding information with the properties `Path`, `Username`, `Password`, and `AuthenticationType`, or pass all the information in the constructor:

```
var de = new DirectoryEntry();
de.Path = "LDAP://platinum/DC=thinktecture, DC=local";
de.Username = "Christian.Nagel@thinktecture.local";
de.Password = "password";

// use the current user credentials
var de2 = new DirectoryEntry("LDAP://DC=thinktecture, DC=local");
```

Even if the construction of the `DirectoryEntry` object is successful, this doesn't mean that the binding was a success. Binding will happen the first time a property is read to avoid unnecessary network traffic. At the first access of the object, you can see if the object exists and if the specified user credentials are correct.

Getting Directory Entries

Now that you know how to specify the binding attributes to an object in Active Directory, you can move on to read the attributes of an object. In the following example, you read the properties of user objects.

The `DirectoryEntry` class has some properties to get information about the object: the `Name`, `Guid`, and `SchemaClassName` properties. The first time a property of the `DirectoryEntry` object is accessed, the binding occurs, and the cache of the underlying ADSI object is filled. (This is discussed in more detail shortly.) Additional properties are read from the cache, and communication with the server isn't necessary for data from the same object.

In the following example, the user object with the common name Christian Nagel in the organizational unit `thinktecture` is accessed:



```
using (var de = new DirectoryEntry())
{
    de.Path = "LDAP://magellan/CN=Christian Nagel, " +
              "OU=thinktecture, DC=cninnovation, DC=local";

    Console.WriteLine("Name: {0}", de.Name);
    Console.WriteLine("GUID: {0}", de.Guid);
    Console.WriteLine("Type: {0}", de.SchemaClassName);
    Console.WriteLine();

    //...
}
```

code snippet `DirectoryServicesSamples/Program.cs`

An Active Directory object holds much more information, with the information available depending on the type of the object; the `Properties` property returns a `PropertyCollection`. Each property is a collection itself, because a single property can have multiple values; for example, the user object can have multiple phone numbers. In this case, you go through the values with an inner `foreach` loop. The collection returned from `properties[name]` is an object array. The attribute values can be strings, numbers, or other types. Here, just the `ToString()` method is used to display the values:

```
Console.WriteLine("Properties: ");
PropertyCollection properties = de.Properties;
foreach (string name in properties.PropertyNames)
{
    foreach (object o in properties[name])
    {
        Console.WriteLine("{0}: {1}", name, o.ToString());
    }
}
```

In the resulting output, you can see all attributes of the specified user object. Some properties such as `otherTelephone` have multiple values. With this property, many phone numbers can be defined. Some of the property values just display the type of the object, `System.__ComObject`; for example, `lastLogoff`, `lastLogon`, and `ntSecurityDescriptor`. To get the values of these attributes, you must use the ADSI COM interfaces directly from the classes in the `System.DirectoryServices` namespace.

```
Name: CN=Christian Nagel
GUID: 0238fd5c-7e67-48bc-985f-c2f1ccf0f86c
Type: user

Properties:
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: user
cn: Christian Nagel
sn: Nagel
givenName: Christian
distinguishedName: CN=Christian Nagel,OU=thinktecture,DC=cninnovation,DC=local
instanceType: 4
whenCreated: 9/25/2009 12:42:05 PM
whenChanged: 9/25/2009 12:42:05 PM
displayName: Christian Nagel
uSNCreated: System.__ComObject
uSNChanged: System.__ComObject
name: Christian Nagel
objectGUID: System.Byte[]
userAccountControl: 66048
badPwdCount: 0
codePage: 0
countryCode: 0
badPasswordTime: System.__ComObject
lastLogoff: System.__ComObject
lastLogon: System.__ComObject
pwdLastSet: System.__ComObject
primaryGroupID: 513
objectSid: System.Byte[]
accountExpires: System.__ComObject
logonCount: 0
sAMAccountName: christian.nagel
sAMAccountType: 805306368
userPrincipalName: christian.nagel@cninnovation.local
objectCategory: CN=Person,CN=Schema,CN=Configuration,DC=cninnovation,DC=local
dsCorePropagationData: 1/1/1601 12:00:00 AM
ntSecurityDescriptor: System.__ComObject
```

With `DirectoryEntry.Properties`, you can access all properties. If a property name is known, you can access the values directly by name:

```
foreach (string homePage in de.Properties["WWWHomePage"])
    Console.WriteLine("Home page: " + homePage);
```

Object Collections

Objects are stored hierarchically in Active Directory. Container objects contain children. You can enumerate these child objects with the `Children` property of the class `DirectoryEntry`. In the other direction, you can get the container of an object with the `Parent` property.

A user object doesn't have children, so you use an organizational unit in the following example.

Non-container objects return an empty collection with the `Children` property. Get all user objects from the organizational unit `thinktecture` in the domain `explorer.local`. The `Children` property

returns a `DirectoryEntries` collection that collects `DirectoryEntry` objects. You iterate through all `DirectoryEntry` objects to display the name of the child objects:



```
using (var de = new DirectoryEntry())
{
    de.Path = "LDAP://magellan/OU=thinkecture, DC=cninnovation, DC=local";

    Console.WriteLine("Children of {0}", de.Name);
    foreach (DirectoryEntry obj in de.Children)
    {
        Console.WriteLine(obj.Name);
    }
}
```

code snippet DirectoryServicesSamples/Program.cs

When you run the program, the common names of the objects are displayed:

```
Children of OU=thinkecture
OU=Admin
CN=Buddhike de Silva
CN=Christian Nagel
CN=Christian Weyer
CN=Consultants
CN=demos
CN=Dominick Baier
CN=Ingo Rammer
CN=Neno Loye
```

In this example, you see all the objects in the organizational unit: users, contacts, printers, shares, and others. If you want to display only some object types, you can use the `SchemaFilter` property of the `DirectoryEntries` class. The `SchemaFilter` property returns a `SchemaNameCollection`. With this `SchemaNameCollection`, you can use the `Add()` method to define the object types you want to see. Here, you are just interested in seeing the user objects, so `user` is added to this collection:

```
using (var de = new DirectoryEntry())
{
    de.Path = "LDAP://magellan/OU=thinkecture, DC=cninnovation, DC=local";

    Console.WriteLine("Children of {0}", de.Name);
    de.Children.SchemaFilter.Add("user");
    foreach (DirectoryEntry obj in de.Children)
    {
        Console.WriteLine(obj.Name);
    }
}
```

As a result, you see only the user objects in the organizational unit:

```
Children of OU=thinkecture
CN=Christian Nagel
CN=Christian Weyer
CN=Dominick Baier
CN=Ingo Rammer
CN=Jörg Neumann
CN=Richard Blewett
```

Cache

To reduce the network transfers, ADSI uses a cache for the object properties. As mentioned earlier, the server isn't accessed when a `DirectoryEntry` object is created; instead, with the first reading of a value from the directory store, all the properties are written into the cache so that a round trip to the server isn't necessary when the next property is accessed.

Writing any changes to objects changes only the cached object; setting properties doesn't generate network traffic. You must use `DirectoryEntry.CommitChanges()` to flush the cache and to transfer any changed data to the server. To get the newly written data from the directory store, you can use `DirectoryEntry.RefreshCache()` to read the properties. Of course, if you change some properties without calling `CommitChanges()` and do a `RefreshCache()`, all your changes will be lost, because you read the values from the directory service again using `RefreshCache()`.

It is possible to turn off this property cache by setting the `DirectoryEntry.UsePropertyCache` property to `false`. However, unless you are debugging your code, it's better not to turn off the cache because of the extra round trips to the server that will be generated.

Creating New Objects

When you want to create new Active Directory objects — such as users, computers, printers, contacts, and so on — you can do this programmatically with the `DirectoryEntries` class.

To add new objects to the directory, first you have to bind to a container object, such as an organizational unit, where new objects can be inserted — you cannot use objects that are not able to contain other objects. The following example uses the container object with the distinguished name `CN=Users, DC=thinktecture, DC=local`:



```
var de = new DirectoryEntry();
de.Path = "LDAP://magellan/CN=Users, DC=cninnovation, DC=local";
```

code snippet DirectoryServicesSamples/Program.cs

You can get to the `DirectoryEntries` object with the `Children` property of a `DirectoryEntry`:

```
DirectoryEntries users = de.Children;
```

The class `DirectoryEntries` offers methods to add, remove, and find objects in the collection. Here, a new user object is created. With the `Add()` method, the name of the object and a type name are required. You can get to the type names directly using ADSI Edit.

```
DirectoryEntry user = users.Add("CN=John Doe", "user");
```

The object now has the default property values. To assign specific property values, you can add properties with the `Add()` method of the `Properties` property. Of course, all of the properties must exist in the schema for the user object. If a specified property doesn't exist, you'll get a `COMException`: "The specified directory service attribute or value doesn't exist":

```
user.Properties["company"].Add("Some Company");
user.Properties["department"].Add("Sales");
user.Properties["employeeID"].Add("4711");
user.Properties["samAccountName"].Add("JDoe");
user.Properties["userPrincipalName"].Add("JDoe@explorer.local");
user.Properties["givenName"].Add("John");
user.Properties["sn"].Add("Doe");
user.Properties["userPassword"].Add("someSecret");
```

Finally, to write the data to Active Directory, you must flush the cache:

```
user.CommitChanges();
```

Updating Directory Entries

Objects in the Active Directory service can be updated as easily as they can be read. After reading the object, you can change the values. To remove all values of a single property, you can call the method `PropertyValueCollection.Clear()`. You can add new values to a property with `Add()`. `Remove()` and `RemoveAt()` remove specific values from a property collection.

You can change a value simply by setting it to the specified value. The following example uses an indexer for `PropertyValueCollection` to set the mobile phone number to a new value. With the indexer a value

can be changed only if it exists. Therefore, you should always check with `DirectoryEntry.Properties.Contains()` to see if the attribute is available:

```
using (var de = new DirectoryEntry())
{
    de.Path = "LDAP://magellan/CN=Christian Nagel, " +
              "OU=thinktecture, DC=cninnovation, DC=local";

    if (de.Properties.Contains("mobile"))
    {
        de.Properties["mobile"][0] = "+43(664)3434343434";
    }
    else
    {
        de.Properties["mobile"].Add("+43(664)3434343434");
    }

    de.CommitChanges();
}
```

The `else` part in this example uses the method `PropertyValueCollection.Add()` to add a new property for the mobile phone number, if it doesn't exist already. If you use the `Add()` method with already existing properties, the resulting effect would depend on the type of the property (single-value or multivalue property). Using the `Add()` method with a single-value property that already exists results in a `COMException`: "A constraint violation occurred." Using `Add()` with a multivalue property, however, succeeds, and an additional value is added to the property.

The `mobile` property for a user object is defined as a single-value property, so additional mobile phone numbers cannot be added. However, a user can have more than one mobile phone number. For multiple mobile phone numbers, the `otherMobile` property is available. `otherMobile` is a multivalue property that allows setting multiple phone numbers, so calling `Add()` multiple times is allowed. Note that multivalue properties are checked for uniqueness. If the second phone number is added to the same user object again, you get a `COMException`: "The specified directory service attribute or value already exists."



Remember to call `DirectoryEntry.CommitChanges()` after creating or updating new directory objects. Otherwise, only the cache gets updated, and the changes are not sent to the directory service.

Accessing Native ADSI Objects

Often, it is much easier to call methods of predefined ADSI interfaces instead of searching for the names of object properties. Some ADSI objects also support methods that cannot be used directly from the `DirectoryEntry` class. One example of a practical use is the `IADsServiceOperations` interface, which has methods to start and stop Windows services. (For more details on Windows services see Chapter 25, "Windows Services.")

The classes of the `System.DirectoryServices` namespace use the underlying ADSI COM objects, as mentioned earlier. The `DirectoryEntry` class supports calling methods of the underlying objects directly by using the `Invoke()` method.

The first parameter of `Invoke()` requires the method name that should be called in the ADSI object; the `params` keyword of the second parameter allows a flexible number of additional arguments that can be passed to the ADSI method:

```
public object Invoke(string methodName, params object[] args);
```

You can find the methods that can be called with the `Invoke()` method in the ADSI documentation. Every object in the domain supports the methods of the `IADs` interface. The user object that you created previously also supports the methods of the `IADsUser` interface.

In the following example, the method `IADsUser.SetPassword()` changes the password of the previously created user object:

```
using (var de = new DirectoryEntry())
{
    de.Path = "LDAP://magellan/CN=John Doe, CN=Users, DC=cninnovation, DC=local";

    de.Invoke("SetPassword", "anotherSecret");
    de.CommitChanges();
}
```

It is also possible to use the underlying ADSI object directly instead of using `Invoke()`. To use these objects, choose **Project** ➤ **Add Reference** to add a reference to the Active DS Type Library (see Figure 52-9). This creates a wrapper class where you can access these objects in the namespace `ActiveDs`.

The native object can be accessed with the `NativeObject` property of the `DirectoryEntry` class. In the following example, the object `de` is a user object, so it can be cast to `ActiveDs.IADsUser`. `IADsUser.SetPassword()` is a method documented in the `IADsUser` interface, so you can call it directly instead of using the `Invoke()` method. By setting the `AccountDisabled` property of `IADsUser` to `false`, you can enable the account. As in the previous examples, the changes are written to the directory service by calling `CommitChanges()` with the `DirectoryEntry` object:

```
ActiveDs.IADsUser user = (ActiveDs.IADsUser)de.NativeObject;
user.SetPassword("someSecret");
user.AccountDisabled = false;
de.CommitChanges();
```

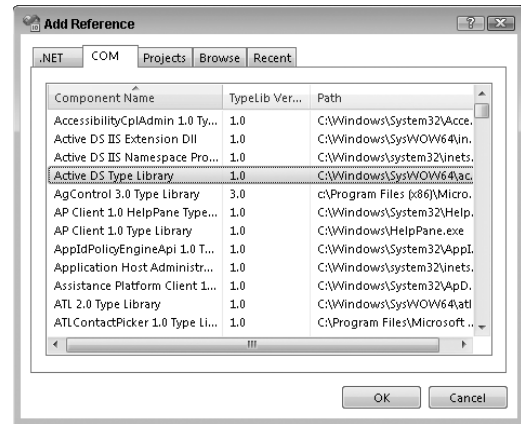


FIGURE 52-9

Beginning with .NET 3.5 the need to invoke the native objects behind the .NET class `DirectoryEntry` is reduced. You can manage users with classes in the namespace `System.DirectoryServices.AccountManagement`. The classes from this namespace are explained later in this chapter.

Searching in Active Directory

Because Active Directory is a data store optimized for *read-mostly* access, you will generally search for values. To search in Active Directory, the .NET Framework provides the `DirectorySearcher` class.

You can use `DirectorySearcher` only with the LDAP provider; it doesn't work with the other providers such as NDS or IIS.

In the constructor of the `DirectorySearcher` class, you can define four important parts for the search: the root where the search starts, the filter, the properties that should be loaded, and the scope of the search. You can also use a default constructor and define the search options with properties.

SearchRoot

The search root specifies where the search should start. The default of `SearchRoot` is the root of the domain you are currently using. `SearchRoot` is specified with the `Path` of a `DirectoryEntry` object.

Filter

The filter defines the values where you want to get hits. The filter is a string that must be enclosed in parentheses.

Relational operators such as `<=`, `=`, and `>=` are allowed in expressions. `(objectClass=contact)` searches all objects of type `contact`; `(lastName>=Nagel)` searches all objects alphabetically where the `lastName` property is equal to or larger than `Nagel`.

Expressions can be combined with the `&` and `|` prefix operators. For example, `(&(objectClass=user)(description=Auth*))` searches all objects of type `user` where the property `description` starts with the string `Auth`. Because the `&` and `|` operators are at the beginning of the expressions, it is possible to combine more than two expressions with a single prefix operator.

The default filter is `(objectClass=*)` so all objects are valid.



The filter syntax is defined in RFC 2254, “The String Representation of LDAP Search Filters.” You can find this RFC at www.ietf.org/rfc/rfc2254.txt.

PropertiesToLoad

With `PropertiesToLoad`, you can define a `StringCollection` of all the properties in which you are interested. Objects can have a lot of properties, most of which will not be important for your search request. You define the properties that should be loaded into the cache. The default properties that are returned if nothing is specified are the path and the name of the object.

SearchScope

`SearchScope` is an enumeration that defines how deep the search should extend:

- `SearchScope.Base` searches only the attributes in the object where the search started, so at most one object is found.
- With `SearchScope.OneLevel`, the search continues in the child collection of the base object. The base object itself is not searched for a hit.
- `SearchScope.Subtree` defines that the search should go down the complete tree.

The default value of the `SearchScope` property is `SearchScope.Subtree`.

Search Limits

A search for specific objects in a directory service can span multiple domains. To limit the search to the number of objects or the time taken, you have some additional properties to define, as shown in the following table.

PROPERTY	DESCRIPTION
ClientTimeout	The maximum time the client waits for the server to return a result. If the server does not respond, no records are returned.
PageSize	With a <i>paged search</i> , the server returns a number of objects defined with the <code>PageSize</code> instead of the complete result. This reduces the time for the client to get a first answer and the memory needed. The server sends a cookie to the client, which is sent back to the server with the next search request so that the search can continue at the point where it finished.
ServerPageTimeLimit	For paged searches, this value defines the time a search should continue to return a number of objects that are defined with the <code>PageSize</code> value. If the time is reached before the <code>PageSize</code> value, the objects that were found up to that point are returned to the client. The default value is -1, which means infinite.
SizeLimit	Defines the maximum number of objects that should be returned by the search. If you set the limit to a value larger than defined by the server (which is 1000), the server limit is used.
ServerTimeLimit	Defines the maximum time the server will search for objects. When this time is reached, all objects that are found up to this point are returned to the client. The default is 120 seconds, and you cannot set the search to a higher value.
ReferralChasing	A search can cross multiple domains. If the root that's specified with <code>SearchRoot</code> is a parent domain or no root was specified, the search can continue to child domains. With this property, you can specify if the search should continue on different servers. <code>ReferralChasingOption.None</code> means that the search does not continue on other servers. The value <code>ReferralChasingOption.Subordinate</code> specifies that the search should go on to child domains. When the search starts at <code>DC=Wrox, DC=com</code> the server can return a result set and the referral to <code>DC=France, DC=Wrox, DC=COM</code>. The client can continue the search in the subdomain. <code>ReferralChasingOption.External</code> means that the server can refer the client to an independent server that is not in the subdomain. This is the default option. With <code>ReferralChasingOption.All</code>, both external and subordinate referrals are returned.
Tombstone	If the property <code>Tombstone</code> is set to <code>true</code> , all deleted objects that match the search are returned, too.
VirtualListView	If large results are expected with the search, the property <code>VirtualListView</code> can be used to define a subset that should be returned from the search. The subset is defined with the class <code>DirectoryVirtualListView</code> .

In the search example, all user objects with a property description value of `Author` are searched in the organizational unit `thinktexture`.

First, bind to the organizational unit `thinktexture`. This is where the search should start. Create a `DirectorySearcher` object where the `SearchRoot` is set. The filter is defined as `(&(objectClass=user)(description=Auth*))`, so that the search spans all objects of type `user` with a description of `Auth` followed by something else. The scope of the search should be a subtree so that child organizational units within `thinktexture` are searched, too:



Available for
download on
Wrox.com

```
using (var de = new DirectoryEntry("LDAP://OU=thinktexture, DC=cninnovation, DC=local"))
using (var searcher = new DirectorySearcher())
{
    searcher.SearchRoot = de;
    searcher.Filter = "(&(objectClass=user)(description=Auth*))";
    searcher.SearchScope = SearchScope.Subtree;
}
```

code snippet DirectoryServicesSamples/Program.cs

The properties that should be in the result of the search are name, description, givenName, and wWWHomePage:

```
searcher.PropertiesToLoad.Add("name");
searcher.PropertiesToLoad.Add("description");
searcher.PropertiesToLoad.Add("givenName");
searcher.PropertiesToLoad.Add("wWWHomePage");
```

You are ready to do the search. However, the result should also be sorted. `DirectorySearcher` has a `Sort` property, where you can set a `SortOption`. The first argument in the constructor of the `SortOption` class defines the property that will be used for a sort; the second argument defines the direction of the sort. The `SortDirection` enumeration has `Ascending` and `Descending` values.

To start the search, you can use the `FindOne()` method to find the first object, or `FindAll()`. `FindOne()` returns a simple `SearchResult`, whereas `FindAll()` returns a `SearchResultCollection`. Here, all authors should be returned, so `FindAll()` is used:

```
searcher.Sort = new SortOption("givenName", SortDirection.Ascending);

SearchResultCollection results = searcher.FindAll();
```

With a `foreach` loop, every `SearchResult` in the `SearchResultCollection` is accessed. A `SearchResult` represents a single object in the search cache. The `Properties` property returns a `ResultPropertyCollection`, where you access all properties and values with the property name and the indexer:

```
SearchResultCollection results = searcher.FindAll();

foreach (SearchResult result in results)
{
    ResultPropertyCollection props = result.Properties;
    foreach (string propName in props.PropertyNames)
    {
        Console.WriteLine("{0}: ", propName);
        Console.WriteLine(props[propName][0]);
    }
    Console.WriteLine();
}
```

It is also possible to get the complete object after a search: `SearchResult` has a `GetDirectoryEntry()` method that returns the corresponding `DirectoryEntry` of the found object.

The resulting output shows the beginning of the list of all thinktecture associates with the properties that have been chosen:

```
givenname: Christian
adspath: LDAP://magellan/CN=Christian Weyer,OU=thinktecture,DC=cninnovation,
DC=local
description: Author
name: Christian Weyer

givenname: Christian
adspath: LDAP://magellan/CN=Christian Nagel,OU=thinktecture,DC=cninnovation,
DC=local
description: Author
name: Christian Nagel

givenname: Dominick
adspath: LDAP://magellan/CN=Dominick Baier,OU=thinktecture,DC=cninnovation,
DC=local
description: Author
name: Dominick Baier
```

```

givenname: Ingo
adspath: LDAP://magellan/CN=Ingo Rammer,OU=thinktecture,DC=cninnovation,
DC=local
description: Author
name: Ingo Rammer

givenname: Jörg
adspath: LDAP://magellan/CN=Jörg Neumann,OU=thinktecture,DC=cninnovation,
DC=local
description: Author
name: Jörg Neumann

```

SEARCHING FOR USER OBJECTS

In this section, you build a WPF application called `UserSearch`. This application is flexible insofar as a specific domain controller, username, and password to access Active Directory can be entered; otherwise, the user of the running process is used. In this application, you access the schema of the Active Directory service to get the properties of a user object. The user can enter a filter string to search all user objects of a domain. It's also possible to set the properties of the user objects that should be displayed.

User Interface

The user interface shows numbered steps to indicate how to use the application (see Figure 52-10):

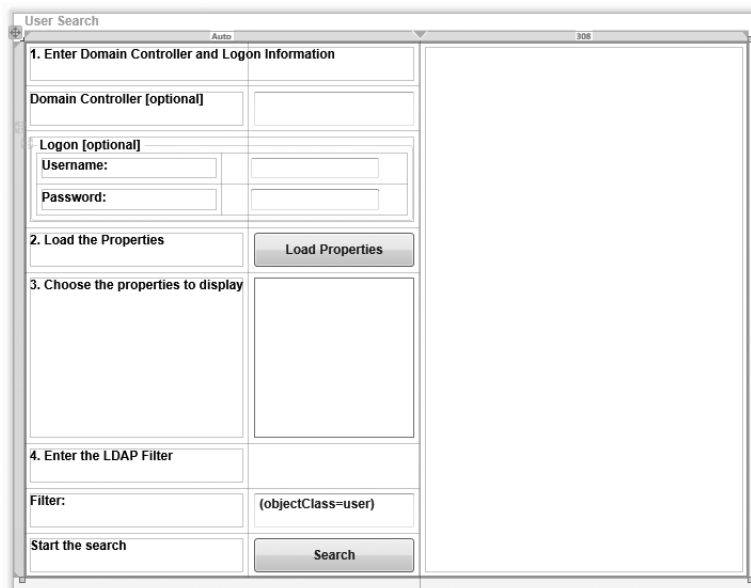


FIGURE 52-10

1. Username, Password, and the Domain Controller can be entered. All this information is optional. If no domain controller is entered, the connection works with serverless binding. If the username is missing, the security context of the current user is taken.
2. A button allows all the property names of the user object to be loaded dynamically in the `listBoxProperties` list box.

3. The properties to be displayed can be selected. The `SelectionMode` of the list box is set to `MultiSimple`.
4. The filter to limit the search can be entered. The default value set in this dialog box searches for all user objects: `(objectClass=user)`.
5. Now the search can start.

Get the Schema Naming Context

This application has only two handler methods: one method for the button to load the properties and one to start the search in the domain. First, you read the properties of the `user` class dynamically from the schema to display it in the user interface.

In the handler `buttonLoadProperties_Click()` method, `SetLogonInformation()` reads the username, password, and host name from the dialog box and stores them in members of the class. Next, the method `SetNamingContext()` sets the LDAP name of the schema and the LDAP name of the default context. This schema LDAP name is used in the call to set the properties in the list box: `SetUserProperties()`.



```
private void OnLoadProperties(object sender, RoutedEventArgs e)
{
    try
    {
        SetLogonInformation();

        SetNamingContext();

        SetUserProperties(schemaNamingContext);
    }
    catch (Exception ex)
    {
        MessageBox.Show(String.Format("check your input! {0}", ex.Message));
    }
}
```

code snippet UserSearch/MainWindow.xaml.cs

In the helper method `SetNamingContext()`, you are using the root of the directory tree to get the properties of the server. You are interested in the value of only two properties: `schemaNamingContext` and `defaultNamingContext`.

```
private void SetNamingContext()
{
    using (var de = new DirectoryEntry())
    {
        string path = "LDAP://" + hostname + "rootDSE";
        de.Username = username;
        de.Password = password;
        de.Path = path;
        schemaNamingContext = de.Properties["schemaNamingContext"][0].ToString();
        defaultNamingContext = de.Properties["defaultNamingContext"][0].ToString();
    }
}
```

Get the Property Names of the User Class

You have the LDAP name to access the schema. You can use this to access the directory and read the properties. You are interested in not only the properties of the `user` class but also those of the base

classes of user: Organizational-Person, Person, and Top. In this program, the names of the base classes are hard-coded. You could also read the base class dynamically with the `subClassOf` attribute.

`GetSchemaProperties()` returns `IEnumerable<string>` with all property names of the specific object type. All the property names are added to the list box:



```
private void SetUserProperties(string schemaNamingContext)
{
    var properties = from p in GetSchemaProperties(schemaNamingContext,
                                                    "User").Concat(
                                                                GetSchemaProperties(schemaNamingContext,
                                                                "Organizational-Person"))
                                                                .Concat(
                                                                GetSchemaProperties(schemaNamingContext, "Top"))
    orderby p
    select p;

    listBoxProperties.DataContext = properties;
}
```

code snippet UserSearch/MainWindow.xaml.cs

In `GetSchemaProperties()`, you are accessing the Active Directory service again. This time, `rootDSE` is not used but rather the LDAP name of the schema that you discovered earlier. The property `systemMayContain` holds a collection of all attributes that are allowed in the class `objectType`:

```
private IEnumerable<string> GetSchemaProperties(string schemaNamingContext,
                                                string objectType)
{
    IEnumerable<string> data;
    using (var de = new DirectoryEntry())
    {
        de.Username = username;
        de.Password = password;

        de.Path = String.Format("LDAP://{0}CN={1},{2}", hostname, objectType,
                                schemaNamingContext);

        PropertyValueCollection values = de.Properties["systemMayContain"];
        data = from s in values.Cast<string>()
                orderby s
                select s;
    }
    return data;
}
```

Step 2 in the application is completed. The `ListBox` control has all the property names of the user objects.

Search for User Objects

The handler for the search button calls only the helper method `FillResult()`:



```
private void OnSearch(object sender, RoutedEventArgs e)
{
    try
    {
        FillResult();
    }
}
```

```

    }
    catch (Exception ex)
    {
        MessageBox.Show(String.Format("check your input! {0}", ex.Message));
    }
}

```

code snippet UserSearch/MainWindow.xaml.cs

In `FillResult()`, you do a normal search in the complete Active Directory Domain as you saw earlier. `SearchScope` is set to `Subtree`, the `Filter` to the string you get from a `TextBox` object, and the properties that should be loaded into the cache are set by the values the user selected in the list box. The `PropertiesToLoad` property of the `DirectorySearcher` is of type `StringCollection` where the properties that should be loaded can be added using the `AddRange()` method that requires a string array. The properties that should be loaded are read from the `ListBox listBoxProperties` with the property `SelectedItems`. After setting the properties of the `DirectorySearcher` object, the properties are searched by calling the `SearchAll()` method. The result of the search inside the `SearchResultCollection` is used to generate summary information that is written to the text box `textBoxResults`:

```

private void FillResult()
{
    using (var root = new DirectoryEntry())
    {
        root.Username = username;
        root.Password = password;
        root.Path = String.Format("LDAP://{0}/{1}",
                                hostname, defaultNamingContext);
        using (var searcher = new DirectorySearcher())
        {
            searcher.SearchRoot = root;
            searcher.SearchScope = SearchScope.Subtree;
            searcher.Filter = textFilter.Text;
            searcher.PropertiesToLoad.AddRange(
                listBoxProperties.SelectedItems.Cast<string>().ToArray());

            SearchResultCollection results = searcher.FindAll();
            var summary = new StringBuilder();
            foreach (SearchResult result in results)
            {
                foreach (string propName in result.Properties.PropertyNames)
                {
                    foreach (object p in result.Properties[propName])
                    {
                        summary.AppendFormat(" {0}: {1}", propName, p);
                        summary.AppendLine();
                    }
                }
                summary.AppendLine();
            }
            textResult.Text = summary.ToString();
        }
    }
}

```

Starting the application gives you a list of all objects where the filter is valid (see Figure 52-11).

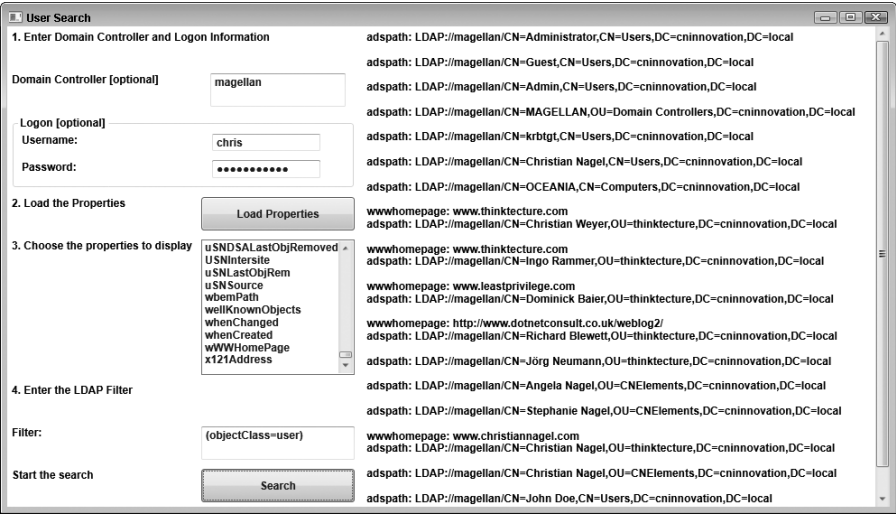


FIGURE 52-11

ACCOUNT MANAGEMENT

Previous to .NET 3.5, it was difficult to create and modify user and group accounts. One way to do that was by using the classes from the `System.DirectoryServices` namespace, or by using the strongly typed native COM interfaces. New since .NET 3.5 is the assembly `System.DirectoryServices.AccountManagement` that offers an abstraction to the `System.DirectoryServices` classes by offering specific methods and properties to search, modify, create, and update users and groups.

The classes and their functionality are explained in the following table.

CLASS	DESCRIPTION
<code>PrincipalContext</code>	With the <code>PrincipalContext</code> , you configure the context of the account management. Here, you can define if an Active Directory Domain, the accounts from the local system, or an application directory should be used. You set this by setting the <code>ContextType</code> enumeration to one of the values <code>Domain</code> , <code>Machine</code> , or <code>ApplicationDirectory</code> . Depending on the context type, you can also define the name of the domain and specify a username and password that are used for access.
<code>Principal</code>	<code>Principal</code> is the base class of all principals. With the static method <code>FindByIdentity()</code> , you can get a <code>Principal</code> identity object. With a principal object, you have access to various properties such as name, description, distinguished name, and the object type from the schema. If you need more control over the principal than is available from the properties and methods of this class, you can use the method <code>GetUnderlyingType()</code> , which returns the underlying <code>DirectoryEntry</code> object.
<code>AuthenticablePrincipal</code>	<code>AuthenticablePrincipal</code> derives from <code>Principal</code> and is the base class for all principals that can be authenticated. There are several static methods to find principals, such as by logon or lockout times, by incorrect password attempts, or by password set time. Using instance methods, you can change the password and unlock an account.

continues

(continued)

CLASS	DESCRIPTION
UserPrincipalComputerPrincipal	UserPrincipal and ComputerPrincipal derive from the base class AuthenticablePrincipal and, thus, have all properties and methods the base class has. UserPrincipal is the object that maps to a user account, and ComputerPrincipal maps to a computer account. With UserPrincipal, you have many properties to get and set information about the user, for example, EmployeeId, EmailAddress, GivenName, and VoiceTelephoneNumber.
GroupPrincipal	Groups cannot authenticate; that's why GroupPrincipal derives directly from the Principal class. With GroupPrincipal, you can get members of the group with the Members property and the GetMembers() method.
PrincipalCollection	The PrincipalCollection contains a group of Principal objects; for example, the Members property from the GroupPrincipal class returns a PrincipalCollection object.
PrincipalSearcher	PrincipalSearcher is an abstraction of the DirectorySearcher class with special use for account management. With PrincipalSearcher, there's no need to know about the LDAP query syntax because this is created automatically.
PrincipalSearchResult<T>	Search methods from the PrincipalSearcher and Principal classes return a PrincipalSearchResult<T>.

The following sections look at some scenarios in which you can use the classes from the System.DirectoryServices.AccountManagement namespace.

Display User Information

The static property Current of the UserPrincipal class returns a UserPrincipal object with information about the currently logged-on user:



```
using (var user = UserPrincipal.Current)
{
    Console.WriteLine("Context Server: {0}", user.Context.ConnectedServer);
    Console.WriteLine(user.Description);
    Console.WriteLine(user.DisplayName);
    Console.WriteLine(user.EmailAddress);
    Console.WriteLine(user.GivenName);
    Console.WriteLine("{0:d}", user.LastLogon);
}
```

code snippet AccountManagementSamples/Program.cs

Running the application displays information about the user:

```
Context Server: Magellan.cninnovation.local
Developer, Author, Trainer, Consultant
Christian Nagel
Christian@ChristianNagel.com
Christian
2009/09/25
```

Create a User

You can use the UserPrincipal class to create a new user. First a PrincipalContext is required to define where the user should be created. With the PrincipalContext, you set the ContextType to an

enumeration value of `Domain`, `Machine`, or `ApplicationDirectory`, depending on whether the directory service, the local accounts of the machine, or an application directory should be used. If the current user does not have access to add accounts to Active Directory, you can also set a user and password with the `PrincipalContext` that is used to access the server.

Next, you can create an instance of `UserPrincipal` passing the principal context, and setting all required properties. Here, the `GivenName` and `EmailAddress` properties are set. Finally, you must invoke the `Save()` method of the `UserPrincipal` to write the new user to the store:



```
using (var context = new PrincipalContext(ContextType.Domain, "cninnovation"))
using (var user = new UserPrincipal(context, "Tom", "P@ssw0rd", true)
    {
        GivenName = "Tom",
        EmailAddress = "test@test.com"
    })
{
    user.Save();
}
```

code snippet AccountManagementSamples/Program.cs

Reset a Password

To reset a password from an existing user, you can use the `SetPassword()` method from a `UserPrincipal` object:



```
using (var context = new PrincipalContext(ContextType.Domain, "cninnovation"))
using (var user = UserPrincipal.FindByIdentity(context, IdentityType.Name,
    "Tom"))
{
    user.SetPassword("Pa$$w0rd");
    user.Save();
}
```

code snippet AccountManagementSamples/Program.cs

The user running this code needs to have the privilege to reset a password. To change the password from an old one to a new one, you can use the method `ChangePassword()`.

Create a Group

A new group can be created in a similar way to creating a new user. Here, just the class `GroupPrincipal` is used instead of the class `UserPrincipal`. As in creating a new user, the properties are set, and the `Save()` method is invoked:



```
using (var ctx = new PrincipalContext(ContextType.Domain, "cninnovation"))
using (var group = new GroupPrincipal(ctx)
    {
        Description = "Sample group",
        DisplayName = "Wrox Authors",
        Name = "WroxAuthors"
    })
{
    group.Save();
}
```

code snippet AccountManagementSamples/Program.cs

Add a User to a Group

To add a user to a group, you can use a `GroupPrincipal` and add a `UserPrincipal` to the `Members` property of the group. To get an existing user and group, you can use the static method `FindByIdentity()`:



```
using (var context = new PrincipalContext(ContextType.Domain))
using (var group = GroupPrincipal.FindByIdentity(
    context, IdentityType.Name, "WroxAuthors"))
using (var user = UserPrincipal.FindByIdentity(
    context, IdentityType.Name, "Stephanie Nagel"))
{
    group.Members.Add(user);
    group.Save();
}
```

code snippet AccountManagementSamples/Program.cs

Finding Users

Static methods of the `UserPrincipal` object allow finding users based on some predefined criteria. The sample here shows finding users who didn't change their passwords within the last 30 days by using the method `FindPasswordSetTime()`. This method returns a `PrincipalSearchResult<UserPrincipal>` collection that is iterated to display the user name, the last logon time, and the time when the password was reset:



```
using (var context = new PrincipalContext(ContextType.Domain, "explorer"))
using (var users = UserPrincipal.FindByPasswordSetTime(context,
    DateTime.Today - TimeSpan.FromDays(30), MatchType.LessThan))
{
    foreach (var user in users)
    {
        Console.WriteLine("{0}, last logon: {1}, " +
            "last password change: {2}", user.Name, user.LastLogon,
            user.LastPasswordSet);
    }
}
```

code snippet AccountManagementSamples/Program.cs

Other methods offered by the `UserPrincipal` class to find users are `FindByBadPasswordAttempt()`, `FindByExpirationTime()`, `FindByLockoutTime()`, and `FindByLogonTime()`.

You can get more flexibility in finding users by using the `PrincipalSearcher` class. This class is an abstraction of the `DirectorySearcher` class and uses this class behind the scenes. With the `PrincipalSearcher` class, you can assign any `Principal` object to the `QueryFilter` property.

In the example here, a `UserPrincipal` object with the properties `Surname` and `Enabled` is set to the `QueryFilter`. This way, all user objects starting with the surname Nag and that are enabled are returned with the `PrincipalSearchResult` collection. The `PrincipalSearcher` class creates an LDAP query string to do the search.



```
var context = new PrincipalContext(ContextType.Domain);

var userFilter = new UserPrincipal(context);
userFilter.Surname = "Nag*";
userFilter.Enabled = true;

using (var searcher = new PrincipalSearcher())
{
    searcher.QueryFilter = userFilter;
    var searchResult = searcher.FindAll();
    foreach (var user in searchResult)
```

```

    {
        Console.WriteLine(user.Name);
    }
}

```

code snippet AccountManagementSamples/Program.cs

DSML

With the namespace `System.DirectoryServices.Protocols`, you can access Active Directory through DSML (Directory Services Markup Language). DSML is a standard defined by the OASIS group (www.oasis-open.org) that allows you to access directory services through a Web service. To make Active Directory available through DSML, you must have at least Windows Server 2003 R2.

Figure 52-12 shows a configuration scenario with DSML. A system that offers DSML services accesses Active Directory via LDAP. On the client system, the DSML classes from the namespace `System.DirectoryServices.Protocols` are used to make SOAP requests to the DSML service.

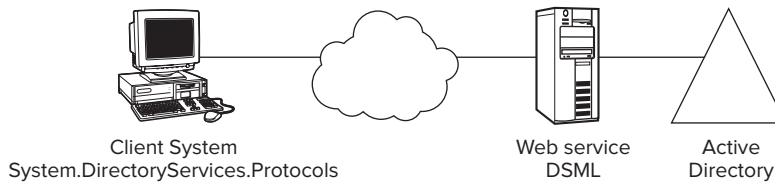


FIGURE 52-12

Classes in System.DirectoryServices.Protocols

The following table shows the major classes in the `System.DirectoryServices.Protocols` namespace.

CLASS	DESCRIPTION
<code>DirectoryConnection</code>	<code>DirectoryConnection</code> is the base class of all the connection classes that can be used to define the connection to the directory service. The classes that derive from <code>DirectoryConnection</code> are <code>LdapConnection</code> (for using the LDAP protocol), <code>DsmlSoapConnection</code> , and <code>DsmlSoapHttpConnection</code> . With the method <code>SendRequest</code> , a message is sent to the directory service.
<code>DirectoryRequest</code>	A request that can be sent to the directory service is defined by a class that derives from the base class <code>DirectoryRequest</code> . Depending on the request type, classes such as <code>SearchRequest</code> , <code>AddRequest</code> , <code>DeleteRequest</code> , and <code>ModifyRequest</code> can be used to send a request.
<code>DirectoryResponse</code>	The result that is returned with a <code>SendRequest</code> is of a type that derives from the base class <code>DirectoryResponse</code> . Examples for derived classes are <code>SearchResponse</code> , <code>AddResponse</code> , <code>DeleteResponse</code> , and <code>ModifyResponse</code> .

Searching for Active Directory Objects with DSML

This section looks at an example of how a search for directory services objects can be performed. As you can see in the code that follows, first a `DsmlSoapHttpConnection` object is instantiated that defines the connection to the DSML service. The connection is defined with the class `DsmlDirectoryIdentifier` that contains an `Uri` object. Optionally, the user credentials can be set with the connection:



```

Uri uri = new Uri("http://dsmlserver/dsml");
var identifier = new DsmlDirectoryIdentifier(uri);

var credentials = new NetworkCredential();
credentials.UserName = "cnagel";

```

```

credentials.Password = "password";
credentials.Domain = "explorer";

var dsmlConnection = new DsmlSoapHttpConnection(identifier, credentials);

```

code snippet DsmlSample/Program.cs

After the connection is defined, the search request can be configured. The search request consists of the directory entry where the search should start, an LDAP search filter, and the definition of what property values should be returned from the search. Here, the filter is set to `(objectClass=user)`, so that all user objects are returned from the search. `attributesToReturn` is set to null, and you can read all attributes that have values. `SearchScope` is an enumeration in the namespace `System.DirectoryServices.Protocols` that is similar to the `SearchScope` enumeration in the namespace `System.DirectoryServices` used to define how deep the search should go. Here, the `SearchScope` is set to `Subtree` to walk through the complete Active Directory tree.

The search filter can be defined with an LDAP string or by using an XML document contained in the `XmlDocument` class:

```

string distinguishedName = null;
string ldapFilter = "(objectClass=user)";
string[] attributesToReturn = null; // return all attributes

var searchRequest = new SearchRequest(distinguishedName,
    ldapFilter, SearchScope.Subtree, attributesToReturn);

```

After the search is defined with the `SearchRequest` object, the search is sent to the Web service by calling the method `SendRequest`. `SendRequest` is a method of the `DsmlSoapHttpConnection` class. `SendRequest` returns a `SearchResponse` object where the returned objects can be read.

As an alternative to invoking the synchronous `SendRequest` method, the `DsmlSoapHttpConnection` class also offers the asynchronous methods `BeginSendRequest` and `EndSendRequest`, which conform to the asynchronous .NET pattern.



The asynchronous pattern is explained in Chapter 20, "Threads, Tasks, and Synchronization."

```

SearchResponse searchResponse =
    (SearchResponse) dsmlConnection.SendRequest(searchRequest);

```

The returned Active Directory objects can be read within the `SearchResponse`. `SearchResponse.Entries` contains a collection of all entries that are wrapped with the type `SearchResultEntry`. The `SearchResultEntry` class has the `Attributes` property that contains all attributes. Each attribute can be read with help of the `DirectoryAttribute` class.

In the code example, the distinguished name of each object is written to the console. Next, the attribute values for the organizational unit (OU) are accessed, and the name of the organizational unit is written to the console. After this, all values of the `DirectoryAttribute` objects are written to the console:

```

Console.WriteLine("\r\nSearch matched {0} entries:",
    searchResponse.Entries.Count);
foreach (SearchResultEntry entry in searchResponse.Entries)
{
    Console.WriteLine(entry.DistinguishedName);

    // retrieve a specific attribute
    DirectoryAttribute attribute = entry.Attributes["ou"];
    Console.WriteLine("{0} = {1}", attribute.Name, attribute[0]);
}

```



```
// retrieve all attributes
foreach (DirectoryAttribute attr in entry.Attributes.Values)
{
    Console.WriteLine("{0}=", attr.Name);

    // retrieve all values for the attribute
    // the type of the value can be one of string, byte[] or Uri
    foreach (object value in attr)
    {
        Console.WriteLine("{0} ", value);
    }
    Console.WriteLine();
}
}
```

Adding, modifying, and deleting objects can be done similarly to searching objects. Depending on the action you want to perform, you can use the corresponding classes.

SUMMARY

This chapter discussed the architecture of Active Directory: the important concepts of domains, trees, and forests. You can access information in the complete enterprise. When writing applications that access Active Directory services, you must be aware that the data you read might not be up to date because of the replication latency.

The classes in the `System.DirectoryServices` namespaces give you easy ways to access Active Directory services by wrapping to the ADSI providers. The `DirectoryEntry` class makes it possible to read and write objects directly in the data store.

With the `DirectorySearcher` class, you can perform complex searches and define filters, timeouts, properties to load, and a scope. By using the global catalog, you can speed up the search for objects in the complete enterprise, because it stores a read-only version of all objects in the forest.

DSML is another API that allows accessing the Active Directory through a Web service interface.

Classes in `System.DirectoryServices.AccountManagement` offer an abstraction to make it easier to create and modify user, group, and computer accounts.

