
Developer Guide

Access Manager 4.2

May 2016

Legal Notice

For information about NetIQ legal notices, disclaimers, warranties, export and other use restrictions, U.S. Government restricted rights, patent policy, and FIPS compliance, see <https://www.netiq.com/company/legal/>.

© 2015 NetIQ Corporation. All Rights Reserved.

For information about NetIQ trademarks, see <https://www.netiq.com/company/legal/>. All third-party trademarks are the property of their respective owners.

Contents

About NetIQ Corporation	5
About this Book and the Library	7
1 Getting Started	9
Development Overview	9
SDK Components	10
Selecting an Integrated Development Environment	10
2 Identity Server Authentication API	11
Prerequisites	11
Understanding the Authentication Class	11
Authentication Class Components	11
How the Authentication Class Operates	12
Creating an Authentication Class	13
Project Requirements	13
doAuthenticate Method	13
Authentication Methods	14
Class Property Methods	15
Status Methods	18
User Information Methods	19
CallbackAuthentication Method	21
Other Methods	21
Understanding the Authentication Class Example	21
Extending the Base Authentication Class	22
Implementing the doAuthenticate Method	22
Prompting for Credentials	22
Verifying Credentials	22
PasswordClass Example Code	23
Localizing the Prompts in Your Authentication Class	27
Creating a Properties File	27
Creating a Resource Class	27
Creating or Modifying a JSP Page	28
Deploying Your Authentication Class	29
3 LDAP Server Plug-In	33
Prerequisites	33
Creating the LDAP Plug-In	33
eDirectory Plug-In	35
Installing and Configuring the LDAP Plug-In	39
Troubleshooting	40
4 The Policy Extension API	41
Getting Started	41
Prerequisites	41
Types of Policy Extensions	42
How the Policy Engine Interacts with an Extension	42
Common Elements and Tasks	46

Implementing Common Elements	46
Initializing the Factory Object	47
Retrieving Information from the Identity Server User Store	48
Implementing the Extension Interface	49
Creating an Extension	55
Creating a Context Data Extension	56
Creating a Condition Extension	60
Creating an Action Extension	63
Installing and Configuring an Extension	65
Installing the Extension on the Administration Console	65
Distributing a Policy Extension to Access Manager Devices	67
Distributing the Extension to Customers	67
Sample Codes	68
Data Extension for External Attribute Source Policy	68
Template Policy Extensions	68
LDAP Group Data Element	69
PasswordClass	69

5 Custom Rule in Risk-Based Authentication 71

Prerequisites	71
Understanding the Rule Class	71
Creating a Custom Rule Class	72
Understanding the Custom Rule Class Example	74
Deploying Your Custom Rule Class	79
Understanding Custom attributes in History SQL Database	81
Custom Rule example	82
Custom Geo Location Data Provider Integration	82
Prerequisites	82
Understanding the Geo Location Provider interface	83
Creating a Custom Geo Location Provider Class	83
Understanding the Custom Geo Location Provider Class Example	83
5.7.5 Deploying Your Custom Geo Location Provider Class	84

A Revisions 87

About NetIQ Corporation

We are a global, enterprise software company, with a focus on the three persistent challenges in your environment: Change, complexity and risk—and how we can help you control them.

Our Viewpoint

Adapting to change and managing complexity and risk are nothing new

In fact, of all the challenges you face, these are perhaps the most prominent variables that deny you the control you need to securely measure, monitor, and manage your physical, virtual, and cloud computing environments.

Enabling critical business services, better and faster

We believe that providing as much control as possible to IT organizations is the only way to enable timelier and cost effective delivery of services. Persistent pressures like change and complexity will only continue to increase as organizations continue to change and the technologies needed to manage them become inherently more complex.

Our Philosophy

Selling intelligent solutions, not just software

In order to provide reliable control, we first make sure we understand the real-world scenarios in which IT organizations like yours operate — day in and day out. That's the only way we can develop practical, intelligent IT solutions that successfully yield proven, measurable results. And that's so much more rewarding than simply selling software.

Driving your success is our passion

We place your success at the heart of how we do business. From product inception to deployment, we understand that you need IT solutions that work well and integrate seamlessly with your existing investments; you need ongoing support and training post-deployment; and you need someone that is truly easy to work with — for a change. Ultimately, when you succeed, we all succeed.

Our Solutions

- ◆ Identity & Access Governance
- ◆ Access Management
- ◆ Security Management
- ◆ Systems & Application Management
- ◆ Workload Management
- ◆ Service Management

Contacting Sales Support

For questions about products, pricing, and capabilities, contact your local partner. If you cannot contact your partner, contact our Sales Support team.

Worldwide:	www.netiq.com/about_netiq/officelocations.asp
United States and Canada:	1-888-323-6768
Email:	info@netiq.com
Web Site:	www.netiq.com

Contacting Technical Support

For specific product issues, contact our Technical Support team.

Worldwide:	www.netiq.com/support/contactinfo.asp
North and South America:	1-713-418-5555
Europe, Middle East, and Africa:	+353 (0) 91-782 677
Email:	support@netiq.com
Web Site:	www.netiq.com/support

Contacting Documentation Support

Our goal is to provide documentation that meets your needs. If you have suggestions for improvements, click **Add Comment** at the bottom of any page in the HTML versions of the documentation posted at www.netiq.com/documentation. You can also email Documentation-Feedback@netiq.com. We value your input and look forward to hearing from you.

Contacting the Online User Community

Qmunity, the NetIQ online community, is a collaborative network connecting you to your peers and NetIQ experts. By providing more immediate information, useful links to helpful resources, and access to NetIQ experts, Qmunity helps ensure you are mastering the knowledge you need to realize the full potential of IT investments upon which you rely. For more information, visit <http://community.netiq.com>.

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 (SDKs) 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 (URIs)
- ♦ Domain Name System (DNS)
- ♦ Web Services Description Language (WSDL)

Other Information in the Library

The library provides the following information resources:

- ♦ [NetIQ Access Manager 4.2 Administration Guide](#)
- ♦ [NetIQ Access Manager 4.2 Installation and Upgrade Guide](#)
- ♦ [NetIQ Access Manager 4.2 Best Practices Guide](#)
- ♦ [Performance and Sizing Guidelines](#)

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.

This section explains how to get started with the Access Manager SDK and contains the following topics:

- ◆ [“Development Overview” on page 9](#)
- ◆ [“Selecting an Integrated Development Environment” on page 10](#)

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@netiq.com.

SDK Components

The Access Manager developer components are included in the [Access Manager Developer Kit](#). However, the complete Access Manager package, including the install, is not included in the NDK. 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. For an evaluation version, see [Novell Downloads](#) and search for Access Manager.

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 the Identity Server. The API presented here allows developers to leverage their own authentication mechanisms within the Access Manager architecture. The following topics are covered:

- ♦ [“Prerequisites” on page 11](#)
- ♦ [“Understanding the Authentication Class” on page 11](#)
- ♦ [“Creating an Authentication Class” on page 13](#)
- ♦ [“Understanding the Authentication Class Example” on page 21](#)
- ♦ [“Localizing the Prompts in Your Authentication Class” on page 27](#)
- ♦ [“Deploying Your Authentication Class” on page 29](#)

Prerequisites

- ♦ Access Manager.
- ♦ Your development environment requires the same installation as outlined in the [NetIQ Access Manager 4.2 Installation and Upgrade Guide](#).
- ♦ Copy the `nidp.jar` and `NAMCommon.jar` files in the following directory of your Identity Server to your development project:
 - ♦ **On Linux:** `/opt/novell/nam/idp/webapps/nidp/WEB-INF/lib`
 - ♦ **On Windows:** `C:\Program Files (x86)\Novell\Tomcat\webapps\nidp\WEB-INF\lib`

Understanding the Authentication Class

Before developing an authentication class, review the following concepts:

- ♦ [“Authentication Class Components” on page 11](#)
- ♦ [“How the Authentication Class Operates” on page 12](#)

Authentication Class Components

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

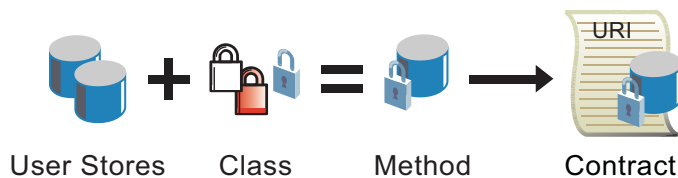
For more detailed information about the Identity Server and its relation to other Access Manager components, see [“Creating Authentication Classes”](#) in the [NetIQ Access Manager 4.2 Administration Guide](#).

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 the 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 illustrates the components of a contract:

Figure 2-1 Local Authentication Components



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.

Creating an Authentication Class

The 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 the Identity Server.

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

- ♦ [“Project Requirements” on page 13](#)
- ♦ [“doAuthenticate Method” on page 13](#)
- ♦ [“Authentication Methods” on page 14](#)
- ♦ [“Class Property Methods” on page 15](#)
- ♦ [“Status Methods” on page 18](#)
- ♦ [“User Information Methods” on page 19](#)
- ♦ [“CallbackAuthentication Method” on page 21](#)
- ♦ [“Other Methods” on page 21](#)

For the Javadoc associated with these methods, see [LocalAuthenticationClass](#).

Project Requirements

The project used to create the custom class must include the `nidp.jar` file shipped with Access Manager. This JAR file is located here:

- ♦ **Linux:** `/opt/novell/nam/idp/webapps/nidp/WEB-INF/lib`
- ♦ **Windows:** `C:\Program Files (x86)\Novell\Tomcat\webapps\nidp\WEB-INF\lib`

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.

The `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
<code>HANDLED_REQUEST</code>	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.
<code>SHOW_JSP</code>	Further information is needed to complete authentication. Typically, this value is returned when a page is returned to query for credentials.
<code>NOT_AUTHENTICATED</code>	The authentication failed.
<code>AUTHENTICATED</code>	The authentication succeeded in identifying a single <code>NIDPPPrincipal</code> object (user).
<code>CANCEL</code>	The authentication process was canceled. This typically occurs only during authentication after a request from a service provider.
<code>PWD_EXPIRING</code>	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.
<code>PWD_EXPIRED</code>	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 `NIDPPPrincipal` 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.

Authentication Methods

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

Method	Description
<code>authenticateWithPassword()</code>	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
<code>authenticateWithQuery()</code>	Takes a string in the form of an LDAP query and a password as its arguments. It succeeds if the query result locates a single user with the associated password. See authenticateWithQuery
<code>findPrincipals()</code>	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 <code>NIDPPPrincipal</code> objects that result from the search. See findPrincipals
<code>findPrincipalsByQuery()</code>	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 <code>NIDPPPrincipal</code> objects that match the query. See findPrincipalsByQuery
<code>getCredentials()</code>	Gets the list of credentials used to authenticate the user or principal. The 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
<code>addCredential()</code>	Adds a credential used for authentication to a user or principal. This method is called by a class so that the Identity Server can call the <code>getCredentials()</code> method. See addCredential
<code>addLDAPCredentials()</code>	Adds an LDAP credential, other than the password, to a user or principal. See addLDAPCredentials
<code>clearCredentials()</code>	Clears the credentials of the user or principal. See clearCredentials

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, the Administration Console displays configuration pages for their required properties. For information about these properties, see “[Creating Authentication Classes](#)” in the *NetIQ Access Manager 4.2 Administration Guide* .

When you deploy your class, the Administration Console has a generic page that allows the administrator to configure property key name and value pairs. As you are creating your class, you need to create a key name and value pair for each configuration item that you want input from the administrator. For example, if you want to allow the administrator to use a different JSP* page for 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, but 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
<code>getProperty()</code>	Obtains specific properties needed by an authentication class. Property values are specified when configuring the authentication class in the Administration Console. See getProperty
<code>getBooleanProperty()</code>	Returns a Boolean value for the specified property and sets a default value if value cannot be found. See getBooleanProperty
<code>getType()</code>	Identifies one of the authentication types known to the 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” on page 17 for a list of supported types. See getType
<code>getProvisionURL()</code>	Gets the URL to call to provision a user and returns the URL to redirect to for user provisioning, or Null if it is not available. See getProvisionURL
<code>getReturnURL()</code>	Returns the URL that any user interactions should post data back to, or Null if it is not available. See getReturnUrl
<code>mustPersist()</code>	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 18 . See mustPersist
<code>isFirstInstance()</code>	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
<code>isCancelAppropriate()</code>	Determines if the option to cancel an authentication is appropriate for this instance. See isCancelAppropriate

Method	Description
<code>isDefinesUser()</code>	<p>Determines if the authentication class instance needs to identify a user. If so, returns True.</p> <p>For more information, see the Identifies User option in “Configuring Authentication Methods” in the <i>NetIQ Access Manager 4.2 Administration Guide</i> .</p> <p>See also isDefinesUser.</p>
<code>isUserIdentification()</code>	<p>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.</p> <p>See isUserIdentification.</p>
<code>isFirstCallAfterPrevMethod()</code>	<p>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.</p> <p>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.</p> <p>See isFirstCallAfterPrevMethod.</p>
<code>isPendingAuthnRequest()</code>	<p>Determines whether there is a pending authentication request from a Service Provider. Returns True if there is a pending request, otherwise, returns False.</p> <p>See isPendingAuthnRequest.</p>
<code>getAuthnRequest()</code>	<p>Gets the request that might have caused this authentication class to be invoked.</p> <p>See getAuthnRequest</p>

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. The Identity Server can reply to such a request with all the contracts that fit the requested authentication type.

The Identity Server supports the following types of authentication classes:

Constant	Description
<code>AuthnConstants.BASIC</code>	Specifies a basic authentication over HTTP. It uses the login page of the browser to prompt the user for a name and a password.
<code>AuthnConstants.PASSWORD</code>	Specifies a form-based authentication using a name and password over HTTP.

Constant	Description
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 multiple authentication method using a smart card.
AuthnConstants.OTHER	Default. Used for all other types not defined above.

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 [“doAuthenticate Method” on page 13](#).

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
<code>setFailure()</code>	Sets a failure state for the current authentication instance. See setFailure
<code>isFailure()</code>	Indicates whether or not the authentication failed. Returns <code>True</code> if authentication failed, otherwise, returns <code>False</code> . See isFailure
<code>setUserErrorMsg()</code>	Sets the error message to be displayed to an end user. See setUserErrorMsg
<code>getUserErrorMsg()</code>	Gets the error message that will be displayed to the end user. See getUserErrorMsg

Method	Description
getLogMsg()	Gets the message for the associated error ID. This method is used primarily by the 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 19.
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 19.

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 only) 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.
LOG_DISABLED_ACCOUNT	The account requested is disabled.
LOG_BAD_CONNECTION	The authentication channel is unable to communicate the user request.

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 the 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 the Identity Server to obtain the credentials verified by an authentication class. See getPrincipal

Method	Description
<code>getPrincipalAttributes()</code>	Gets the attributes for the principal that has been authenticated. See getPrincipalAttributes
<code>getPrincipalUserAuthority()</code>	Gets the user authority for the identified principal, assuming that <code>m_Principal</code> has been set. See getPrincipalUserAuthority
<code>getUserAuthorityCount()</code>	Gets the number of searchable user authorities. See getUserAuthorityCount
<code>getUserAuthority()</code>	Gets a specific user authority. The <code>getUserAuthorityCount()</code> method returns the index range. See getUserAuthority
<code>setPrincipal()</code>	Sets the principal to be authenticated by this class. See setPrincipal
<code>setPrincipalAttributes()</code>	Sets attributes for a principal that has been authenticated. See setPrincipalAttributes
<code>setSessionProperties()</code>	Sets user session properties that can be used later by other custom authentication classes as well as risk-based authentication rules. With the following code snippet, you can set session properties using custom authentication class: <pre>// 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);</pre>
<code>getSessionProperties()</code>	Gets user session properties that were previously set by other custom authentication classes. With the following code snippet, you can get session properties using custom authentication class: <pre>// 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("ExternalEmail");</pre>

CallbackAuthentication Method

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

To perform the STS authentication, you need to implement the `cbAuthenticate` method in the authentication class.

For a sample implementation of `cbAuthenticate`, see [“PasswordClass Example Code” on page 23](#).

Other Methods

The following tables lists other useful methods:

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

Understanding the 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:

- ◆ [“Extending the Base Authentication Class” on page 22](#)
- ◆ [“Implementing the doAuthenticate Method” on page 22](#)
- ◆ [“Prompting for Credentials” on page 22](#)
- ◆ [“Verifying Credentials” on page 22](#)
- ◆ [“PasswordClass Example Code” on page 23](#)

Extending the Base Authentication Class

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

The `LocalAuthenticationClass` defines a single abstract method, `doAuthenticate()`, which must be implemented by new classes. During user authentication, the 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.

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 the 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.

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.

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()`.

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
STSAAuthenticationClass, 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);

    // 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);

    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 NIDPPPrincipal 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 NIDPPPrincipal cbAuthenticate(CallbackHandler cbHandler) {
        PasswordValidationCallback pwdCallback = new PasswordValidationCallback();
        Callback[] callbacks = new Callback[] { pwdCallback };

        NIDPPPrincipal 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;
}
}
}

```

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:

- ♦ [“Creating a Properties File” on page 27](#)
- ♦ [“Creating a Resource Class” on page 27](#)
- ♦ [“Creating or Modifying a JSP Page” on page 28](#)

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.

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;
    }
}

```

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%>">

```


General Local Liberty SAML 1.1 SAML 2.0

User Stores | **Classes** | Methods | Contracts | Defaults

New | Delete

Name

Name/Password - Basic

Name/Password - Form

Secure Name/Password - Basic

Secure Name/Password - Form

5 From this page, click **New**.

Create Authentication Class

Step 1 of 2: Specify name and java class.

Display name:

Java class: ▼

Java class path:

6 Fill in the following fields:

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

Java class: For a new 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.

7 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 returned to your class in the `props` parameter when your class is called.

8 Click **Finish**.

9 To configure a method for your class, click **Methods > New**, and select your class for the **Class** field.

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 [“Configuring Authentication Methods”](#) in the *NetIQ Access Manager 4.2 Administration Guide*.

10 Click **Finish**.

11 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 [“Configuring Authentication Contracts”](#) in the *NetIQ Access Manager 4.2 Administration Guide*.

12 (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, see [“Supported Authentication Class Types”](#) on page 17 and [“Specifying Authentication Defaults”](#) in the *NetIQ Access Manager 4.2 Administration Guide* .

- 13 Click **Finish** > **OK**.
- 14 restart The Identity Server.
- 15 On the Identity Servers page, click **Update**.
- 16 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 3.0 SP4 or above. The three supported LDAP servers are eDirectory™, Active Directory, and Sun ONE. Any other directory types require an LDAP Server plug-in.

Prerequisites

To develop an LDAP server plug-in:

- ♦ Meet all system requirements of the Identity Servers and Access Gateways. See the [NetIQ Access Manager 4.2 Installation and Upgrade Guide](#).
- ♦ Install and configure all components of Access Manager. For detailed installation and configuration information, see the [NetIQ Access Manager 4.2 Installation and Upgrade Guide](#) and “Setting Up a Basic Access Manager Configuration” in the [NetIQ Access Manager 4.2 Administration Guide](#).
- ♦ Have an integrated Java development environment.
- ♦ Copy the `NAMCommon.jar` file in the following directory of your Identity Server to your development project:
 - ♦ **Linux:** `/opt/novell/nam/idp/webapps/nidp/WEB-INF/lib`
 - ♦ **Windows:** `C:\Program Files (x86)\Novell\Tomcat\webapps\nidp\WEB-INF\lib`

Creating the LDAP Plug-In

The project used to create the plug-in must include the `NAMCommon.jar` file shipped with Access Manager. This JAR file is located in the following directory:

- ♦ **Linux:** `/opt/novell/nam/idp/webapps/nidp/WEB-INF/lib`
- ♦ **Windows:** `C:\Program Files (x86)\Novell\Tomcat\webapps\nidp\WEB-INF\lib`

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
<code>getDirectoryName()</code>	Needs to return the name you want displayed for your directory type. For eDirectory, this method returns “Novell eDirectory” for this string.
<code>getGUIDAttributeName()</code>	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.
<code>getMemberAttributeName()</code>	Needs to return the name of the attribute that is used to identify an object as a member of a group. For eDirectory, this is the member attribute.

Method	Description
<code>getUserClassName()</code>	Needs to return the name of the class that is used to create users. For <code>eDirectory</code> , this is the <code>User</code> class.
<code>getUserNameNamingAttrName()</code>	Needs to return the name of the attribute that is used to name users. For <code>eDirectory</code> , this is the <code>cn</code> attribute.
<code>preUserAccountCreation()</code>	Needs to return an <code>attributes</code> object that contains an array of attributes, with each member containing the name of an attribute and its value. This <code>attributes</code> 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 <code>eDirectory</code> , this also includes the <code>sn</code> attribute.

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

Method	Description
<code>postUserAccountCreation()</code>	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 <code>strCorrelationId</code> parameter that you can use to match the user with the user in the <code>preUserAccountCreation()</code> method.
<code>onCreateConnection()</code>	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.
<code>onCreateConnectionException()</code>	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 <code>AuthenticationException</code> parameter.
<code>onCreateConnectionException()</code>	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 <code>OperationNotSupportedException</code> parameter.

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 [“eDirectory Plug-In” on page 35](#).

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
{
    public String getDirectoryName()
    {
        return "Novell eDirectory";
    }

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

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

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

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

    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;
}
}

```

Installing and Configuring the LDAP Plug-In

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

1 Copy the plug-in class file to the Identity Server:

1a Copy it to the following directory under the correct directory structure as per the class package:

♦ Linux:

- ♦ 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`

♦ Windows:

- ♦ If you want to use a LDAP-plugin class file: `C:\Program Files (x86)\Novell\Tomcat\webapps\nidp\WEB-INF\classes`
- ♦ If you want to use a LDAP-plugin class in a jar file: `C:\Program Files (x86)\Novell\Tomcat\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.

1b Repeat [Step 1a](#) for each Identity Server in the cluster.

2 To associate an LDAP Server plug-in with the Custom1, Custom2, or Custom3 directory type, modify the `web.xml` file on the Identity Server:

2a In a text editor, open the following file:

- ♦ **Linux:** `/opt/novell/nam/idp/webapps/nidp/WEB-INF/web.xml`
- ♦ **Windows:** `C:\Program Files (x86)\Novell\Tomcat\webapps\nidp\WEB-INF\web.xml`

2b Add an entry for the `ldapStorePlugins` 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 following format:

```
custom1:classname;custom2:classname;custom3:classname
```

2c Repeat [Step 2a](#) through [Step 2b](#) on each Identity Server in the cluster.

3 In the Administration Console, configure the 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 the 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 [“Troubleshooting” on page 40](#).

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}.” on page 40](#)
- ♦ [“300105030=Cannot instantiate LDAP Store Plugin class: {0}. Error: {1}.” on page 40](#)
- ♦ [“300105031=An unknown or unsupported user store directory type {0} was found for the user store named {1}. Defaulting to eDirectory!” on page 40](#)

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 39](#)).

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 is a new feature that has been added to Access Manager 3.1. It 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, describes how to create the three possible types of extensions, then explains how to install and use the extension in an Access Manager policy.

- ♦ [“Getting Started” on page 41](#)
- ♦ [“Common Elements and Tasks” on page 46](#)
- ♦ [“Creating an Extension” on page 55](#)
- ♦ [“Installing and Configuring an Extension” on page 65](#)
- ♦ [“Sample Codes” on page 68](#)

Getting Started

The following sections explain the requirements for developing an extension and provide an overview of the possible types of extensions and an overview of how the Access Manager policy engine interacts with an extension.

- ♦ [“Prerequisites” on page 41](#)
- ♦ [“Types of Policy Extensions” on page 42](#)
- ♦ [“How the Policy Engine Interacts with an Extension” on page 42](#)

Prerequisites

- ♦ Access Manager 4.0 installed and configured. For detailed installation and configuration information, see the [NetIQ Access Manager 4.2 Installation and Upgrade Guide](#) and [“Setting Up a Basic Access Manager Configuration”](#) in the [NetIQ Access Manager 4.2 Administration Guide](#).
- ♦ A basic understanding of the Access Gateway Authorization policies and Access Gateway Identity Injection policies. See [Access Manager Policies](#) in the [NetIQ Access Manager 4.2 Administration Guide](#).
- ♦ An integrated Java development environment.
- ♦ Copy the `npxpe.jar` file from the following directory of your Access Manager device to your development environment:
 - ♦ **Linux:** `/opt/novell/nam/idp/webapps/nidp/WEB-INF/lib` (for roles)
 - or
 - `/opt/novell/nam/mag/webapps/nesp/WEB-INF/lib` (for other policies)

- ♦ **Windows:** C:\Program Files (x86)\Novell\Tomcat\webapps\nidp\WEB-INF\lib (for roles)
or
C:\Program Files\Novell\Tomcat\webapps\nesp\WEB-INF\lib (for other policies)

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 action.

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.

Actions extensions are used in Access Gateway Authorization policies.

- ♦ **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.

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.

Condition extensions are used in Access Gateway Authorization policies and Identity Server Role policies.

- ♦ **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.

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.

Data extensions can be used in Access Gateway Authorization policies, Access Gateway Identity Injection policies, Identity Server Role policies, External Attribute Source policies.

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

As the policy engine is configuring the policy, it calls the extension when it encounters an external element. The engine expects the extension to return an object that is specific to the type of extension, unless an exception occurs. The object contains any data that the extension needs for processing, and the object is returned to the policy engine with the data the engine needs to continue processing the policy. For specific details, see the following:

- ♦ [“How the Policy Engine Interacts with a Condition Extension” on page 43](#)
- ♦ [“How the Policy Engine Interacts with a Data Extension” on page 44](#)
- ♦ [“How the Policy Engine Interacts with an Action Extension” on page 45](#)

How the Policy Engine Interacts with a Condition Extension

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

This process is illustrated in the following code snippet:

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

During the next part of the configuration phase, the policy engine calls the `NxpeCondition.initialize` method, passing an `NxpeParameterList` object for the configuration parameters. The configuration parameters are used to initialize the `NxpeCondition` object and are the parameters that the extension needs for evaluating the condition. The values for these configuration parameters are retrieved at evaluation from the `NxpeInformationContext` object that is passed by the policy engine.

The `initialize` method is guaranteed to be 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;
}
```

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.initialize()` 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.

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 string 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 required to override this method to implement the supported action.

The following code snippet illustrates this process:

```

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

```

Common Elements and Tasks

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

- ♦ “Implementing Common Elements” on page 46
- ♦ “Initializing the Factory Object” on page 47
- ♦ “Retrieving Information from the Identity Server User Store” on page 48
- ♦ “Implementing the Extension Interface” on page 49

For information about the Extension API interfaces and class, see the [Javadoc API Reference](#).

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

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 <code>False</code> .
ConditionTrue	Condition	The compared values match, so the condition evaluation resolved to <code>True</code> .
ConditionUnknown	Condition	The values could not be compared, so the results are unknown. This is comparable to the <code>Result</code> on <code>Condition</code> <code>Error</code> 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 <code>NxpeException</code> class.
ErrorCodeComponent		Reserved.

Return Code	Extension Type	Description
ErrorConfigInitialization	All	The initialize method for the extension encountered an error. This result can be returned with the <code>NxpeException</code> class.
ErrorDataUnavailable	Context Data	The requested data is not available. This result can be returned with the <code>NxpeException</code> class.
ErrorIllegalArgument	All	The <code>informationContext</code> object contains an unknown parameter. This result can be returned with the <code>NxpeException</code> 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 <code>NxpeException</code> class.
ErrorNoMemory		Reserved
GeneralFailure	All	Unknown error. This result can be returned with the <code>NxpeException</code> class.
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.

Constructors in the `NxpeException` Class

The `NxpeException` class allows you to 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” on page 46](#).
- ◆ 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

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    }
12
13    public NxpeContextDataElement getInstance(
14        String strName,
15        int iEnumerativeValue,
16        String strParameter)
17        throws NxpeException
18    {
19        return (new LDAPGroupDataElement(strName, iEnumerativeValue,
strParameter));
20
21    }
22
23 } /* 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 23 implement the factory interface that creates an LDAPGroupDataElement object.

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

Retrieving Information from the Identity Server User Store

All extensions need to access an external data store and retrieve information from it. You need to know the type of data that your extension is going to 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 the Identity Server to log in as a user in the external data store. This method assumes that the user has the same credentials in the Identity Server user store and the external data store.
- ◆ You can create an LDAP attribute in the 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 the Administration Console to enter a username and password for accessing the external data store. The password is entered in clear text in the 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 programming 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 57](#)
- ◆ [“Available Configuration Parameters for a Condition Extension” on page 61](#)
- ◆ [“Available Configuration Parameters for an Action Extension” on page 64](#)

Implementing the Extension Interface

All extensions need to perform the following tasks.

- ◆ [“Task 1: Specifying the Required Import Files” on page 49](#)
- ◆ [“Task 2: Defining the Configuration Parameters” on page 50](#)
- ◆ [“Task 3: Retrieving Configuration Parameters before Policy Evaluation” on page 51](#)
- ◆ [“Task 4: Implementing the Extension Methods” on page 51](#)
- ◆ [“Task 5: Retrieving Configuration Parameters at Policy Evaluation” on page 52](#)
- ◆ [“Task 6: Connecting with the External Data Source” on page 53](#)
- ◆ [“Task 7: Returning from an Extension” on page 53](#)
- ◆ [“Task 8: Error Handling” on page 55](#)
- ◆ [“Task 9: Performing Extension-Specific Tasks” on page 55](#)

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 `NxpeException` class contains the defined constructors for throwing exceptions. For more information, see [“Constructors in the `NxpeException` Class” on page 47](#).

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 `NxpeParameterList` 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 46](#).

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 the 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.

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 what the LDAP group extension defines 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_GROUPOFUNIQUE NAMES);
    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 });
}
```

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);
}

```

The `NxpeContextDataElement` introduces a new element with additional methods. These methods help you control the duration for which data returned from the extension interface should be 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 52.

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.

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.

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.

Task 8: Error Handling

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

```
catch (NamingException e)
{
    if (debug)
    {
        e.printStackTrace();
    }

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

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.

Creating an Extension

You can create the following types of extensions:

- ◆ [“Creating a Context Data Extension” on page 56](#)
- ◆ [“Creating a Condition Extension” on page 60](#)
- ◆ [“Creating an Action Extension” on page 63](#)

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:

- ♦ [“Context Data Interfaces and Methods” on page 56](#)
- ♦ [“Available Configuration Parameters for a Data Context Extension” on page 57](#)

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

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.
	<code>getInstance</code>	Creates the <code>NxpeContextDataElement</code> object.
NxpeContextDataElement		Contains the methods required to create a context data element that can be used for injection, for activating roles, or in a condition.
	<code>initialize</code>	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 <code>configurationValues</code> parameter is valid only during the lifetime of the <code>initialize</code> method. If your extension needs to preserve this configuration data, you must maintain a reference. The <code>get</code> methods in this interface allow you to retrieve information about the parameters when the policy is being evaluated.
	<code>getEnumerativeValue</code>	Returns -1. Reserved for future releases.
	<code>getName</code>	Retrieves the name of the data element of the policy.
	<code>getParameter</code>	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 the parameter values that you need from the policy engine in order to perform the evaluation.

When you configure a condition in a policy in the Administration Console, 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.

Available Configuration Parameters for a Data Context Extension

You can use any of the data items listed in the Table 4.x to create configuration parameters that allow you 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 57](#)
- ♦ [Table 4-2, “Configuration Parameters for an Identity Injection Policy,” on page 58](#)
- ♦ [Table 4-3, “Configuration Parameters for an Authorization Policy,” on page 58](#)
- ♦ [Table 4-4, “Configuration Parameters for an External Attribute Source Policy,” on page 59](#)

Table 4-1 Configuration Parameters for a Role Policy

Data Item	Returns
Authentication IDP	The name of the 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.

Data Item	Returns
Roles from Identity Provider	The names of the Roles assigned to the user by the 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 4-2 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	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 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.

Data Item	Returns
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.

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

Data Item	Returns
Authentication IDP	The name of the 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.

Data Item	Returns
Roles from Identity Provider	The names of the Roles assigned to the user by the 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.

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.

- ♦ [“Interfaces and Methods for a Condition Extension” on page 60](#)
- ♦ [“Available Configuration Parameters for a Condition Extension” on page 61](#)

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.

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 61](#)
- ◆ [Table 4-6, “Configuration Parameters for an Authorization Condition,” on page 62](#)

Table 4-5 Configuration Parameters for a Role Condition

Data Item	Returns
Authentication IDP	The name of the 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 the 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.

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.

- ◆ [“Action Interfaces and Methods” on page 63](#)
- ◆ [“Actions” on page 63](#)
- ◆ [“Available Configuration Parameters for an Action Extension” on page 64](#)

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.

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 acknowledgement 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.

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.

Installing and Configuring an Extension

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

- ◆ [“Installing the Extension on the Administration Console” on page 65](#)
- ◆ [“Distributing a Policy Extension to Access Manager Devices” on page 67](#)
- ◆ [“Distributing the Extension to Customers” on page 67](#)

Installing the Extension on the 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	<p>A display name for the extension.</p> <p>A description for the extension.</p>
What you need to know	<p>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.</p> <p>The name of the Java class that is used by the extension. Each data type usually uses a different Java factory class.</p> <p>The filename of the extension.</p> <p>The type of data the extension manipulates.</p> <p>Authorization Policy: Can be used to return:</p> <ul style="list-style-type: none">◆ 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. <p>Identity Injection Policy: A data extension that retrieves data for injecting into a header.</p> <p>Identity Role Policy: Can be used to return:</p> <ul style="list-style-type: none">◆ 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 <p>External Attribute Source Policy: You can use it to:</p> <ul style="list-style-type: none">◆ 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 the 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, you need to create a configuration for each extension in the file.

- 1 Copy the JAR file to a location that you can browse to from the Administration Console.
- 2 In the 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 the 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:** From the drop-down list, select the type of extension you have uploaded.
 - Type:** From the drop-down list, 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:** From the drop-down list, 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, click the down-arrow, then 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” on page 67](#).

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.
 - ♦ **Linux:** Enter the following command:
Identity Server: `/etc/init.d/novell-idp restart`
Access Gateway: `/etc/init.d/novell-mag restart`
 - ♦ **Windows:** Enter the following commands:
`net stop Tomcat7`
`net start Tomcat7`
- 4 (Conditional) If the extension is for an Authorization policy or an Identity Injection policy, update the Access Gateway.
- 5 (Conditional) If the extension is for a Role policy, update the Identity Server.

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.

You need to import your JAR file and configure it as described in [“Installing the Extension on the Administration Console” on page 65](#). After it has been configured, you can select to 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, you need to document any parameter the user needs to modify after the import.

To export an extension:

- 1 In the 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.

Sample Codes

You can find the sample codes for the following extensions in `netiq-access-manager-sdk-4_2-extensions.tar.gz` in the [NetIQ Access Manager 4.2 Developer Kit](#).

For more information, see [NetIQ Access Manager - Sample Code](#).

- ◆ [“Data Extension for External Attribute Source Policy”](#) on page 68
- ◆ [“Template Policy Extensions”](#) on page 68
- ◆ [“LDAP Group Data Element”](#) on page 69
- ◆ [“PasswordClass”](#) on page 69

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 the 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 the Identity servers to retrieve attributes from external sources.

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

Template Policy Extensions

This includes the following two types:

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

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 the Administration Console.
- ◆ Implementation details of the extension factory and extension classes.

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

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 the Administration Console.
- ◆ Implementation details of the extension factory and extension classes.

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

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 the Administration Console.
- ◆ Implementation details of the extension factory and extension classes.

The policy extension example includes:

- ◆ `PolicyActionExtnDenyFactoryTemplate.java`
- ◆ `PolicyActionExtnDenyTemplate.java`
- ◆ `PolicyActionExtnPermitFactoryTemplate.java`
- ◆ `PolicyActionExtnPermitTemplate.java`

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`.

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 [“Understanding the Authentication Class Example” on page 21](#) and [“Deploying Your Authentication Class” on page 29](#).

5 Custom Rule in Risk-Based Authentication

This document explains how to create a Custom Rule Class for risk based authentication. The API presented here allows developers to leverage their own risk based custom rule mechanisms within the Risk based Authentication architecture. The following topics are covered:

- ◆ “Prerequisites” on page 71
- ◆ “Understanding the Rule Class” on page 71
- ◆ “Creating a Custom Rule Class” on page 72
- ◆ “Understanding the Custom Rule Class Example” on page 74
- ◆ “Deploying Your Custom Rule Class” on page 79
- ◆ “Understanding Custom attributes in History SQL Database” on page 81
- ◆ “Custom Geo Location Data Provider Integration” on page 82

Prerequisites

- ◆ Access Manager 4.1 or later.
- ◆ Your development environment requires the same installation as outlined in the [NetIQ Access Manager Installation Requirements](#)
- ◆ Copy the nidp.jar, NAMCommon.jar and risk-*.jar files in the following directory of your Identity Server to your development project:
 - ◆ On Linux: `/opt/novell/nam/idp/webapps/nidp/WEB-INF/lib`
 - ◆ On Windows: `C:\Program Files (x86)\Novell\Tomcatwebapps\nidp\WEB-INF\lib`

Understanding the Rule Class

Risk evaluation is done using a set of rules configured. You can configure the in-built default 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 rule as a custom rule. As shown in the below figure, Risk Engine evaluate all the rules configured one-by-one, and evaluates the Risk Score with Risk Level for the connecting user.

Figure 5-1 Risk Engine evaluating Rules



Risk Engine collects all the activity details of the connecting user and passes it on to the rules for evaluation. These include IP address of the connecting client, HTTP headers, Cookies, User attributes, user historical data etc.

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 will be added in case if the rule (Condition) fails.

Creating a Custom Rule Class

You can create the custom Rule class by extending the `com.novell.nam.nidp.risk.core.rules.Rule` class. This class is available with `risk-core.jar` file. You class must override the abstract method called 'evaluate()' in the custom class. This method should contain the business logic for the custom rule and this method should return 'true' if the rule condition is success. If not the method should return 'false'.

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

Authentication Methods	Description
<code>evaluate ()</code>	Takes <code>HttpContext</code> , <code>LocationContext</code> , <code>DeviceContext</code> , <code>UserContext</code> and <code>ResponseObject</code> as its arguments. Example of using these classes are provided in the code below. Returns True, if the rule evaluation passes. If failed, false will be returned and risk score will be considered for this rule.
<code>isHistoricalDataEnabled()</code>	Returns true if historical data is enabled for the rule
<code>getName()</code>	Returns the name of the Rule inString
<code>getPriority()</code>	Returns the priority of the rule in integer.
<code>isExceptionRule()</code>	Returns true if this rule is a Privileged Rule.
<code>isRuleEnabled()</code>	Returns true if this rule is enabled
<code>isNATed()</code>	Returns true if Nat setting is enabled for this server
<code>setType()</code>	Takes String or List as argument. This is used as part of the constructor to inform the RiskEngine to get the type of History data this Rule needs

Authentication Methods	Description
<code>clearType()</code>	Clears the Types set so far
<code>getType()</code>	Returns the List of Types set by this Rule
<code>isHistoryEnabled()</code>	Same as <code>isHistoricalDataEnabled()</code>
<code>getBoolean()</code>	Takes name of the property in String as argument and returns its boolean value. These are Rule properties set as part of the configuration.
<code>getProperty()</code>	Takes name of the Property in String and returns the value that is configured for this Rule in String
<code>getLong()</code>	Takes name of the property in String as argument and returns its long value. These are Rule properties set as part of the configuration.
<code>getInteger()</code>	Takes name of the property in String as argument and returns its int value. These are Rule properties set as part of the configuration.
<code>getClientIP()</code>	Takes <code>HttpContext</code> & <code>LocationContext</code> as arguments and returns IP of the connecting client in String
<code>isServerNATed()</code>	Same as <code>isNATed()</code>
<code>isNegateResult()</code>	Returns true if negate results options is enabled for the rule
<code>getReturnValue()</code>	Evaluated result is passed to it and this applies <code>isNegateResult</code> on it
<code>getRiskScore()</code>	Returns the risk score assigned to this rule in int
<code>SaveOnSuccessfulAuth()</code>	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.
<code>getRequiredAttributes()</code>	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
<code>getM_HTTPHeaders()</code>	Returns the name/value map of http headers of the connecting client
<code>getCookieValue()</code>	Returns the value of the cookie in String. Takes name of the cookie as argument in String

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

Authentication Methods	Description
<code>getClientIPAddress()</code>	Returns the client IP from the Http Request object

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

Authentication Methods	Description
------------------------	-------------

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

User session properties that are set by a custom authentication class can be used as part of the custom risk authentication rules. HTTPContext that is passed 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("ExternalEmail");
// Continue evaluation.
}
```

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());

        /*
        * Following commented code shows the way 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 will
    be 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 could 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());
}
}
}

```

Deploying Your Custom Rule Class

1. Create a jar file for your custom rule class and any associated classes.
2. Copy the jar file to the following location in the Identity Server:
 - ◆ Linux: /opt/novell/nam/idp/webapps/nidp/WEB-INF/lib
 - ◆ Windows: C:\Program Files (x86)\Novell\Tomcatwebapps\nidp\WEB-INF\lib

If the Identity Server is in a cluster, the file needs to be copied to all members of the cluster.
3. In the Administration Console, click **Access Manager > Policies > Risk Configuration > Rules > New**.

Figure 5-2 Custom Rule Name

Rule Name:

Rule Definitions

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

Rule Definitions: Select the 'custom rule' to configure the custom rule

4. Fill in the following fields:

Figure 5-3 Custom Rule Details

Rule Name:

Rule Definitions

Rule1 - Custom Rule

Custom Class Name: ⓘ

Check user history ⓘ

Negate Result ⓘ

Class Properties

Property Name: Value: ✖

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 the parameters and values which will be passed 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.

Figure 5-4 Specifying Risk Score for Custom Rule

Rule Group	Risk Score for New Rule	Add as Privileged Rule
<input checked="" type="checkbox"/> grp1	50	<input type="checkbox"/>
Add New Rule Group: <input type="text" value="Group Name"/>	<input type="text"/>	<input type="checkbox"/>

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 the Identity Server.
8. On the Identity Servers page, click **Update**.
9. Update any associated devices that are using this Identity Server configuration.

Understanding Custom attributes in History SQL Database

Risk module has a feature to save historical data of the user login as part of the SQL database. Custom rule examples explain how to read the existing parameters from the historical database. If you have requirement to create a new attribute in the database for your custom rules to use, then you could do it as follows:

1. Create the custom tables as below:

```
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. The table name should be '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>
e.g) custom_string_userlogintime
4. The attribute name should match the database column name.
5. Currently the following data types are supported for the custom attributes:

- ◆ String
- ◆ Int
- ◆ Char
- ◆ Boolean
- ◆ Date

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 below:

```
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 below:

```
(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.

Custom Geo Location Data Provider Integration

This section documents describes how to integrate the custom geo location data provider. The API presented here allows developers to integrate the custom geo location data provider within RISK based authentication of the Access Manager architecture. The following topics are covered:

- ♦ [“Prerequisites” on page 82](#)
- ♦ [“Understanding the Geo Location Provider interface” on page 83](#)
- ♦ [“Creating a Custom Geo Location Provider Class” on page 83](#)
- ♦ [“Understanding the Custom Geo Location Provider Class Example” on page 83](#)
- ♦ [“5.7.5 Deploying Your Custom Geo Location Provider Class” on page 84](#)

Prerequisites

- ♦ Access Manager 4.1 or later
- ♦ Your development environment requires the same installation as outlined in the ["NetIQ Access Manager Installation Requirements \(https://www.netiq.com/documentation/access-manager-41/\)"](https://www.netiq.com/documentation/access-manager-41/)
- ♦ Copy the nidp.jar, NAMCommon.jar and risk-*.jar and third party Geo Location data provider jar files in the following directory of your Identity Server to your development project:
 - ♦ On Linux: /opt/novell/nam/idp/webapps/nidp/WEB-INF/lib
 - ♦ On Windows: C:\Program Files (x86)\Novell\Tomcatwebapps\nidp\WEB-INF\lib

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. It returns the Geo Location information as GeoLocation Bean.

Creating a Custom Geo Location Provider Class

You can create the custom geo location provider class as follows:

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;  
}
```

We can create the Custom Provider class by implements the above interface. We should override the above `init()` and `readGeoLocInfo()` methods.

Extending Abstract Provider Class

```
import com.novell.nam.nidp.risk.core.geoloc.AbstractProvider;  
  
public abstract class AbstractProvider implements Provider {  
    abstract public void init(Properties props);  
  
    abstract public GeoLocBean readGeoLocInfo(InetAddress IPAddress)  
    throws GeoLocException;  
  
    public AbstractProvider(Properties props){  
        init(props);  
    }  
}
```

We can create the Custom Provider class by extending the above `AbstractProvider` class. We should override the above `init()` and `readGeoLocInfo()` abstract methods.

Understanding the Custom Geo Location 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 for
// this 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 geo location information
// from any external provider using webservice calls or any sources

    return null;
}
}

```

5.7.5 Deploying Your Custom Geo Location Provider Class

1. Create a jar file for your custom geo location provider class and any associated classes.
2. Copy the jar files to the following location in the Identity Server:
 - Linux: /opt/novell/nam/idp/webapps/nidp/WEB-INF/lib
 - Windows: C:\Program Files (x86)\NovellTomcatwebapps\nidp\WEB-INF\lib
3. If the Identity Server is in a cluster, the file needs to be copied to all members of the cluster.
4. In the Administration Console, click **Access Manager > policies > Risk Configuration > GeoLocation**
5. Select **Custom Provider** from the drop-down and fill in the following fields:

Figure 5-5 Specify Geo Location Rule Name

The screenshot shows the 'Risk Configuration' console with the 'Geolocation' tab selected. The 'Enable Location Profiling' checkbox is checked. Under 'Geolocation Provider Configuration', the 'Geolocation Provider' is set to 'Custom Provider', the 'Provider Name' is 'maxmind_local_D B', and the 'Java Class Path' is 'j.custom.risk.core.geoloc.providers.MaxMindLocalDB'. Below this, the 'Provider Properties' section contains an '+ Add Property' button and a single property with 'Property Name' 'citydbfile' and 'Value' 't\novell/GeoLiteCity.db'.

Provider Name: Specify a name that the Administration Console can use to identify this custom provider.

Java Class Path: This allows you to specify the path name of your custom Geo Provider Java class.

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

Property Name: Name of the parameter.

Value: Value of the parameter.

6. Click **OK**.
7. Restart the Identity Server.
8. On the Identity Servers page, click **Update**.
9. Update any associated devices that are using this Identity Server configuration.

A

Revisions

This section outlines all the changes that have been made to the Access Manager SDK documentation (in reverse chronological order).

May 2015	Added a new chapter: RBA Custom Rule
August 2012	Made the following changes in LDAP Server Plug-in: <ul style="list-style-type: none">◆ Changed the code under section 3.3 Directory Plug-In.◆ Replaced <code>nidp.jar</code> with <code>NAMCommon.jar</code>.◆ Replaced <code>com.novell.nidp.common.authority.ldap.jndi.LDAPStorePlugin</code> with <code>com.novell.nam.common.ldap.jndi.LDAPStorePlugin</code>. Added table 4-4 Configuration Parameters for an External Attribute Source Policy Added the 'External Attribute Source Policy' references in sections 4.1.2, 4.3.1, and 4.4.1. Added a new section: 4.5 Sample Codes. Fixed broken links. Changed the template to NetIQ.
February 2012	Removed Appendix A, "Identity Injection Java Plug-In. This API has been deprecated. Modified the tomcat paths for Linux and added paths for Windows.
November 2009	Removed two unsupported LocalClassAuthentication methods (<code>showErrorJSP</code> and <code>getIDPPProviders</code>) from the manual.
March 2009	Split the SDK into two SDKs, one for Access Manager 3.0.4 and one for Access Manager 3.1.
January 2009	Added information on how to use the policy extension API. This new feature is only available in Access Manager 3.1. For more information, see Chapter 4, "The Policy Extension API," on page 41 . Deprecated the Identity Injection Java Plug-In interface (Appendix A, "Identity Injection Java Plug-In). It has been replaced by the policy extension API in Access Manager 3.1.
July 2008	Added information about the LDAP server plug-in that you can create to extend the directory types that the Identity Server supports for user stores. See Chapter 3, "LDAP Server Plug-In," on page 33 .
April 2008	Added information about the <code>doAuthenticate()</code> method that was added in Access Manager 3.0 SP3 and takes advantage of secret store unlocking. See "doAuthenticate Method" on page 13 .
October 10, 2007	Added information about additional methods and grouped the methods. Added information about localizing an authentication class. See Chapter 2, "Identity Server Authentication API," on page 11 . Added a section on how to design a plug-in. (Section A.4, "Designing the Plug-In,")

May 14, 2007	Added Chapter 2, "Identity Server Authentication API," on page 11.
December 6, 2006	Added the component to the Novell Developer Kit as an Early Access (unsupported) document.
