Access Manager 5.0
SDK Guide

March 2021

Legal Notice

Contents

About this Book and the Library

This document explains how to incorporate various security management features of NetIQ Access Manager with your proprietary applications. Unlike many software development kits (SDK) that rely on application programming interfaces to expose application functionality, this component primarily leverages how Access Manager extends existing Liberty Alliance, OASIS, SAML, and other specifications in defining and exchanging user identities.

This document will be updated as new functionality is released for developers to enhance the capabilities of Access Manager with your own applications and Web services.

Intended Audience

The audience for this documentation includes advanced network security software engineers and experienced network administrators who understand the Liberty Alliance, Java* development, and secure networking issues to enforce the security requirements the Liberty Alliance.

Specifically, you should have advanced understanding of Internet protocols such as:

- Extensible Markup Language (XML)
- Simple Object Access Protocol (SOAP)
- Security Assertion Markup Language (SAML)
- Public Key Infrastructure (PKI) digital signature concepts and Internet security
- Secure Socket Layer/Transport Layer Security (SSL/TSL)
- Hypertext Transfer Protocol (HTTP and HTTPS)
- Uniform Resource Identifiers (URI)
- Domain Name System (DNS)
- Web Services Description Language (WSDL)

Other Information in the Library

You can access other information resources in the library at the following locations:

- Access Manager Product Documentation (https://www.microfocus.com/documentation/access-manager/index.html)
- Access Manager Developer Resources (https://www.microfocus.com/documentation/access-manager/developer-documentation-5.0/)

NOTE: Contact namsdk@microfocus.com for any query related to Access Manager SDK.

---

1 Getting Started

NetIQ Access Manager provides a component-based framework for building secure federated identity network applications based on Liberty Alliance project standards. This framework is designed to help developers make a rapid transition into Liberty's architecture.

The Liberty components enable the convenience of single sign-on and secure business-to-employee, business-to-customer, and business-to-business relationships across a variety of applications within a trusted Web services model. All components are standards-based and designed for maximum interoperability.

In this Chapter

1.1 Development Overview

This SDK describes how to design a flexible and expandable access management system to enable your applications to interact with the identity management capabilities of Access Manager, including federation, provisioning, and the secure delivery of identity information (user name and password, and X.509 certificates) to client-based applications.

The SDK is designed for those who want to develop new applications or integrate existing applications with the standards-based security architecture of Access Manager. It allows NetIQ partners and third-party developers to do the following:

- Leverage the identity management and policy capabilities of the product.
- Provide access to various product features, including:
  - Liberty-based federated identity
  - Secure credential exchange
  - User provisioning services
  - Authentication and authorization methods and policies
  - SAML assertion generation and processing

NOTE: To coordinate the development of Liberty-enabled access management applications within the NetIQ industry framework, contact namsdk@microfocus.com.

1.1.1 SDK Components

The Access Manager developer components are included in the Access Manager Developer Kit (https://www.microfocus.com/documentation/access-manager/developer-documentation-5.0/samplecodes/main.html).

However, the complete Access Manager package, including the install, is not included in the SDK. For complete current product information, see the NetIQ Access Manager Product Site.

The SDK does not include the JAR files required from the product to compile your extension. You need access to an Access Manager installation to obtain these files.

1.2 Selecting an Integrated Development Environment

The Java applications can be developed on a number of open source IDEs, such as Eclipse* and NetBeans*.

2 Identity Server Authentication API

This section provides details about how to create a custom authentication class for Identity Server. The API presented here allows developers to leverage their own authentication mechanisms within the Access Manager architecture.

2.1 Prerequisites for Creating a Custom Authentication Class

- Access Manager is installed.
- Your development environment requires the same installation as outlined in the NetIQ Access Manager 5.0 Installation and Upgrade Guide.
- Download `nidp.jar` and `NAMCommon.jar` from the `/opt/novell/nam/idp/webapps/nidp/WEB-INF/lib` directory and add these to your development project.

  For information about how to download a file, see Downloading Files from a Server in the NetIQ Access Manager 5.0 Administration Guide.

  For information about how to add a file, see Adding Configurations to a Cluster in the NetIQ Access Manager 5.0 Administration Guide.

2.2 Understanding the Authentication Class

2.2.1 Authentication Class Components

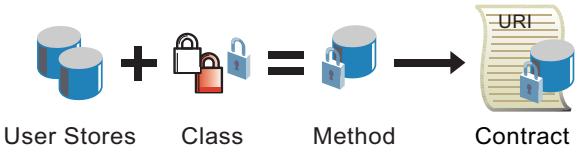
Identity Server is the central authentication and identity access point for all services performed by Access Manager. Identity Server supports numerous ways for users to authenticate. These include name/password, RADIUS token-based authentication, and X.509 digital certificates.

For more information about Identity Server and its relation to other Access Manager components, see “Creating Authentication Classes” in the NetIQ Access Manager 5.0 Administration Guide.

The configuration and interaction of the following entities defines how authentication takes place within Identity Server:

- **User Stores:** The LDAP directory that stores the user credentials. Access Manager can be configured to use the following directories: eDirectory™, Active Directory*, or Sun One*. Users set up their user stores when creating Identity Server configuration.
- **Authentication Classes:** The code (a Java class) that implements a particular authentication type (name/password, RADIUS, and X.509) or means of obtaining credentials. This is what you create with this API.
- **Authentication Methods:** Pairs an authentication class with one or more user stores, primarily to identify authenticated users. Authentication methods also can be designed to identify entities other than end users.
- **Authentication Contracts:** The basic unit of authentication within Identity Server. Contracts are identified by a unique uniform resource identifier (URI) that can be used by Access Gateways and agents to protect resources. Contracts are comprised of one or more authentication methods used to uniquely identify a user.

Figure 2-1 *Local Authentication Components*



2.2.2 How the Authentication Class Operates

Figure 2-2 illustrates an example of how an authentication class is used to authenticate to an Identity Server. It uses a single user store located on an LDAP server to verify name and password credentials.

Figure 2-2 *How the Authentication Class Handles a User Request.*

1. A user initializes an authentication request from a browser.
2. The request causes the default authentication class to execute. This class defines what credentials are required for authentication, and it returns a response prompting the user for the required credentials (that is, username, password, x509 certificate, etc.). The user enters the credentials.
3. The class obtains the credentials, then passes them to the user store for verification and validation.
4. If credentials are valid, the user store returns the user's DN (or other information specified by the method) and allows user access. If the information is not valid, access is denied.

The authentication API also enables you to implement more complex authentication using X.509 certificates, data generated by token devices, biometric data, or other data you specify. In such instances, you must specify the outside resources that contain the credential stores that are configured to validate the required user credentials.

2.3 Creating an Authentication Class

Identity Server architecture provides a programming interface that allows you to create a custom authentication class that can be plugged in to the Access Manager system. Custom authentication classes can define additional ways of obtaining and validating end-user credentials. You use the Access Manager Administration Console to identify your custom classes and specify any needed initialization properties. Custom classes must be configured to be in the class path of Identity Server.

The following sections explain project requirements and methods for creating a custom class:

For the Javadoc associated with these methods, see LocalAuthenticationClass.

2.3.1 Project Requirements

The project used to create the custom class must include nidp.jar shipped with Access Manager.

2.3.2 doAuthenticate Method

A customized authentication class must extend the abstract class `com.novell.nidp.authentication.local.LocalAuthenticationClass`, which is found in `nidp.jar`. The base class contains a single required constructor. Your custom class must implement one of two methods, either doAuthenticate(), which is preferred, or authenticate(), which was used in previous releases of this SDK.

The doAuthenticate() method is new in Access Manager 3.0 SP3. Previous releases used the authenticate() method. The older method is still supported, but new classes created for SP3 and later should use the doAuthenticate() method because it performs additional Novell SecretStore® checks. SecretStore now supports a security flag that locks the SecretStore when secrets are modified. The doAuthenticate() method performs checks to determine the state of the SecretStore. If it is locked, it prompts the user to supply the passphrase that can be used to unlock the SecretStore. If you use the older authenticate() method and the SecretStore is locked, no indication of this state is returned. The SecretStore remains locked, and Access Manager cannot retrieve the secrets for policies or applications that require them.

Identity Server calls the doAuthenticate() method during its interaction with the class. Multiple calls to authenticate often are made to collect the necessary authentication credentials. The method returns a value indicating any of the following authentication states:

Constant	Description
HANDLED_REQUEST	The request has been handled and a response provided. Further processing or information is needed to complete authentication. Typically, this value is returned when a page is returned to query for credentials.
SHOW_JSP	Further information is needed to complete authentication. Typically, this value is returned when a page is returned to query for credentials.
NOT_AUTHENTICATED	The authentication failed.
AUTHENTICATED	The authentication succeeded in identifying a single NIDPPrincipal object (user).
CANCEL	The authentication process was canceled. This typically occurs only during authentication after a request from a service provider.
PWD_EXPIRING	Although authentication is successful, a user's password is about to expire. This condition causes a redirection to the expired password servlet if one is defined on the authentication contract.
PWD_EXPIRED	Authentication is unsuccessful, because the user's password is expired. This condition causes a redirection to the expired password servlet if one is defined on the authentication contract.

When the doAuthenticate() method succeeds, it needs to return AUTHENTICATED. It can succeed only when it obtains a single NIDPPrincipal object from a user store using the credentials obtained to verify the principal. After credentials are obtained, each user store is searched to locate a user identified by the credentials. Each user store is searched until one of the follow conditions is met:

- **Successful authentication:** Indicates that a single user/object is located.
- **Unsuccessful authentication with an error:** Indicates that more than one user/object is located.

2.3.3 Authentication Methods

When implementing the doAuthenticate method(), you can use the following methods to retrieve and manage authentication credentials:

Method	Description
authenticateWithPassword()	Takes a user ID and password as its arguments. The method succeeds if a user with the given ID and password is located. See authenticateWithPassword

Method	Description
authenticateWithQuery()	Takes a string in the form of an LDAP query and a password as its as its arguments. It succeeds if the query result locates a single user with the associated password. See authenticateWithQuery
findPrincipals()	Locates the users in a directory that match the specified user ID. The method does not do any password verification. It returns an array of NIDPPrincipal objects that result from the search. See findPrincipals
findPrincipalsByQuery()	Locates the users in a directory that match the specified LDAP query. The method does not do any password verification. It returns an array of NIDPPrincipal objects that match the query. See findPrincipalsByQuery
getCredentials()	Gets the list of credentials used to authenticate the user or principal. Identity Server uses this method to obtain the credentials verified by an authentication class for possible later use with an Identity Injection policy. An authentication class does not typically call this method. See getCredentials
addCredential()	Adds a credential used for authentication to a user or principal. This method is called by a class so that Identity Server can call the getCredentials() method. See addCredential
addLDAPCredentials()	Adds an LDAP credential, other than the password, to a user or principal. See addLDAPCredentials
clearCredentials()	Clears the credentials of the user or principal. See clearCredentials

2.3.4 reCAPTCHA Methods

To override the class properties, use the following properties at the Method Properties level. These must be set at the Method pages on the Properties page as custom properties. This is done because no modifications are done to the Method Properties to add convenient ways to set the captcha variables for the method. The interface changes are done only at the Authentication Class level.

Method	Description
recaptchaEnable	“true” or “false” if reCAPTCHA is enabled
recaptchaThreshold	“0” indicates always show the reCAPTCHA on the login page, “2” indicates that the failed login count must be 2 or more times before the reCAPTCHA displays

Method	Description
recaptchaSiteKey	The Google reCAPTCHA Site Key
recaptchaSecretKey	The Google reCAPTCHA Secret Key

2.3.5 Class Property Methods

Typically, classes have properties assigned to them. The installed Identity Server authentication classes have associated properties. Because these classes and their properties are known, Administration Console displays configuration pages for their required properties. For information about these properties, see "Creating Authentication Classes" in the NetIQ Access Manager 5.0 Administration Guide.

When you deploy your class, Administration Console has a generic page that allows an administrator to configure property key name and value pairs. When you create a class, you need to create a key name and value pair for each configuration item for which you want input from the administrator. For example, if you want to allow the administrator to use a different JSP page in the login form, you can create a key name of JSP with an expected value of filename. You would use the getProperty() method to obtain the value of the JSP key name. If the method returns null, you would have your code use your default JSP page. You need to document any key names that you create and the type of value that it requires, and make this information available to the administrator.

The class property methods return all values as strings. However, you can manipulate the string value as required by your code. For example, if your key name requires a number and the administrator configures the key name with a letter value, you need to decide how to handle such an error (continue and use a default value or throw an exception). As a minimum, the error should be logged, so that the administrator can discover the cause of the configuration problem.

The following methods are available for retrieving information about configuration properties:

Method	Description
getProperty()	Obtains specific properties needed by an authentication class. Property values are specified when configuring the authentication class in Administration Console. See getProperty
getBooleanProperty()	Returns a Boolean value for the specified property and sets a default value if the value cannot be found. See getBooleanProperty
getType()	Identifies one of the authentication types known to Identity Server. The value returned by this method is used primarily when a service provider initiates an authentication request by asking for a specific authentication type. When such a request is made, a check of all executed contracts is made. If a contract has executed a method by using a class that defines the particular type, the authentication succeeds. See Supported Authentication Class Types for a list of supported types. See getType

Method	Description
getProvisionURL()	Gets the URL to call to provision a user and returns the redirect URL for user provisioning, or Null if it is not available. See getProvisionURL
getReturnURL()	Returns the URL to which a user interaction should post data, or Null if it is not available. See getReturnUrl
mustPersist()	Indicates whether the class must persist for interaction with the user during the entire authentication session. If this is the case, returns True. For more information about persistence, see “Class Persistence” on page 16. See mustPersist
isFirstInstance()	Determines if this authentication class instance is the first instance after the system was started or was reconfigured. Returns True if it is the first instance. See isFirstInstance
isCancelAppropriate()	Determines if the option to cancel an authentication is appropriate for this instance. See isCancelAppropriate
isDefinesUser()	Determines if the authentication class instance needs to identify a user. If so, returns True. For more information, see the **Identifies User** option in see Creating Authentication Methods in the Access Manager 5.0 Administration Guide. See also isDefinesUser.
isUserIdentification()	Determines if this authentication class instance is the result of an assertion being returned to an unauthenticated session. The request for authentication is the result of an assertion from an identity provider, and it is necessary to identify the user for the purpose of completing the federation process. See isUserIdentification.
isFirstCallAfterPrevMethod()	Defines the sequence of the authentication process after a method is called and determines if this authentication class instance is the result of an assertion being returned to an unauthenticated session. This is useful to determine if an authentication class begins execution immediately after the successful completion of another class. This enables a class to know if credentials were actually used by the previous class. See isFirstCallAfterPrevMethod.
isPendingAuthnRequest()	Determines whether there is a pending authentication request from a Service Provider. Returns True if there is a pending request, otherwise, returns False. See isPendingAuthnRequest.
getAuthnRequest()	Gets the request that might have caused this authentication class to be invoked. See getAuthnRequest

2.3.5.1 Supported Authentication Class Types

When you create an authentication class, you must specify an authentication type. An authentication type is required, because some service providers request contracts, not by URI, but by authentication type. Identity Server can reply to such a request with all the contracts that fit the requested authentication type.

Identity Server supports the following types of authentication classes:

Constant	Description
AuthnConstants.BASIC	Specifies a basic authentication over HTTP. It uses the login page of a browser to prompt a user for name and password.
AuthnConstants.PASSWORD	Specifies a form-based authentication using a name and password over HTTP.
AuthnConstants.PROTECTED_BASIC	Specifies a basic authentication over HTTPS. It uses the login page of the browser to prompt the user for a name and a password.
AuthnConstants.PROTECTED_PASSWORD	Specifies a form-based authentication using a name and password over HTTPS.
AuthnConstants.X509	Specifies authentication using an X.509 certificate.
AuthnConstants.TOKEN	Specifies a token-based authentication type.
AuthnConstants.SMARTCARD	Specifies a smart-card-based authentication method.
AuthnConstants.SMARTCARDPKI	Specifies a multi-authentication method using a smart card.
AuthnConstants.OTHER	Default. Used for all other types not defined above.

2.3.5.2 Class Persistence

Persistence of a class is session based. A session is created when a user is prompted to provide credentials for a contract. Each method of a contract gets executed in the order defined in the contract. When a method executes, it creates an instance of the class. The class can persist between requests for credentials if necessary. If keeping state is not required by the class, then it does not need to persist. By default, classes persist. If this is not the desired behavior, use the mustPersist() method to return False.

If the class is configured to persist, the instance of the class persists as long as the doAuthenticate() or authenticate() method of the class returns HANDLED_REQUEST. When this method returns any other value, the instance of the class is removed. For a list of possible return values, see Section 2.3.2, “doAuthenticate Method,” on page 11.

2.3.6 Status Methods

The following methods allow you to set status information about the authentication instance, to retrieve status information about the instance, to set and get error messages, and to log messages.

Method	Description
setFailure()	Sets a failure state for the current authentication instance. See setFailure
isFailure()	Indicates whether or not the authentication failed. Returns True if authentication failed, otherwise, returns False. See isFailure
setUserErrorMsg()	Sets the error message to be displayed to an end user. See setUserErrorMsg
getUserErrorMsg()	Gets the error message that will be displayed to the end user. See getUserErrorMsg
getLogMsg()	Gets the message for the associated error ID. This method is used primarily by Identity Server to obtain the credentials verified by an authentication class. See getLogMsg
setErrorMsg()	Sets the error message to be seen by the end user, as well as the error message to be put into the log file. See setErrorMsg. See “Authentication Error Messages” on page 17.
setErrorMsg()	Sets the error message to be seen by the end user, as well as the error message with a parameter to be put into the log file. See setErrorMsg. See “Authentication Error Messages” on page 17.

2.3.6.1 Authentication Error Messages

The following error messages have been defined for the LocalAuthenticationClass and are returned:

Value	Error Message Description
LOG_INCORRECT_PASSWORD	The password entered does not match any of those authorized in the specified user stores.
LOG_INTRUDER_DETECTION	(eDirectory) The user account is locked because of intruder detection.
LOG_RESTRICTED_ACCOUNT	(eDirectory only) This account has restricted access and the user is attempting to access it during a time period when the account has been configured to deny access.

Value	Error Message Description
LOG_DISABLED_ACCOUNT	The account requested is disabled.
LOG_BAD_CONNECTION	The authentication channel is unable to communicate the user request.

2.3.7 User Information Methods

The following methods allow you to set the identity of who has been authenticated and to set values for any associated attributes. If the instance is persistent, you can retrieve this same information. User authorities are the LDAP servers that Identity Server has been configured to use for verifying authentication credentials. The principal user authority is the LDAP server that was used to verify the user's credentials.

Method	Description
getPrincipal()	Gets the principal authenticated by this class. This value is Null if the authentication class is set to not define a user or if the authentication fails. This method is used primarily by Identity Server to obtain the credentials verified by an authentication class. See getPrincipal
getPrincipalAttributes()	Gets the attributes for the principal that has been authenticated. See getPrincipalAttributes
getPrincipalUserAuthority( )	Gets the user authority for the identified principal, assuming that m_Principal has been set. See getPrincipalUserAuthority
getUserAuthorityCount()	Gets the number of searchable user authorities. See getUserAuthorityCount
getUserAuthority()	Gets a specific user authority. The getUserAuthorityCount() method returns the index range. See getUserAuthority
setPrincipal()	Sets the principal to be authenticated by this class. See setPrincipal
setPrincipalAttributes()	Sets attributes for a principal that has been authenticated. See setPrincipalAttributes

Method	Description

setSessionProperties()	Sets user session properties that can be used later by other custom authentication classes and risk-based authentication rules.
	
	With the following code snippet, you can set session properties by using custom authentication class:
	
	`// Create a new HashMap`
	
	`HashMap<String, Object> map = new HashMap<String, Object>();`
	
	`map.put("Name", "InuputName");`
	
	`map.put("ExternalEmail", "email@email.com");`
	
	`// Call the API to set the Map`
	
	`setSessionProperties(map);`

getSessionProperties()	Gets user session properties that were previously set by other custom authentication classes.
	
	With the following code snippet, you can set session properties by using custom authentication class:
	
	`// Create a new HashMap`
	
	`HashMap<String, Object> map = new HashMap<String, Object>();`
	
	`// Get the session properties from session`
	
	`map = getSessionProperties();`
	
	`String email = (String)map.get("ExernalEmail");`

getPrincipalAttributesFromPreferredLDAPReplica()	Gets the attributes for the authenticated Principal, from the last used User Store replica for a particular user session. Use this method when there is a need to read attributes soon after the same were written to the User Store. This will avoid any errors that may occur in case the attributes have not been synchronized across all replicas yet.
setPrincipalAttributesInPreferredLDAPReplica()	Sets the attributes for the authenticated Principal, in the last used User Store replica for a particular user session. Use this method to avoid any errors that may occur in case the attributes have not been synchronized across all replicas yet.

2.3.8 CallbackAuthentication Method

To use a custom authentication class in the WS-Trust/STS/OAuth Resource Owner credential authentication, implement the `com.novell.nidp.authentication.local.CallbackAuthentication` interface in the authentication class.

To perform the STS/OAuth Resource Owner credential authentication, you need to implement the `cbAuthenticate` method in the authentication class.

For a sample implementation of `cbAuthenticate`, see PasswordClass Example Code.

2.3.9 Other Methods

The following tables lists other useful methods:

Method	Description
showError()	Causes an error JSP to be executed to display an error message. See showError
showJSP()	Forwards execution to a specific JSP. See showJSP
escapeName()	Escapes characters typed by the user. See escapeName
initializeRequest()	Initializes the authentication class with the current request/response. Normally, this method is called only by Identity Server when it initializes the authentication class with the current request/response. See initializeRequest.

2.4 Authentication Class Example

This section demonstrates how a password authentication class might be implemented by using the PasswordClass. All authentication classes are derived from the LocalAuthenticationClass, so you need to understand the key methods within it:

- Section 2.4.1, “Extending the Base Authentication Class,” on page 21
- Section 2.4.2, “Implementing the doAuthenticate Method,” on page 21
- Section 2.4.3, “Prompting for Credentials,” on page 21
- Section 2.4.4, “Verifying Credentials,” on page 21
- Section 2.4.5, “PasswordClass Example Code,” on page 22
- Section 2.4.6, “Accessing a Principal Object from an Authentication Method,” on page 26

2.4.1 Extending the Base Authentication Class

Authentication classes extend the base class LocalAuthenticationClass as shown on lines 11 and 12 of PasswordClass Example Code. The LocalAuthenticationClass has a single constructor that must be called as shown in lines 20 - 23. Identity Server uses this constructor to pass the necessary properties and user store information defined in Administration Console to the class.

The LocalAuthenticationClass defines a single abstract method, doAuthenticate(), which must be implemented by new classes. During user authentication, Identity Server creates an instance of an authentication class and calls the authenticate() method, which in turn calls the doAuthenticate() method. By default, the class instance remains persistent, allowing the state to be preserved between requests/responses while credentials are obtained. If persistence is not needed, the mustPersist() method can be overloaded to return False so new instances of the class are created upon each call to the authenticate() method.

2.4.2 Implementing the doAuthenticate Method

Lines 43 - 65 in the PasswordClass Example Code show how the doAuthenticate() method is used. Return values from this method indicate to Identity Server that the class has succeeded or failed to authenticate a user or that additional user credentials are required and must be obtained.

The call to the isFirstCallAfterPrevMethod() method on line 49 determines if the call to the class is following a successful authentication by another class executed by a method. If that is the case, any credentials provided for the previous class most likely are not valid for this class and should not be tested for (line 52). In this example, the handlePostedData() method is called to obtain and validate a username and password entered by a user.

2.4.3 Prompting for Credentials

When lines are encountered in the PasswordClass Example Code, it has been determined that a page needs to be returned through the execution of a JSP to enable credentials to be prompted for and returned. Tests are made to determine if provisioning should be enabled, and if a **Cancel** button and federated providers should be displayed. The return value of HANDLED_REQUEST or SHOW_JSP indicates that the class has responded to the request and requires more information to proceed.

2.4.4 Verifying Credentials

The handlePostedData() method does much of the important work of this example (lines 74 - 114 in the PasswordClass Example Code). Lines 81 - 100 attempt to obtain the credentials.

Line 86 provides an example of obtaining a class property configured by an administrator. In this case, a query can be defined by the administrator that can be used to look up a user instead of using the username and password. If the query is used, the authenticateWithQuery method is called at line 88. If a query is not available, the authenticateWithPassword() method is called at line 98.

If the credentials correctly identify the user, the value AUTHENTICATED is returned. If they fail to identify the user, NOT_AUTHENTICATED is returned.

When eDirectory is the user store and a password has either expired or is expiring, the return values PWD_EXPIRED and PWD_EXPIRING can be returned respectively. See lines 102 - 108.

Line 111 demonstrates how an attribute is used to set an error message that is displayed to the user by calling the method getUserErrorMsg().

2.4.5 PasswordClass Example Code

```
package com.novell.nidp.authentication.local;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Properties;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.eclipse.higgins.sts.api.ISecurityInformation;
import org.eclipse.higgins.sts.api.IUsernameToken;

import com.novell.nidp.NIDPConstants;
import com.novell.nidp.NIDPException;
import com.novell.nidp.NIDPPrincipal;
import com.novell.nidp.NIDPSession;
import com.novell.nidp.NIDPSessionData;
import com.novell.nidp.authentication.AuthnConstants;
import com.novell.nidp.common.authority.PasswordExpiredException;
import com.novell.nidp.common.authority.PasswordExpiringException;
import com.novell.nidp.common.authority.UserAuthority;
import com.novell.nidp.common.protocol.AuthnRequest;
import com.novell.nidp.liberty.wsc.cache.WSCCacheEntry;
import com.novell.nidp.logging.NIDPLog;
import com.novell.nidp.saml.SAMLAuthMethods;
import com.novell.security.sso.SecretStore;
import com.sun.xml.wss.impl.callback.UsernameCallback;

public class PasswordClass extends LocalAuthenticationClass implements
STSAuthenticationClass, CallbackAuthentication {
    private String m_Error;

    // for NRL
    LocalAuthenticationClass basicClass = null;

    /**
     * Constructor for form based authentication
     *
     * @param props
     *            Properties associated with the implementing class
     * @param uStores
     *            List of ordered user stores to authenticate against
     */
    public PasswordClass(Properties props, ArrayList<UserAuthority> uStores) {
        super(props, uStores);
        // for NRL
        if (m_LECP)
            basicClass = new BasicClass(props, uStores);
    }

    /**
     * Get the authentication type this class implements
     *
     * @return returns the authentication type represented by this class
     */
    public String getType() {
```

```
        return AuthnConstants.PASSWORD;
    }

    public void initializeRequest(HttpServletRequest request, HttpServletResponse
response, NIDPSession session, NIDPSessionData data, boolean following, String url)
{
        super.initializeRequest(request, response, session, data, following, url);
        if (basicClass != null)
            basicClass.initializeRequest(request, response, session, data,
following, url);
    }

    /**
     * Perform form based authentication. This method gets called on each
     * response during authentication process
     *
     * @return returns the status of the authentication process which is one of
     *         AUTHENTICATED, NOT_AUTHENTICATED, CANCELLED, HANDLED_REQUEST,
     *         PWD_EXPIRING, PWD_EXPIRED
     */
    protected int doAuthenticate() {
        // If this is the first time the class is called following another
        // method
        // we want to display the form that will get the credentials. This
        // method
        // prevents a previous form from providing data to the next form if any
        // parameter names end up being the same
        if (!isFirstCallAfterPrevMethod()) {
            // This wasnt first time method was called, so see if data can be
            // processed
            int status = handlePostedData();
            if (status != NOT_AUTHENTICATED)
                return status;
        }

        String jsp = getProperty(AuthnConstants.PROPERTY_JSP);
        if (jsp == null || jsp.length() == 0)
            jsp = NIDPConstants.JSP_LOGIN;

        m_PageToShow = new PageToShow(jsp);
        m_PageToShow.addAttribute(NIDPConstants.ATTR_URL,(getReturnURL() != null?
getReturnURL():m_Request.getRequestURL().toString()));
        if (getAuthnRequest() != null && getAuthnRequest().getTarget() != null)
            m_PageToShow.addAttribute("target", getAuthnRequest().getTarget());

        String username = m_Request.getParameter(NIDPConstants.PARM_USERID);
        if (username != null) // user name is already present
            m_PageToShow.addAttribute("username", username);
        // They failed logging in, check if Captcha is required
                // Is CAPTCHA really required, if it is, include reCaptchaSiteKey
        if (isCaptchaRequired())
m_PageToShow.addAttribute(NIDPConstants.ATTR_RECAPTCHA_SITEKEY,getProperty(AuthnCo
nstants.PROPERTY_RECAPTCHA_SITEKEY));//this attribute will trigger Captcha to
display.
        // If we are displaying in the credential window and the error has not
        // been displayed yet, go ahead and show it. This can happened when
        // the wrong credentials as posted from a third party site
        String option = m_Request.getParameter("option");
        if (option != null && option.equals("credential") && m_Error != null) {
            m_PageToShow.addAttribute(NIDPConstants.ATTR_LOGIN_ERROR, m_Error);
            m_Error = null;
        }

        return SHOW_JSP;
    }
```

```
    protected int doAuthenticateNRL() {
        /*
         * Presently NRL always gets the credentials passed in the Basic header
         * over Liberty LECP So, invoking basic class todo the processing
         */
        int status = basicClass.doAuthenticate();
        if (basicClass.getPrincipal() != null) {
            this.setPrincipal(basicClass.getPrincipal());
            this.m_Credentials = basicClass.getCredentials();
        } else {
            this.m_ExpiredPrincipal = basicClass.getExpiredPrincipal();
           this.setErrorMsg(basicClass.getUserErrorMsg(), basicClass.getLogMsg());
            setFailure();
            this.m_PasswordException = basicClass.getPasswordException();
        }
        return status;
    }

    /**
     * Get and process the data that is posted from the form
     *
     * @return returns the status of the authentication process which is one of
     *         AUTHENTICATED, NOT_AUTHENTICATED, CANCELLED, HANDLED_REQUEST,
     *         PWD_EXPIRING, PWD_EXPIRED
     */
    private int handlePostedData() {
        // Look for a name and password
        String id = m_Request.getParameter(NIDPConstants.PARM_USERID);
        String password = m_Request.getParameter(NIDPConstants.PARM_PASSWORD);
        // Check if the recaptcha-response is needed or valid
       // this will return true if captcha is not required or false if captcha is
required and the recaptcha-response is not valid
      // if captcha is invalid, FAIL the login, don't even check username/password
        if
(!verifyRecaptcha(m_Request.getParameter(AuthnConstants.RECAPTCHA_RESPONSE)))
        {
            setErrorMsg(NIDPMainResDesc.LOGIN_FAILED,
NIDPMainResDesc.RECAPTCHA_NOT_OPTIONAL,null);
            m_Error = getUserErrorMsg();
            return NOT_AUTHENTICATED;
        }

        setUserId(id);


        // Check to see if admin has setup for a custom query
        String ldapQuery = checkForQuery();

        try {
            // using admin defined attributes for query
            if (ldapQuery != null) {
                if (authenticateWithQuery(ldapQuery, password))
                    return AUTHENTICATED;
            }

            // If using default of name and password
            else {
                if (id == null || id.length() == 0)
                    return NOT_AUTHENTICATED;

                if (authenticateWithPassword(id, password))
                    return AUTHENTICATED;
            }
        } catch (PasswordExpiringException pe) {
            return PWD_EXPIRING;
        } catch (PasswordExpiredException pe) {
```

```
            return PWD_EXPIRED;
        }

        m_Error = getUserErrorMsg();
        return NOT_AUTHENTICATED;
    }

    public NIDPPrincipal handleSTSAuthentication(ISecurityInformation
securityInformation) {
        IUsernameToken usernameToken = (IUsernameToken)
securityInformation.getFirst(IUsernameToken.class);

        if (null != usernameToken) {
            try {
                if (authenticateWithPassword(usernameToken.getUsername(),
usernameToken.getPassword()))
                    return getPrincipal();
            } catch (PasswordExpiringException pe) {
                return getPrincipal();
            } catch (PasswordExpiredException pe) {
            }
        }
        return null;
    }

    @Override
    public NIDPPrincipal cbAuthenticate(CallbackHandler cbHandler) {
        PasswordValidationCallback pwdCallback = new PasswordValidationCallback();
        Callback[] callbacks = new Callback[] { pwdCallback };

        NIDPPrincipal principal = null;
        try {
            cbHandler.handle(callbacks);
            if (pwdCallback.getUsername() != null) {

                String query = getProperty(AuthnConstants.PROPERTY_QUERY);
                String ldapQuery = null;
                boolean status = false;
                if (query != null)
                {
                  ldapQuery = getLDAPQueryString(query,pwdCallback.getUsername());
                  if (authenticateWithQuery(ldapQuery,pwdCallback.getPassword()))
                    status = true;
                }
                else if (authenticateWithPassword(pwdCallback.getUsername(),
pwdCallback.getPassword()))
                status = true;

              if ( status == true ) {
                    principal = getPrincipal();
                    principal.setAuthMethod(SAMLAuthMethods.PASSWORD);
                    return principal;
                }
            }

        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (UnsupportedCallbackException e) {
```

```
            if(NIDPLog.isLoggableWSTrustFine())
                NIDPLog.logWSTrustFine("The caller doesn't support password
callback: " + e.getMessage());
        } catch (PasswordExpiredException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (PasswordExpiringException e) {
            principal = getPrincipal();
            principal.setAuthMethod(SAMLAuthMethods.PASSWORD);
            return principal;
        }
        return null;
    }
}
```

2.4.6 Accessing a Principal Object from an Authentication Method

For a custom class, the following is an example of how to read the principal object from a previous authentication method:

```
NIDPPrincipal principal = (NIDPPrincipal)m_Properties.get("Principal");
if (principal == null)
if ( m_Session.isAuthenticated() && ( m_Session.getSubject().getPrincipal() != null
) )
principal = m_Session.getSubject().getPrincipal();
```

Use this code in the `doAuthenticate()` method.

2.5 Localizing the Prompts in Your Authentication Class

You need to create a JSP page for displaying the login prompts. When doing so, you might want to allow the prompts to be displayed in multiple languages.

To enable the text so that it can be displayed in multiple languages, you need to do the following:

- Section 2.5.1, “Creating a Properties File,” on page 26
- Section 2.5.2, “Creating a Resource Class,” on page 27
- Section 2.5.3, “Creating or Modifying a JSP Page,” on page 28

2.5.1 Creating a Properties File

You need to create a list of the strings to be displayed when prompting users for login credentials and reacting to their input. You need to create a string constant for each string and place the string constant and string in a properties file. The following properties file contains some sample string constants for a few of the prompts that your JSP page might need.

```
LOGIN=Login
USERNAME_PROMPT=Username:
CONTACT_ADMINISTRATOR_PROMPT=Contact your system administrator.
SAMPLE_AUTH_FAILED_MSG=Authentication Failed.
CONTINUE_PROMPT=Continue
CONTINUE_TITLE=Continue
LOGIN_ERROR_PROMPT=Authentication Error.
```

The name for this properties file needs to end with the Java defined constants for each language. For the English version for use in the United States, the file would end with `en_US.properties`, for example, `SampleResources_en_US.properties`. The base portion of the name (in this example, `SampleResources`) stays the same for all languages.

You need to create such a file, with the appropriately translated strings and name, for each language you want to support.

2.5.2 Creating a Resource Class

You need to extend the `com.novell.nidp.resource.NIDPResDesc` class with a resource class that knows how to call your properties files and retrieve the strings. The following sample code extends the `NIDPResDesc` class with a class called `SampleResDesc`, defines the base name for the properties file (`SampleResources`), and defines a string constant for each string in the properties file.

```
#############need a package name line ####################
import com.novell.nidp.resource.NIDPResDesc;

public class SampleResDesc extends NIDPResDesc
{
    private static final String SAMPLE_BUNDLE_BASENAME =
                          "SampleResources";
    private static final String KEYS_PREFIX = "";

 // Names of localized strings and messages
 public static final String LOGIN = "LOGIN";
 public static final String USERNAME_PROMPT = "USERNAME_PROMPT";
 public static final String CONTACT_ADMINISTRATOR_PROMPT =
                              "CONTACT_ADMINISTRATOR_PROMPT";
 public static final String SAMPLE_AUTH_FAILED_MSG =
                              "SAMPLE_AUTH_FAILED_MSG";
 public static final String CONTINUE_PROMPT = "CONTINUE_PROMPT";
 public static final String CONTINUE_TITLE = "CONTINUE_TITLE";
 public static final String LOGIN_ERROR_PROMPT = "LOGIN_ERROR_PROMPT";

 private static SampleResDesc m_instance = null;

    private SampleResDesc()
    {
        super(SAMPLE_BUNDLE_BASENAME, KEYS_PREFIX);
    }

    public static SampleResDesc getInstance()
    {
        if (null == m_instance)
        {
            m_instance = new SampleResDesc();|
        }
        return m_instance;
    }

}
```

2.5.3 Creating or Modifying a JSP Page

The JSP page generates the prompts for user credentials by calling your extended resource class to retrieve the strings. The following snippet of a JSP page gets the local language code, and then calls the extended resource class (SampleResDesc) to display the string for the string constant USERNAME_PROMPT.

```
.
.
.
<%@ page import="com.novell.nidp.*" %>
<%@ page import="com.novell.nidp.resource.*" %>
<%
      Locale locale = request.getLocale();
      String strLanguageCode = locale.getLanguage();
      String strLanguageCodeLowercase = strLanguageCode.toLowerCase();
      NIDPResource sampleResource = NIDPResourceManager.getInstance()
            .get(SampleResDesc.getInstance(), locale);
%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//<%=strLanguageCode%>">
<html lang="<%=strLanguageCode%>">
.
.
.
<table>
     <tr>
           <td>
                 <label style="width: 100px"><%= sampleResource.
                  getString(SampleResDesc.USERNAME_PROMPT) %></label>
           </td>
           <td width="100%">
                 <input id="My_User_ID" type="text" class="smalltext"
                       name="My_User_ID" size="30">  
                 <input alt="<%=sampleResource.getString0
                         (SampleResDesc.LOGIN)%>"
                       border="0" name="loginButton"
                       src="<%=request.getContextPath() %>/images/
                         <%=strLanguageCodeLowercase%>/
                           btnlogin_<%=strLanguageCodeLowercase%>.gif"
                       type="image" value="Login"
                       onclick="mySubmit()">
.
.
.
```

For your authentication class, this JSP snippet needs to be extended to include the prompts for the other authentication credentials your class requires. As you add prompts to the JSP page, these constants need to be added to your resource class and properties files.

The name of this JSP page needs to correspond to the page you call in your authentication class. See lines 109 - 111 in the sample class (Section 2.4.5, “PasswordClass Example Code,” on page 22).

When you create the jar file for your authentication class, the properties files and the Java resource file need to be included. The JSP page for your authentication class needs to be added to the `/opt/novell/nids/lib/webapp/jsp` directory of Identity Server. For information about how to add a file, see Adding Configurations to a Cluster in the NetIQ Access Manager 5.0 Administration Guide.

2.6 Deploying Your Authentication Class

1. Create a jar file for your authentication class and any associated classes.
2. Add the jar file to the `/opt/novell/nam/idp/webapps/nidp/WEB-INF/lib` directory using Advanced File Configurator.

   For information about how to add a file, see Adding Configurations to a Cluster in the NetIQ Access Manager 5.0 Administration Guide.
3. (Conditional) If you created a custom JSP page for your authentication class, add it to the `/opt/novell/nids/lib/webapp/jsp` directory of Identity Server using Advanced File Configurator.
4. Click **Devices** > **Identity Servers** > **Edit** > **Local** > **Classes** > **New**.
5. Specify the following details:

   Display name: Specify a name that Administration Console can use to identity this class.

   Java class: Select **Other**. This allows you to specify the path name of your Java class.

   Java class path: Specify the name of your Java class.
6. Click **Next**, and specify any needed properties of your class.

   This is dependent upon your class. You need to specify properties only if your class requires them.

   This information is sent to your class in the `props` parameter when your class is called.
7. Click **Finish**.
8. To configure a method for your class, click **Methods** > **New**, and select your class in **Class**.

   When you configure a method, you specify which user stores can be used for authentication. This information is returned to your class in the `uStores` parameter when your class is called.

   For more information, see Creating Authentication Methods in the NetIQ Access Manager 5.0 Administration Guide.
9. Click **Finish**.
10. To configure a contract for your class, click **Contracts** > **New**, and move your class to be a value in the **Methods** list.

    For more information, see Creating Authentication Contracts in the NetIQ Access Manager 5.0 Administration Guide.
11. (Optional) Default contracts can be specified for each authentication type that might be required by a service provider. These contracts are executed when a request for a specific authentication type comes from a service provider.

    For more information, seeSupported Authentication Class Types and “Specifying Authentication Defaults” in the Access Manager 5.0 Administration Guide.
12. Click **Finish** > **OK**.
13. Restart Identity Server.
14. On Identity Servers page, click **Update**.
15. Update any associated devices that are using this Identity Server configuration.

3 LDAP Server Plug-In

An LDAP Server plug-in module is a Java class that allows an unsupported LDAP server to be used with Access Manager. The three supported LDAP servers are eDirectory™, Active Directory, and Sun ONE. Any other directory types require an LDAP Server plug-in.

3.1 Prerequisites

To develop an LDAP server plug-in:

- ❒ Meet all system requirements of Identity Servers and Access Gateways. See the NetIQ Access Manager 5.0 Installation and Upgrade Guide.
- ❒ Install and configure all components of Access Manager. For installation and configuration information, see the NetIQ Access Manager 5.0 Installation and Upgrade Guide and “Setting Up a Basic Access Manager Configuration” in the Access Manager 5.0 Administration Guide.
- ❒ Have an integrated Java development environment.
- ❒ Download `NAMCommon.jar` from the `/opt/novell/nam/idp/webapps/nidp/WEB-INF/lib` directory and add it to your development project.

  For information about how to download a file, see Downloading Files from a Server in the NetIQ Access Manager 5.0 Administration Guide.

  For information about how to add a file, see Adding Configurations to a Cluster in the NetIQ Access Manager 5.0 Administration Guide.

3.2 Creating the LDAP Plug-In

The project used to create the plug-in must include the `NAMCommon.jar` file shipped with Access Manager.

To create an LDAP Server plug-in, you need to create a public class that extends the abstract the `com.novell.nam.common.ldap.jndi.LDAPStorePlugin` class.

In your public class, you need to implement the following methods:

Method	Description
getDirectoryName()	Needs to return the name you want displayed for your directory type. For eDirectory, this method returns “Novell eDirectory” for this string.

Method	Description
getGUIDAttributeName()	Needs to return the name of the globally unique ID attribute that uniquely identifies all objects in this type of directory. For eDirectory, this is the GUID attribute.
getMemberAttributeName()	Needs to return the name of the attribute that is used to identity an object as a member of a group. For eDirectory, this is the member attribute.
getUserClassName()	Needs to return the name of the class that is used to create users. For eDirectory, this is the User class.
getUserNameNamingAttrName( )	Needs to return the name of the attribute that is used to name users. For eDirectory, this in the cn attribute.
preUserAccountCreation()	Needs to return an attributes object that contains an array of attributes, with each member contain the name of an attribute and its value. This attributes object needs to contain all the attributes that are required to create a user in the LDAP directory. This usually consists of the name of the object class, the naming attribute, and a password. For eDirectory, this also includes the sn attribute.

The following methods can be implemented, and might be required for your LDAP directory:

Method	Description
postUserAccountCreation()	Modifies a user's attributes after the user has been created. Some LDAP directories do not let you set a password until after the user account has been created. The method contains a strCorrelationId parameter that you can use to match the user with the user in the preUserAccountCreation() method.
onCreateConnection()	Allows the plug-in to check the connection creation parameters and modify them, if needed. This method is called just before a connection is created with the LDAP directory.
onCreateConnectionException()	Allows you to customize the exception that is thrown when the process to create an LDAP connection fails and throws an authentication exception. This method is overloaded and requires an AuthenticationException parameter.
onCreateConnectionException()	Allows you to customize the exception that is thrown when the process to create an LDAP connection fails and throws a connection exception. This method is overloaded and requires an OperationNotSupportedException parameter.

Method	Description
getFailedLoginCountAttributeName()	Allows you to enable the reCAPTCHA feature. When the reCAPTCHA feature is enabled, the login page shows the Google reCAPTCHA box, so that the user trying to log in can confirm `I am not a robot`. The reCAPTCHA box does not appear unless the failed login count exceeds a specific number. This method returns the name of the attribute that is used to retrieve the bad password count or login intruder attempts. This method returns `loginIntruderAttempts for eDirectory` and `badPwdCount` for Active Directory.

If you are upgrading from Access Manager 4.2.x or earlier, add the following lines to the LDAP plug-in to avoid errors:

```
    /**
     * Returns the schema name of the Failed Login Attempts attribute for this
directory type. This is the attribute that indicates the attribute name for the
Failed Login Count. For example, for eDirectory, this method might return
'loginIntruderAttempts'.
     *
     * @return The schema name of the Failed Login Attempts attribute.
     */
    public abstract String getFailedLoginCountAttributeName();
    /**
public String getFailedLoginCountAttributeName(){
     return "<name of the attribute>"
}
```

For Active Directory, replace `<name of the attribute>` with `badPwdCount`. For eDirectory, replace `<name of the attribute>` with `loginIntruderAttempts`.

For details about the LDAPStorePlugin class and methods, see the Javadoc API Reference.

For an example plug-in that extends the LDAPStorePlugin class and implements the required methods and some of the optional methods, see Section 3.3, “eDirectory Plug-In,” on page 33.

3.3 eDirectory Plug-In

The following code is from the eDirectory plug-in:

```
package com.novell.nam.common.ldap.jndi;

import javax.naming.AuthenticationException;
import javax.naming.OperationNotSupportedException;
import javax.naming.directory.Attributes;
import javax.naming.directory.BasicAttributes;
import javax.naming.ldap.ExtendedRequest;
import javax.naming.ldap.ExtendedResponse;

import com.novell.nam.common.ldap.jndi.ext.GetEffectiveRightsRequest;
import com.novell.nam.common.ldap.jndi.ext.GetEffectiveRightsResponse;
import com.novell.nam.common.ldap.jndi.ext.NdsAttributeRights;
import com.novell.nam.common.ldap.jndi.ext.NdsEntryRights;
import com.novell.nam.common.ldap.jndi.ext.NdsRights;
```

```
public class LDAPStorePluginEDir extends LDAPStorePlugin
{
  @Override
    public String getDirectoryName()
    {
      return "Novell eDirectory";
    }

  @Override
    public String getGUIDAttributeName()
    {
      return "GUID";
    }

  @Override
    public String getMemberAttributeName()
    {
      return "member";
    }

  @Override
    public String getUserClassName()
    {
      return "User";
    }

  @Override
    public String getUserNamingAttrName()
    {
      return "cn";
    }

  @Override
  public String getFailedLoginCountAttributeName()
  {
    return "loginIntruderAttempts";
  }

  public Attributes preUserAccountCreation(String strCorrelationId, String name,
String password, String context)
    {
        Attributes  attrs = new BasicAttributes();
        attrs.put(JNDIConstants.LDAP_ATTR_OBJECTCLASS,"User");
        attrs.put(JNDIConstants.LDAP_ATTR_CN,name);
        attrs.put(JNDIConstants.LDAP_ATTR_SN,"NAM Generated");
        attrs.put("userPassword",password);
        return attrs;
    }

    public void onCreateConnectionException(AuthenticationException ae)
    throws JNDIException
  {
      // Check the return message to see if we can interpret it.
        String strDetails = ae.getMessage();
        // Look for "Incorrect Password"
        int iIdxLdapErrorCode = strDetails.indexOf(" 49 ");
        int iIdxNDSErrorCode = strDetails.indexOf("(-669)");
        if ((-1 != iIdxLdapErrorCode) && (-1 != iIdxNDSErrorCode))
        {
            if (iIdxLdapErrorCode < iIdxNDSErrorCode)
            {   // The user typed in an incorrect password
              throw new JNDIExceptionIncorrectPassword(ae,
ae.getLocalizedMessage());
            }
        }
        // Look for Expired Password
```

```
        iIdxLdapErrorCode = strDetails.indexOf(" 49 ");
        iIdxNDSErrorCode = strDetails.indexOf("(-222)");
        if ((-1 != iIdxLdapErrorCode) && (-1 != iIdxNDSErrorCode))
        {
            if (iIdxLdapErrorCode < iIdxNDSErrorCode)
            {   // The password for this user account has expired.
              throw new JNDIExceptionExpiredPassword(ae, ae.getLocalizedMessage());
            }
        }
  }

  public void onCreateConnectionException(OperationNotSupportedException onse)
    throws JNDIException
  {
    // Check the return message to see if we can interpret it.
        String strDetails = onse.getMessage();
        // Look for "Incorrect Password"
        int iIdxLdapErrorCode = strDetails.indexOf(" 53 ");
        if (iIdxLdapErrorCode != -1)
        {
            int iIdxNDSErrorCode = strDetails.indexOf("(-220)");

            // Check for account disabled (or a restriction has disabled the
account)
            if (iIdxNDSErrorCode != -1 && iIdxLdapErrorCode < iIdxNDSErrorCode)
                throw new JNDIExceptionDisabledAccount(onse,
onse.getLocalizedMessage());

            // Check for intruder detection disablement
            iIdxNDSErrorCode = strDetails.indexOf("(-218)");
            if (iIdxNDSErrorCode != -1 && iIdxLdapErrorCode < iIdxNDSErrorCode)
                throw new JNDIExceptionRestrictedAccount(onse,
onse.getLocalizedMessage());

            // Check for intruder detection disablement
            iIdxNDSErrorCode = strDetails.indexOf("(-197)");
            if (iIdxNDSErrorCode != -1 && iIdxLdapErrorCode < iIdxNDSErrorCode)
              throw new JNDIExceptionIntruderDetection(onse,
onse.getLocalizedMessage());
        }
  }

  public boolean supportsEffectiveRightsRetrieval()
  {
    return true;
  }

  public ExtendedRequest getEntryEffectiveRightsExtendedRequest(String objectDN,
String trusteeDN)
  {
    return new GetEffectiveRightsRequest(objectDN, trusteeDN);
  }

  public int getEntryEffectiveRights(ExtendedResponse response)
  {
    if (response instanceof GetEffectiveRightsResponse)
    {
      NdsRights rights = ((GetEffectiveRightsResponse)response).getRights();
      return rights.getRights();
    }
    return 0;
  }

  public ExtendedRequest getAttributeEffectiveRightsExtendedRequest(String
objectDN, String trusteeDN)
  {
```

```
    return new GetEffectiveRightsRequest(objectDN, trusteeDN,
NdsRights.ALL_ATTRIBUTES_RIGHTS);
  }

  public int getAttributeEffectiveRights(ExtendedResponse response)
  {
    if (response instanceof GetEffectiveRightsResponse)
    {
      NdsRights rights = ((GetEffectiveRightsResponse)response).getRights();
      return rights.getRights();
    }
    return 0;
  }

  public boolean hasEntrySupervisorRights(int iEntryRights)
  {
    return new NdsEntryRights(iEntryRights).hasSupervisor();
  }

  public boolean hasEntryBrowseRights(int iEntryRights)
  {
    return new NdsEntryRights(iEntryRights).hasBrowse();
  }

  public boolean hasEntryRenameRights(int iEntryRights)
  {
    return new NdsEntryRights(iEntryRights).hasRename();
  }

  public boolean hasEntryDeleteRights(int iEntryRights)
  {
    return new NdsEntryRights(iEntryRights).hasDelete();
  }

  public boolean hasEntryAddRights(int iEntryRights)
  {
    return new NdsEntryRights(iEntryRights).hasAdd();
  }

  public boolean hasAttributeCompareRights(int iAttributeRights)
    {
    return new NdsAttributeRights(NdsRights.ALL_ATTRIBUTES_RIGHTS,
iAttributeRights).hasCompare();
    }

    public boolean hasAttributeReadRights(int iAttributeRights)
    {
    return new NdsAttributeRights(NdsRights.ALL_ATTRIBUTES_RIGHTS,
iAttributeRights).hasRead();
    }

    public boolean hasAttributeWriteRights(int iAttributeRights)
    {
    return new NdsAttributeRights(NdsRights.ALL_ATTRIBUTES_RIGHTS,
iAttributeRights).hasWrite();
    }

    public boolean hasAttributeSelfRights(int iAttributeRights)
    {
    return new NdsAttributeRights(NdsRights.ALL_ATTRIBUTES_RIGHTS,
iAttributeRights).hasSelf();
    }

    public boolean hasAttributeSupervisorRights(int iAttributeRights)
    {
    return new NdsAttributeRights(NdsRights.ALL_ATTRIBUTES_RIGHTS,
```

```
iAttributeRights).hasSupervisor();
    }

  public boolean hasObjectSearchRights(int iEntryRights, int iAttributeRights)
  {
    NdsEntryRights entryRights = new NdsEntryRights(iEntryRights);
    NdsAttributeRights attributeRights = new
NdsAttributeRights(NdsRights.ALL_ATTRIBUTES_RIGHTS, iAttributeRights);
    if (entryRights.hasSupervisor())
    {  // Supervisor entry rights are sufficient for doing a user search
      return true;
    }
    if (entryRights.hasBrowse())
    {  // Browse entry rights plus supervisor/compare attribute rights are
sufficient for doing a user search
      if (attributeRights.hasSupervisor() || attributeRights.hasCompare())
      {
        return true;
      }
    }
    return false;
  }
}
```

3.4 Installing and Configuring the LDAP Plug-In

After you have created your plug-in, configure Access Manager to use it.

1. Add the plug-in class file to Identity Server to the following directory using Advanced File Configurator under the correct directory structure as per the class package:
   - If you want to use a LDAP-plugin class file: `/opt/novell/nam/idp/webapps/nidp/WEB-INF/classes`
   - If you want to use a LDAP-plugin class in a jar file: `/opt/novell/nam/idp/webapps/nidp/WEB-INF/lib`

   If your class package name is com.acme.ldap.plugin, you need to create the `com`, `acme`, `ldap`, and `plugin` directories.

   For information about how to add a file, see Adding Configurations to a Cluster in the NetIQ Access Manager 5.0 Administration Guide.
2. To associate an LDAP Server plug-in with the Custom1, Custom2, or Custom3 directory type, modify the Identity Server `web.xml` file.

   2a Open the web.xml file.

   For information about how to modify a file, see Modifying Configurations in the NetIQ Access Manager 5.0 Administration Guide.

   2b Add an entry for the ladpStorePlugins context parameter. Your entry should look similar to the following to associate the com.acme.plugin.Sample1Plugin with the Custom1 directory type.

   ```
   <context-param>
   <param-name>ldapStorePlugins</param-name>
   <param-value>custom1:com.acme.ldap.plugin.Sample1Plugin</param-value>
   </context-param>
   ```

   You can add up to three values using the `custom1:classname;custom2:classname;custom3:classname` format.

3 In Administration Console, configure Identity Server to use the new directory type for a user store.

3a Click **Access Manager** > **Identity Servers** > **Edit** > **Local**.

3b Either select the name of a user store or click **New**.

3c For the **Directory type**, select the custom string you have configured in Step 2.

3d Complete one of the following:

- For a new user store, configure the other required values, then click **Finish**.
- For a modified user store, modify the other options to fit the new directory type, then click **OK**.

3e Update Identity Server.

4 (Optional) To verify that the new directory type is functioning correctly, log in to the user portal by using the credentials of a user in the user store.

If you encounter any errors, see Section 3.5, “Troubleshooting,” on page 38.

3.5 Troubleshooting

If problems with LDAP Server plug-ins are detected, the following error messages are issued during Access Manager initialization. To log these messages to the `catalina.out` file, set the application component file logger to the warning level or higher.

300105029: Cannot load LDAP Store Plugin class: {0}. Error: {1}.

Cause: The java.lang.Class.forName() method failed to load the LDAP Store Plugin class.

Action: Verify that a valid Java class file is available in Access Manager's class path for the referenced plug-in class file. Check the modifications you made to the web.xml file (see Step 2 on page 37).

300105030=Cannot instantiate LDAP Store Plugin class: {0}. Error: {1}.

Cause: The java.lang.Class.newInstance() method failed to instantiate the LDAP Store Plug-in class.

Action: Verify that a valid Java class file is available in Access Manager's class path for the referenced plug-in class file. Also, ensure that the LDAP Store Plug-in has a zero parameter constructor.

300105031=An unknown or unsupported user store directory type {0} was found for the user store named {1}. Defaulting to eDirectory!

Cause: A user store was configured with an unrecognized directory type. The configuration was manually modified to include an invalid directory type specifier or the configuration has been corrupted.

Action: Examine the supplied error detail and take applicable actions. If the directory type is wrong, reconfigure the user store with the correct directory type. If the configuration is corrupted, delete the user store configuration, then re-create it.

4 The Policy Extension API

The policy extension API allows you to enhance the Access Manager policy engine so that an external module can perform the following types of tasks:

- Evaluate a condition and return results that Access Manager can use to determine enforcement.
- Provide data from an external source that Access Manager can use to evaluate a condition or to inject into an HTTP header.
- Provide actions that are performed when the policy conditions evaluate to True.

This section describes the basic characteristics of a policy extension, how to create the possible types of extensions, ands how to install and use the extension in an Access Manager policy.

4.1 Getting Started

4.1.1 Prerequisites

- ❒ Install and configure all components of Access Manager. For installation and configuration information, see the NetIQ Access Manager 5.0 Installation and Upgrade Guide and “Setting Up a Basic Access Manager Configuration” in the Access Manager 5.0 Administration Guide.
- ❒ A basic understanding of Access Gateway Authorization and Access Gateway Identity Injection policies. See “Access Manager Policies” in the Access Manager 5.0 Administration Guide.
- ❒ An integrated Java development environment.
- ❒ Download `nxpe.jar` from the `/opt/novell/nam/idp/webapps/nidp/WEB-INF/lib` directory (for roles) or `/opt/novell/nam/mag/webapps/nesp/WEB-INF/lib` (for other policies) of your Identity Server and add these to your development project.

  For information about how to download a file, see Downloading Files from a Server in the NetIQ Access Manager 5.0 Administration Guide.

  For information about how to add a file, see Adding Configurations to a Cluster in the NetIQ Access Manager 5.0 Administration Guide.

4.1.2 Types of Policy Extensions

You can use the policy extension API to create the following types of policy extensions:

- **Action:** This type of extension allows a new action to be added to the policy. When the policy is evaluated and the conditions evaluate to true, the extension is called so that it can execute its action. The action can be a permit, deny, or obligation. Actions extensions are used in Access Gateway Authorization policies.

  For example, when a user is denied access to an Access Gateway resource, the extension generates a dynamic page that is displayed to the user and updates a database with the details of the attempted access.

- **Condition:** This type of extension allows a new condition to be added to the policy. When the policy is evaluated, the extension is called to evaluate the condition and is responsible for returning a True, False, or Error value for the condition. Condition extensions are used in Access Gateway Authorization policies and Identity Server Role policies.

  For example, the Acme company requires historical sales records to be available via the corporate Intranet. Access to the records is granted according to regular procedures set up by the accounting department. The accounting department manages the access rights in a database that supports SQL. In order for Access Manager to take advantage of the access granting process already in place in the accounting department, a condition extension is created that queries the accounting access rights database and returns true, false, or error.

- **Data:** This type of extension retrieves data from an external source that can then be injected into a policy and used as input for evaluating a condition or an action. Data extensions can be used in Access Gateway Authorization policies, Access Gateway Identity Injection policies, Identity Server Role policies, External Attribute Source policies.

  For example, suppose a policy needs to use the role assignments made in an Oracle* database to determine whether a user is assigned an Access Manager role. The data extension could retrieve the role assignments from the database and return them in a string object that could be used by Access Manager in evaluating the condition for the Role policy.

4.1.3 How the Policy Engine Interacts with an Extension

When the policy engine processes a policy, the first step is to configure the policy. The following elements can be marked as external elements in the policy:

- Conditions
- Data elements
- Actions

When the policy engine configures a policy, it calls the extension if it encounters an external element. The engine expects the extension to return an extension type-specific object, unless an exception occurs. The object contains the data that the extension needs for processing, and the object is returned to the policy engine with the required data to continue processing the policy.

For specific details, see the following sections:

- Section 4.1.3.1, “How the Policy Engine Interacts with a Condition Extension,” on page 41
- Section 4.1.3.2, “How the Policy Engine Interacts with a Data Extension,” on page 41
- Section 4.1.3.3, “How the Policy Engine Interacts with an Action Extension,” on page 42

4.1.3.1 How the Policy Engine Interacts with a Condition Extension

When the policy engine processes a policy and encounters a condition marked as an extension, it instantiates an object that must comply with the NxpeConditionFactory interface. It then calls the getInstance method and expects an NxpeCondition object from the extension unless NxpeException is thrown by the NxpeConditionFactory object.

This process is illustrated in the following code snippet:

```
public interface NxpeConditionFactory
{
    NxpeCondition getInstance()
        throws NxpeException;
} /* NxpeConditionFactory */
```

The policy engine then calls the NxpeCondition.initialize method and sends an NxpeParameterList object for configuration parameters. Configuration parameters are used to initialize the NxpeCondition object. The extension needs these parameters for evaluating the condition. Values for these parameters are retrieved at evaluation from the NxpeInformationContext object that is sent by the policy engine.

The initialize method is called before any other method, followed by a method that sets an ID for the condition.

The following code snippet illustrates this process:

```
public interface NxpeCondition
{
    void initialize(
            NxpeParameterList   configurationValues)
        throws NxpeException;

        NxpeResult evaluate(
                        NxpeInformationContext   informationContext,
                        NxpeResponseContext      responseContext)
                throws NxpeException;

    void setInterfaceId(
            String interfaceId)
        throws NxpeException;
}
```

4.1.3.2 How the Policy Engine Interacts with a Data Extension

When the policy engine is processing a policy and encounters a data element marked as an extension, the engine instantiates an object that must comply with the NxpeContextDataElementFactory interface. The engine then calls the getInstance() method, passing the name, enumerativeValue, and parameter as arguments, and expects the extension to return an NxpeContextDataElement object unless the NxpeContextDataElementFactory object throws an NxpeException. The following code snippet illustrates this process:

```
public interface NxpeContextDataElementFactory
{
     NxpeContextDataElement getInstance(
            String    name,
            int       enumerativeValue,
            String    parameter)
        throws NxpeException;

} /* NxpeContextDataElementFactory */
```

During the next part of the configuration phase, the policy engine calls the NxpeContextDataElement.initialize() method, passing an NxpeParameterList object with configureParameters. The configureParameters are used to initialize the NxpeContextDataElement object and are the parameters required during policy evaluation. It is expected that the values for these configureParameters are retrieved from the NxpeInformationContext object passed by the policy engine.

The following code snippet illustrates this process:

```
public interface NxpeContextDataElement
{
    void initialize(
            NxpeParameterList   configurationValues)
        throws NxpeException;

    String getName();

    int getEnumerativeValue();

    String getParameter();

    Object getValue(
            NxpeInformationContext  informationContext,
            NxpeResponseContext     responseContext)
        throws NxpeException;

} /* NxpeContextDataElement */
```

The policy engine calls the NxpeContextDataElement.intialize() method to initialize a component in preparation for policy evaluation. Derived classes are required to implement this method. This method is guaranteed to be called before any other method is called, because it is part of object construction.

The configurationValues parameter contains a list of the configuration data required by the external ContextDataElement handler. If the context data element wants to preserve configuration data, it must maintain a reference to the configuration value parameters.

4.1.3.3 How the Policy Engine Interacts with an Action Extension

When the policy engine is processing a policy and encounters an action marked as an extension, the engine instantiates an object that must comply with the NxpeActionFactory interface. The engine then calls the getInstance() method, and expects the extension to return an NxpeAction object unless the NxpeActionFactory object throws an NxpeException.

This process is illustrated in the following code snippet:

```
public interface NxpeActionFactory
{
        NxpeAction   getInstance()
        throws NxpeException;

} /* NxpeActionFactory */
```

During the next part of the configuration phase, the policy engine calls the NxpeAction.initialize() method, passing an NxpeParameterList object with the configureParameters. The configureParameters are used to initialize the NxpeAction object. The configureParameters are those parameters needed during NxpePolicy.evaluate(). It is expected that the values for these configureParameters are retrieved from the NxpeInformationContext passed by the policy engine.

The following code snippet illustrates this process:

```
public interface NxpeAction
{
        void initialize(
            NxpeParameterList   configurationValues)
        throws NxpeException;
```

The NxpeParameterList is a list of configuration data required by the external action extension. If the action extension wants to preserve configuration data, the extension must maintain a reference to the configuration value parameters.

The second method called is the setInterfaceId method, which sets up a value for trace evaluation. The interfaceId parameter sets a unique sting value for the action. The following code snippet illustrates this last step in the NxpeAction interface.

```
    void setInterfaceId(
            String   interfaceId)
        throws NxpeException;

} /* NxpeAction */
```

The policy engine calls the doAction method to initiate the action. It has the following parameters:

- The informationCtx parameter contains the policy enforcement Point information context to query for values
- The responseCtx is a reflection object for communicating detailed response information back to the application. This is additional information and does not replace the need to place an action completion status in the return value from this call.

This method returns an NxpeResult, which contains an error code, permit, deny, or obligation. Derived classes are require to override this method to implement the supported action.

The following code snippet illustrates this process:

```
    NxpeResult doAction(
            NxpeInformationContext  informationCtx,
            NxpeResponseContext     responseCtx)
        throws NxpeException;
```

4.2 Common Elements and Tasks

As you develop your extension, the extension needs to perform the following tasks:

- Section 4.2.1, “Implementing Common Elements,” on page 44
- Section 4.2.2, “Initializing the Factory Object,” on page 45
- Section 4.2.3, “Retrieving Information from Identity Server User Store,” on page 46
- Section 4.2.4, “Implementing the Extension Interface,” on page 47

For information about the Extension API interfaces and class, see the Javadoc API Reference.

4.2.1 Implementing Common Elements

Each extension type has two interfaces:

- A factory interface that contains the method for initializing an extension object with data from the engine that the extension can use to retrieve data from an external source or to evaluate a condition or an action.
- An extension interface that contains the methods that need to be implemented for the specific type of extension. For example, the NxpeCondition interface contains the method for evaluating the condition and returning True, False, or Error.

All extensions need to implement both interfaces for the extension type and use the NxpeResult class for return codes and the NxpeException class for exceptions.

4.2.1.1 Return Codes in the NxpeResult Class

The NxpeResult class allows an extension to return the following values:

Return Code	Extension Type	Description
Cancel		Reserved
ConditionFalse	Condition	The compared values do not match, so the condition evaluation resolved to False.
ConditionTrue	Condition	The compared values match, so the condition evaluation resolved to True.
ConditionUnknown	Condition	The values could not be compared, so the results are unknown. This is comparable to the Result on Condition Error option when creating a policy.
Deny	Action	A deny action was applied.
ErrorBadData	Context Data	The data cannot be parsed. This result can be returned with the NxpeException class.
ErrorCodeComponent		Reserved.
ErrorConfigInitialization	All	The initialize method for the extension encountered an error. This result can be returned with the NxpeException class.
ErrorDataUnavailable	Context Data	The requested data is not available. This result can be returned with the NxpeException class.
ErrorIllegalArgument	All	The informationContext object contains an unknown parameter. This result can be returned with the NxpeException class.
ErrorIllegalState		Reserved
ErrorInterfaceUnavailable	All	The extension has not implemented one of the required methods in the interface. This result can be returned with the NxpeException class.
ErrorNoMemory		Reserved
GeneralFailure	All	Unknown error. This result can be returned with the NxpeException class.

Return Code	Extension Type	Description
NoAction		Reserved for use by the policy engine.
Obligation	Action	An obligation action was performed.
Pending		Reserved.
Permit	Action	A permit action was performed.
Success		Reserved for use by the policy engine.

4.2.1.2 Constructors in the NxpeException Class

You can use a constructor that throws exceptions with the following information:

- No information
- With a string message
- With a string message and a cause
- With a result from the NxpeResult class. See Return Codes in the NxpeResult Class.
- With a cause and a result from the NxpeResult class
- With a string message and a result from the NxpeResult class
- With a string message, a cause, and a result from the NxpeResult class

4.2.2 Initializing the Factory Object

All extension types need to implement the factory interface for the extension type and initialize an object specific to its type. The policy engine uses this object to send the parameter information about the user making the request to the extension. The extension uses this object to return its results to the policy engine.

The following code sample illustrates how to implement the factory interface. It uses the NxpeContextDataElementFactory to create an LDAPGroupDataElement object.

```
1  package ContextDataElement;
2
3  import com.novell.nxpe.NxpeContextDataElement;
4  import com.novell.nxpe.NxpeContextDataElementFactory;
5  import com.novell.nxpe.NxpeException;
6
7  public final class LDAPGroupDataElementFactory implements
NxpeContextDataElementFactory
8  {
9      public LDAPGroupDataElementFactory()
10     {
11     }
11     public NxpeContextDataElement getInstance(
12             String strName,
13             int iEnumerativeValue,
14             String strParameter)
15         throws NxpeException
16     {
17         return (new LDAPGroupDataElement(strName, iEnumerativeValue,
strParameter)
);
18   }
19 } /* LDAPGroupDataElementFactory */
```

The package line needs to be replaced with the package line for your extension.

All extensions need the three import lines for the factory interface. The first two import lines vary with the type of extension you are creating, but you need to import the factory interface and the extension interface.

Lines 7 through 19 implement the factory interface that creates an LDAPGroupDataElement object.

The other factory interfaces are very similar and are as easy to implement.

4.2.3 Retrieving Information from Identity Server User Store

All extensions need to access an external data store and retrieve information from it. You must know the type of data that your extension will retrieve, and then design how you are going to retrieve it.

If the extension needs to establish a connection to the external data store and log in to retrieve information, consider using one of the following methods:

- The extension can use the credentials that authenticated the user to Identity Server to log in as a user in the external data store. This method assumes that the user has the same credentials in Identity Server user store and the external data store.
- You can create an LDAP attribute in Identity Server user store and store an X.509 certificate that you can use to access the external data store.
- You can create configuration parameters that allow the administrator of Administration Console to enter a username and password for accessing the external data store. The password is entered in clear text in Administration Console, so this is not a secure method. To minimize the security risk, you can create a special user in the external data store whose rights are restricted to retrieving only the information required by the extension. If the retrieved information is not sensitive, this simple solution might not present a security risk.

When you create configuration parameters, you need to provide documentation for the administrator who installs the extension. Each configuration parameter requires a name, an ID, and a mapping to a data item. You need to document these for the administrator.

The name and ID you create to fit your programing requirements. These must be mapped to a data item available for the extension type.

NOTE: The data items are returned as strings or as string arrays if they are multivalued.

Your external data store and the methods available for accessing its data determine whether any of the data items are useful in making the connection to the external data store.

For the data items specific to an extension type, see the following:

- “Available Configuration Parameters for a Data Context Extension” on page 54
- “Available Configuration Parameters for a Condition Extension” on page 58
- “Available Configuration Parameters for an Action Extension” on page 60

4.2.4 Implementing the Extension Interface

All extensions need to perform the following tasks.

- Task 1: Specifying the Required Import Files
- Task 2: Defining the Configuration Parameters
- Task 3: Retrieving Configuration Parameters before Policy Evaluation
- Task 4: Implementing the Extension Methods
- Task 5: Retrieving Configuration Parameters at Policy Evaluation
- Task 6: Connecting with the External Data Source
- Task 7: Returning from an Extension
- Task 8: Error Handling
- Task 9: Performing Extension-Specific Tasks

4.2.4.1 Task 1: Specifying the Required Import Files

All extensions need a package line and the following import lines. The package line for the sample needs to be replaced with the package line for your extension. The first import line needs to be modified to import the interface for the extension type you are creating. The other import lines are standard for all extensions.

```
package ContextDataElement;

import com.novell.nxpe.NxpeContextDataElement;
import com.novell.nxpe.NxpeException;
import com.novell.nxpe.NxpeInformationContext;
import com.novell.nxpe.NxpeParameter;
import com.novell.nxpe.NxpeParameterList;
import com.novell.nxpe.NxpeResponseContext;
import com.novell.nxpe.NxpeResult;
```

The NxpeExpection class contains the defined constructors for throwing exceptions. For more information, see “Constructors in the NxpeException Class” on page 45.

The NxpeInformationContext class contains methods that allow you to gather information about extension evaluation.

The NxpeParameter class contains methods that allow you to retrieve information about a specific configuration parameter.

The NxpeParamaterList class contains methods that allow you to retrieve information about the configuration parameters you have defined for the extension.

The NxpeResponseContext class contains methods that allow you to configure the information that is sent with the results, such as logging or trace entry.

The NxpeResult class contains the methods and constants to set the return value for the extension. For more information, see “Return Codes in the NxpeResult Class” on page 44.

4.2.4.2 Task 2: Defining the Configuration Parameters

If your extension requires configuration parameters, you need to define them. The following code snippet contains the parameters for the LDAP group extension. These are the name and ID values that are configured on the Extension Details page (**Policies** > **Extensions** > **[Extension Name]**).

```
private static final String USER_STORE_NAME = "User Store";
    private static final int EV_USER_STORE = 11;

    private static final String AUTHENTICATION_NAME = "Authentication";
    private static final int EV_AUTHENTICATION = 211;
    private static final String DEFAULT_AUTHENTICATION = "simple";

    private static final String DIRECTORY_TYPE_NAME = "Directory Type";
    private static final int EV_DIRECTORY_TYPE = 222;
    private static final String DEFAULT_DIRECTORY_TYPE = "unknown";

    private static final String PROVIDER_URL_NAME = "User Store Replica";
    private static final int EV_PROVIDER_URL = 31;
    private static final String DEFAULT_PROVIDER_URL = "ldap://localhost:389";

    private static final String LDAP_USER_DN_NAME = "LDAP User DN";
    private static final int EV_LDAP_USER_DN = 41;

    private static final String SECURITY_PRINCIPAL_NAME = "Security Principal";
    private static final int EV_SECURITY_PRINCIPAL = 51;

    private static final String SECURITY_CREDENTIALS_NAME = "Security Credentials";
    private static final int EV_SECURITY_CREDENTIALS = 52;

    private static final String SEARCH_CONTEXT_NAME = "Search Context";
    private static final int EV_SEARCH_CONTEXT = 61;

    private static final String DEBUG_NAME = "Debug";
    private static final int EV_DEBUG = 91;
```

Not all of the parameters need to be defined in Administration Console. If you want the administrator to decide the value that is mapped to the parameter, then you need to document the parameter and let the administrator select the mapping.

This is also a good place to define any other static constants your extension needs.

4.2.4.3 Task 3: Retrieving Configuration Parameters before Policy Evaluation

If your extension needs to be aware of some parameter values before it is called during policy evaluation, you can retrieve the values during the initialize method. Each extension interface (NxpeAction, NxpeCondition, NxpeContextDataElement) has an initialize method that contains a configurationValues object. The following code snippet illustrates the LDAP group extension defined for this method. The setDebug line shows how to obtain the current value for the debug parameter.

```
public void initialize(
          NxpeParameterList configurationValues)
       throws NxpeException
   {
       this.configurationValues = configurationValues;

       setDebug(configurationValues);

       strProviderURL = DEFAULT_PROVIDER_URL;
       strAuthentication = DEFAULT_AUTHENTICATION;
       strDirectoryType = DEFAULT_DIRECTORY_TYPE;

       StringBuffer sbLdapFilter = new StringBuffer(128);

       // setup filter
       sbLdapFilter.append("(|(objectClass=");
       sbLdapFilter.append(CLS_GROUP);
       sbLdapFilter.append(")(objectClass=");
       sbLdapFilter.append(CLS_GROUPOFNAMES);
       sbLdapFilter.append(")(objectClass=");
       sbLdapFilter.append(CLS_GROUPOFUNIQUENAMES);
       sbLdapFilter.append("))");

       strLdapFilter = new String(sbLdapFilter);

       // setup search controls
       searchControls = new SearchControls();
       searchControls.setTimeLimit(0);
       searchControls.setReturningObjFlag(true);
       searchControls.setSearchScope(SearchControls.SUBTREE_SCOPE);
       searchControls.setReturningAttributes(new String[] { ATTR_CN });

   }
```

4.2.4.4 Task 4: Implementing the Extension Methods

Besides having an initialize method, each extension interface has a few other methods that need to be implemented. The NxpeContextDataElement interface has the get methods. The following code snippet illustrates how the LDAP Group extension implements three of these methods.

```
public int getEnumerativeValue()
    {
        return (iEnumerativeValue);
    }
    public String getName()
    {
        return (strName);
    }
    public String getParameter()
    {
        return (strParameter);
```

NxpeContextDataElement introduces a new element with additional methods. Using these methods, you can set the duration for which the data returned from the extension interface is cached by Access Manager.

```
public int getValidForSeconds()
    {
       return -1;
    }
    public int getValidForSeconds ()
    {
     return 0;
    }
   public int getValidForSeconds ()
    {
     return n;
    }
```

getValidForSeconds informs the policy engine about how often data needs to be queried. Specify *0* as the return value to query data for each request. Specify *-1* as the return value to cache the data. Substitute *n* with the number of seconds to indicate validity of the data.

The fourth method (the getValue method) is described in the next section. See "Task 5: Retrieving Configuration Parameters at Policy Evaluation" on page 50.

4.2.4.5 Task 5: Retrieving Configuration Parameters at Policy Evaluation

All extension interfaces have a method for retrieving configuration parameters at policy evaluation. The NxpeCondition interface has an evaluate method with an informationContext object. The NxpeAction interface has a doAction method with a informationCxt object. The NxpeContextDataElement interface has a getValue method with an informationContext object. The informationContext object contains information about the user and the user's request that you need. You populate this object with the parameters that you need to evaluate the policy, and the policy engine supplies the values.

The following code snippet illustrates how the LDAP Group extension retrieves parameter values:

```
public synchronized Object getValue(
            NxpeInformationContext informationContext,
            NxpeResponseContext responseContext)
        throws NxpeException
    {
        LdapContext ldapContext = null;

        String strUserStore = getUserStore(informationContext);
        String strProviderURL = getProviderURL(informationContext);
        String strAuthentication = getAuthentication(informationContext);
        String strDirectoryType = getDirectoryType(informationContext);

        String strLDAPUserDN = getLDAPUserDN(informationContext);
        String strDN = getSecurityPrincipal(informationContext);

        if (strLDAPUserDN == null)
        {
            strLDAPUserDN = strDN;
        }

        String strPassword = getSecurityCredentials(informationContext);
        String strSearchContext = getSearchContext(informationContext);
```

Notice that this code snippet does not have an ending parenthesis. All the main work of the extension is done in this method. The next two tasks (Task 6: Connecting with the External Data Source and Task 7: Returning from an Extension) are performed within the getValue method.

4.2.4.6 Task 6: Connecting with the External Data Source

How you connect to the external data source in your extension is specific to the type of data source you are using. The following code snippet from the LDAP Group extension file illustrates how to connect to an LDAP user store:

```
try
        {
            HashSet<String> groupDNs = new HashSet<String>();

            ldapContext = newInitialLdapContext(strDN, strPassword);

            NamingEnumeration neGroups = ldapContext.search(strSearchContext,
strLdapMemberFilter, searchControls);
```

This piece of code is very specific to LDAP.

4.2.4.7 Task 7: Returning from an Extension

The following code snippet from the LDAP Group extension illustrates the tasks you need to complete as you return the results of your extension action/evaluation to the policy engine:

```
while (neGroups.hasMore())
          {
              Attribute cn;
              SearchResult srGroup = (SearchResult) neGroups.next();
              String strGroupDN = srGroup.getNameInNamespace();

              groupDNs.add(strGroupDN);

              if (debug)
              {
                  System.out.println("LDAPGroupDataElement: \"" + strGroupDN +
"\"");

              }
          }

          String[] strGroupDNs = new String[groupDNs.size()];

          groupDNs.toArray(strGroupDNs);
          return (strGroupDNs);
```

This code searches through the LDAP search results, retrieves the DN of any group found, adds it to the array, then returns the array.

This task is specific to the purpose of the extension. If the purpose of the extension is to evaluate a condition and determine whether the user matches the condition, the code for this task should show the extension obtaining the user's value for the condition, comparing that value to the expected value, then return True for a match, False for a mismatch, and Error if extension cannot perform the evaluation.

4.2.4.8 Task 8: Error Handling

Each extension must handle potential error conditions. The following snippet illustrate how the LDAP Group extension handles potential errors:

```
catch (NamingException e)
        {
            if (debug)
            {
                e.printStackTrace();
            }
            throw (new NxpeException(NxpeResult.ErrorDataUnavailable, e));
        }
        finally
        {
            if (ldapContext != null)
            {
                try
                {
                    ldapContext.close();
                }
                catch (NamingException e)
                {
                    if (debug)
                    {
                        System.out.println(e.getMessage());
                    }
```

4.2.4.9 Task 9: Performing Extension-Specific Tasks

After your extension has implemented all the required interface methods, the rest of the code implements what the extension requires to perform its purpose. Everything that follows the

********** LDAPGroupDataElement/private **********

comment in the `LDAPGroupDataElement.java` file shows how the LDAP Group extension performs its required tasks. For example, you can see how the extension retrieves parameter information from the policy engine, such as the user's DN, security credentials, and user store information. With this information the extension interacts with the LDAP user store and retrieves the groups the user belongs to.

4.3 Creating an Extension

You can create the following types of extensions:

- Section 4.3.1, "Creating a Context Data Extension," on page 53
- Section 4.3.2, "Creating a Condition Extension," on page 57
- Section 4.3.3, "Creating an Action Extension," on page 59

4.3.1 Creating a Context Data Extension

A context data extension can be used for a Role policy, an Authorization policy, an Identity Injection policy, or an External Attribute Source policy. When the extension is used for an Authorization policy, it can only be used to evaluate a condition. When it is used for a Role policy, it can be designed to do the following:

- A condition to determine whether the user meets the requirements for a role assignment
- An action for activating roles based on the values returned by the extension.

When the extension is used for an Identity Injection policy, it injects data into the Authentication header, the custom header, or the query string.

The following sections describe the interfaces, methods, and configuration parameters available for a context data extension:

- Section 4.3.1.1, “Context Data Interfaces and Methods,” on page 53
- Section 4.3.1.2, “Available Configuration Parameters for a Data Context Extension,” on page 54

For sample code for this type of extension, see the `LDAPGroupDataElement.java` and `LDAPGroupDataElementFactory.java` file.

4.3.1.1 Context Data Interfaces and Methods

When creating a context data element extension, you need to implement the following interfaces and methods:

Interface	Method	Purpose
NxpeContextDataElementFactory		Contains the method required to create a context data element object.
	getinstance	Creates the NxpeContextDataElement object.
NxpeContextDataElement		Contains methods required to create a context data element that can be used for injection, for activating roles, or in a condition.
	initialize	Called by policy engine and therefore must be implemented. It initializes the element and sends configuration values you have requested to your extension. These parameters contain valid information only if the parameters contain information independent of the request that triggers policy evaluation. The data in the configurationValues parameter is valid only during the lifetime of the initialize method. If your extension needs to preserve this configuration data, you must maintain a reference. The get methods in this interface allow you to retrieve information about the parameters when the policy is being evaluated.
	getEnumerativeValue	Returns -1. Reserved for future releases.
	getName	Retrieves the name of the data element of the policy.
	getParameter	Retrieves the string value of the parameter of the policy.

Interface	Method	Purpose
	getValue	Called by the policy engine when a request triggers a policy evaluation. The informationContext object contains parameter values that you need from the policy engine for the evaluation.

When you configure a condition in a policy, you select a condition and a value. The condition sets up the left operand for the comparison and the value sets up the right operand for the comparison.

4.3.1.2 Available Configuration Parameters for a Data Context Extension

You can use any of the data items listed in the Table 4.1 to create configuration parameters to retrieve information about the request and the user making the request. Select the parameters that are useful for your extension. Many of the available data items might not be useful for your implementation.

- Table 4-1, “Configuration Parameters for a Role Policy,” on page 54
- Table 4-2, “Configuration Parameters for an Identity Injection Policy,” on page 55
- Table 4-3, “Configuration Parameters for an Authorization Policy,” on page 55
- Table 4-4, “Configuration Parameters for an External Attribute Source Policy,” on page 56

Table 4-1 *Configuration Parameters for a Role Policy*

Data Item	Returns
Authentication IDP	The name of Identity Server that authenticated the user.
Authenticating Contact	The URI of the contract that the user used for authentication.
Authentication Method	The name of the method the user used for authentication.
Authentication Type	The type of authentication the user used, such as Name Password, Secure Name Password, x509, Smart Card, Smart Card PKI, and Token.
Credential Profile	The credentials the user used for authentication, such as LDAP Credentials (CN, DN, and password), X509 Credentials (with certificate subject, with certificate issuer, with public certificate, and with serial number), and SAML Credentials. If a custom contract is created that uses other credentials for authentication, these credentials are not available within the credential profile.
LDAP Group	The DNs of any LDAP groups the user belongs to. If it is multi-valued, this item returns a string array.
LDAP OU	The DNs of any OUs that are part of the user’s DN. If it is multi-valued, this item returns a string array.
LDAP Attribute	The value or values stored in the specified LDAP attribute. If it is multi-valued, this item returns a string array.
Liberty User Profile	The value or values stored in the specified Liberty User Profile attribute.
Roles from Identity Provider	The names of the Roles assigned to the user by Identity Server when the user authenticated. If it is multi-valued, this item returns a string array.

Data Item	Returns
User Store	The name of the user store that authenticated the user.
User Store Replica	The URL of the replica that authenticated the user.
String Constant	The static value the administrator has been instructed to enter.

Table 4-2 *Configuration Parameters for an Identity Injection Policy*

Data Item	Returns
Authenticating Contact	The URI of the contract that the user used for authentication.
Client IP	The IP address of the user.
Credential Profile	Credentials the user used for authentication, such as LDAP Credentials (CN, DN, and password), X509 Credentials (with certificate subject, with certificate issuer, with public certificate, and with serial number), and SAML Credentials. If a custom contract has been created that uses other credentials for authentication, these credentials aren't available within the credential profile.
LDAP Attribute	The value or values stored in the specified LDAP attribute. If it is multi-valued, this item returns a string array.
Liberty User Profile	The value or values stored in the specified Liberty User Profile attribute.
Proxy Session Cookie	The session cookie associated with the user.
Roles	The roles that have been assigned to the user
Shared Secret	The value of the specified shared secret.
String Constant	The static value the administrator has been instructed to enter.

Table 4-3 *Configuration Parameters for an Authorization Policy*

Data Item	Returns
Authentication Contract	The URI of the contract used for authentication or the URI of the specified contract.
Client IP	The IP address of the user.
Credential Profile	The credentials of the user. You can ask for LDAP credentials (username, DN, and password), X.509 credentials (public certificate subject, public certificate issuer, public certificate, serial number), or the SAML assertion.
Current Date	The date when the request was sent.
Day of Week	The day when the request was sent.
Current Day of Month	The day of the month when the request was sent.
Current Time of Day	The time of day when the request was sent.
HTTP Request Method	The HTTP method in the request.

Data Item	Returns
LDAP Attribute	The value of the specified LDAP attribute.
LDAP OU	The value of any OUs in the user's DN.
Liberty User Profile	The value of the specified Liberty attribute.
Roles	The roles that have been assigned to the user.
URL	The URL of the current request.
URL Scheme	The HTTP scheme (HTTP or HTTPS) of the current request.
URL Host	The hostname specified in the URL of the current request.
URL Path	The path specified in the URL of the current request.
URL File Name	The filename specified in the URL of the current request.
URL File Extension	The file extension specified in the URL of the current request.
X-Forwarded-For IP	The value in the X-Forwarded-For header in the current request.
String Constant	The static value the administrator has been instructed to enter.

Table 4-4 *Configuration Parameters for an External Attribute Source Policy*

Data Item	Returns
Authentication IDP	The name of Identity Server that authenticated the user.
Authenticating Contact	The URI of the contract that the user used for authentication.
Authentication Method	The name of the method the user used for authentication.
Authentication Type	The type of authentication the user used, such as Name Password, Secure Name Password, x509, Smart Card, Smart Card PKI, and Token.
Credential Profile	Credentials the user used for authentication, such as LDAP Credentials (CN, DN, and password), X509 (with certificate subject, with certificate issuer, with public certificate, and with serial number), and SAML Credentials. If a custom contract uses other credentials for authentication, these credentials are not available within the credential profile.
LDAP Group	DNs of any LDAP groups the user belongs to. If it is multi-valued, this item returns a string array.
LDAP OU	DNs of any OUs that are part of the user's DN. If it is multi-valued, this item returns a string array.
LDAP Attribute	The values stored in the specified LDAP attribute. If it is multi-valued, this item returns a string array.
Liberty User Profile	The values stored in the specified Liberty User Profile attribute.
Roles from Identity Provider	The names of the Roles assigned to the user by Identity Server when the user authenticated. If it is multi-valued, this item returns a string array.
User Store	The name of the user store that authenticated the user.

Data Item	Returns
User Store Replica	The URL of the replica that authenticated the user.
String Constant	The static value the administrator has been instructed to enter.

4.3.2 Creating a Condition Extension

A condition extension can be used in a Role policy or an Authorization policy. In both types of policy, the policy engine provides the extension with some data about the user and the request. The extension retrieves additional data from an external source, then evaluates the condition. The extension returns True, False, or Error to the policy engine.

The following sections describe the interfaces, methods, and configuration parameters available for a condition extension.

- Section 4.3.2.1, “Interfaces and Methods for a Condition Extension,” on page 57
- Section 4.3.2.2, “Available Configuration Parameters for a Condition Extension,” on page 58

4.3.2.1 Interfaces and Methods for a Condition Extension

When creating a condition extension, you need to implement the following interfaces and methods:

Interface	Method	Purpose
NxpeConditionFactory		Contains the method required to create a condition object.
	getInstance	Creates the NxpeCondition object.
NxpeCondition		Contains the methods required to evaluate the condition for a policy.
	initialize	Called by policy engine and therefore must be implemented. It initializes the element and passes to your extension any configuration values you have requested. These parameters contain valid information only if the parameters contain information independent of the request that triggers policy evaluation. The data in the configurationValues parameter is valid only during the lifetime of the initialize method. If your extension needs to preserve this configuration data, you must maintain a reference.
	evaluate	Called by the policy engine when the condition extension needs to be evaluated for a policy. The informationContext parameter contains the parameter information the extension needs from the policy engine to evaluate the condition. The responseContext parameter contains the results of the extension’s evaluation of the condition.
	setInterfaceId	Sets the unique string value for the condition. This value is used for tracing evaluation.

4.3.2.2 Available Configuration Parameters for a Condition Extension

You can use the configuration parameters to gather information about the user. You can then use this information when evaluating your condition and use it to determine whether the condition should return True or False. The available configuration parameters depend upon whether it is a condition for a Role policy or a condition for a Authorization policy. Select the parameters that are useful for your extension. Many of the available data items might not be useful for your implementation.

- Table 4-5, “Configuration Parameters for a Role Condition,” on page 58
- Table 4-6, “Configuration Parameters for an Authorization Condition,” on page 59

Table 4-5 *Configuration Parameters for a Role Condition*

Data Item	Returns
Authentication IDP	The name of Identity Server that authenticated the user.
Authenticating Contact	The URI of the contract that the user used for authentication.
Authentication Method	The name of the method the user used for authentication.
Authentication Type	The type of authentication the user used, such as Name Password, Secure Name Password, x509, Smart Card, Smart Card PKI, and Token.
Credential Profile	The credentials the user used for authentication, such as LDAP Credentials (CN, DN, and password), X509 Credentials (with certificate subject, with certificate issuer, with public certificate, and with serial number), and SAML Credentials. If a custom contract has been created that uses other credentials for authentication, these credentials are not available within the credential profile.
LDAP Group	The DNs of any LDAP groups the user belongs to. If it is multi-valued, this item returns a string array.
LDAP OU	The DNs of any OUs that are part of the user’s DN. If it is multi-valued, this item returns a string array.
LDAP Attribute	The value or values stored in the specified LDAP attribute. If it is multi-valued, this item returns a string array.
Liberty User Profile	The value or values stored in the specified Liberty User Profile attribute.
Roles from Identity Provider	The names of the Roles assigned to the user by Identity Server when the user authenticated. If it is multi-valued, this item returns a string array.
User Store	The name of the user store that authenticated the user.
User Store Replica	The URL of the replica that authenticated the user.
String Constant	The static value the administrator has been instructed to enter.

Table 4-6 *Configuration Parameters for an Authorization Condition*

Data Item	Returns
Authentication Contract	The URI of the contract used for authentication or the URI of the specified contract.
Client IP	The IP address of the user.
Credential Profile	The credentials of the user. You can ask for LDAP credentials (username, dn, and password), X.509 credentials (public certificate subject, public certificate issuer, public certificate, serial number), or the SAML assertion.
Current Date	The date when the request was sent.
Day of Week	The day when the request was sent.
Current Day of Month	The day of the month when the request was sent.
Current Time of Day	The time of day when the request was sent.
Destination IP	The destination IP address of the request.
HTTP Request Method	The HTTP method in the request.
LDAP Attribute	The value of the specified LDAP attribute.
LDAP OU	The value of any OUs in the user's DN.
Liberty User Profile	The value of the specified Liberty attribute.
Roles	The roles that have been assigned to the user.
URL	The URL of the current request.
URL Scheme	The HTTP scheme (HTTP or HTTPS) of the current request.
URL Host	The hostname specified in the URL of the current request.
URL Path	The path specified in the URL of the current request.
URL File Name	The filename specified in the URL of the current request.
URL File Extension	The file extension specified in the URL of the current request.
X-Forwarded-For IP	The value in the X-Forwarded-For header in the current request.
String Constant	The static value the administrator has been instructed to enter.

4.3.3 Creating an Action Extension

There are the three types of actions: deny, permit, and obligation. The following sections describe the interfaces, methods, and configuration parameters available for an action extension.

- Section 4.3.3.1, "Action Interfaces and Methods," on page 60
- Section 4.3.3.2, "Actions," on page 60
- Section 4.3.3.3, "Available Configuration Parameters for an Action Extension," on page 60

4.3.3.1 Action Interfaces and Methods

When creating an action extension, you need to implement the following interfaces and methods:

Interface	Method	Purpose
NxpeActionFactory		Contains the methods required to create an action object.
	getInstance	Creates the NxpeAction object.
NxpeAction		Contains the methods required to implement a deny, permit, or obligation action.
	Initialize	Called by the policy engine and therefore must be implemented. It initializes the element and passes to your extension any configuration values you have requested. These parameters contain valid information only if the parameters contain information independent of the request that triggers policy evaluation. The data in the configurationValues parameter is valid only during the lifetime of the initialize method. If your extension needs to preserve this configuration data, you must maintain a reference.
	doAction	Called by the policy engine when the action extension needs to be evaluated for a policy. The informationCtx parameter contains the parameter information the extension needs from the policy engine to evaluate the condition. The responseCtx parameter contains the results of the action.
	setInterfaceId	Sets the unique string value for the action. This value is used for tracing the action during policy evaluation.

4.3.3.2 Actions

A policy rule can have multiple obligation actions but only one terminating action of either permit or deny. A permit or deny action needs to return either success or failure to the policy engine. An obligation action can return either success or failure; the policy engine just needs the acknowledgment that the obligation extension has performed its action.

An extension that implements an obligation action can use the doAction method to enter a log or audit event in another system or send an email message.

An extension that implements a deny or permit action can use the doAction method to ask another database or policy to evaluate a condition and then return the results of that evaluation to the Access Manager policy engine.

4.3.3.3 Available Configuration Parameters for an Action Extension

You can use any of the data items in the list to retrieve information about the user and the user's request to create a configuration parameter. Your extension can then use this information in determining the type of action to take. Select the parameters that are useful for your extension. Many of the available data items might not be useful for your implementation.

Data Item	Returns
Authentication Contract	The URI of the contract used for authentication or the URI of the specified contract.
Client IP	The IP address of the user.
Credential Profile	The credentials of the user. You can ask for LDAP credentials (username, dn, and password), X.509 credentials (public certificate subject, public certificate issuer, public certificate, serial number), or the SAML assertion.
Current Date	The date when the request was sent.
Day of Week	The day when the request was sent.
Current Day of Month	The day of the month when the request was sent.
Current Time of Day	The time of day when the request was sent.
HTTP Request Method	The HTTP method in the request.
LDAP Attribute	The value of the specified LDAP attribute.
LDAP OU	The value of any OUs in the user's DN.
Liberty User Profile	The value of the specified Liberty attribute.
Roles	The roles that have been assigned to the user.
URL	The URL of the current request.
URL Scheme	The HTTP scheme (HTTP or HTTPS) of the current request.
URL Host	The hostname specified in the URL of the current request.
URL Path	The path specified in the URL of the current request.
URL File Name	The filename specified in the URL of the current request.
URL File Extension	The file extension specified in the URL of the current request.
X-Forwarded-For IP	The value in the X-Forwarded-For header in the current request.
String Constant	The static value the administrator has been instructed to enter.

4.4 Installing and Configuring an Extension

After you have created your extension, you need to install it, configure it, and distribute it.

- Section 4.4.1, "Installing the Extension on Administration Console," on page 62
- Section 4.4.2, "Distributing a Policy Extension to Access Manager Devices," on page 63
- Section 4.4.3, "Distributing the Extension to Customers," on page 64

4.4.1 Installing the Extension on Administration Console

To install an extension, you need to have access to the JAR file and know the following information about the extension or extensions contained within the file.

What you need to create	A display name for the extension. A description for the extension.
What you need to know	The policy type of the extension, which defines the policy type it can be used with. You should know whether it is an extension for an Access Gateway Authorization policy, an Access Gateway Identity Injection policy, or an Identity Server Role policy. The name of the Java class that is used by the extension. Each data type usually uses a different Java factory class. The filename of the extension. The type of data the extension manipulates. **Authorization Policy:** Can be used to return: ◆ An action of deny, permit, or obligation. ◆ A condition that the extension evaluates and returns either true or false. ◆ A data element that the extension retrieves and the policy can use for evaluating a condition. **Identity Injection Policy:** A data extension that retrieves data for injecting into a header. **Identity Role Policy:** Can be used to return: ◆ A condition that the extension evaluates and returns either true or false ◆ A data element that the extension retrieves, which can be used in evaluating a condition or used to assign roles **External Attribute Source Policy:** You can use it to: ◆ Get attributes from the external sources. ◆ Create shared secrets. This shared secret then can be used in configuring other policies or can be used by Identity Servers in their attribute sets. The names, IDs, and mapping type of any configuration parameters. Configuration parameters allow the policy engine to pass data to the extension, which the extension can then use to retrieve data or as part of its evaluation.

If the file contains more than one extension, create a configuration for each extension in the file.

1. Copy the JAR file to a location that you can browse to from Administration Console.
2. In Administration Console, click **Policies** > **Extensions**.
3. To upload the file, click **Upload** > **Browse**, select the file, then click **Open**.
4. (Conditional) If you want this JAR file to overwrite an existing version of the file, select **Overwrite existing *.jar file**.
5. Click **OK**.

The file is uploaded to Administration Console, but nothing is visible on the Extensions page until you create a configuration.

6 To create an extension configuration, click **New**, then fill in the following fields:

Name: Specify a display name for the extension.

Description: (Optional) Specify the purpose of the extension and how it should be used.

Policy Type: Select the type of extension you have uploaded.

Type: Select the data type of the extension.

Class Name: Specify the name of the class that creates the extension, for example com.acme.policy.action.successActionFactory.

File Name: Select the JAR file that contains the Java class that implements the extension and its corresponding factory. This should be the file you uploaded in Step 3.

7 Click **OK**.

8 (Conditional) If the extension requires data from Access Manager, click the name of the extension.

9 In the **Configuration Parameters** section, click **New**, specify a name and ID, then click **OK**.

The developer of the extension must supply the name and ID that the extension requires.

10 In the **Mapping** column, select the required data type.

The developer of the extension must supply the data type that is required. If the data type is a data string, then the developer needs to explain the type of information you need to supply in the text field.

11 (Conditional) If the extension requires more than one data item, repeat Step 9 and Step 10.

12 Click **OK**.

The extension is now available for the policy type it was created for.

13 (Conditional) If the class can be used for multiple policy types, you need to create an extension configuration for each policy type.

For example, if an extension can be used for both an Identity Injection policy and a Role policy, you need to create an entry for both. The **File Name** option should contain the same value, but the other options should contain unique values.

14 Continue with Distributing a Policy Extension to Access Manager Devices.

4.4.2 Distributing a Policy Extension to Access Manager Devices

To distribute the policy extension to the devices that need it:

1 Create a Role, Identity Injection, or Authorization policy that uses the extension.

2 Assign the policy to a device:

- For a Role policy, enable it for an Identity Server.
- For an Authorization policy, assign it to a protected resource.
- For an Identity Injection policy, assign it to a protected resource.

IMPORTANT: Do not update the device at this time. The JAR files must be distributed before you update the device.

---

3 Distribute the JAR files:

3a Click **Policies** > **Extensions**.

3b Select the extension, then click **Distribute JARs**.

3c Restart services on the devices listed for reboot.

Identity Server: `/etc/init.d/novell-idp restart`

Access Gateway: `/etc/init.d/novell-mag restart`

4 (Conditional) If the extension is for an Authorization policy or an Identity Injection policy, update Access Gateway.

5 (Conditional) If the extension is for a Role policy, update Identity Server.

4.4.3 Distributing the Extension to Customers

You can distribute the extension as either a JAR file or as a ZIP file. If the extension contains multiple types of extensions or contains multiple configuration parameters, you might want to consider distributing the extension as a ZIP file.

Import your JAR file and configure it as described in Installing the Extension on Administration Console. After it has been configured, you can export it as a ZIP file. Your users can then import the ZIP file, and each extension type you have created is imported with its configuration parameters. In the documentation you create for the extension, document any parameter the user needs to modify after the import.

To export an extension:

1 In Administration Console, click **Policies** > **Extensions**.

2 Select all the extensions that are part of your JAR file.

If you have more than one JAR file, you can select the extensions that belong to it and include them in the same export.

3 Click **Export**, specify a name for the file, then click **OK**.

4 Follow your browser prompts to save the file to disk.

4.5 Sample Codes

You can find the sample codes for the following extensions in the NetIQ Access Manager SDK Sample Code page.

4.5.1 Data Extension for External Attribute Source Policy

This example demonstrates how an External Attribute Source policy retrieves information from external sources. It provides details about:

- How to configure and install the External Attribute Source Data policy extension in Administration Console.
- Implementation details of the extension factory and extension classes.
- How to use the information retrieved from the External Attribute Source policies as shared secret. It also explains how to use that shared secret to configure other policies or use them in Identity Servers to retrieve attributes from external sources.

The policy extension example includes `NameAttributeFromMailIDFactory.java` and `NameAttributeFromMailID.java`.

4.5.2 Template Policy Extensions

This includes the following two types:

- Template Condition Policy
- Template Data Policy
- Template Action Policy

4.5.2.1 Template Condition Policy

You can use this example as a template to implement a policy extension of type `Condition` that is com.novell.nxpe.NxpeCondition. This example provides a basic framework that can be used as a starting point for creating data policy (com.novell.nxpe.NxpeContextDataElement.) extensions. It provides details about:

- How to configure and install a `Condition` policy extension in Administration Console.
- Implementation details of the extension factory and extension classes.

The policy extension example includes `PolicyConditionExtnFactoryTemplate.java` and `PolicyConditionExtnTemplate.java`.

4.5.2.2 Template Data Policy

You can use this example as a template to implement a policy extension of type `Data` that is com.novell.nxpe.NxpeContextDataElement. This example provides a basic framework that can be used as a starting point for creating such policy extensions. It provides details about:

- How to configure and install the Data policy extension in Administration Console.
- Implementation details of the extension factory and extension classes.

The policy extension example includes `PolicyDataExtnFactoryTemplate.java` and `PolicyDataExtnTemplate.java`.

4.5.2.3 Template Action Policy

You can use this example as a template to implement a policy extension of type `Action` that is com.novell.nxpe.NxpeContextActionElement. The action policy extension are of the following types: Permit, Deny, and Obligation. This example provides a basic framework that can be used as a starting point for creating such policy extensions. It provides details about:

- How to configure and install the Action policy extension - Permit, Deny, and Obligation, in Administration Console.
- Implementation details of the extension factory and extension classes.

The policy extension example includes:

- PolicyActionExtnDenyFactoryTemplate.java
- PolicyActionExtnDenyTemplate.java
- PolicyActionExtnPermitFactoryTemplate.java
- PolicyActionExtnPermitTemplate.java

4.5.3 LDAP Group Data Element

This example illustrates how a policy extension can use external data sources to obtain information. This policy extension connects to the required LDAP repository, runs a search on it, and returns the results. An Identity Injection policy is created in this example that uses this policy extension.

The policy extension example includes `LDAPGroupDataElement.java` and `LDAPGroupDataElementFactory.java`.

4.5.4 PasswordClass

This authentication class extends the base class LocalAuthenticationClass and performs a form based authentication. The policy extension example includes `passwordClass.java`.

For more information, see Section 2.4, “Authentication Class Example,” on page 20 and Section 2.6, “Deploying Your Authentication Class,” on page 29.

5 Custom Rule in Risk-based Authentication

This section explains how to create a custom rule class for risk-based authentication. The API explained here allows developers to use their own risk-based custom rule mechanisms within the risk-based authentication architecture.

In this Chapter

5.1 Prerequisites

- The latest version of Access Manager is installed. See NetIQ Access Manager 5.0 Installation and Upgrade Guide.
- Download nidp.jar, NAMCommon.jar, and risk-*.jar files from `/opt/novell/nam/idp/webapps/nids/WEB-INF/lib` by using Advanced File Configurator and add these to your development project.

  For information about how to download a file, see Downloading Files from a Server in the NetIQ Access Manager 5.0 Administration Guide.

  For information about how to add a file, see Adding Configurations to a Cluster in the NetIQ Access Manager 5.0 Administration Guide.

5.2 Understanding the Rule Class

Risk evaluation is done using a set of rules. You can configure the in-built rules that are provided in the product. If you have a requirement that is not achievable using these rules, then you can write your own custom rule.

Risk Engine evaluates all configured rules one-by-one, and evaluates the Risk Score with Risk Level for the connecting user.

Risk Engine collects all activity details of the connecting user and sends these to the rules for evaluation. These include IP address of the connecting client, HTTP headers, Cookies, User attributes, user historical data, and so forth.

The Risk Engine architecture provides a programming interface that allows you to create a custom rule class. This rule can be configured like any other rule for Risk Engine. Whenever the Risk Engine evaluates this rule, corresponding risk core is added when the rule (condition) fails.

5.3 Creating a Custom Rule Class

You can extend the `com.novell.nam.nidp.risk.core.rules.Rule` class to create a custom rule class. This class is available in `risk-core.jar`. This class must override the abstract method called `evaluate()` in the custom class. This method must contain the business logic for the custom rule and this method must return `true` if the rule condition is met. Else, the method must return `false`.

Class Details of `com.novell.nam.nidp.risk.core.rules.Rule`:

Authentication Methods	Description
`evaluate()`	Takes HTTPContext, LocationContext, DeviceContext, UserContext, and ResponseObject as its arguments. Example of using these classes are provided in the code below. Returns true if the rule evaluation is successful. If failed, false is returned and risk score is considered for this rule.
`isHistoricalDataEnabled()`	Returns true if historical data is enabled for the rule
`getName()`	Returns the name of the Rule inString
`getPriority()`	Returns the priority of the rule in integer.
`isExceptionRule()`	Returns true if this rule is a Privileged Rule.
`isRuleEnabled()`	Returns true if this rule is enabled

Authentication Methods	Description
isNATed()	Returns true if Nat setting is enabled for this server
setType()	Takes String or List as argument. This is used as part of the constructor to inform the Risk Engine to get the type of History data this Rule needs
clearType()	Clears the Types set so far
getType()	Returns the List of Types set by this Rule
isHistoryEnabled()	Same as isHistoricalDataEnabled()
getBoolean()	Takes name of the property in String as argument and returns its boolean value. These are Rule properties set as part of the configuration.
getProperty()	Takes name of the Property in String and returns the value that is configured for this Rule in String
getLong()	Takes name of the property in String as argument and returns its long value. These are Rule properties set as part of the configuration.
getInteger()	Takes name of the property in String as argument and returns its int value. These are Rule properties set as part of the configuration.
getClientIP()	Takes HTTPContext & LocationContext as arguments and returns IP of the connecting client in String
isServerNATed()	Same as isNATed()
isNegateResult()	Returns true if negate results options is enabled for the rule
getReturnValue()	Evaluated result is passed to it and this applies isNegateResult on it
getRiskScore()	Returns the risk score assigned to this rule in int
SaveOnSuccessfulAuth()	Return true in your custom rule class, if you want to set a cookie back to the browser. You will need to write a small piece of code to set the cookie value. Example of this will be provided in this document.
getRequiredAttributes()	Override this method in your class. This must return Array of String of user attributes that is required for your rule to evaluate the risk.

Class Details of com.novell.nam.nidp.risk.context.HTTPContext:

Authentication Methods	Description
getM_HTTPHeaders()	Returns the name/value map of http headers of the connecting client.
getCookieValue()	Returns the cookie value in string. Takes the cookie name as argument in string.

Class Details of com.novell.nam.nidp.risk.context.LocationContext:

Authentication Methods	Description
GetClientIPAddress()	Returns the client IP from the Http Request object

Class Details of com.novell.nam.nidp.risk.context.UserContext:

Authentication Methods	Description
`getUserLoginTimeStamp()`	Returns the long value of Clients login time. Its same value as returned by `Calendar.getInstance().getTimeInMillis()`
`get()`	Returns Object for the provided name. This could be Attribute of the user that was requested using `getRequiredAttributes()` or could be the History Record requested through `setType()` of Rule class. Examples of this method will be part of Custom Rule example codes.

You can use the user session properties, which are set by a custom authentication class, as part of custom risk authentication rules. HTTPContext that is sent to the rule evaluation contains this information.

With the following code snippet, you can get the previously set session values by using a custom risk rule class:

Inside evaluate method,

```
public boolean evaluate(HTTPContext httpContext, LocationContext lContext,
DeviceContext dContext, UserContext uContext, ResponseObject rspObject)
{
String email = (String)httpContext.getSessionContext().get("ExernalEmail");
// Continue evaluation.
}
```

5.4 Understanding the Custom Rule Class Example

The following example explains how to create a custom rule class:

```
import java.util.Base64;
import java.util.Map;
import java.util.Properties;
import com.novell.nam.nidp.risk.context.DeviceContext;
import com.novell.nam.nidp.risk.context.HTTPContext;
import com.novell.nam.nidp.risk.context.LocationContext;
import com.novell.nam.nidp.risk.context.UserContext;
import com.novell.nam.nidp.risk.core.rules.Rule;
import com.novell.nam.nidp.risk.util.ResponseObject;
public class CustomRuleTmpl extends Rule {
  /**
   * @param configProps
   * All the configuration will be passed to the constructor.
   *
   * Pass the type of user historical data you want.
   *
   */
public CustomRuleTmpl(Properties configProps) {super(configProps);
/*
 * Check all the properties that is configured
*/
      printProperties(configProps);
    if ( isHistoricalDataEnabled())
      {

// Enter all the user attributes that you need from the history database. Generally
you would need one or two values.
setType(HistoricalAttributeEntries.IP.name());
```

```
/*
* The following commented code shows how to get other historical data from
database.
*   setType(HistoricalAttributeEntries.LASTLOGGEDINTIME.name());
*   setType(HistoricalAttributeEntries.CITY.name())
*   setType(HistoricalAttributeEntries.COUNTRY.name());
*   setType(HistoricalAttributeEntries.REGION.name());
*   setType(HistoricalAttributeEntries.RISKSCORE.name());
*   setType(HistoricalAttributeEntries.LOGINRESULT.name());
*   setType(HistoricalAttributeEntries.RISKCATEGORY.name());
*   setType(HistoricalAttributeEntries.RISKSCORE.name());
*   setType(HistoricalAttributeEntries.REGIONCODE.name());
*   setType(HistoricalAttributeEntries.METROCODE.name());
*   setType(HistoricalAttributeEntries.POSTCODE.name());
     *
     *
     * Or you could even set it using an array List
     *   clearType(); // Clear the previously set rule type values
   *  ArrayList<String> historyAttributes = newArrayList<String>();
   *  historyAttributes.add ( HistoricalAttributeEntries.IP.name());
   * historyAttributes.add (HistoricalAttributeEntries.LASTLOGGEDINTIME.name());
   *  setType(historyAttributes);
   */
    }
  }
private void printProperties(Properties configProps) {
    System.out.println("Configured properties are: -");
        for (Entry<Object, Object> e: configProps.entrySet())
    System.out.println("Name :" + e.getKey() + "Value : " + e.getValue());
}

/* (non-Javadoc)
   * @see
com.novell.nam.nidp.risk.core.rules.Rule#evaluate(com.novell.nam.nidp.risk.context
.HTTPContext,
com.novell.nam.nidp.risk.context.LocationContext,
com.novell.nam.nidp.risk.context.DeviceContext,
com.novell.nam.nidp.risk.context.UserContext,
com.novell.nam.nidp.risk.util.ResponseObject)
   *
   * This method evaluates the rule and is called in the order of the priority.
   *
   * Parameters
   * HttpContext- Contains all the request http header information
   * LocationContext- Contains information about the client location (IP)
   * DeviceContext- Contains device information
   * UserContext- Contains user information that includes user attributes, roles,
and historical login data of the user.
   *   ResponseObject- Can be used for setting cookies, headers and user attributes
on completion of the risk calculation.
   *
     * Return Values
   * true- on successful evaluation of the rule.
   * false- if failed to evaluate the rule. In this case, configured risk score is
considered.
    *
  *   This method will have 3 sections
  * 1) Pre-evaluation: To get all the parameters of the user login
  * 2)Evaluate the rule: Apply the use case to the evaluation using the parameters
  * 3)Post-evaluation: - Set result, cookie and history parameters if needed
   */

@Override
    public boolean evaluate(HTTPContext httpContext, LocationContext
    lContext,      DeviceContext dContext, UserContext uContext,
    ResponseObject rspObject) {
```

```
    boolean returnValue = false;
    if ( m_ruleEnabled)
      {
      /* ######## Pre-Evaluation Section ####################*/
      getHTTPHeaderInformation(httpContext);
      getCookieInformation(httpContext, "JSESSIONID");
      getLocationParameter(lContext);
      getUserContext(uContext);
      /* ############### Evaluation Section ###################*/
      {
        /*
         * Change the return value according logic of the evaluation
         */
if ( true )
            returnValue = true;
        }
      /* ############### Post-Evaluation Section ###################*/
      /*
 * Execute the post evaluation method to consider other configuration like negate
result
       */
//  rspObject.setUserAttr(HistoricalAttributeEntries.IP.name(), clientIP);

 return getReturnValue(returnValue);
         }
    return true;
    }

    /*
   * Get all the user context/attributes
   */
    private void getUserContext(UserContext uContext) {
      // TODO Auto-generated method stub
    getUserAttribute(uContext);
    getUserRoles(uContext);
    getHistoricalData(uContext);
  }
  /*
   * Get the historical data of the user from the configured database
   */
    private void getHistoricalData(UserContext uContext) {
    // It will get all the passed transaction for the user in the past.
   // If the transaction you looking for is not found, that mean it has failed for
that log in.
    HistoryRecord records =
(HistoryRecord)uContext.get(HistoricalAttributeEntries.IP.name());
    if ( records != null)
      {
      System.out.println("Printing past entries from the History, in this example
its the IP used by the user");
      for( Object o : records.getValue() )
              System.out.println("< " + (String)o + "
>n");
    }
    }
  /*
     * Get the user's current role information
     */
      private void getUserRoles(UserContext uContext) {
String[] values = (String[])
uContext.get(UserProfile.Constants.ROLES.name());
    RiskLog.debug("Roles of the user are ");
      for ( String role : values)
      RiskLog.debug(" " + role + ",");
    }
  /*
```

```
  * Get the user's ldap attributes.
     *
  * NOTE: To get attributes here, you must return the name of the attributes, you
need, using method getRequiredAttributes();

   */
     private void getUserAttribute(UserContext uContext) {
   // Value will be null if attribute name is not set as part of
getRequiredAttributes()
    String mail = (String) uContext.get("mail");
    String carlicense = (String) uContext.get("carlicense");
    System.out.println("Mail attribute of the user is " + mail + ",
and the carlicense is " + carlicense);
  }
/*
* This method should return the name of the user ldap attributes required during
evaluation of the rule. You can configure those in the custom rule properties and
can pass the value here.
   */
     @Override
     public String[] getRequiredAttributes() {
         // TODO Auto-generated method stub
         String[] attributes = new String[2];
    attributes[0] = "mail";
      attributes[1] = "carlicense";
      return attributes;
    }
  /*
  * Get the location parameter of the user
    *
  */
  private void getLocationParameter(LocationContext lContext) {
    String clientIP = lContext.getClientIPAddress();
      RiskLog.debug("Client Ip address for this request is = " + clientIP);
    Properties props = new Properties();
      Provider provider;

       try {
       provider = GeoLocationFactory.getProvider
      RiskEngine.getInstance().getCoreProps().getProperty("geolocation.provider"),
    null, props);
      GeoLocBean geoLoc = provider.readGeoLocInfo(InetAddress
            .getByName(clientIP));
      System.out.println("Country  = " + geoLoc.getCountry());
        System.out.println("Country code = " + geoLoc.getCountryCode());
        System.out.println("City  = " + geoLoc.getCity());
        } catch (GeoLocException | UnknownHostException
e) {
      // TODO Auto-generated catch block
          System.out.println("Geo location configuration exception
" + e.getLocalizedMessage());
      e.printStackTrace();
          }
  }
  /*
   * Get a specific cookie out of headers
      */
      private void getCookieInformation(HTTPContext httpContext,
String cookieName) {
    String cookieValue = httpContext.getCookieValue(cookieName);
    RiskLog.debug("Cookie Name = " + cookieName + "
Value = " + cookieValue);
```

```
 }
/*
       * Get all http Context information.
     * Contains all http headers that is part of the request, including cookies.
   */
     private void getHTTPHeaderInformation(HTTPContext httpContext) {
      Map<String, String> headers = httpContext.getM_HTTPHeaders();
      Iterator itr = headers.entrySet().iterator();
      for  ( Map.Entry< String, String> entry : headers.entrySet()
)
        RiskLog.debug("Header Name = " + entry.getKey()
 + " Value = " + entry.getValue());
  }
}
```

5.5 Deploying Your Custom Rule Class

1. Create a jar file for your custom rule class and any associated classes.
2. Add the jar file to `/opt/novell/nam/idp/webapps/rba-core/WEB-INF/lib` by using Advanced File Configurator.

   For information about how to add a file, see Adding Configurations to a Cluster in the NetIQ Access Manager 5.0 Administration Guide.
3. In Administration Console, click **Policies> Risk Configuration > Rules > New**.

   Rule name: Specify a name that Administration Console can use to identity this custom rule

   Rule Definitions: Select the 'custom rule' to configure the custom rule
4. Specify the following details:

   Custom class Name: Specify the name of your Java class

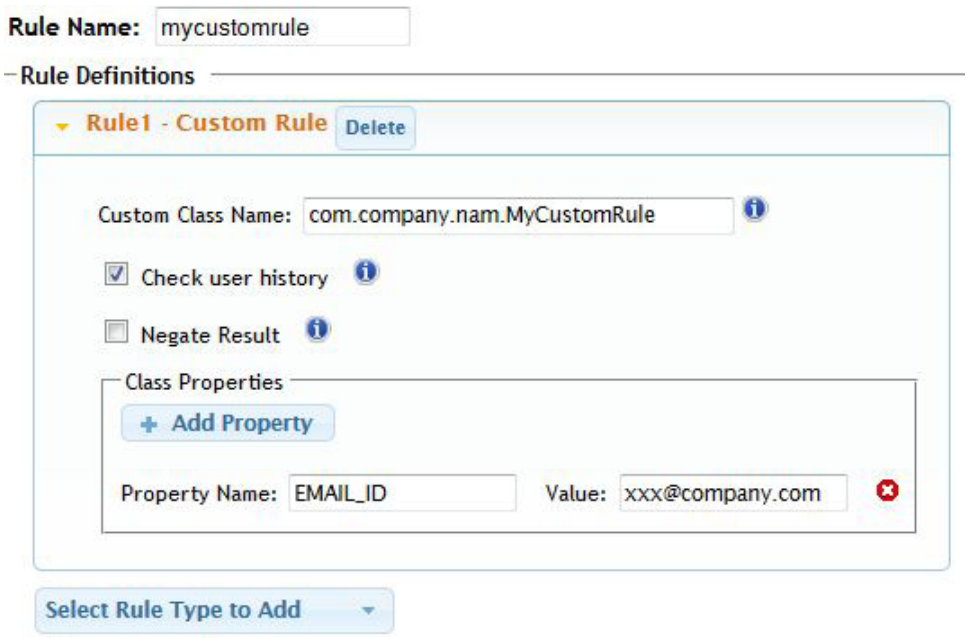
   Check User History: Select this option if you are using the user's history data in you custom class

   Negate Result: Select this option to reverse the output of the rule condition

   Class Property: Specify parameters and values which will be sent to the custom class at runtime.

   Property Name: Name of the parameter.

   Value: Value of the parameter.



5. Click **Next** and specify the risk score for the rule.

   Rule Group: Select the group name.

   Risk Score: Specify the risk score for the custom rule.

   Privileged Rule: Select if the custom rule is a privileged rule.
6. Click **Finish > OK**.
7. Restart Identity Server.
8. On the Identity Servers page, click **Update**.
9. Update any associated devices that are using this Identity Server configuration.

5.6 Understanding Custom Attributes in History SQL Database

The Risk module enables you to save historical data of the user login to the external database. Custom rule examples explain how to read the existing parameters from an historical database. To create a new attribute in the database for your custom rules, perform the following steps:

1. Create the custom tables as follows:

```
CREATE TABLE netiq_risk.extra
 (
   id         VARCHAR(32)  NOT NULL,
     custom_string_entry1       VARCHAR2(100),
   custom_int_entry2      INTEGER,
     custom_char_entry3       CHAR(1),
   CONSTRAINT fk_extra_id FOREIGN KEY (id) REFERENCES netiq_risk.usr(id)
)
```

2. Specify the of the table as 'extra'.
3. The column name (attribute) should start with 'custom' followed by the data type of the column, like `custom_<datatype>_<name of the attribute>`

For example, `custom_string_userlogintime`

4. Ensure that the attribute name matches with the database column name.

   Access Manager supports the following data types for custom attributes:

   - String
   - Int
   - Char
   - Boolean
   - Date

5.6.1 Custom Rule example

As part of your customer class constructor, set the type of the history you are looking for.

```
//Get the last login time of the user
  setType(HistoricalAttributeEntries.LASTLOGGEDINTIME.name());
//Get the custom string user login time of the user
  setType("custom_string_userlogintime");
```

As part of the evaluate() method, you can access these custom values as follows:

```
HistoryRecord records =
   (HistoryRecord)uContext.get("custom_string_userlogintime");
    String value = (String)records.getValue().get(0);
```

At the end of the evaluate() method, you can set the value of the custom attribute as follows:

```
(ResponseObject)rspObject.setUserAttr("custom_string_userlogintime","12:02:01");
```

Post evaluation of the risk, this will be set to the extra table on the SQL database.

5.7 Custom Geolocation Data Provider Integration

This section describes how to integrate the custom geolocation data provider. The API presented here allows you to integrate the custom geolocation data provider with risk-based authentication.

5.7.1 Prerequisites

- The latest version of Access Manager is installed.
- Your development environment requires the same installation as outlined in the NetIQ Access Manager 5.0 Installation and Upgrade Guide.
- Download nidp.jar, NAMCommon.jar and risk-*.jar and third-party Geo Location data provider jar files from `/opt/novell/nam/idp/webapps/rba-core/WEB-INF/lib` and add these files to your development project by using Advanced File Configurator.

For information about how to download a file, see Downloading Files from a Server in the NetIQ Access Manager 5.0 Administration Guide.

For information about how to add a file, see Adding Configurations to a Cluster in the NetIQ Access Manager 5.0 Administration Guide.

5.7.2 Understanding the Geo Location Provider interface

Method	Description
init()	Takes Properties as its arguments. This properties object contains the parameters which are passed through the Admin Console for this Custom class. The method used to initialize the Geo Location Provider Class.
readGeoLocInfo()	Takes InetAddress as its arguments. Returns the Geo Location information as Geolocation Bean.

5.7.3 Creating a Custom Geolocation Provider Class

You can create the custom geolocation provider class as follows:

- Implementing Provider Interface
- Extending Abstract Provider Class

5.7.3.1 Implementing Provider Interface

```
import com.novell.nam.nidp.risk.core.geoloc.Provider;

public interface Provider {

  public void init(Properties props);
      public GeoLocBean readGeoLocInfo(InetAddress IPAddress) throws
GeoLocException;
}
```

You can create the Custom Provider class by implementing this interface. Override the `init()` and `readGeoLocInfo()` methods.

5.7.3.2 Extending Abstract Provider Class

```
import com.novell.nam.nidp.risk.core.geoloc.AbstractProvider;

public abstract class AbstractProvider implements Provider {

abstract public GeoLocBean readGeoLocInfo(InetAddress IPAddress)
throws GeoLocException;

    public AbstractProvider(Properties props){
    init(props);
   }
   }
```

You can create a custom provider class by extending the AbstractProvider class. Override the above init() and readGeoLocInfo() abstract methods.

5.7.4 Custom Geolocation Provider Class Example

```
import com.novell.nam.nidp.risk.core.geoloc.AbstractProvider;
import com.novell.nam.nidp.risk.core.geoloc.exception.GeoLocException;
import com.novell.nam.nidp.risk.core.geoloc.model.GeoLocBean;

public class MyCustomGeoProvider extends AbstractProvider {

  public MyCustomGeoProvider (Properties props) {
  super(props);

  }

// The argument 'props' contains the configuration parameters which are provided in
the admin console forthis custom class.
  @Override
  public void init(Properties props) {

  }

// This method should return the geo location information
@Override
public GeoLocBean readGeoLocInfo(InetAddress IPAddress)
throws GeoLocException                              {
// read the geolocation information from any external provider using web service
calls or any sources

    return null;
  }

}
```

5.7.5 Deploying Your Custom Geolocation Provider Class

1. Create a jar file for your custom geolocation provider class and any associated classes.
2. Add jar files to the `/opt/novell/nam/idp/webapps/rba-core/WEB-INF/lib` directory by using Advanced File Configurator.

   For information about how to add a file, see Adding Configurations to a Cluster in the NetIQ Access Manager 5.0 Administration Guide.
3. In Administration Console, click **policies> Risk Configuration > Geolocation**.
4. Select **Custom Provider** from the list and specify the following details:

   Provider Name: A name that Administration Console can use to identity this custom provider.

   Java Class Path: The path name of your custom Geo Provider Java class.

   Class Property: The parameters and values which will be passed to the custom class at runtime.

   Property Name: The name of the parameter.

   Value: The value of the parameter.
5. Click **OK**.
6. Restart Identity Server.
7. On the Identity Servers page, click **Update**.
8. Update any associated devices that are using this Identity Server configuration.