Take advantage of IBM Tivoli Directory Server's LDAP Controls via Java and JNDI

Subtree deletion, password policy and more

Stefan Zoerner (szoerner@de.ibm.com)
Advisory IT Architect
IBM Germany

Skill Level: Intermediate

Date: 11 Jun 2005

Controls allow the LDAP protocol to be extended without changing the protocol itself. This article provides information about some important controls implemented by IBM Tivoli Directory Server. It describes what controls are, and presents the API portion of JNDI which deals with them. With the help of examples, starring the *Tree Delete Control* and the *Password Policy Control*, it demonstrates how to employ controls in arbitrary Java components using JNDI.

Introduction

Like many other directory solutions, IBM Tivoli Directory Server supports LDAP, the Lightweight Directory Access Protocol. Within LDAP v3, there are three standardized ways to extend the functionality of a server:

1. SASL (Simple Authentication and Security Layer) Mechanisms
2. Extended Operations
3. Controls

The SASL framework (RFC 2222) allows support for different authentication methods. An extended operation is used to define a completely new operation, while a control modifies or enhances the behaviour of an existing one. The "Server-side Sorting" control (RFC 2891) for instance is an extension of the search operation. Normally, LDAP search results are returned by the server in an arbitrary order. The control provides functionality to sort the members of a search result on the server according to certain attributes, ascending or descending.

Using LDAP Controls

1.3.18...? (OIDs, Object Identifiers)

OIDs (object identifiers) are strings composed by dot-separated numbers. They are constructed hierarchically; new OIDs are formed by appending

a number to an existing OID. The permission to do so is given to the organization or person that owns or originated the OID. The OID 1.3.18 for instance belongs to IBM, which can allocate arbitrary OIDs below it. An OID is assigned to each LDAP control (the same is true for several other objects in a directory).

LDAP Controls may be sent as part of a request from a client to the server (request control), or may be sent together with a result from the server back to the client (response control). It is even possible to have both controls in a single operation. The LDAP specification (RFC 2251) defines how to append controls to messages, and which general information must be provided when doing so. The specification does not contain specific controls. Their definition is normally provided by software vendors, though some controls have become de-facto standards (or even RFCs), and are supported by different products.

In addition to the OID (see sidebar "1.3.18...?") of the control and optional control specific parameters, a client also indicates whether the functionality of the control is critical or not. Normally an LDAP server does not support all controls. In the case where the server does not implement a requested control, but the client marked the control as critical, the whole operation fails. If the unsupported control is not marked as critical, the server performs the operation as usual (i.e. without the added functionality).

Each LDAP v3 compliant server publishes its supported controls via attribute values of the Root DSE (Root DSA-specific Entry). The same is true for extended operations and SASL mechanisms. In the case of controls, the attribute `supportedcontrol` is used, its values can be fetched with an appropriate LDAP search. Listing 1 contains such a search with the command line tool ldapsearch. Because the scope "base" is used and no base DN is stated in the options, the Root DSE is returned. As a result the tool prints out the OIDs of the supported controls of the addressed server.

Listing 1. Examine controls supported by an LDAP server

```
$ ldapsearch -h magritte -p 389 -s base "(objectclass=*)" supportedcontrol

supportedcontrol=2.16.840.1.113730.3.4.2
supportedcontrol=1.3.18.0.2.10.5
supportedcontrol=1.2.840.113556.1.4.473
supportedcontrol=1.2.840.113556.1.4.319
supportedcontrol=1.3.6.1.4.1.42.2.27.8.5.1
supportedcontrol=1.2.840.113556.1.4.805
supportedcontrol=2.16.840.1.113730.3.4.18
supportedcontrol=1.3.18.0.2.10.15
supportedcontrol=1.3.18.0.2.10.18
$
```

Controls supported by IBM Tivoli Directory Server

The examination of supported controls as shown above works for all standards-compliant LDAP servers -- but of course the output will differ. Listing 1 displays the result for an IBM Tivoli Directory Server 5.2. Table 1 introduces some of the controls. The product documentation (e.g. Appendix F of "IBM Directory Server C-Client SDK

Programming Reference") provides information about all nine of them (OID, usage, parameters). Tivoli Directory Server 6.0 provides even more controls.

Table 1. Selected controls supported by IBM Tivoli Directory Server

OID	Name	Short description of functionality
1.2.840.113556.1.4.805	Tree Delete	Deletes an entire subtree of a container entry. Defined in an expired Internet Draft.
1.2.840.113556.1.4.319	Simple Paged Results	Allows a client to control the rate at which an LDAP server returns the results of an LDAP search operation. Defined in RFC 2696.
1.2.840.113556.1.4.473	Server-side Sorting	Allows a client to receive search results sorted by a list of criteria, where each criterion represents a sort key. Defined in RFC 2891.
2.16.840.1.113730.3.4.2	Manage DSAIT	Allows manipulation of referral and other special objects as normal objects. Defined in RFC 3296.
1.3.6.1.4.1.42.2.27.8.5.1	Password policy	Contains various warnings and errors associated with password policy. Defined in an expired Internet Draft.

In the following sections, some of the control functionality described above will be used within Java components using JNDI.

Controls in the Java Naming and Directory Interface

The Java Naming and Directory Interface (JNDI) provides access to different kinds of naming and directory services. It offers a common interface to connect to several naming services, including RMI , CORBA, DNS, LDAP servers, file systems and even the Windows registry. In order to integrate all these different services, and still remain extensible, JNDI makes use of a plugin architecture with so-called service providers. JNDI has been part of the Java 2 Standard Edition (J2SE) since version 1.3. It is widely available with the runtime environment, and the natural choice to interact with an LDAP server from Java. For older JDK versions (1.1 and above) it is possible to download and install JNDI as a standard extension.

The classes and interfaces of JNDI reside in different packages. The `javax.naming` package contains the fundamental interfaces for accessing naming services (the `Context` interface is of particular importance). A name service permits the binding of an object to a name (e.g. in DNS, an IP address is bound to a domain name). Once bound, the object can be retrieved using the name. The `javax.naming.directory` package provides classes and interfaces to access directory services. The primary difference between naming services and directory services is that directory services support reading and writing the attributes of bound objects. Directory services also permit searches for objects whose attribute values meet certain criteria. Finally the `javax.naming.spi` package includes classes and interfaces for the implementation of service providers (spi stands for Service Provider Interface).

Listing 2 presents a JNDI example which prints out the values of the attribute `supportedcontrol` within the Root DSE of an arbitrary LDAP server. The output is identical to Listing 1. Connection data (service provider, LDAP URL) is provided when the `InitialDirContext` is created, in form of a `Hashtable` object. Alternatively, the configuration can be done via the environment, e.g. with the file jndi.properties (here is an example). The JNDI documentation and the JNDI Tutorial (see resources) describe the different configuration options in detail.

Listing 2. Examine controls supported by an LDAP server (JNDI version)

```
package dw.samples.ldapcontrols;

import java.util.Hashtable;

import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.Attribute;
import javax.naming.directory.Attributes;
import javax.naming.directory.InitialDirContext;

public class ShowSupportedControls {

    public static void main(String[] args) throws NamingException {

        Hashtable env = new Hashtable();
        env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
        env.put(Context.PROVIDER_URL, "ldap://magritte:389/");

        InitialDirContext ctx = new InitialDirContext(env);
        Attributes attrs = ctx.getAttributes("", new String[] { "supportedcontrol" });

        Attribute attr = attrs.get("supportedcontrol");
        for (int i = 0; i < attr.size(); ++i) {
            System.out.println(attr.getID()+"="+attr.get(i));
        }
    }
}
```

The package `javax.naming.ldap` contains LDAP v3 specific features, including interfaces which describe extended operations and controls. The fact that JNDI provides extra classes and interfaces for LDAP in a specific package illustrates how important the LDAP directory service is.

Figure 1. Control portion of JNDI/LDAP

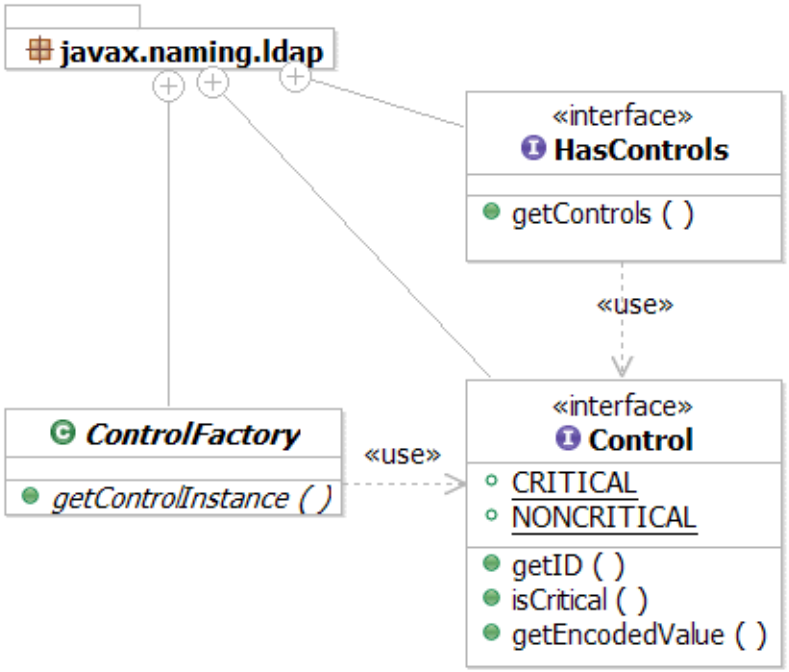


Figure 1 shows the classes and interfaces within JNDI, which deal with LDAP controls. No surprise, the interface `Control` is essential. All request and response controls are instances of classes, which implement `Control`. The method `getId` returns the OID of the control. With `isCritical` a client can state whether the support of the control is significant for processing. Finally `getEncodedValue` provides data specific to the control, e.g. sort criteria in case of the "Server-side Sorting". The value is ASN.1 BER encoded (see below). In the case of controls that don't have any specific data or parameters, the method returns `null`.

An Example: Tree Delete Control

By default, LDAP compliant directories forbid the deletion of entries with children. It is only permitted to delete leaf entries. However the Tree Delete Control provided by IBM Tivoli Directory Server 5.2/6.0 extends the delete operation and allows the removal of sub trees within a directory using a single delete request. In addition to Tivoli Directory Server this control is also supported by Microsoft Active Directory.

When using JNDI methods to delete an entry within a directory, it is also not possible to remove whole subtrees at once. This is an expected behaviour because the underlying JNDI service provider uses LDAP. It is also possible to utilize the described LDAP control within JNDI to overcome this limitation. In order to demonstrate this, Listing 3 contains a Java class which implements the JNDI interface `Control` (see above). The method `getId` provides the OID of the Tree Delete Control. `isCritical` signals that the client considers the support of the control as critical. This property could be made configurable, but in this case a constant

value is adequate. Since the definition of the control does not provide specific parameters, the method `getEncodedValue` returns `null`.

Listing 3. A simple class for the Tree Delete Control

```
package dw.samples.ldapcontrols;

import javax.naming.ldap.Control;

public class TreeDeleteControl implements Control {

    public String getID() {
        return "1.2.840.113556.1.4.805";
    }

    public boolean isCritical() {
        return Control.CRITICAL;
    }

    public byte[] getEncodedValue() {
        return null;
    }
}
```

In order to use the control within a Java component the `DirContext` (the extension of `javax.naming.Context` for directory services) must be replaced with the subclass specialized for LDAP (`javax.naming.ldap.LdapContext`). The construction of the corresponding `InitialLdapContext` in Listing 4 is done with parameters from the environment to save space (default constructor, jndi.properties), take care that a principal with appropriate privileges is used. The example demonstrates the use of the control class of Listing 3 to remove an entry (leaf or subtree). The entry is retrieved via a `lookup` call. Before deletion, an object of the class `TreeDeleteControl` is attached to the context as a request control. It is therfore sent with the delete operation (`unbind`) to the server and ensures that the complete subtree is removed, i.e. the entry itself and all its potential subnodes.

If the addressed server does not implement the Tree Delete Control, an `OperationNotSupportedException` is raised (LDAP error code 12, "unavailable critical extension"). In case that the control has not been marked as critical, but the entry to be removed has child entries, the deletion fails as well. The result would be a `ContextNotEmptyException` (LDAP error code 66, "not allowed on non-leaf"). Because the behaviour for a non-critical control would differ on the same server depending on whether the entry is a leaf or not, the control is marked as critical in this case. This avoids a situation where the removal of an entry sometimes works and sometimes fails.

Listing 4. Use of the Tree Delete Control

```
LdapContext ctx = new InitialLdapContext();

LdapContext entry = (LdapContext) ctx.lookup("ou=subtree");
entry.setRequestControls(new Control[] { new TreeDeleteControl() });
entry.unbind("");
```

Do we lose portability?

Not all LDAP servers support the Tree Delete Control (e.g. OpenLDAP, Sun Java System Directory Server do not). In general, the use of vendor specific controls raises the risk of creating client code which cannot be used uniformly with arbitrary servers. We will use the Tree Delete Control to demonstrate an elegant detection of the absence of functionality and the execution of alternate code in such a case. This approach may be used as a blueprint for other controls as well.

As described above, the absence of a critical control causes the JNDI service provider to throw a specific exception. In Listing 5, the exception is caught and handled by invoking the `deleteRecursively` method, which implements the removal of complete subtrees on its own with multiple delete requests. Before removing the entry given as parameter, the method calls itself recursively for all immediate child nodes. Therefore it is certain that all children have been removed, and the entry is a leaf which can safely be deleted.

Listing 5. Use of the Tree Delete Control with fallback

```
package dw.samples.ldapcontrols;

import javax.naming.Binding;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;
import javax.naming.OperationNotSupportedException;
import javax.naming.ldap.Control;
import javax.naming.ldap.InitialLdapContext;
import javax.naming.ldap.LdapContext;

public class PortableDeletion {

    public static void main(String[] args) throws NamingException {

        LdapContext ctx = new InitialLdapContext();
        LdapContext entry = (LdapContext) ctx.lookup("ou=subtree");
        entry.setRequestControls(new Control[] { new TreeDeleteControl() });
        try {
            entry.unbind("");
            System.out.println("Entry " + entry.getNameInNamespace()
                + " deleted");
        } catch (OperationNotSupportedException e) {
            entry.setRequestControls(new Control[0]);
            deleteRecursively(entry);
        }
    }

    public static void deleteRecursively(LdapContext entry)
            throws NamingException {

        NamingEnumeration enum = entry.listBindings("");
        while (enum.hasMore()) {
            Binding b = (Binding) enum.next();
            if (b.getObject() instanceof LdapContext) {
                deleteRecursively((LdapContext) b.getObject());
            }
        }
        entry.unbind("");
        System.out.println("Entry " + entry.getNameInNamespace() + " deleted");
    }
}
```

This approach preserves the portability of the client code, and in cases where the Tree Delete Control is supported, such as Tivoli Directory Server and Active Directory, the client may benefit from the server-specific (and normally more efficient) operation.

Using Control classes provided by Sun's JNDI Provider

The implementation of the control class for the Tree Delete Control was rather simple. This was primarily due to the fact that the control doesn't need any specific data (the information that the control should be used is sufficient for the server to perform the task). Furthermore no response control had to be implemented. If the request control must transport specific data, or a response control returns results which have to be evaluated and provided to the caller, the implementation is more complex. Such an example will be described later on (Password Policy Control).

Fortunately it is not always necessary to create custom Java classes for an LDAP control in order to benefit from it. In many cases existing implementations may be used. This is particularly true for two very helpful controls related to searches supported by IBM Tivoli Directory Server: Sorted Search and Paged Result Sets. The first allows a client to receive search results sorted by a list of criteria (normally, LDAP search results are provided by the server in arbitrary order). The second one allows management of the amount of data returned from a search request.

JNDI/LDAP Booster Pack and J2SE 5.0 (Tiger)

A special class library for JNDI and LDAP has long been available from Sun on the JNDI product page; the so-called JNDI/LDAP Booster Pack. It contains numerous classes for controls of various directory vendors. Implementations for popular LDAP extensions and helper classes for group operations in directories (e.g. adding and removing members) are included as well. The Booster pack is therefore useful apart from its support for LDAP controls as well.

The control classes in the Booster pack integrate seamlessly in the JNDI framework, their implementation corresponds to the concepts shown above, and so does their usage. It is only necessary to set the classpath, which must contain the relevant jar file (ldapbp.jar, which stands for LDAP Booster Pack).

Some LDAP controls are so popular, it was decided to include the corresponding Java classes directly in the Java 2 Standard Edition (J2SE), as part of the Tiger release (version 5.0). Table 2 provides an overview of all LDAP controls which are supported directly with Java classes in the Booster Pack and/or J2SE 5.0. Among these are several controls, which are implemented by IBM Tivoli Directory Server 5.2 and 6.0, for instance "Proxied Authorization" and "Manage DSAIT".

Table 2. Overview of controls and control classes within Booster Pack and J2SE

Short name	OID	Class in JNDI/LDAP Booster Pack 1.0	Class in J2SE 5.0 (Tiger)

Authorization ID	2.16.840.1.113730.3.4.16	`com.sun.jndi.ldap.ctl.AuthorizationIDControl`	-
Dir Sync	1.2.840.113556.1.4.841	`com.sun.jndi.ldap.ctl.DirSyncControl`	-
Get Effective Rights	1.3.6.1.4.1.42.2.27.9.5.2	`com.sun.jndi.ldap.ctl.GetEffectiveRightsControl`	-
Manage DSAIT	2.16.840.1.113730.3.4.2	-	javax.naming.ldap.ManageReferralControl
Password Expired	2.16.840.1.113730.3.4.4	`com.sun.jndi.ldap.ctl.PasswordExpiredResponseControl`	-
Password Expiring	2.16.840.1.113730.3.4.5	`com.sun.jndi.ldap.ctl.PasswordExpiringResponseControl`	-
Proxied Authorization	2.16.840.1.113730.3.4.18	`com.sun.jndi.ldap.ctl.ProxiedAuthorizationControl`	-
Real Attributes Only	2.16.840.1.113730.3.4.17	`com.sun.jndi.ldap.ctl.RealAttributesOnlyControl`	-
Server-side Sorting	1.2.840.113556.1.4.473	`com.sun.jndi.ldap.ctl.SortControl`	javax.naming.ldap.SortControl
Simple Paged Results	1.2.840.113556.1.4.319	`com.sun.jndi.ldap.ctl.PagedResultsControl`	javax.naming.ldap.PagedResultsControl
Tree Delete	1.2.840.113556.1.4.805	`com.sun.jndi.ldap.ctl.TreeDeleteControl`	-
Virtual Attributes Only	2.16.840.1.113730.3.4.19	`com.sun.jndi.ldap.ctl.VirtualAttributesOnlyControl`	-
Virtual List View	2.16.840.1.113730.3.4.9	`com.sun.jndi.ldap.ctl.VirtualListViewControl`	-

The table provides links to the javadoc pages for the J2SE implementations. For the two searching-related controls mentioned above, the javadoc contains code snippets for their usage. Examples for "Server-side Sorting" and "Manage DSAIT" are also contained in the JNDI Tutorial. The rest of this article focuses on "Password Policy", which is functionality specific to IBM Tivoli Directory Server, from which one can also benefit with the help of an LDAP control. Note "Password Policy" is missing in the table above. "Password Expired" and "Password Expiring" are distinct controls and are not supported by IBM Tivoli Directory Server. However their functionality is covered by "Password Policy".

Password policy in IBM Tivoli Directory Server

Passwords play an important role in IT security. In order to protect them better it is quite common that users are prevented by the system administration from choosing weak passwords. Passwords are required to have a certain complexity (e.g. minimum length or mixture of letters, numbers and special characters). In addition users are often forced to change their passwords periodically (e.g. at least every three months), and prevented from reusing recently used passwords ("password history"). After a certain number of failed authentication attempts, an account may be locked by the system.

In many cases directories are used as user registries, which usually implies the storage of passwords. IBM Tivoli Directory Server supports a password policy allowing the flexible customization of rules regarding password security. Among the features are

- Enforcing password syntax rules
- Maintaning a password history for each user
- Locking an account after a sequence of unsuccessful bind attempts
- Encryption of stored passwords

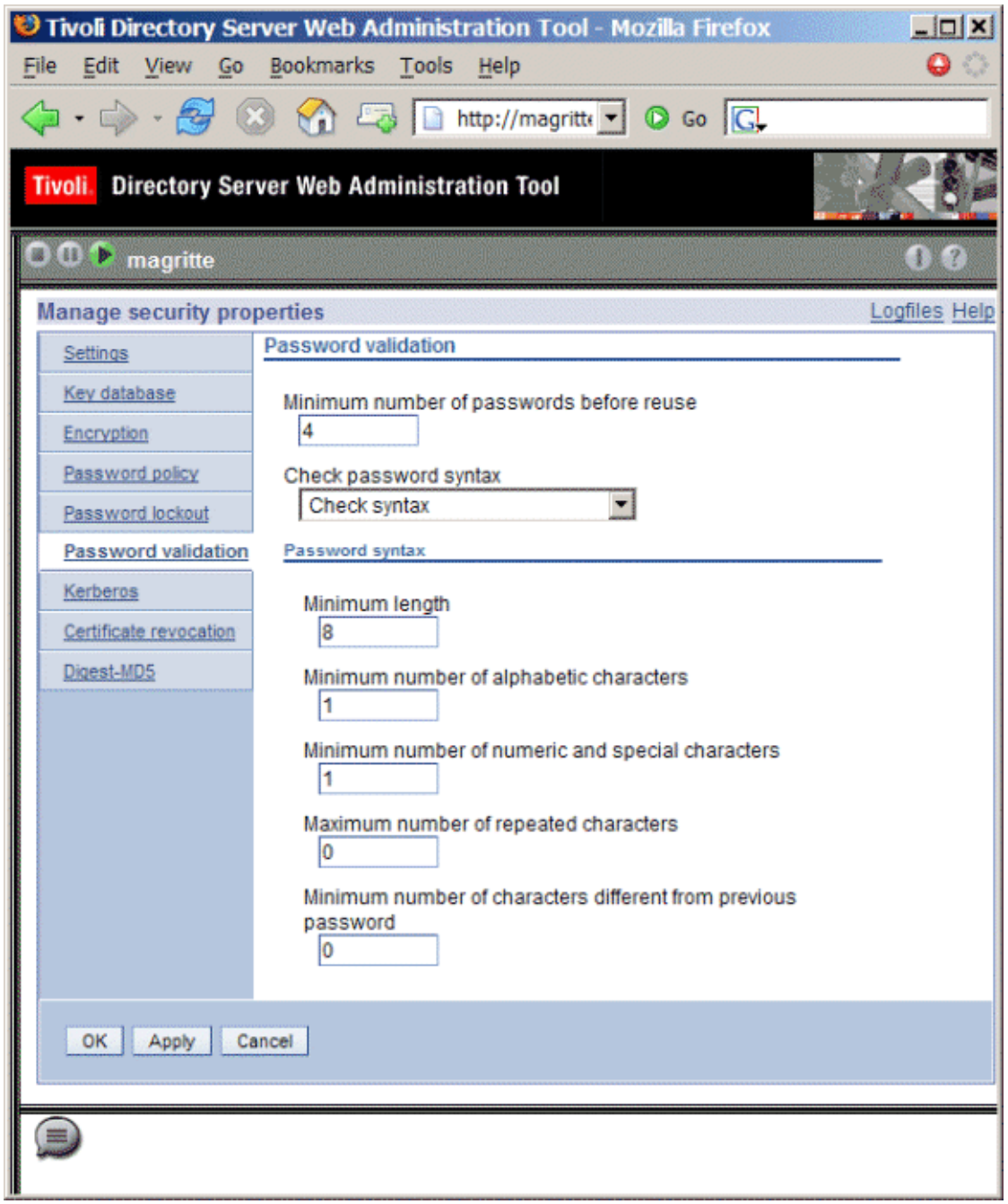
Configuration of Password Policy in Tivoli Directory Server

There are two options to configure password policy:

- Interactive (Tivoli Directory Server Web Administration Tool)
- Command line (creation of an LDIF file)

Figure 2 shows the form of the webbased administration tool to maintain password validation properties (password syntax rules). Among these are the length of the password history (4 in this case), the minumum length of a password (8) and the character mix. Further changes are possible in the "Password policy" and the "Password lockout" screens. Detailed information about this tool is included in the product documentation.

Figure 2. Configuring password policies within the Web Administration Tool



The second configuration option is the creation of an LDIF file with modifications (LDIF stands for LDAP Data Interchange Format, the format is defined in RFC 2849, which also contains examples) followed by its application to the server via the command line tool ldapmodify. Listing 6 displays such a configuration file. The settings for password policies are stored in the special entry "CN=PWDPOLICY". The LDIF file contains several modifications of its attribute values, some of them corresponding to those in Figure 2. There are various other attributes available for fine-grained customization, described in the product documentation (IBM Tivoli Directory Server Administration Guide, chapter Securing the directory).

Listing 6. An LDIF file for password policy configuration

```
dn: CN=PWDPOLICY
changetype: modify
replace: ibm-pwdpolicy
ibm-pwdpolicy: true
-
replace: pwdchecksyntax
pwdchecksyntax: 2
-
replace: pwdMustChange
pwdMustChange: false
-
replace: pwdinhistory
pwdinhistory: 4
-
replace: pwdminlength
pwdminlength: 8
-
replace: passwordminalphachars
passwordminalphachars: 1
-
replace: passwordminotherchars
passwordminotherchars: 1
-
# maximum password age in seconds (90 days)
replace: pwdmaxage
pwdmaxage: 7776000
```

Listing 7 presents a command line call to apply the configuration to the server. Subsequent to the configuration of the password policies an easy way to demonstrate their effect are the client command line tools, which are included in the IBM Tivoli Directory Server package.

Listing 7. Applying the configuration to the server

```
$ ldapmodify -D cn=root -w ? -f enablePwdPolicy.ldif
Enter password ==>
modifying entry CN=PWDPOLICY
```

Testing the configuration via command line

Listing 8 displays some calls for a normal user with DN "cn=Stefan Zoerner,dc=example,dc=com" and password "Start001" (command line options -D and -w). Note that it does not make sense to employ a user with administrative rights in this example (e.g. "cn=root"), because the password policies don't apply for such users (i.e. he/she may set any password to an arbitrary value). IBM Tivoli Directory Server 6.0 allows specifying a separate password policy for administrative IDs

Listing 8. Demonstration of password policy capabilities with command line tools

```
$ ldapsearch -D "cn=Stefan Zoerner,dc=example,dc=com"
             -w Start001 -s base (objectclass=*) port
ldap_simple_bind:  Warning, time before expiration is 404011
port=389

$ ldapchangepwd -D "cn=Stefan Zoerner,dc=example,dc=com" -w Start001 -n My1Secret
ldap_simple_bind:  Warning, time before expiration is 403966
changing password for entry cn=Stefan Zoerner,dc=example,dc=com

$ ldapchangepwd -D "cn=Stefan Zoerner,dc=example,dc=com" -w My1Secret -n 1234567
changing password for entry cn=Stefan Zoerner,dc=example,dc=com
Constraint violation --- Error, Password too short

$ ldapchangepwd -D "cn=Stefan Zoerner,dc=example,dc=com" -w My1Secret -n Start001
changing password for entry cn=Stefan Zoerner,dc=example,dc=com
Constraint violation --- Error, Password in History
```

ASN.1 and BER

ASN.1 (Abstract Syntax Notation One) is a common notation for describing abstract types and values. The specific data of an LDAP request and/or response control is normally defined in this notation (see Listing 9 for an example).

BER (Basic Encoding Rules for ASN.1) is a set of rules for representing ASN.1 objects as octet strings. The `getEncodedValue` method of a JNDI Control returns the value encoded like this as a byte array.

Unfortunately, Java does not provide direct support for encoding and decoding BER, but there a several class libraries availabe for this task. See the Resources for more information on ASN.1, BER and related Java technologies.

The first call is a simple, non-anonymous search request. During authentication (bind) the server provides a warning, saying that the password of the user will expire in approximately five days (404011 seconds). The same warning appears during the second operation, which successfully changes the password to the new value "My1Secret" via the command line tool ldapchangepwd. After that, the listing contains two unsuccessful attempts to change the password of the user to the values "1234567" and "Start001" respectively. The first value is too short (7 characters instead of 8 or more), while the second one repeats a value from the password history.

The Password Policy Control

IBM Tivoli Directory Servers uses a control in conjunction with password policy functionality. The OID of the control is "1.3.6.1.4.1.42.2.27.8.5.1". The request control does not have any custom data (it includes OID and critical option only). It is sent along with a request (e.g. bind, search) to demand the corresponding response control in addition to the result from the server. The response control value contains the relevant information. The control has already been silently used in the command line examples in Listing 8 to carry errors and warnings from the server to the client (the program ldapchangepwd or ldapsearch respectively).

Listing 9 contains the syntax definition of the response control value in ASN.1 notation (see sidebar "ASN.1 and BER"). It is taken from Appendix F of "IBM Directory Server C-Client SDK Programming Reference," which contains the syntax definitions for all controls supported by IBM Tivoli Directory Server. The response of Password Policy is a sequence of two optional elements, the warning and the error part. Normally only one of them is present, but in certain circumstances both are present. An empty sequence indicates that no error or warning occurred.

Listing 9. Password Policy: Syntax defintion of response control value in ASN.1

```
PasswordPolicyResponseValue ::= SEQUENCE {
  warning [0] CHOICE OPTIONAL {
    timeBeforeExpiration [0] INTEGER (0 .. maxInt),
    graceLoginsRemaining [1] INTEGER (0 .. maxInt)
  }
  error [1] ENUMERATED OPTIONAL {
    passwordExpired       (0),
    accountLocked         (1),
    changeAfterReset      (2),
    passwordModNotAllowed (3),
    mustSupplyOldPassword (4),
    invalidPasswordSyntax (5),
    passwordTooShort      (6),
    passwordTooYoung      (7),
    passwordInHistory     (8)
  }
}
```

The warning part contains either the number of seconds until the user's password will expire (`timeBeforeExpiration`), or the number of times the server is willing to tolerate the use of the expired password (`graceLoginsRemaining`). The error part contains a value between 0 and 8 to denote the specific error situation. Table 3 briefly explains their meaning.

Table 3. Error Codes of Password Policy Response Control

Value	Name	Meaning
0	passwordExpired	The password is expired, i.e. its age exceeded the maximum configured in attribute `pwdmaxage` of the "CN=PWDPOLICY" entry.
1	accountLocked	The account is locked due to consecutive unsuccessful logon attempts (fine-grained configuration possible with attributes `pwdLockout`, `pwdMaxFailure`, `pwdFailureCountInterval` and `pwdLockoutDuration`).
2	changeAfterReset	The password must be changed by the user in order to perform arbitrary operations (she/he is allowed to bind and modify her/his password). Configuration is done via attribute `pwdMustChange`.
3	passwordModNotAllowed	The user is not allowed to change his/her password (attribute `pwdAllowUserChange`).
4	mustSupplyOldPassword	The old password must be sent along with the new one to modify it (attribute `pwdSafeModify`).
5	invalidPasswordSyntax	The quality of the password is insufficient, i.e. it does not adhere to the configured syntax rules (attributes `passwordMinAlphaChars`, `passwordMinOtherChars` and `passwordMaxRepeatedChars`)
6	passwordTooShort	The new password is not long enough with respect to attribute `pwdMinLength`.
7	passwordTooYoung	The minimum time between password changes (attribute `pwdMinAge`) is violated. This feature is used to prevent old passwords being reused by changing the password repeatedly until the desired value is out of the password history and can be reused again.
8	passwordInHistory	The new password has already been used. The length of the password history is configured in attribute `pwdInHistory`.

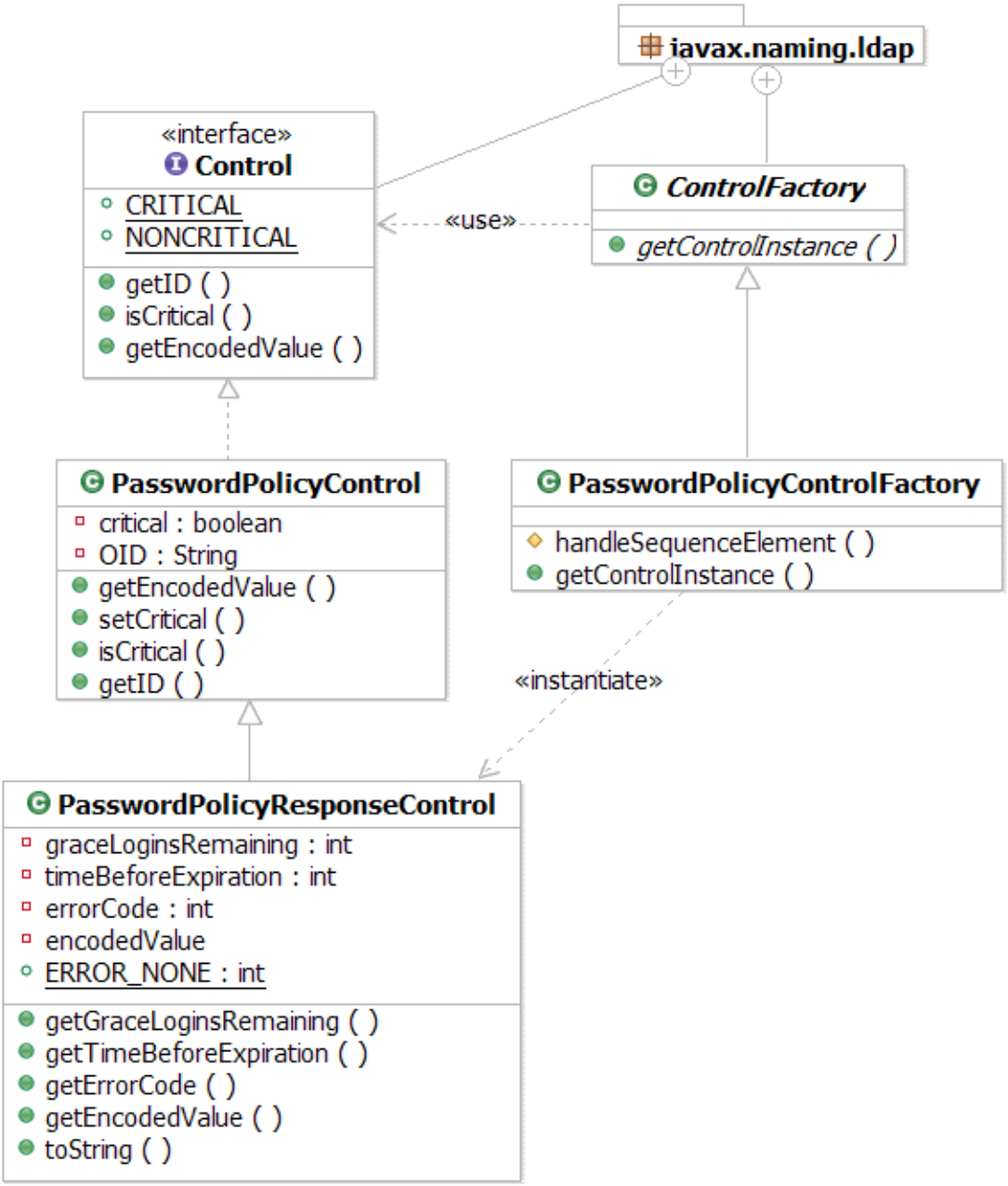
It is clear that using the information returned by the Password Policy Control will add value. Imagine a web application that uses IBM Tivoli Directory Server as a user registry. After a successful logon the system may display a warning ("Your password will expire in 3 days.") and recommend an action ("Would you like to change it?"). If desired, the system may also present detailed messages in case of an unsuccessful authentication ("Your account is locked."). During password changes, it is also possible to give the user hints in case of a failure (see error codes 5-8 in Table 3.). The client command line tools provided by the product have built-in support

(cf. Listing 8) for this information. But how can one benefit from the Password Policy Control within Java custom application development?

Implementation of Java classes for the password policy control

Unfortunately, Password Policy is not in the set of controls which are supported by J2SE 5.0 and the LDAP Booster pack (cf. Table 2). In order to employ it within a Java component, custom Java classes have to be created. These classes must implement the corresponding JNDI interfaces. The class diagram in Figure 3 gives an overview of an implementation of such classes.

Figure 3. Overview of the implementation



The class `PasswordPolicyControl` implements `javax.naming.ldap.Control`. Its objects are used as request controls, furthermore the class acts as superclass for `PasswordPolicyResponseControl`. Because the request control of Password Policy does not carry along any specific value, the implementation is as simple as the `TreeDeleteControl` example (cf. Listing 3). `getID` returns the corresponding OID value of the control, `getEncodedValue` return `null`. However, unlike the `TreeDeleteControl` class, the `critical` property is configurable (via constructor and setter).

The class `PasswordPolicyResponseControl` serves as response control and extends the request control with the control specific data, as defined in Listing 9. This is accomplished by the attributes `timeBeforeExpiration`, `graceLoginsRemaining` and `errorCode`, all of which have getters and (protected) setters. `getEncodedValue` returns the unchanged value of the response control as byte array, the other getters and the methods `hasError` and `hasWarning` allow convenient access to the state. The method `toString` provides a textual representation of the control value. In particular, it includes the message text for any error code (e.g. "password expired") that might be present.

The only challenging task within the implementation is the decoding of the BER encoded value of the response control in order to set the attributes (e.g. error code). This is done by the class `PasswordPolicyControlFactory`, which extends `javax.naming.ldap.ControlFactory` and therefore integrates in the JNDI framework. The component will be invoked by the JNDI LDAP service provider when of a response control is received. Initially, the response control contains only the BER encoded data. The control factory decides whether it would like to transform the control object into a more specific object. This is done by the method `getControlInstance`, which is invoked with the original response control as parameter (see Listing 10). First it checks whether the OID of the passed control matches with the one from Password Policy Control. If not, the method returns `null` and signals that the factory is not responsible for this control (in this case the service provider calls other control factories). Otherwise an object of class `PasswordPolicyResponseControl` is created.

Listing 10. Fragment of class PasswordPolicyControlFactory

```
package dw.samples.ldapcontrols;

import java.io.ByteArrayInputStream;
import java.io.IOException;

import javax.naming.ldap.Control;
import javax.naming.ldap.ControlFactory;

import netscape.ldap.ber.stream.*;

public class PasswordPolicyControlFactory extends ControlFactory {

    public Control getControlInstance(Control ctl) {
        Control result = null;

        if (ctl.getID().equals(PasswordPolicyControl.OID)) {
            try {
                PasswordPolicyResponseControl rctl =
                        new PasswordPolicyResponseControl();
                if (ctl.getEncodedValue() != null) {
                    rctl.setEncodedValue(ctl.getEncodedValue());
                    int[] bread = { 0 };
                    BERSequence seq = (BERSequence)
                        BERElement.getElement(new SpecificTagDecoder(),
                           new ByteArrayInputStream(ctl.getEncodedValue()), bread);

                    for (int i = 0; i < seq.size(); i++) {
                        handleSequenceElement(seq.elementAt(i), rctl);
                    }
```

```
                }
                result = rctl;
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return result;
    }

    protected void handleSequenceElement(BERElement element,
            PasswordPolicyResponseControl target) {
        // ... (omitted)
    }
}
```

If the control object passed to the factory method contains control specific data (`getEncodedValue()` is not `null`), the byte array must be decoded. This is best done using a class library, since the J2SE does not provide the functionality, and implementation from scratch would be costly and error-prone. Some alternatives for this purpose are listed in the resources at the end of the article. When choosing the library please check whether it is appropriate for your application (license, maturity). IBM Tivoli Directory Server ships with its own BER package ("IBMLDAPJavaBer.jar") which is unfortunately not officially supported for external use. The sample code also includes a control factory which uses this package (see download, file PasswordPolicyControlFactoryIbmBer.java).

The example implementation presented here uses Netscape Directory SDK for Java, which provides suitable classes in package `netscape.ldap.ber.stream`. In order to decode data, a subclass of `BERTagDecoder` must be implemented (`SpecificTagDecoder`). The actual decomposing according to the ASN.1 definition (Listing 9) occurs primarily in the method `handleSequenceElement`. This method inspects an individual sequence element, determines whether it is an error or a warning, interprets the data, and assigns the results to the created control object (parameter target). Due to space limitations, details are omitted, but the complete source code is available for download. Please note that in order to compile and run the example the jar file "ldapjdk.jar" from the Netscape Directory SDK for Java is needed.

Demonstration of functionality

Listing 11 shows the use of the control classes implemented above within a Java component (a class with main method in this case). Within the sample code a user with DN "cn=Stefan Zoerner,dc=example,dc=com" logs on to the system and tries to change his/her password several times. In order to benefit from the Password Policy Control two things have to be done. First a request control has to be created and sent along with the requests. This is done by assigning an array containing the control object to the context (see creation of `InitialLdapContext` and method (`setRequestControls`). Second, the control factory has to be declared within the JNDI settings in order to transform response controls for the Password Policy Control to PasswordPolicyResponseControl objects. A special JNDI property (`LdapContext.CONTROL_FACTORIES`, the value is "java.naming.factory.control") is used

for that. The value of the property may contain several control factories as a colon-separated list. For the LDAP Booster Pack control classes in Table 2, an appropriate factory exists within the same package.

The method `handleResponseControls` in the example just iterates over the response controls it finds, and prints them out. Due to a verbose `toString` method within the `PasswordPolicyResponseControl` class, the relevant information is visible on the console (see Listing 12).

Listing 11. Sample code that uses the Control class

```
package dw.samples.ldapcontrols;

import java.util.Hashtable;

import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.BasicAttribute;
import javax.naming.directory.DirContext;
import javax.naming.directory.ModificationItem;
import javax.naming.ldap.Control;
import javax.naming.ldap.InitialLdapContext;
import javax.naming.ldap.LdapContext;

public class TestSetPassword {

    public static final String USER_DN  = "cn=Stefan Zoerner,dc=example,dc=com";
    public static final String USER_PWD = "Start001";

    public static void main(String[] args) throws NamingException {

        Control[] rctls = { new PasswordPolicyControl(true) };

        Hashtable env = new Hashtable();
        env.put(Context.INITIAL_CONTEXT_FACTORY,
                "com.sun.jndi.ldap.LdapCtxFactory");
        env.put(Context.PROVIDER_URL, "ldap://magritte:389/dc=example,dc=com");
        env.put(Context.SECURITY_PRINCIPAL, USER_DN);
        env.put(Context.SECURITY_CREDENTIALS, USER_PWD);

        env.put(LdapContext.CONTROL_FACTORIES,
                "dw.samples.ldapcontrols.PasswordPolicyControlFactory");

        System.out.println("logon on with DN " + USER_DN);
        LdapContext ctx = new InitialLdapContext(env, rctls);
        handleResponseControls(ctx);

        String[] passwords = new String[] {
                "My1Secret",
                "abc001",    // too short
                "01234567",  // numbers only
                USER_PWD     // current password

        };

        ctx.setRequestControls(rctls);
        for (int i = 0; i < passwords.length; i++) {
            ModificationItem replacePw = new ModificationItem(
                    DirContext.REPLACE_ATTRIBUTE, new BasicAttribute(
                            "userPassword", passwords[i]));

            try {
                System.out.println("Try to set password to ["
```

```
                    + passwords[i] + "]");
                ctx.modifyAttributes("cn=Stefan Zoerner",
                        new ModificationItem[] { replacePw });
            } catch (NamingException ne) {
                System.out.println(ne);
            }
            handleResponseControls(ctx);
        }

    }

    public static void handleResponseControls(LdapContext ctx)
            throws NamingException {

        Control[] ctls = ctx.getResponseControls();
        for (int j = 0; ctls != null && j < ctls.length; j++) {
            System.out.println(ctls[j]);
        }
    }
}
```

Listing 12 contains a sample output for the program in Listing 11. Results of the `toString` method of the `PasswordPolicyResponseControl` are highlighted. Of course it would also be possible to react to the response controls other than just printing out the message. After casting to the concrete type, the getters offer convenient access to attributes like `errorCode` or `timeBeforeExpiration`.

Listing 12. Sample output for Listing 11

```
logon on with DN cn=Stefan Zoerner,dc=example,dc=com
PasswordPolicyResponseControl,
warning: time before expiration is 431981
Try to set password to [My1Secret]
PasswordPolicyResponseControl (no error, no warning)
Try to set password to [abc001]
javax.naming.directory.InvalidAttributeValueException:
[LDAP: error code 19 - Constraint Violation]; remaining name 'cn=Stefan Zoerner'
PasswordPolicyResponseControl, error: password too short
Try to set password to [01234567]
javax.naming.directory.InvalidAttributeValueException:
[LDAP: error code 19 - Constraint Violation]; remaining name 'cn=Stefan Zoerner'
PasswordPolicyResponseControl, error: invalid password syntax
Try to set password to [Start001]
javax.naming.directory.InvalidAttributeValueException:
[LDAP: error code 19 - Constraint Violation]; remaining name 'cn=Stefan Zoerner'
PasswordPolicyResponseControl, error: password in history
```

If an LDAP bind operation should be performed by the construction of an `InitialLdapContext` (e.g. to implement logon functionality), and this creation fails, there is no way to examine the response controls. An exception is thrown, and there is no context object available to invoke `getResponseControls`. It is not possible in this situation to learn from a Password Policy Control that the account used for the bind is locked. One option is to bind anonymously (or with a technical user) first, and afterward use the context obtained by this call to rebind with the actual credentials. When the request control for Password Policy is set before the rebind, the corresponding response control is available after catching possible exceptions. The download code contains an example (file "FailedBindWithControls.java"). Unfortunately, this workaround is not always applicable.

Conclusion

In this article some interesting LDAP extensions provided by IBM Tivoli Directory Server with the help of controls were introduced. It was demonstrated how they can be used from Java/JNDI. An existing package (LDAP Booster Pack) and standard components (J2SE 5.0) as well as custom application development methods were presented. An area needing particular attention is the encoding and decoding of control specific data (ASN.1 BER), which was done in the last example in order to interpret the response control value. Because most LDAP controls use this type of encoding for their payload, the example can also be used as a blueprint for other controls with custom data that must be integrated with a JNDI program. Applying the final example (Password Policy Control) in an application which uses Tivoli Directory Server for authentication can provide a better user experience.

Acknowledgements

The author would like to thank Manuel Burckas, Mark McConaughy, Philipp Schoepf, Jason Todoroff, Stephan Waespe and Julius for providing feedback, contributing their ideas and reviewing this article.

Downloads

Description	Name	Size	Download method
Java source code of listings	t-controlSamples.zip	10 KB	HTTP

Information about download methods

Resources

- IBM Tivoli Directory Server is a powerful, security-rich and standards-compliant enterprise directory for corporate intranets and the Internet.
- *Understanding LDAP - Design and Implementation*, an IBM Redbook, provides an introduction to LDAP and an overview on IBM Tivoli Directory server.
- The IBM Tivoli Directory Server Documentation offers detailed information about the product.
- The JNDI Tutorial by Sun Microsystems contains information about accessing LDAP servers with JNDI as well as examples for the usage of LDAP controls.
- At the JNDI Product Page the JNDI/LDAP Booster Pack, which contains specific classes for controls, is avalaible for download.
- In his note A Layman's Guide to a Subset of ASN.1, BER, and DER Burton Kaliski gives an introduction to ASN.1 and BER.
- The Apache Directory Project started a subproject for encoding and decoding ASN.1 data structures in Java.
- The Netscape Directory SDK for Java also contains classes for encoding and decoding of BER data. Its source code is now available at the Mozilla Directory SDK project, Sun Microsystems offers binary distributions for download (Sun ONE Directory SDK for Java 4.1).
- Another Option is the Novell Developer Kit (LDAP Classes for Java, aka JLDAP) which provides Java classes for ASN.1 and BER en-/decoding as well. Source code of this package is available through the OpenLDAP project.
- This technote describes the operational attributes for password policy in IBM Tivoli Directory Server 5.1 and 5.2.

About the author

Stefan Zoerner

Stefan Zoerner is an Advisory IT Architect at IBM e-business Innovation Center in Hamburg, Germany. Since 1996, he has been occupied intensively with Java technologies. In 2004, Stefan published a German LDAP book for Java developers. Currently he uses IBM Tivoli Directory Server in a portal development project.