Endeca® Platform Services

Security Guide

Version 6.1.1• December 2011

Contents

Copyright and disclaimer

Preface

Endeca® InFront enables businesses to deliver targeted experiences for any customer, every time, in any channel. Utilizing all underlying product data and content, businesses are able to influence customer behavior regardless of where or how customers choose to engage — online, in-store, or on-the-go. And with integrated analytics and agile business-user tools, InFront solutions help businesses adapt to changing market needs, influence customer behavior across channels, and dynamically manage a relevant and targeted experience for every customer, every time.

InFront Workbench with Experience Manager provides a single, flexible platform to create, deliver, and manage content-rich, multichannel customer experiences. Experience Manager allows non-technical users to control how, where, when, and what type of content is presented in response to any search, category selection, or facet refinement.

At the core of InFront is the Endeca MDEX Engine,™ a hybrid search-analytical database specifically designed for high-performance exploration and discovery. InFront Integrator provides a set of extensible mechanisms to bring both structured data and unstructured content into the MDEX Engine from a variety of source systems. InFront Assembler dynamically assembles content from any resource and seamlessly combines it with results from the MDEX Engine.

These components — along with additional modules for SEO, Social, and Mobile channel support — make up the core of Endeca InFront, a customer experience management platform focused on delivering the most relevant, targeted, and optimized experience for every customer, at every step, across all customer touch points.

About this guide

This guide describes the Endeca security features and the major tasks involved in using them to develop a secure Endeca implementation.

Who should use this guide

This guide is for developers who are responsible for implementing security features in Endeca applications.

Conventions used in this guide

This guide uses the following typographical conventions:

Code examples, inline references to code elements, file names, and user input are set in `monospace` font. In the case of long lines of code, or when inline monospace text occurs at the end of a line, the following symbol is used to show that the content continues on to the next line: ¬

When copying and pasting such examples, ensure that any occurrences of the symbol and the corresponding line break are deleted and any remaining space is closed up.

Contacting Endeca Customer Support

The Endeca Support Center provides registered users with important information regarding Endeca software, implementation questions, product and solution help, training and professional services consultation as well as overall news and updates from Endeca.

You can contact Endeca Standard Customer Support through the Support section of the Endeca Developer Network (EDeN) at *http://eden.endeca.com*.

Chapter 1

Introduction to Endeca Security Features

This section provides a high-level look at the security features that are available in the Endeca Information Access Platform.

Endeca Access Control System

The Endeca Access Control System is used to authenticate a user's identity against and obtain authorization information from a variety of external systems, such as an LDAP directory. The authorization information is used to control which records are retrieved during a query.

In any application that protects secure information, one of the first requirements is to clearly identify those users who should be granted access. By using the Endeca Access Control System, you can authenticate an end-user's identity against an external directory. The security architecture of the Access Control System is based on the Java Authentication and Authorization Service (JAAS) or the .NET Framework.

The Access Control System permits the Endeca Information Access Platform to support multiple authentication mechanisms in a simple plug-in fashion. Currently, the Endeca Information Access Platform supplies two authentication plug-ins:

- LDAPLoginModule plug-in
- FileLoginModule plug-in

In both cases, the authentication is performed at the Endeca Presentation API layer.

The authentication plug-ins also provide the application programmer with flexibility in how credentials are obtained from the user. The plug-ins are configured via a common configuration file.

Related Links

Access Control System Configuration on page 43
This section describes how to configure the Endeca Access Control System for your implementation.

LDAP directory authentication

The first method of authentication is to store user-relevant information in an LDAP directory.

The LDAP (Lightweight Directory Access Protocol) directory typically stores security-related information, such as a user's login name and what groups that user belongs to. Once login information is received

from the end user, it is passed to the Endeca LDAPLoginModule plug-in, which then communicates with the LDAP server, which performs the actual validation against its LDAP directory.

The LDAP server can also return a user's group membership information, if it is stored in the LDAP directory. This information will later be used to define the user's access privileges in the Endeca implementation.

The LDAP-based plug-in is implemented with the Presentation API `LDAPLoginModule` class.

Related Links

User Authentication with LDAP on page 65
This section explains how to authenticate users via the Endeca `LDAPLoginModule` plug-in.

File-based authentication

A second method of authentication is to store user-relevant information in a file.

The Access Control System provides a second login plug-in for a standalone directory local to the Endeca environment. A file in this native directory can store the names, passwords, and group memberships of valid users, similar to a UNIX `/etc/passwd` file that contains basic user attributes.

This type of user directory can be used if no other security infrastructure exists. However, its most common use is to allow developers to test security solutions quickly and simply as an Endeca implementation is being developed. If the site has access to an LDAP server, then the implementation can easily be switched to use the LDAP authentication described in the previous section.

The file-based authentication plug-in is implemented with the Presentation API `FileLoginModule` class.

Related Links

File-based User Authentication on page 73
This section explains how to authenticate users via the Endeca `FileLoginModule` plug-in. This plug-in handles logins authenticated against a password file.

Stacked authentication

The Java version of the Endeca Access Control System supports stacked authentication.

In the stacked authentication configuration, both login plug-ins are used to authenticate the user.

For example, you can use the LDAPLoginModule plug-in to authenticate the username and password of the end-user and then use the FileLoginModule plug-in to retrieve that end user's group affiliations.

This flexibility is important to a site that has a master LDAP directory, but wants to allow additional access control to a small number of users who can be supported by one administrator. The administrator can specify group membership via the local file, which is simple to maintain.

Related Links

Using stacked authentication on page 70
The Java version of the Endeca Access Control System also supports the notion of stacked authentication.

Endeca Access Control Lists for records

Endeca records can contain ACL properties.

A user's entitlement filter defines the Endeca records that the user may access via the Endeca MDEX Engine. The entitlement filter works in the context of permission properties that are tagged on those records.

For example, if a record's `Endeca.ACL.Allow.Read` property is set to the "research" group, then only users whose entitlement filter identifies them as being members of "research" will have access to that record.

A record can have multiple users or groups in its permission properties. If there are none, the record will not be matched by any entitlement filters and it will not be accessible to anyone. In contrast, to make a record available to all users, it must be tagged with a group or collection of groups that encompasses all users.

Related Links

Using ACLs for document access control on page 55
The Endeca Access Control System allows sensitive information to be indexed and presented in a MDEX Engine in such a way that only authorized personnel can search and navigate those records.

Tagging Endeca records

Record permissions are assigned during processing of the source data by the Endeca Data Foundry.

A source document's permissions are represented by an ACL (Access Control List) property.

Typically, these permissions are extracted from the source document's Windows or UNIX file system information by the Endeca Crawler and then mapped to Endeca ACL properties by the pipeline's property mapper. The permissions may also be set in the pipeline with an Access Rules component.

The Endeca ACL properties and the user entitlement filter thus assure that authenticated users can access only those records to which they have permission.

Related Links

Configuring the property mapper on page 59
The property mapper maps a record's source properties to Endeca properties or dimensions.

SSL

The Endeca IAP software supports SSL communications for its components.

Safe and trusted communication among the various components of your Endeca implementation ensures that the data being transmitted will not be compromised.

The Endeca software allows you configure Version 3.0 of the Secure Sockets Layer (SSL) protocol for the important communication endpoints. You can configure:

- Base SSL only, using encryption for keys up to 4096 bits. This configuration does not use mutual authentication between components.
- Mutually authenticated SSL, using both encryption and certificates for authentication of a component.

Using SSL for encrypted communications

The Endeca IAP software allows you to configure SSL between other Endeca components in several flexible combinations.

As an example of this flexibility, the EAC (Endeca Application Controller) can use SSL communication in three places:

- Between the EAC Central Server and the public Web service interface.
- Between the EAC Central Server and its Agents.
- With the MDEX Engine.

These SSL features are easily turned on and off via configuration settings in the EAC configuration files.

Note that the Endeca components that do not currently support SSL are the Log Server and Developer Studio.

Related Links

SSL Configuration on page 15
This section describes how to configure your Endeca implementation to use SSL connections among the various Endeca components.

Mutual authentication between Endeca components

You can configure mutual authentication for your SSL connections.

In addition to encryption, you can impose a higher level of security by configuring the SSL-enabled Endeca components to require mutual authentication. You can use an Endeca utility to create a Certification Authority (CA), which is then used to sign certificates for use by the various components.

During communication initialization, each component confirms that the certificate it is receiving from the other party has been signed by the CA. A bearer's possession of a signed certificate implicitly grants full access to the Endeca component it is contacting.

You create the CA file with the `enecerts` utility.

SSL certificate utilities

The standard Endeca MDEX Engine installation includes an `enecerts` utility.

This utility allows you to perform the following SSL certificate operations:

- Generate your own set of SSL certificate files.
- Create custom certificates with private keys that are larger or smaller than the default 1024-bit size.
- Provide your own CA file to use for mutual authentication among Endeca components.

With these SSL certificate files, you can configure SSL communications among Endeca components, such as an SSL-enabled EAC Central Server.

The Endeca software also includes a utility that can convert a PEM-format key to a standard Java KeyStore (JKS) format.

Related Links

Using Endeca SSL Certificate Utilities on page 37

This section describes how to use the Endeca `enecerts` utility to generate standard and custom SSL certificate files to be used for SSL connections between the various Endeca components. It also documents an Endeca utility that can convert a PKCS12-format key to a standard Java KeyStore (JKS) format.

Chapter 2

SSL Configuration

This section describes how to configure your Endeca implementation to use SSL connections among the various Endeca components.

Endeca system communications

The SSL (Secure Sockets Layer) protocol is designed to help protect the privacy and integrity of data while it is transferred across a network.

In an Endeca system, network communication occurs at multiple points. In addition to the connections that are established with the user's browser and with the LDAP server, there are other connections that are established between individual Endeca components.

Note: For information about configuring Endeca Workbench to use SSL for Web browser connections, see the *Workbench Administrator's Guide*. For information about configuring SSL to work with the deprecated Control Interpreter, see the *Control System Guide*.

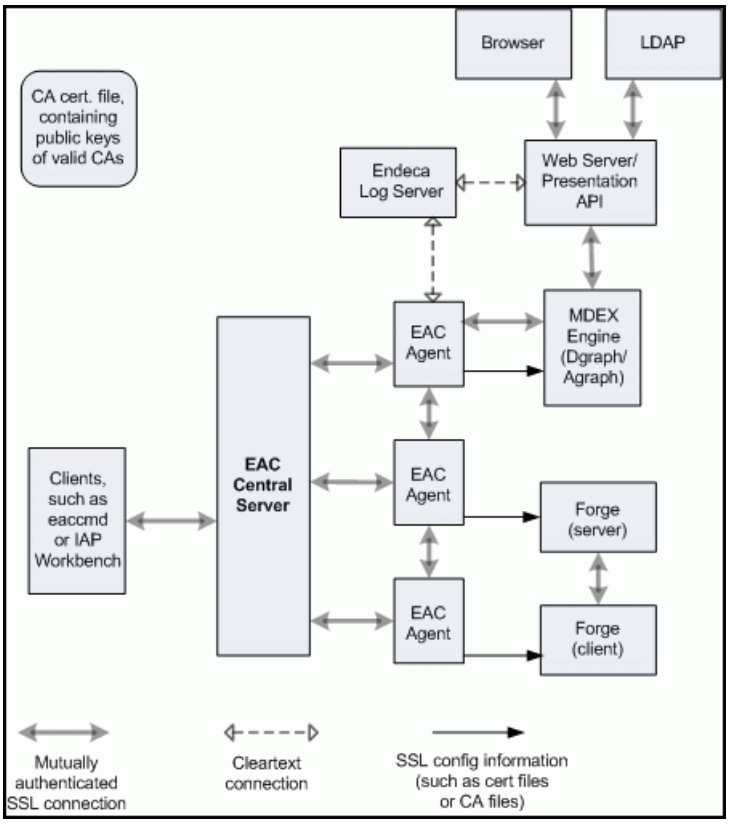
In a standard Endeca implementation, these communication links can be unencrypted. However, for highly secure implementations, these links may be encrypted with mutually-authenticated SSL.

The following illustration shows what type of communications can exist among the components of an Endeca secure implementation in an Endeca Application Controller (EAC) environment. It is important to keep in mind that the Application Controller can use SSL communication in three places:

- Between the EAC Central Server and its clients
- Between the EAC Central Server and its Agents
- With the MDEX Engine

In each case, because authentication is mutual, the host or client must contain both the keystore and the truststore.

As the illustration shows, not all components can use SSL. For example, the Endeca Log Server currently cannot be configured to use SSL.



Authentication among components

In addition to providing encryption, the SSL connections allow communicating Endeca components to mutually authenticate and implement a passport model of access control.

By using the `enecerts` utility, you can create a Certification Authority (CA) that can be used to sign certificates for use by the various components.

During communication initialization, each component confirms that the certificate it is receiving from the other party has been signed by the CA. A bearer's possession of a signed certificate (its passport) implicitly grants full access to the Endeca component it is contacting.

For example, when the Endeca Presentation API in the front-end contacts an Endeca MDEX Engine, it is allowed to make queries because both components have exchanged certificates.

Configuring SSL for the EAC

Making SSL work with the Endeca Application Controller involves generating a set of certificates and specifying their location in various configuration files (along with other configuration information).

The Endeca Application Controller can use SSL to mutually encrypt the HTTP channel between the EAC Central Server and its client (such as `eaccmd` or Endeca Workbench), as well as between the Central Server and its Agents. The following topics describe how to implement these connections.

Creating Application Controller certificates

You must create SSL keystores and truststores for the EAC components.

This task requires the use of the `enecerts` utility.

The following set of SSL keystores and truststores are necessary to configure SSL for the EAC:

- A keystore file (named `eac.ks` in the examples below).
- A truststore file (named `ca.ks` in the examples below).

Both files should be stored in the `$ENDECA_CONF\conf` directory on UNIX (`%ENDECA_CONF%\conf` on Windows). Because these certificates are not shipped with the Endeca IAP, you must generate them.

To create the SSL keystores and truststores for EAC:

1. Run the `enecerts` utility to generate a set of SSL certificates, among which are the `eneCert.pem` and `eneCA.pem` certificates.

   The `enecerts` utility is documented in Chapter 3 of this guide.
2. Run the `endeca-key-importer` utility to convert the `eneCert.pem` certificate to a `keystore.ks` keystore and the `eneCA.pem` Certificate Authority file to a `truststore.ks` truststore.

   The `endeca-key-importer` utility is also documented in Chapter 3 of this guide.
3. Rename the certificates as follows:
   - Rename the `keystore.ks` keystore to `eac.ks`.
   - Rename the `truststore.ks` truststore to `ca.ks`.
4. Copy both files to the `$ENDECA_CONF\conf` directory on UNIX (`%ENDECA_CONF%\conf` on Windows).

Alternatively at step 3, you can choose other filenames or retain the generated filenames. In this case, you would specify those names in the configuration files that are described in the following topics. For the purposes of the SSL configuration procedures in later topics, the `eac.ks` and `ca.ks` filenames will be used.

Enabling SSL security in the EAC

After creating the keystore and truststore certificates, you can configure the EAC components for SSL.

SSL in the Endeca Application Controller is disabled by default. To enable SSL security (between the client and the EAC Central Server, between the Central Server and an Agent, or between Agents), you need to:

- Enable the SSL version of the appropriate EAC WAR file (`eac-ssl.war` replaces `eac.war` for the Central Server and `eac-agent-ssl.war` replaces `eac-agent.war` for the Agent).
- Modify the `server.xml` file for the Tomcat that is hosting the EAC.

These procedures are explained in the following two topics.

Configuring the SSL version of the EAC WAR file

The SSL version of the EAC WAR file must be used for SSL.

When you install the EAC Central Server or Agent, the non-SSL version of the EAC WAR file is installed by default. After creating the SSL certificates, you must configure the SSL version of this file.

To enable the SSL version of the EAC WAR files:

1. Stop the Endeca HTTP Service.
2. Navigate to `%ENDECA_CONF%\conf\Standalone\localhost` (on Windows) or `$ENDECA_CONF/conf/Standalone/localhost` (on UNIX).
3. Open the `eac.xml` file, which is for the Central Server.
4. In the `docBase` attribute, replace the `eac-<buildNumber>.war` with the `eac-<build-number>-ssl.war` version.
   This file now points to the SSL-enabled version of the WAR file for the Central Server.
5. Save and close the `eac.xml` file.
6. In the same directory, open the `eac-agent.xml` file, which is for the Agent.
7. In the `docBase` attribute, replace the `eac-agent-<buildNumber>.war` with the `eac-agent-<buildNumber>-ssl.war` version.
   This file now points to the SSL-enabled version of the WAR file for the Agent.
8. Save and close the `eac-agent.xml` file.
9. Restart the Endeca HTTP Service.

If you want to restore the non-SSL versions at a later date, you can edit `eac.xml` or `eac-agent.xml` as needed.

Modifying the server.xml

You must modify the server.xml file for the Tomcat that is hosting the EAC.

Before you can use SSL with the EAC, you must edit its `server.xml` file as described below. Before beginning, make sure that you have generated keystore and truststore certificates for the EAC.

To enable the HTTPS connector in Tomcat:

1. Stop the Endeca HTTP Service.
2. Navigate to `%ENDECA_CONF%\conf` (on Windows) or `$ENDECA_CONF/conf` (on UNIX).
3. Open the `server.xml` file.
4. Remove the comments around the Connector element for port 8443, so that the result looks like this:

```
<!-- Define a SSL HTTP/1.1 Connector on port 8443 -->
<Connector port="8443" maxHttpHeaderSize="8192" SSLEnabled="true"
   maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
   enableLookups="false" disableUploadTimeout="true"
   acceptCount="100" scheme="https" secure="true"
   clientAuth="true" sslProtocol="TLS"
   keystoreFile="conf/eac.ks" keystorePass="eacpass"
   truststoreFile="conf/ca.ks" truststorePass="eacpass"
   URIEncoding="UTF-8"/>
```

5. For the `keystorePass` and `truststorePass` attributes, make sure that the passphrases match those in the certificates.
6. Optionally, change the port number to something other than 8443 if you do not want to use that default.
7. Change the `redirectPort` attribute on the regular HTTP connector to point to this same port. Alternatively, you can comment out the non-SSL connector in the `server.xml` file.

Note: If you are using eaccmd, do not comment out the non-SSL connector in the EAC Central Server's server.xml file.

8. Save and close the `server.xml` file.
9. Restart the Endeca HTTP Service.

The tag specifies an explicit location for the Java keystore and a passphrase to allow it to use the Application Controller keystore in the Tomcat conf directory. If you remove these attributes, Tomcat uses the default keystore in the user's home directory and assumes a passphrase of "changeit".

Enabling SSL for EAC clients

EAC clients should also be enabled for SSL.

In addition to implementing SSL on the EAC Central Server, you must also configure its client (such as eaccmd or Endeca Workbench) to use SSL. Clients and servers each require both a truststore and a keystore to communicate using SSL mutual authentication.

Replacing the default certificate files with custom keys

In a production environment, you may want to obtain certificates from a certificate authority, such as Verisign.

If you do obtain obtain certificates from a certificate authority, keep in mind that `ca.ks` has to have the CA key of the certificate authority needed to verify the server certificate. In the case of most major certificate authorities, the CA certificates are already stored in the default Java keystore (`<java_sdk_installation>\jre\lib\security\cacerts`) and should be used automatically.

In the case of the keystore, you can do either of the following:

- Create a new keystore and name it `eac.ks`.
- Alternatively, update the `server.xml` file to point to a different keystore that you already use. The latter method requires the name of the host, the type of keystore, and the password in the `server.xml` file.

For information on how to import a certificate, see the Java keytool reference at:

http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/keytool.html

Enabling SSL for eaccmd and EAC Web services

The eaccmd and EAC Web services scripts use command-line JVM arguments to enable SSL.

Enabling SSL for eaccmd

The eaccmd script (`eaccmd.bat` on Windows, or `eaccmd.sh` on UNIX) contains two JVM arguments that tell it to use the keystore and truststore.

Because they do not affect non-SSL operation, eaccmd passes these arguments by default:

```
IF EXIST %ENDECA_CONF%\conf\ca.ks (
   SET TRUSTSTORE=%ENDECA_CONF%\conf\ca.ks
) ELSE (
   SET TRUSTSTORE=%EAC_ROOT%\..\workspace\conf\ca.ks
)
```

```
IF EXIST %ENDECA_CONF%\conf\eac.ks (
   SET KEYSTORE=%ENDECA_CONF%\conf\eac.ks
) ELSE (
   SET KEYSTORE=%EAC_ROOT%\..\workspace\conf\eac.ks
)
...
SET JVM_ARGS=%JVM_ARGS% -Djavax.net.ssl.trustStore=%TRUSTSTORE%
  -Djavax.net.ssl.trustStoreType=JKS
  -Djavax.net.ssl.trustStorePassword=eacpass
SET JVM_ARGS=%JVM_ARGS% -Djavax.net.ssl.keyStore=%KEYSTORE%
  -Djavax.net.ssl.keyStoreType=JKS
  -Djavax.net.ssl.keyStorePassword=eacpass
```

Enabling SSL for the EAC Web services

The same SET JVM_ARGS command line-arguments documented above can be used with any AXIS client.

SSL settings in the eac.properties file

The `eac.properties` file contains SSL-related settings.

The `eac.properties` file is the general configuration file for the Endeca Application Controller. The following section describes the SSL-related settings you can specify in `eac.properties` and provides a configuration file example.

The SSL keystores in the `eac.properties` file are used when the EAC Central Server or Agent is acting as a client to other Agents, and not when the Agent is acting as a server. In the latter case, the SSL configuration information resides in the `server.xml` file.

SSL Setting	Description
`com.endeca.eac.sslKeyStore`	Path to the JKS keystore.
`com.endeca.eac.sslKey¬ StorePassphrase`	The passphrase associated with the keystore.
`com.endeca.eac.sslTrustStore`	Path to the JKS truststore.
`com.endeca.eac.sslTrust¬ StorePassphrase`	The passphrase associated with the truststore.

UNIX Example

```
...
# This must be a JKS key store type
com.endeca.eac.sslKeyStore=/usr/local/endeca/PlatformSer¬
vices/workspace/conf/eac.ks
com.endeca.eac.sslKeyStorePassphrase=eacpass

# This must be a JKS trust store type
com.endeca.eac.sslTrustStore=/usr/local/endeca/PlatformSer¬
vices/workspace/conf/ca.ks
com.endeca.eac.sslTrustStorePassphrase=eacpass
```

Windows Example

```
...
# This must be a JKS key store type
```

```
com.endeca.eac.sslKeyStore=C:\\Endeca\\PlatformSer¬
vices\\workspace/conf/eac.ks
com.endeca.eac.sslKeyStorePassphrase=eacpass

# This must be a JKS trust store type
com.endeca.eac.sslTrustStore=C:\\Endeca\\PlatformSer¬
vices\\workspace/conf/ca.ks
com.endeca.eac.sslTrustStorePassphrase=eacpass
```

Configuring SSL between Endeca Workbench and the Central Server

Endeca Workbench can be configured to communicate with a Central Server using mutually authenticated SSL.

Endeca Workbench reads in its configuration from the `webstudio.properties` file. Among other things, the file contains a set of properties and default values that specify the location of the SSL truststore and keystores used by Endeca Workbench. The `webstudio.properties` file is stored in:

- `%ENDECA_TOOLS_CONF%\conf` (on Windows)
- `$ENDECA_TOOLS_CONF/conf` (on UNIX)

To enable SSL communication with the Central Server in Endeca Workbench:

1. Generate your own EAC certificates.
2. Upload the keystore and truststore files to the Endeca Workbench server. Store the files in `%ENDECA_TOOLS_CONF%\conf` (on Windows) or `$ENDECA_TOOLS_CONF/conf` (on UNIX).
3. Open the `webstudio.properties` file and uncomment the properties in the SSL section. On Windows machines, make sure that each backslash in a path is escaped with a preceding backslash, as in this example:

   ```
   # The SSL settings for connecting to an SSL-enabled EAC
   # Configure your key store and trust store information here.
   javax.net.ssl.trustStore=C:\\Endeca\\Workbench\\workspace/conf/ca.ks
   javax.net.ssl.trustStoreType=JKS
   javax.net.ssl.trustStorePassword=eacpass
   javax.net.ssl.keyStore=C:\\Endeca\\Workbench\\workspace/conf/eac.ks
   javax.net.ssl.keyStoreType=JKS
   javax.net.ssl.keyStorePassword=eacpass
   ```

4. Update the properties to reference the paths, names, and passphrases of your custom keystore and truststore.

Setting	Description
`javax.net.ssl.trustStore`	Absolute path to the JKS truststore.
`javax.net.ssl.trustStoreType`	The type of truststore. Must be JKS.
`javax.net.ssl.trustStorePassword`	The passphrase for the truststore.
`javax.net.ssl.keyStore`	Absolute path to the JKS keystore.
`javax.net.ssl.keyStoreType`	The type of keystore. Must be JKS.
`javax.net.ssl.keyStorePassword`	The passphrase for the truststore.

Configuring Endeca Workbench to use the SSL port for the Central Server

There are two ways of configuring the SSL port for Endeca Workbench.

By default, Endeca Workbench uses the non-SSL port to initiate connections with the EAC Central Server, which is then forwarded to the SSL port using an internal redirect, thereby establishing a mutually authenticated connection.

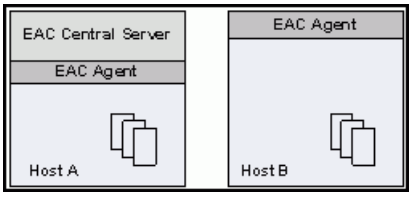
If you have disabled the non-SSL port on the EAC Central Server, or if you do not want to rely on this redirect, Endeca Workbench can initiate an HTTPS connection directly to the SSL port of the Application Controller. You can configure this behavior in Endeca Workbench by specifying the SSL port for the EAC Central Server, and selecting the Use HTTPS option. For more information about specifying the EAC Central Server in Endeca Workbench, see the Endeca Workbench Help.

If Endeca Workbench attempts to connect to the Application Controller on the SSL port with HTTP, or on the non-SSL port with HTTPS, the connection fails and an error message displays.

SSL interactions in an EAC environment

This section outlines the configuration requirements to run the Endeca Application Controller with various SSL elements enabled.

SSL may be on or off on a per-Agent basis, as long as the EAC Central Server or any other Agent needing to communicate with an SSL-enabled Agent have the appropriate client-side SSL configuration settings in the `eac.properties` file.



In all of the following examples, we assume two machines:

- Host A, which is running both an EAC Central Server and an EAC Agent
- Host B, which is running only an EAC Agent

Example One: SSL enabled everywhere

In the case where SSL is enabled everywhere, the configuration would be as follows:

Host A	Host B
1. The SSL-enabled version of the EAC Central Server, `eac-ssl.war`.	1. The SSL-enabled version of the Agent, `eac-agent-ssl.war`.
2. The SSL-enabled version of the Agent, `eac-agent-ssl.war`.	2. SSL settings in `eac.properties`: • a. Keystore identifying this server. • b. Truststore identifying the CA shared by all servers.
3. SSL settings in `eac.properties`: • a. Keystore identifying this server.	3. The Tomcat SSL connector enabled in `server.xml` as follows: • a. The same keystore as in 2a.

Host A	Host B
• b. Truststore identifying the CA shared by all servers.	• b. The same truststore as in 2b. • c. Client authentication turned on.
4. The Tomcat SSL connector enabled in `server.xml` as follows: • a. The same keystore as in 3a. • b. The same truststore as in 3b. • c. Client authentication turned on.	

Note: When the Agents are communicating with the EAC Central Server via SSL, you might note some related exceptions in the log file. These messages are benign and can safely be ignored.

Example Two: SSL disabled everywhere

In the case where SSL is disabled everywhere, the configuration would be as follows:

Host A	Host B
1. The non-SSL version of the EAC Central Server, `eac.war`.	1. The non-SSL version of the Agent, `eac-agent.war`.
2. The non-SSL version of the Agent, `eac-agent.war`.	2. No SSL settings in `eac.properties`.
3. No SSL settings in `eac.properties`.	3. The Tomcat SSL connector disabled in `server.xml`.
4. The Tomcat SSL connector disabled in `server.xml`.	

Note: In the `eac.properties` file, it is a good idea to comment the SSL settings out, rather than remove them, in case you choose to enable SSL at a later time.

Example Three: SSL enabled on the EAC Central Server only

In this scenario, the EAC Central Server is SSL-enabled, but the Agents are not.

Note: The SSL configuration for the EAC Central Server does not affect the Agent on the same server.

Host A	Host B
1. The SSL-enabled version of the EAC Central Server, `eac-ssl.war`.	1. The non-SSL version of the Agent, `eac-agent.war`.
2. The non-SSL version of the Agent, `eac-agent.war`.	2. No SSL settings in `eac.properties`.
3. No SSL settings in `eac.properties`.	3. The Tomcat SSL connector disabled in `server.xml`.

Host A	Host B
4. The Tomcat SSL connector enabled in `server.xml` as follows: • a. Keystore identifying this server. • b. Truststore identifying the CA shared by all servers. • c. Client authentication turned on.	

Example Four: SSL enabled on the EAC Agents only

In this scenario, the EAC Central Server is not SSL-enabled, but both Agents are SSL-enabled.

Host A	Host B
1. The non-SSL version of the EAC Central Server, `eac.war`.	1. The SSL-enabled version of the Agent, `eac-agent-ssl.war`.
2. The SSL-enabled version of the Agent, `eac-agent-ssl.war`.	2. SSL settings in `eac.properties`, for EAC Agent to Agent communication.
3. SSL settings in `eac.properties`, for EAC Central Server to Agent communication and Agent to Agent communication.	3. The Tomcat SSL connector enabled in `server.xml` as follows: • a. Keystore identifying this server. • b. Truststore identifying the CA shared by all servers. • c. Client authentication turned on.
4. The Tomcat SSL connector enabled in `server.xml` as follows: • a. Keystore identifying this server. • b. Truststore identifying the CA shared by all servers. • c. Client authentication turned on.	

Configuring stronger encryption

You can configure stronger encryption by using the BCC package.

The Bouncy Castle Crypto (BCC) package is included in your Endeca Information Access Platform installation. This package is a Java implementation of cryptographic algorithms and provides stronger encryption than the native JCE implementation. For example, RSA authentication and key exchange is supported for up to 4096-bit keys.

The package also contains the BCC provider, which is a JCE-compliant provider that is a wrapper built on top of the BCC light-weight API.

Before you integrate the BCC package, make sure that you are running Java 2 SDK version 1.4.x or later. Earlier versions of the Java 2 SDK do not support the stronger cryptographic capabilities of the BCC package.

To integrate the BCC package:

1. Find the BCC JAR file, which should have a name similar to the following: `bcprov-jdk14-121.jar` (the exact name of the file depends on its version number).

   Note: This file is shipped in the `$ENDECA_ROOT/lib/java` directory on UNIX (`%ENDECA_ROOT%\lib\java` on Windows). You may notice that the version of the JDK that is shipped with the Platform Services package is higher than the version indicated in this file's name. It is important to note that this file is compatible with the later version of the JDK with which it is shipped.

2. Copy the BCC JAR file to the `[JAVA_HOME]/jre/lib/ext` directory.
3. The JCE policy files shipped with the Java 2 SDK allow strong but limited cryptography to be used. To use the stronger encryption, replace them with the JCE Unlimited Strength Jurisdiction Policy Files, which you can download from the java.sun.com site (e.g., *http://java.sun.com/j2se/1.4.2/download.html*).
4. Unpack the JCE Unlimited Strength policy files (named `local_policy.jar` and `US_export_policy.jar`) and copy them to the `[JAVA_HOME]/jre/lib/security` directory. Note that they will be overwriting files of the same name, so you may want to first move the original files to another location.
5. Edit the `[JAVA_HOME]/jre/lib/security/java.security` file to add the Bouncy Castle provider. To add the Bouncy Castle provider to the java.security file, use an entry with this format (where *n* is the preference order of the provider):

   ```
   security.provider.n=org.bouncycastle.jce.provider.BouncyCastleProvider
   ```

   It is recommended that you not put the Bouncy Castle provider as the first name in the preference order. It is up to you to determine the actual order of the providers, but the following example is one recommended ordering.

Example of ordering providers

```
# List of providers and their preference orders
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.crypto.provider.SunJCE
security.provider.3=sun.security.jgss.SunProvider
security.provider.4=org.bouncycastle.jce.provider.BouncyCastleProvider
security.provider.5=com.sun.net.ssl.internal.ssl.Provider
```

Configuring SSL on the application server

The application server should be configured to use SSL.

In order to implement a secure application, you need to configure your application server to use SSL between the application server and client browsers. You may want to configure SSL as mutual (two-way) authentication in which both the server and the client browser are required to present a certificate to each other in order to successfully negotiate an SSL connection.

Because the exact details of configuring SSL differ from one application server to another, please consult your application server documentation for instructions.

Configuring SSL for the MDEX Engine

You can configure the MDEX Engine to use SSL and, optionally, mutual authentication when communicating with the Presentation API and other Endeca system components.

When configuring the MDEX Engine for SSL, keep in mind that you will be using the following two certificate files:

- `eneCert.pem` – The certificate used by all clients and servers to specify their identity when using SSL. This certificate file should be thought of as the identity of the Endeca system, or as the identity of all components of the Endeca system.
- `eneCA.pem` – The Certificate Authority (CA) file used by all clients and servers that wish to authenticate the other endpoint of a communication channel.

Likewise, you can configure Forge to use SSL between the Forge server and Forge client in a parallel Forge implementation. However, you must use a control script for this configuration because there is no Forge SSL interface in Endeca Workbench. For details, see the *Endeca Control System Guide*.

Application Controller configuration

The EAC Agent that controls the MDEX Engine should also be SSL-enabled.

When a Dgraph or Agraph is configured to require SSL connections, the Endeca Application Controller Agent that runs that program uses mutually authenticated SSL to communicate with it.

Note: In the Application Controller, there is no explicit interface for implementing SSL security for the Forge component. Instead, you must use command-line options.

Provisioning security for the Dgraph or Agraph

When a Dgraph or Agraph is provisioned to use SSL, when that component is started, it is given arguments on the command line that instruct the server to require mutually authenticated SSL for communication.

The Dgraph and Agraph components both include a section in their provisioning to define certificates to use for SSL. You configure this capability in the component definitions for the Dgraph and Agraph by adding the `ssl-configuration` element.

The settings are as follows:

- `cert-file` – The `cert-file` setting specifies the path of the `eneCert.pem` certificate file that is used by the MDEX Engine processes (Dgraph or Agraph) to present to any client. This is also the certificate that the Application Controller Agent should present to the MDEX Engine when trying to talk to the MDEX Engine. The file name can be a path relative to the component's working directory.
- `ca-file` – The `ca-file` setting specifies the path of the `eneCA.pem` Certificate Authority file that the MDEX Engine processes (Dgraph or Agraph) uses to authenticate communications with other Endeca components. If SSL has been enabled, using this setting will turn on mutual authentication. The file name can be a path relative to the component's working directory.
- `cipher` – The `cipher` setting is an optional cipher string (such as RC4-SHA) that specifies the minimum cryptographic algorithm that the Dgraph/Agraph processes will use during the SSL negotiation. If you omit this setting, the SSL software will try an internal list of ciphers, beginning with AES256-SHA.

The following is an example of an `ssl-configuration` element:

```
<ssl-configuration>
  <cert-file>/usr/local/endeca/workspace/etc/eneCert.pem</cert-file>
  <ca-file>/usr/local/endeca/workspace/etc/eneCA.pem</ca-file>
  <cipher>AES128-SHA</cipher>
</ssl-configuration>
```

Making changes to certificate values

The certificate values used are the ones that were specified by the configured files on disk at the time that the component was started. If the contents of these files change to specify different certificates after the component has been started, it has no effect on the running component. However, the next time the component is started, the new values are used.

Configuring the MDEX Engine for SSL in Endeca Workbench

In Endeca Workbench, you use the EAC Admin Console to configure the MDEX Engine (and Aggregated MDEX Engine) for SSL.

The EAC Admin Console page will let you specify the location of the SSL certificate files and the SSL cipher for the MDEX Engine. By clicking the Show Advanced Options of the MDEX Engine component, you will see three SSL-related fields: SSL Cert File, SSL CA File, and SSL Cipher. The following table lists the meanings of the SSL fields.

SSL Field	Description
SSL Cert File	The path of the `eneCert.pem` certificate that the MDEX Engine presents to any client. This is also the certificate that the Application Controller Agent should present to the MDEX Engine when trying to communicate with the MDEX Engine. The file name can be a path relative to the component's working directory.
SSL CA File	The path of the `eneCA.pem` Certificate Authority file that the MDEX Engine uses to authenticate communications with other Endeca components. The file name can be a path relative to the component's working directory.
SSL Cipher	An optional cipher string (such as `RC4-SHA`) that specifies the minimum cryptographic algorithm that the MDEX Engine will use during the SSL negotiation. If you leave this field blank, the SSL software will try to obtain a working cryptographic algorithm from its internal list of ciphers, beginning with `AES256-SHA`.

To configure SSL for the Endeca Workbench:

1. Log in to Endeca Workbench with the application that you want to administer.
2. Click the **EAC Administration** link.
3. Expand the MDEX Engine component and click **Show Advanced Options**.
4. In the SSL Cert File field, enter the full path of the SSL certificate file.
5. In the SSL CA File field, enter the full path of the SSL certificate authority file.
6. In the SSL Cipher field, either leave it empty or enter an SSL cipher string.
7. To save the configuration, click **Update**.

Note: If the MDEX Engine is running, you must stop it before you can save your changes.

Specifying cipher strings

When configuring SSL for the MDEX Engine and Forge, you should specify a cipher string to indicate which type of cryptographic algorithm will be used.

You set this cipher string in the `cipher` element when you provision the components, either in a provisioning file or in Endeca Workbench.

Keep in mind that the cipher string specifies the minimum cryptographic algorithm that you want to use. If, during the SSL negotiation between components, the Endeca system determines that a stronger algorithm is needed, then it will automatically use a stronger cipher suite. For example, if you specify the `AES128-SHA` cipher string, the system may actually use the stronger `AES256-SHA` cryptographic algorithm to make the SSL connection.

If you omit the `cipher` element, the SSL software will try to obtain a working cryptographic algorithm from its internal list of ciphers, starting with the `AES256-SHA` cipher. To make sure that you get the exact cryptographic algorithm that you want, you should specifically set it via the `cipher` element.

Some of the available cipher strings are listed in the following table.

Cipher string	Resulting cryptographic algorithm
AES128-SHA	KeyExchange=RSA, Authentication=RSA, Encryption=AES (128-bit), MessageDigestHash=SHA-1
AES256-SHA	KeyExchange=RSA, Authentication=RSA, Encryption=AES (256-bit), MessageDigestHash=SHA-1
DES-CBC3-SHA	KeyExchange=RSA, Authentication=RSA, Encryption=3DES (168-bit), MessageDigestHash=SHA-1
RC4-SHA	KeyExchange=RSA, Authentication=RSA, Encryption=RC4 (128-bit), MessageDigestHash=SHA-1
RC4-MD5	KeyExchange=RSA, Authentication=RSA, Encryption=RC4 (128-bit), MessageDigestHash=MD5

Configuring SSL for JSP applications

You can configure your JSP application to use SSL for communications with the MDEX Engine.

The application used as an example in this section is based on the JSP version of the Endeca reference implementation. Tomcat is used as the application server, with the JSP implementation being located in the Tomcat `webapps` directory.

To successfully run the user authentication process, you must perform the following tasks to set up the application server:

1. Write a `HostnameVerifier` class.
2. Create a JKS-format keystore certificate.
3. Configure an SSL connector on which the application can be accessed.
4. Start the application server with the appropriate keystore and truststore system properties.

These tasks are described in the following sections. An additional section also explains how to use PKCS12 certificates instead of JKS-format keystores.

Writing a HostnameVerifier class

You need to write a host name verifier that validates the host.

A host name verifier validates that the host to which an SSL connection is made is the intended or authorized party. In an Endeca JSP application, you use the `AuthHttpENEConnection.setHostnameVerifier()` method to set the host name verifier. Because this method takes a `javax.sun.net.ssl.HostnameVerifier` object type , you must create your own `HostnameVerifier` class.

During testing, you may want to use a null version of the `HostnameVerifier` class, which always returns true. The Java code for such a class is used in the example below. In a production environment, you would want to write a class that actually verifies that the host name is an acceptable match with the server's authentication scheme.

To write and implement your `HostnameVerifier` class:

1. Create a .java file with the following Java code. Note that the example creates a package named `myverifier`.

```
package myverifier;
import javax.net.ssl.HostnameVerifier;
import javax.net.ssl.SSLSession;
/**
 * Create a class to trust all hosts, so always returns true
 */

public class NullHostnameVerifier implements HostnameVerifier {
  public boolean verify(String urlHostname, SSLSession sslSession) {
    return true;
  }
}
```

2. Compile the .java file, as in the following example.

```
javac NullHostnameVerifier.java
```

3. Place the resulting .class file where it can be imported into your application. For example, if your application is located in the `C:\Tomcat\webapps\endeca_jspref` directory, then place the .class file in the following location.

```
C:\Tomcat\webapps\endeca_jspref\WEB-INF\classes\myverifier
```

4. Import the class into your application, as in the following example.

```
<%@ page errorPage="error.jsp" %>
<%@ page import="com.endeca.navigation.*" %>
<%@ page import="com.endeca.logging.*" %>
<%@ page import="myverifier.NullHostnameVerifier" %>
```

When the `AuthHttpENEConnection.setHostnameVerifier()` method is used in your application, your `NullHostnameVerifier` class provides the `verifier` object:

```
//Instantiate a connection object for the MDEX Engine
AuthHttpENEConnection nec = new AuthHttpENEConnection(emeHost, emePort);
// Enable the SSL connection with our NullHostnameVerifier class
nec.setHostnameVerifier(new NullHostnameVerifier());
```

Creating a JKS-Format keystore certificate

The application in this example uses a certificate in the standard Java KeyStore (JKS) format.

You can produce a JKS-format client certificate by converting the `eneCert.p12` certificate key that you generated with the `enecerts` utility. You will need a third-party utility to convert the key.

The following sections will assume that `eneCert.jks` is the name of the resulting JKS-format client certificate.

Configuring the SSL connector

You must enable an SSL HTTP/1.1 connector on the application server.

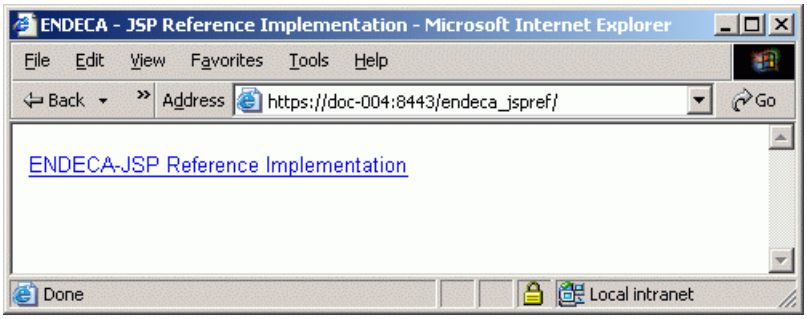
The JSP application will run on the port of this SSL HTTP/1.1 connector.

To enable the SSL connector, modify the Tomcat server.xml file with an entry similar to this example:

```
<!-- Define a SSL HTTP/1.1 Connector on port 8443 -->
<Connector port="8443" maxHttpHeaderSize="8192" SSLEnabled="true"
  maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
  enableLookups="false" disableUploadTimeout="true"
  acceptCount="100" debug="0" scheme="https" secure="true"
  clientAuth="false" sslProtocol="TLS"
  keystoreFile="C:\Endeca\PlatformServices\workspace\conf\eneCert.jks"
  keystorePass="endeca"
  truststoreFile="C:\Endeca\PlatformServices\workspace\conf\eneCert.jks"
  truststorePass="endeca"
  URIEncoding="UTF-8"/>
```

The example uses port 8443 and uses the `eneCert.jks` file as both the keystore and truststore file. However, mutual authentication is not enabled because the `clientAuth` attribute is set to `false`.

When the application server is running, you can access the application by using `https` in the browser URL, along with the host name, SSL port number, and name of the application, as in the following example:



Starting the application server with the keystores

When you start the Tomcat application server, you must specify the location and passphrase of the keystore and truststore files.

You used the following JVM `java -D` system property command arguments to specify the keystore and truststore files:

- `-Djavax.net.ssl.keyStore` specifies the keystore file.
- `-Djavax.net.ssl.keyStorePassword` specifies the passphrase of the keystore.

- `-Djavax.net.ssl.trustStore` specifies the truststore file to use to validate client certificates.
- `-Djavax.net.ssl.trustStorePassword` specifies the passphrase to access the truststore file.

One way to set these system properties is to use the `java` command from the command line.

A second method is to set the command arguments to the value of the Tomcat `CATALINA_OPTS` environment variable. This variable provides Java runtime options when the server is started.

You can set the `CATALINA_OPTS` environment variable in an existing Tomcat startup file (.bat on Windows or .sh on UNIX) or create a wrapper file that sets the variable and then calls the Tomcat startup file.

For example, the following Windows batch file can be placed in the Tomcat `bin` directory and used to start the server:

```
@echo off
setlocal

set CLIENT_CERT=C:\Endeca\PlatformServices\workspace\etc\eneCert.jks
set CATALINA_OPTS=-Djavax.net.ssl.keyStore=%CLIENT_CERT%
  -Djavax.net.ssl.keyStorePassword=endeca
  -Djavax.net.ssl.trustStore=%CLIENT_CERT%
  -Djavax.net.ssl.trustStorePassword=endeca
cd c:\tomcat\bin
call c:\tomcat\bin\startup.bat
endlocal
```

Note that the values for the `set CATALINA_OPTS` command are on separate lines for ease of reading, but should be on the same command line in the batch file.

Using PKCS12 keystores

You can use PKCS12 keystores instead of JKS-format client certificates.

The previous sections assume that the Tomcat application server is using a JKS-format client certificate. However, the Tomcat server version 5.0 and higher supports the use of PKCS12 keystores. Therefore, you can use the `eneCert.p12` certificate key that you generated with the `enecerts` utility.

To set up a PKCS12 keystore on a Tomcat server:

1. Edit the `JAVA_HOME/jre/lib/security/java.security` file and change the default keystore type:

   ```
   # Default keystore type.
   keystore.type=pkcs12
   ```

2. Configure the SSL connector by editing the Tomcat `server.xml` file with an entry similar to the following example. Note that the `keystoreType` and `truststoreType` attributes are set to "PKCS12" because you are not using the default JKS format.

   ```
   <!-- Define a SSL Coyote HTTP/1.1 Connector on port 8443 -->
   <Connector port="8443"
     maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
     enableLookups="false" disableUploadTimeout="true"
     acceptCount="100" debug="0" scheme="https" secure="true"
     clientAuth="false" sslProtocol="TLS"
     keystoreType="PKCS12"
     keystoreFile="C:\Endeca\MDEXEngine\workspace\etc\eneCert.p12"
     keystorePass="endeca"
     truststoreType="PKCS12"
   ```

```
  truststoreFile="C:\Endeca\MDEXEngine\workspace\etc\eneCert.p12"
  truststorePass="endeca" />
```

3. Start Tomcat with a batch file or script similar to the following Windows batch file example. (Note that the values for the `set CATALINA_OPTS` command are on separate lines for ease of reading, but should be on the same command line in the batch file.)

```
@echo off
setlocal
set CLIENT_CERT=C:\Endeca\MDEXEngine\workspace\etc\eneCert.p12
set CATALINA_OPTS=-Djavax.net.ssl.keyStoreType=PKCS12
  -Djavax.net.ssl.keyStore=%CLIENT_CERT%
  -Djavax.net.ssl.keyStorePassword=endeca
  -Djavax.net.ssl.trustStore=%CLIENT_CERT%
  -Djavax.net.ssl.trustStorePassword=endeca
cd c:\tomcat\bin
call c:\tomcat\bin\startup.bat
endlocal
```

Configuring SSL for ASP.NET Applications

You can configure your ASP.NET application to use SSL for communications with the MDEX Engine.

The general procedure for configuring your ASP.NET application is:

1. Configure the MDEX Engine to run SSL.
2. Convert your private certificate (`eneCert.pem`) to a DER format.
3. Import one or both of your certificates (`eneCert.p12` and `eneCA.pem`) to your local machine store (that is, the Local Computer/Personal certificate store).
4. Give the ASP.NET process permission to use the certificate.
5. Modify the application's entryp-point file to use SSL.

Except for step 1, the steps are explained in detail in the following sections.

Converting the private certificate to the DER format

You need to convert your private certificate into a DER (Distinguished Encoding Rules) format.

The DER format, which is one of the formats for X.509 certificates, provides a platform-independent method of encoding certificates for transmission between devices and applications. For these instructions, it is assumed that the certificate to be converted is the `eneCert.pem` certificate that you created with the `enecerts` utility and stored in the `%ENDECA_CONF%\etc` directory.

Use the openssl.exe program (in the `%ENDECA_MDEX_ROOT%\bin` directory) to convert the certificate. Assuming that you have opened a command prompt and have navigated to the `%ENDECA_CONF%\etc` directory, the conversion command is:

```
openssl x509 -inform PEM -outform DER -in eneCert.pem -out eneCert.der
```

The `eneCert.der` certificate will be later used in the `controller.aspx` file.

Importing the certificates to the local machine store

Your Personal Information Exchange (PKCS12-format) key file must be imported to your Local Computer\Personal certificate store.

The procedure in this section assumes that you are using the `eneCert.p12` private key that that you created with the `enecerts` utility and stored in the `%ENDECA_CONF%\etc` directory.

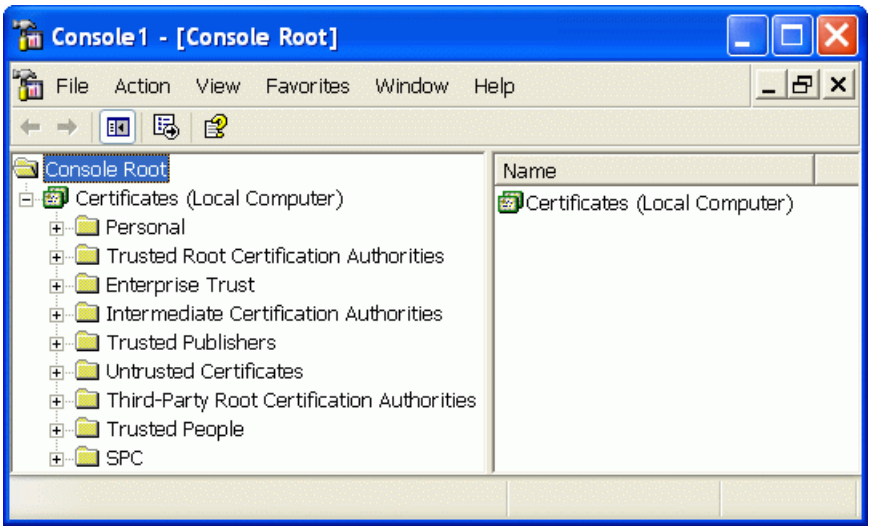
Using the Microsoft Management Console

To import the certificates, use the Microsoft Management Console (MMC) with the Certificates snap-in.

If your MMC does not have the Certificates snap-in, add it as follows:

1. Start the MMC by selecting **Run** from the Start menu, typing in `MMC`, and then clicking **OK**.
2. From the File menu, select **Add/Remove Snap-in**.
3. At the Add/Remove Snap-in dialog, select a name from the **Snap-ins added to** drop-down box. This procedure assumes that you have selected the default, `Console Root`.
4. At the Add/Remove Snap-in dialog, click **Add**. The Add Standalone Snap-in dialog box is displayed with a list of the snap-ins that are installed on your computer.
5. At the Add Standalone Snap-in dialog, select `Certificates` and click **Add**.
6. At the Certificates snap-in dialog, select `Computer account` and click **Next**.
7. At the Select Computer dialog, select `Local computer` and click **Finish**.
8. You are returned to the Add Standalone Snap-in dialog. Click **Close**.
9. You are returned to the Add/Remove Snap-in dialog. The dialog should have `Certificates (Local Computer)` in the list. Click **OK**.

As a result, the Console Root window now has the Certificates (Local Computer) snap-in rooted at the Console Root folder, as shown in this example:



Importing the private certificate

The procedure to import the `eneCert.p12` certificate is described in this topic.

To import the `eneCert.p12` (PKCS12) private certificate:

1. Start the MMC by selecting **Run** from the Start menu, typing in `MMC`, and then clicking **OK**.
2. In the console tree, expand **Certificates (Local Computer)** and right-click on **Personal**.
3. Point to **All Tasks** and then click **Import** to start the Certificate Import Wizard.
4. In the wizard, follow these steps:
   a) In the Welcome dialog, click **Next**.
   b) In the File to Import dialog, click **Browse** and navigate to the `eneCert.p12` private key. After you open the key, click **Next**.
   c) In the Password dialog, type the password used to encrypt the private key and click **Next**.
   d) In the Certificate Store dialog, select the **Place all certificates in the following store** button and select **Personal** as the certificate store. When you have done this, click **Next**.
   e) To exit the Certificate Store Wizard, click **Finish**. Click **OK** when you get the confirmation message.

To verify that the PKCS12-format key was successfully imported, click the **Local Computer\Personal\Certificates** link. The right-hand panel should list a key that was issued to "Endeca User".

Give permissions to the ASP.NET account

The account that runs the ASP.NET process must be given permission to use the PKCS12-format key that was imported to the local machine store.

The name of the the ASP.NET account is typically "NETWORK SERVICE" (although it may use another name on your system).

To assign the permissions, you must first download and install the Microsoft Windows HTTP Services Certificate Configuration Tool, which is available at the following Microsoft Web site:

http://www.microsoft.com/downloads/details.aspx?familyid=c42e27ac-3409-40e9-8667-c748e422833f

The tool installs to the `C:\Program Files\Windows Resource Kits\Tools` directory by default; `winhttpcertcfg.exe` is the name of the executable.

To run the tool, open a command prompt, navigate to the tool's installation directory, and issue this command:

```
WinHttpCertCfg -g -c LOCAL_MACHINE\MY -s "Endeca" -a "NETWORK SERVICE"
```

If the command is successful, you should see output like this:

```
Microsoft (R) WinHTTP Certificate Configuration Tool
Copyright (C) Microsoft Corporation 2001.

Matching certificate:
E=support@endeca.com
CN=Endeca User
O=Endeca Technologies
C=US

Granting private key access for account:
   NT AUTHORITY\NETWORK SERVICE
```

Modifying the application's entry-point file

The `controller.aspx` file must be modified with new function calls.

In the ASP.NET reference implementation, the `controller.aspx` file is the entry point into the Endeca application. If your application uses a different file structure, the information in this section will apply to the entry-point file for your application.

To enable SSL for the application, you must add two function calls to the file:

- The `HttpENEConnection.EnableSSL()` method enables an SSL connection to the MDEX Engine, using a previously-created X.509 v.3 certificate.
- `AcceptAllCertificatePolicy` class.

The next sections describe the syntax of these calls.

After modifying the `controller.aspx` file, you can run the application in the same way as before.

EnableSSL method

The signature of the `HttpENEConnection.EnableSSL()` method is:

```
HttpENEConnection.enableSSL(X509Certificate clientCertificate)
```

where *clientCertificate* is an `X509Certificate` object (from the .NET Framework Class Library) that is an implementation of an X.509 v.3 certificate.

The .NET `X509Certificate.CreateFromCertFile` method was used to create the `clientCertificate` object from the `eneCert.der` certificate (the `ASN.1 DER` format is the only certificate format supported by this class).

You should place the `EnableSSL()` method immediately after the `HttpENEConnection` connection object is instantiated, as shown in the example below.

AcceptAllCertificatePolicy class

The `AcceptAllCertificatePolicy` class is intended for situations where you want to prevent `Host Not Found` exceptions that are thown if the host name on the certificate does not match the name of the server. One example is if you are using the certificates that you generated with the `enecerts` utility. Note that you may not want to use this class if you are using your own custom certificates and want to verify the host name.

The signature of the `AcceptAllCertificatePolicy` class is:

```
AcceptAllCertificatePolicy(X509Certificate certificateToAccept)
```

where *certificateToAccept* is the same `X509Certificate` object used with the `HttpENEConnection.EnableSSL()` method. The X.509 certificate is set by the .NET Framework `ServicePointManager.CertificatePolicy` property to override any host name mismatches.

You can put the code after the X509 certificate is created from the DER-format version.

Example of a modified controller.aspx file

```
// Set the MDEX Engine connection
HttpENEConnection nec = new HttpENEConnection(ENEHost, ENEPort);
// Create the X509 certificate from the DER version
X509Certificate privateCert =
   X509Certificate.CreateFromCertFile(@"C:\Endeca\MyCerts\eneCert.der");
// Enable SSL for the connection, using the new X509 certificate.
nec.EnableSSL(privateCert);
// Now update the certificate validation with a custom policy.
```

```
// Required because Endeca certificates throw Host Not Found exceptions.
ServicePointManager.CertificatePolicy =
   new AcceptAllCertificatePolicy(privateCert);
// Create an ENEQuery for the MDEX Engine.
...
```

Chapter 3

Using Endeca SSL Certificate Utilities

This section describes how to use the Endeca `enecerts` utility to generate standard and custom SSL certificate files to be used for SSL connections between the various Endeca components. It also documents an Endeca utility that can convert a PKCS12-format key to a standard Java KeyStore (JKS) format.

Certificate files used by Endeca components

You configure SSL among the standard Endeca components by using a set of certificate files.

The certificate files are listed in the following table:

Certificate file	Description
`eneCert.pem`	Certificate file used by all Endeca clients and servers to specify their identity when using SSL. This certificate should be thought of as the identity of the Endeca system, or as the identity of all components of the Endeca system.
`eneCA.pem`	Certificate authority file used by all Endeca clients and servers to authenticate the other endpoint of a communication channel.
`eneCA.key`	Private key that is used by the `enecerts` certificate authority program to sign the `eneCert.pem` certificate.
`eneCA.cer`	Certificate authority file for import into browsers such as Microsoft Internet Explorer.
`eneCert.p12`	Personal Information Exchange (PKCS12-format) key file for import into browsers such as Microsoft Internet Explorer.

Because these certificate files are not provided in the Endeca IAP packages, you must use the Endeca-provided `enecerts` utility (documented in the next topic) to generate them. When you do, you should store them in the following directory:

- On UNIX: `$ENDECA_CONF/etc`
- On Windows: `%ENDECA_CONF%\etc`

In addition to the files listed above, the Endeca Application Controller keystore (`eac.ks`) and the client truststore (`ca.ks`) can be produced by the `endeca-key-importer` utility. This utility converts the `eneCert.pem` and `eneCA.pem` certificates from a PEM format to the standard Java KeyStore (JKS) format.

For .NET-based applications, you can use these files to secure connections between the .NET client and the EAC Central Server.

Generating SSL certificates

You can use the `enecerts` utility program to generate new SSL certificate files.

The two typical scenarios for generating SSL certificates are:

- You are setting up SSL for the first time and need to generate the set of standard certificates.
- You want to generate custom certificates, such as those with a private key size greater than the default 1024 bits.

The `enecerts` utility resides in the `$ENDECA_MDEX_ROOT/bin` directory (`%ENDECA_MDEX_ROOT%\bin` on Windows) under the name `enecerts` (`enecerts.exe` on Windows).

Generating standard SSL certificates on UNIX

This procedure shows how to generate the set of standard certificates with a 1024-bit private key size on UNIX platforms.

To generate the SSL certificates on a UNIX machine:

1. Make sure that the `$ENDECA_MDEX_ROOT` environment variable is set on your machine.
2. Change to the `$ENDECA_CONF/etc` directory, where the certificate files should reside.
3. Run the `enecerts` utility that creates the certificates:
   ```
   $ENDECA_MDEX_ROOT/bin/enecerts
   ```
4. Enter an export password of your choice.

If the programs finishes successfully, it displays the list of certificates that it generated.

Generating standard SSL certificates on Windows

This procedure shows how to generate the set of standard certificates with a 1024-bit private key size on Windows platforms.

To generate the SSL certificates on a Windows machine:

1. Open a command prompt.

   Note: Make sure you are using a new command prompt window, not one that is left over from earlier tasks.

2. To ensure that the MDEX Engine environment variables are set for this user process, change to the MDEX Engine root directory and run the `mdex_setup.bat` script.
3. Change to the `%ENDECA_CONF%\etc` directory.
4. Run the `enecerts` utility that creates the certificates:
   ```
   %ENDECA_MDEX_ROOT%\bin\enecerts
   ```
5. Enter an export password of your choice.

If the programs finishes successfully, it displays the list of certificates that it generated.

Generating custom certificates

You can use the `enecerts` utility to generate customized certificates.

You can generate two types of customized certificates by:

- Specifying a private key size larger or smaller than the default 1024-bit size.
- Using your own CA file and private key to generate the `eneCert.pem` certificate.

The next two sections describe these operations.

Specifying a different certificate key size

The `--keysize` flag of the `enecerts` utility lets users specify the size of the generated private key. The flag syntax is:

```
--keysize bits
```

where *bits* is the private key size in bits (default value is 1024).

For example, the following Windows command creates certificates with a private key size of 2048 bits:

```
enecerts --keysize 2048
```

Keep in mind that using larger keys will slow system performance. A recommended alternative to the default 1024-bit size is a key size of 512 bits, which will give you a good balance between security and performance considerations.

Using your CA file to generate certificates

By default, the `enecerts` utility produces the `eneCert.pem` certificate (used by all clients and servers to specify their identity when using SSL) and the `eneCA.pem` CA certificate (used by all clients and servers that wish to authenticate the other endpoint of a communication channel).

If you have your own CA certificate and private-key files, you can use the `--CAkey` and `--CAcert` flags to generate the `eneCert.pem` certificate. The private-key file (.key extension) is used to digitally sign the public key that is generated by the `enecerts` utility. Both flags must be used for this operation.

The syntax for the `--CAkey` flag is:

```
--CAkey private-key
```

where *private-key* is your own .key file with the private key for the CA that should be used to sign the generated certificate.

The syntax for the `--CAcert` flag is:

```
--CAcert cert-pem
```

where *cert-pem* is your CA certificate (.pem extension). This file is the same type of file as the default `eneCA.pem` CA certificate.

For example, the following Windows command creates a signed certificate file using your own CA certificate and private-key files:

```
enecerts --CAkey myCA.key --CAcert myCA.pem
```

You would then use the resulting `eneCert.pem` certificate and your CA file (`myCA.pem` in the example) to configure SSL for your Endeca components. If you have multiple machines in your deployment, you must also copy these files to the other machines.

Copying the SSL certificates to other machines

All machines that are running your deployment must use the same SSL certificates.

If you have multiple machines in your deployment, the standard or custom SSL certificates should be created only once, on one machine. You must then copy them to the `$ENDECA_CONF/etc` directory (on UNIX) or the `%ENDECA_CONF%\etc` directory (on Windows) on all other machines. All of the machines must use the same SSL certificates.

Importing SSL certificates in Internet Explorer

Depending on the details of your deployment, you may have to import the SSL certificates to your browser.

Typically, you do not need to import the SSL certificates in the Internet Explorer browser. For example, you can run the Endeca reference implementations without importing SSL certificates to your browser.

However, you must import the certificates in Internet Explorer if you are using the deprecated Endeca JCD with SSL and you want to connect directly to it via your browser. For details, see the *Endeca Control System Guide*.

To import the SSL certificates in Internet Explorer:

1. If you created the SSL certificates on a UNIX machine, copy the `eneCert.p12` and `eneCA.cer` files to the Windows machine.
2. Open Internet Explorer. From the Tools menu, choose **Internet Options**.
3. In the Internet Options dialog box, click the **Content** tab and then the **Certificates** button to display the Certificates dialog box.
4. If imported certificates are listed from any previous Endeca installations, delete them.
5. In the Certificates dialog box, click **Import** to launch the Certificate Import wizard. This allows you to import the standard or custom SSL certificates that you created using the enecerts utility. Follow these steps:
   a) In the Welcome screen, click **Next**.
   b) In the File to Import screen, browse to `eneCA.cer`, which is located in the `%ENDECA_CONF%\etc` directory or the directory to which you copied the file. You may have to change the **File of Type** option to `X.509 Certificate` to see the `eneCA.cer` file.
   c) In the Certificate Store screen, choose `Automatically select the certificate store based on the type of certificate`.
   d) In the Completing the Certificate Import Wizard screen, click **Finish**. (If you receive a Security Warning, click **Yes**.) When you see the confirmation message, click **OK**.
   e) Relaunch the Certificate Import wizard.
   f) In the File to Import screen, browse to the `eneCert.p12` file, which is located in the same directory as the `eneCA.cer` file.
   g) In the Password screen, enter the password you used when you created the SSL certificates.
   h) In the Certificate Store screen, choose `Automatically select the certificate store based on the type of certificate`.
   i) In the Completing the Certificate Import Wizard screen, click **Finish**. When you see the confirmation message, click **OK**.
6. Close the Certificates window.
7. Click OK in the Internet Options window.

If an SSL-enabled JCD is running, you can test the access to it with the imported SSL certificate files:

1. In Internet Explorer, enter a command similar to the following in the Address box to open the Endeca JCD home page:
   ```
   https://localhost:8088
   ```
2. If a Client Authentication dialog appears, select the certificate to use (for example, the `Endeca User` certificate) and click **Yes**.
3. The browser should display a page titled **Endeca JCD** followed by a list of links. You can click on any link to execute that command.

Converting PEM-format keys to JKS format

The Endeca Key Importer is a certificate conversion utility that allows you to convert PEM-format certificates to the standard Java KeyStore (JKS) format.

With the Endeca Key Importer utility, you can:

- Convert the `eneCert.pem` certificate file to a `keystore.ks` keystore file.
- Convert the `eneCA.pem` Certificate Authority file to a `truststore.ks` truststore file.

The Java keystores can then be used for communication between Endeca components that are configured for SSL (for example, between an EAC Agent and the MDEX Engine if both are SSL-enabled).

The Endeca Key Importer utility is provided as a JAR file, which is named `endeca-key-importer.jar` and is shipped in the `$ENDECA_ROOT/lib/java` directory for UNIX platforms and `%ENDECA_ROOT%\lib\java` for Windows platforms.

The usage syntax for the utility is:

```
endeca-key-importer.jar input_dir output_dir password
```

where:

- *input_dir* is the directory that contains the *eneCert.pem* and *eneCA.pem* certificates.
- *output_dir* is the destination directory for the `keystore.ks` and `truststore.ks` converted files.
- *password* is the passphrase for the keystores. Note that this should also be the passphrase of the original `eneCA.pem` certificate.

A passphrase that contains spaces or special characters may be double-quoted or escaped according to the rules of your command shell. Note that the Key Importer utility will neither accept nor create key materials that are not protected with a passphrase.

This example on a Windows platform shows how to run the utility:

```
java -jar %ENDECA_ROOT%\lib\java\endeca-key-importer.jar pems keystores
mypass
```

In the example, `pems` is the directory that contains the PEM certificates, `keystores` is the destination directory for the keystores, and `mypass` is the password for the certificates.

When the command finishes, it outputs instructions as to how to run the Java `keytool` utility that you can use to validate both resulting keystores.

Enabling .NET SSL communication with EAC

You can encrypt the HTTP connection between your .NET client and the Endeca Application Manager's EAC Central Server.

This section assumes that you have already implemented SSL in the EAC Central Server, including the generation of the EAC `eac.ks` keystore and the `ca.ks` client truststore. Therefore, the section outlines what you need to do on the .NET client side.

After the certificate for the CA has been installed, you can write your .NET client. In order to enable the .NET client code to communicate with the HTTPS file enabled in the Tomcat `server.xml` file on the EAC Central Server, you must edit the URL in the generated client code so that it points to `https://<eachost>:<eacsslport>` rather than the default `http://<eachost>:<eacport>`.

Modifying the ICertificatePolicy interface

The ICertificatePolicy interface is used to validate security certificates in a .NET application.

Because you can use the Endeca-generated certificates on different servers, the host name of the server cannot be validated. This can result in warning or error messages.

You can modify this policy to bypass the error conditions and point to your custom implementation. To do so, implement the `ICertificatePolicy` interface with a your custom policy, and then set the `SecurityPointManager.CertificatePolicy` to point to it.

The following example demonstrates such an override:

```
class ExampleDotNetSetup
  {
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
      try
      {
        // Set the certificate policy if you are using endeca certs
        // to ignore hostname matching
        ServicePointManager.CertificatePolicy = new MyCertificateValida¬
tion();
       // Create service stubs change url to be https://eachost:eacsslport

        // and invoke services...
        Console.Out.WriteLine("It worked");
      }
      catch (Exception e)
      {
        Console.Out.WriteLine(e);
      }
    }
}
```

For more information, contact Endeca Professional Services, or see the Microsoft Developer's Network (MSDN) documentation on `ICertificatePolicy`, which is located here:

```
http://msdn.microsoft.com/library/default.asp?url=/library/
en-us/cpref/html/frlrfsystemneticertificatepolicyclasstopic.asp
```

Chapter 4

Access Control System Configuration

This section describes how to configure the Endeca Access Control System for your implementation.

About the Access Control System

The Endeca Access Control System is used for the authentication and authorization of your application's end-users.

The Endeca Access Control System can be used for two purposes:

- For authentication of users, to reliably and securely control which records an authorized user may view.
- For authorization of users, to ensure they have the access control rights (permissions) required to do the actions performed.

Some details of the configuration of the Access Control System depend on whether your application is using Java or .NET. The following topics will point out these differences when applicable; otherwise, the configuration topic under discussion will apply to both .NET and Java.

Authentication framework

The type of authentication framework used depends on whether you use the Java or .NET version of the Presentation API.

JAAS authentication framework

To use the Endeca Access Control System on machines running the Java version of the Presentation API, you need the Java Authentication and Authorization Service (JAAS) to function as its framework.

JAAS implements a Java version of the Pluggable Authentication Module (PAM) framework, which permits applications to remain independent from underlying authentication technologies. The PAM framework allows the use of new or updated authentication technologies without requiring modifications to your application. Currently, the Endeca Information Access Platform supplies two authentication plug-ins: an LDAP-based plug-in and a local file-based plug-in. These plug-ins are named:

- `com.endeca.navigation.LDAPLoginModule`
- `com.endeca.navigation.FileLoginModule`

You can obtain JAAS by using Java 2 SDK version 1.4.x (or later), which includes JAAS.

.NET authentication framework

To use the Endeca Access Control System on machines running the .NET version of the Presentation API, you need the .NET Framework. The .NET Framework permits applications to remain independent from underlying authentication technologies.

You can obtain the .NET Framework version redistributable package from the downloads section of the Microsoft Web site.

The .NET Framework also supports the LDAP-based plug-in and the local file-based plug-in. Their names are:

- `Endeca.Navigation.AccessControl.LDAPLoginModule.LDAPLoginModule`
- `Endeca.Navigation.AccessControl.FileLoginModule.FileLoginModule`

The plug-in classes are in the `Endeca.Navigation.AccessControl.dll` file, which is shipped in the `%Endeca_Root%\lib\Endeca.NET` directory

Access Control configuration file

The Access Control System is configured via its own configuration file.

The Access Control System configuration file consists of one or more authentication configuration entries that specify which type of login authentication module will be used.

The general format of the configuration entries are given in this topic. Details on the entries are in the following topic.

Format of configuration entries for Java

If you are using the Java framework, the format for a configuration entry is:

```
ConfigEntryName {
   LoginModuleClass1 ModuleFlag
   ModuleOptions;
   LoginModuleClass2 ModuleFlag
   ModuleOptions;
};
```

The `LoginModuleClass` parameter is either `com.endeca.navigation.LDAPLoginModule` or `com.endeca.navigation.FileLoginModule`. Note that in the Java version, you can specify multiple LoginModule classes, as long as each name is unique.

Format of configuration entries for .NET

If you are using the .NET framework, the format for a configuration entry is:

```
ConfigEntryName {
   LoginModuleClass ModuleFlag
   ModuleOptions;
};
```

The `LoginModuleClass` parameter is either `Endeca.Navigation.AccessControl.LDAPLoginModule.LDAPLoginModule` or `Endeca.Navigation.AccessControl.FileLoginModule.FileLoginModule`. Note that in the .NET version, you can specify only one LoginModule class.

Configuration entry parameters

An authentication configuration entry for Endeca's Access Control System has the following parameters.

The configuration parameters apply to both Java and .NET frameworks. Differences between the two versions are noted in the table.

Parameter	Value
`ConfigEntryName`	Endeca must be used as the name of the configuration entry.
`LoginModuleClass`	These Endeca LoginModule classes can be specified for Java implementations: • `com.endeca.navigation.LDAPLoginModule` • `com.endeca.navigation.FileLoginModule` These Endeca LoginModule classes can be specified for .NET implementations: • `Endeca.Navigation.AccessControl.LDAPLoginModule.LDAPLoginModule` • `Endeca.Navigation.AccessControl.FileLoginModule.FileLoginModule`
`ModuleFlag` (Java framework)	The ModuleFlag value controls the overall behavior as authentication proceeds down the stack (i.e., the list of LoginModules). The value must be one of the following: `Required` – The LoginModule is required to succeed. If it succeeds or fails, authentication still continues to proceed down the LoginModule list. `Requisite` – The LoginModule is required to succeed. If it succeeds, authentication still continues to proceed down the LoginModule list. If it fails, control immediately returns to the application (authentication does not proceed down the LoginModule list). `Sufficient` – The LoginModule is not required to succeed. If it does succeed, control immediately returns to the application (authentication does not proceed down the LoginModule list). If it fails, authentication continues down the LoginModule list. `Optional` – The LoginModule is not required to succeed. If it succeeds or fails, authentication still continues to proceed down the LoginModule list.
`ModuleFlag` (.NET framework)	The ModuleFlag value controls the overall behavior of the authentication process. The value must be: `Required` – The LoginModule is required to succeed.
`ModuleOptions`	See the descriptions of the LoginModules in the following topics for details of each LoginModule's options.

ModuleFlag values for Java

When setting the ModuleFlag value for a Java framwork, keep in mind that the overall authentication succeeds only if all `Required` and `Requisite` LoginModules succeed. If a `Sufficient` LoginModule

is configured and succeeds, then only the `Required` and `Requisite` LoginModules prior to that `Sufficient` LoginModule need to have succeeded for the overall authentication to succeed. If no `Required` or `Requisite` LoginModules are configured for an application, then at least one `Sufficient` or `Optional` LoginModule must succeed.

Configuration examples

The following is an example of an Access Control System configuration file for a Java framework running on a UNIX machine:

```
Endeca {
   com.endeca.navigation.FileLoginModule required
   passwordFile="/usr/local/endeca/etc/passwd";
};
```

This is the .NET version of the same configuration running on a Windows machine:

```
Endeca {
   Endeca.Navigation.FileLoginModule.FileLoginModule required
   passwordFile="c:\\endeca\\navigationengine\\etc\\passwd";
};
```

In both examples, `Endeca` is the name of the configuration entry. The FileLoginModule is being used, with `required` as its module flag. The `passwordFile` module option specifies the location of the password file.

Specifying the location of the configuration file

This topic describes how the Java and .NET frameworks locate the Access Control configuration file.

A sample login configuration file (named `Login.conf`) is provided in the `%ENDECA_CONF%\etc` directory (`$ENDECA_CONF/etc` on UNIX). You can modify this file (including changing the file name) and store it in any directory.

How the Java framework finds the configuration file

In a Java framework, JAAS finds the login configuration file by examining the value of the `java.se¬curity.auth.login.config` property within the Java JVM in which the Access Control System is running. If this property is not set, the system will look for a file named `.java.login.config` in the home directory of the user who started the JVM.

How you set the `java.security.auth.login.config` property depends on the specifics of your application server. A common method to specify the configuration file is to use a JVM `-Djava.security.auth.login.config` command-ine argument, as in this example:

```
java -Djava.security.auth.login.config=Login.conf weblogic.server
```

A second method is to edit the `JAVA_HOME/jre/lib/security/java.security` file and add the name of the login configuration file, as in this Windows example:

```
# Default login configuration file
login.config.url.1=file:C:/EndecaProjects/SSL/Login.conf
```

Note that even though the path is on Windows, you must use forward slashes because the path is actually a URL.

Please consult your application server documentation for full details on how to set this property.

How the .NET framework finds the configuration file

The .NET Framework finds the login configuration file by examining the value of a registry setting, which is set by the Endeca installation program to the `%ENDECA_CONF%\etc\Login.conf` path.

Configuring the LDAPLoginModule plug-in

The `LDAPLoginModule` plug-in allows the Endeca Access Control System to authenticate users and obtain authorization information from an LDAP server.

The `LDAPLoginModule` has a large number of configuration parameters that allow it to be used with most LDAP configurations.

In Java, the `LDAPLoginModule` is:

```
com.endeca.navigation.LDAPLoginModule
```

In .NET, the `LDAPLoginModule` is:

```
Endeca.Navigation.AccessControl.LDAPLoginModule.LDAPLoginModule
```

LDAPLoginModule templates

The `LDAPLoginModule` allows templates to be supplied for certain configuration parameters.

These templates allow values from the authentication operation (such as the username and password) and the values from the user or group objects to be substituted into the parameter value. Any text not appearing in a `%{}` escape in a template is copied literally.

The table shows how the `%{}` escapes are expanded. Note that the templates apply to both the Java and .NET frameworks.

Escape	Description
%{#username}	The username value that will be provided to the `AuthHttpENEConnection.Login()` method via a `CallbackHandler` instance.
%{#dn}	The distinguished name of the user object.
%{#logname}	The username value provided by the `loginName` parameter for rebinding to the LDAP server.
%{#fieldname}	The value in the `fieldName` field of the user (or group object when used in the `groupTemplate` parameter) under consideration.

Selecting values from an escape sequence

Any escape sequence can have a path field value selected from it by appending a colon and the index of the field.

For example, if the value in the `%{#dn}` field is:

```
cn=joe,ou=People,dc=foo,dc=com
```

then the value "People" will be substituted for `%{#dn:1}`, while "joe" will be substituted for `%{#dn:0}`.

LDAPLoginModule required parameters for Java

These configuration parameters are required for the Java framework version of the `LDAPLoginModule` plug-in.

Parameter	Definition
userPath	The distinguished name pattern to match to find a user. The username entered by the user at login will be substituted for the `{username}` value. The default value is: `/ou=People,dc=endeca,dc=com??sub?(cn=%{#username:1})`
groupPath	A template that specifies the set of objects that contain the user's group memberships. The resulting information is used to construct an entitlement filter for the user. You can specify this parameter multiple times. The default value is: `/ou=Groups,dc=endeca,dc=com?cn?sub?(uniqueMember=%{#dn})`
groupTemplate	A template that specifies how to produce individual group names from the set of groups returned from the `groupPath` query. The default value is: `%{cn}`
serverInfo	A URL specifying the name and port of the LDAP server to be used for authentication. You can specify multiple LDAP servers. Note that the protocol portion of the URL (that is, `ldap://`) must be in all-lowercase. The default value is: `ldap://web1.endeca.com:1234`

Prepending strings to user and group names

For login purposes, you can set up the `LDAPLoginModule` plug-in to prepend strings to usernames and/or group names. Make sure to add the prepend string before the `%{}` escape sequence.

For example, assume you want to prepend the string "user:" to usernames and "group:" to group names. You would specify the `userPath` and `groupPath` parameters similar to the following example:

```
groupTemplate="%{cn}"
userPath="/ou=People,dc=endeca,dc=com??sub?(cn=user:%{#username})"
groupPath="/ou=Groups,dc=endeca,dc=com?sub?(uniqueMember=group:%{#dn})";
```

Specifying Multiple LDAP Servers

You can specify multiple LDAP servers with multiple instances of the `serverInfo` parameter, by using the format:

```
serverInfo.n = "ldap://server_url:port"
```

For example:

```
serverInfo.0="ldap://web01.endeca.com:1234"
serverInfo.1="ldap://web02.endeca.com:1230"
serverInfo.2="ldap://web03.endeca.com:1334"
```

If you specify multiple LDAP servers, the servers are assumed to be equivalent.

The choice of which LDAP server to contact is made randomly. If an LDAP server cannot be reached, the `LDAPLoginModule` plug-in proceeds through the remaining servers in order of configuration, wrapping if necessary.

For example, if five servers are configured and Server 3 is the first to be contacted, the remaining order of contact is Server 4, Server 5, Server 1, and finally Server 2.

LDAPLoginModule required parameters for .NET

These configuration parameters are required for the .NET framework version of the `LDAPLoginModule` plug-in.

Parameter	Definition
userURL	A template that specifies the location of the LDAP server and the distinguished name pattern to match to find the user to be authenticated. The username entered by the user at login will be substituted for the `{username}` value. The default value is: `ldap://xyz.com/ou=People,dc=endeca,dc=com??sub?(cn=%{#username:1})`
groupURL	A template that specifies the set of objects that contain the user's group memberships. The resulting information is used to construct an entitlement filter for the user. The default value is: `ldap://xyz.com/ou=Groups,dc=endeca,dc=com?cn?sub?(uniqueMember=%{#dn})`
groupTemplate	A template that specifies how to produce individual group names from the set of groups returned from the `groupPath` query. The default value is: `%{cn}`

LDAPLoginModule optional configuration parameters

These configuration parameters are optional for the `LDAPLoginModule` plug-in.

The optional parameters listed in this table apply to both the Java and .NET versions of the `LDAPLoginModule` plug-in.

Parameter	Definition
ldapBindAuthentication	If set to true, tells the `LDAPLoginModule` to authenticate users by rebinding as the user to the LDAP system, thereby employing the LDAP system's own authentication mechanism. The default value is `true`.
loginName	A template login name that will be used to rebind to the LDAP server if `ldapBindAuthentication` is `true`. The default value is: `%{#dn}`
passwordAttribute	The name of the attribute on the user object that contains the user's password. Used only if `ldapBindAuthentication` is set to `false`. The field specified must contain the user's password in clear text. The default value is the `userPassword` attribute.

Parameter	Definition
checkPasswords	Tells the `LDAPLoginModule` whether to check passwords during logins. If set to `false`, only the user name is used for logins. The default value is `true`.
serviceUsername	The username of the administrator login to the LDAP server that the `LDAPLoginModule` should use to find user objects. For example: `"cn=Manager,dc=foo,dc=com"` If no value is specified for this option, the `LDAPLoginModule` will authenticate anonymously. The default value is `""`.
servicePassword	The password to use in conjunction with the `serviceUsername` value. The default value is `""`.
useSSL	Tells the `LDAPLoginModule` whether or not to make mutually authenticated SSL connections to the LDAP server. If you set the parameter, make sure that you have configured the LDAP server to use SSL. The default value is `false`.

Additional parameters for Java

The following table lists the optional parameters that apply to only the Java version of the `LDAPLoginModule` plug-in.

Parameter	Definition
serviceAuthentication	Specifies the method of authentication that should be used in connecting to the LDAP server as the administrator account. The supported values are these strings: • none • simple (this is the default) • EXTERNAL
authentication	Specifies the method of authentication that should be used in connecting to the LDAP server as a user account. The supported values are these strings: • none • simple (this is the default) • EXTERNAL
keyStoreLocation	Specifies the location of the Java keystore, which stores keys and certificates. The keystore is where Java gets the certificates to be presented for authentication. The location of the keystore is OS-dependant, but is often stored in a file named `.keystore` in the user's home directory. The default value is `""`.
keyStorePassphrase	Specifies the passphrase used to open the keystore file. The default value is `""`.

LDAPLoginModule configuration examples

These sample login configuration files show how to configure the Access Control System to use an LDAP server for authentication.

Java example using an LDAP server

```
Endeca {
   com.endeca.navigation.LDAPLoginModule required
   ldapBindAuthentication="false"
   serviceUsername="cn=Manager,dc=endeca,dc=com"
   servicePassword="nosecret"
   checkPasswords="false"
   groupTemplate="%{cn}"
   useSSL="true"
   serverInfo.0="ldap://web01.qa.endeca.com:1234"
   serverInfo.1="ldap://web02.qa.endeca.com:1234"
   serverInfo.2="ldap://web03.qa.endeca.com:1234"
   userPath="/ou=People,dc=endeca,dc=com??sub?(cn=%{#username})"
   groupPath="/ou=Groups,dc=endeca,dc=com?sub?(uniqueMember=%{#dn})";
   keyStoreLocation="/localdisk/endeca/ldap/keystore"
   keyStorePassphrase="changeit"
};
```

.NET example using an LDAP server

```
Endeca {
  Endeca.Navigation.AccessControl.LDAPLoginModule.LDAPLoginModule required

  ldapBindAuthentication="false"
  serviceUsername="cn=Manager,dc=endeca,dc=com"
  servicePassword="nosecret"
  checkPasswords="false"
  groupTemplate="%{cn}"
  useSSL="false"
  userURL="ldap://web01.xyz.com:1234/ou=People,
    dc=endeca,dc=com??sub?(cn=%{#username})"
  groupURL="ldap://web01.xyz.com:1234/ou=Groups,
    dc=endeca,dc=com?sub?(uniqueMember=%{#dn})";
};
```

.NET example using an Active Directory server

```
Endeca {
  Endeca.Navigation.AccessControl.LDAPLoginModule.LDAPLoginModule required

  ldapBindAuthentication="true"
  serviceUsername="Administrator@Ad.com"
  servicePassword="endeca"
  useSSL="false"
  userURL="ldap://ad.com/cn=Users,dc=AD,dc=COM??one?
    (sAMAccountName=%{#username})"
  loginName="%{#username}@ad.com"
  groupURL.0="ldap://ad.com/cn=Users,dc=AD,dc=COM?memberOf?one?
    (sAMAccountName=%{#username})"
  groupURL.1="ldap://ad.com/cn=Builtin,dc=AD,dc=COM?name?one?
    (&(objectClass=group) (member=%{#dn}))"
  groupTemplate.0="%{memberOf:0}"
```

```
  groupTemplate.1="%{name}";
};
```

Configuring the FileLoginModule plug-in

The `FileLoginModule` is a simple LoginModule that reads login information from a flat file.

The file contains user, password, and group information that the `FileLoginModule` uses to authenticate the user.

In Java, the `FileLoginModule` is:

```
com.endeca.navigation.FileLoginModule
```

In .NET, the `FileLoginModule` is:

```
Endeca.Navigation.AccessControl.FileLoginModule.FileLoginModule
```

FileLoginModule configuration parameters

The `FileLoginModule` has one required parameter and one optional parameter.

The `FileLoginModule` parameters apply to both the Java and .NET frameworks.

Parameter	Definition
passwordFile	Required. Specifies the file in which user, password, and group information is stored. There is no default. On both UNIX and Windows platforms, you can use single forward slashes in the path (see Example 1 and Example 2 below). On Windows platforms, you can use double backslashes, in which the first backslash escapes the second one (see Example 3).
checkPasswords	Optional. Tells the `FileLoginModule` whether or not to check passwords during logins. If set to `false`, only the user name is used for logins. The default is `true`.

passwordFile examples

Example 1: a file path on a UNIX platform:

```
passwordFile="/usr/local/endeca/PlatformServices/workspace/etc/passwd"
```

Example 2: a file path using forward slashes on a Windows platform:

```
passwordFile="c:/Endeca/PlatformServices/workspace/etc/passwd"
```

Example 3: a file path using backward slashes on a Windows platform:

```
passwordFile="c:\\Endeca\\PlatformServices\\workspace\\etc\\passwd"
```

Password file format

The password file contains a series of user entries that specify the password and groups of each user.

Each user entry uses this format:

```
username:cleartextpassword:group1,group2,...groupN
```

The three fields are delimited by colons. Note that the *cleartextpassword* and *group* fields can be empty.

The following is a sample password file:

```
dave:en958:development,allcompany
sally:lopper39:development,allcompany
john:jhn931:marketing,allcompany
```

In this sample file:

- Dave has a password of "en958" and is allowed to see records in the `development` group.
- Sally has a password of "lopper39" and is also allowed to see records in the `development` group.
- John has a password of "jhn931" can see records in the `marketing` group.
- All three users can see records in the `allcompany` group.

FileLoginModule configuration examples

These sample login configuration files show how to configure the Access Control System to use a password file for authentication.

Java example on a UNIX machine

```
Endeca {
  com.endeca.navigation.FileLoginModule required
  passwordFile="/usr/local/endeca/PlatformServices/workspace/etc/passwd"
  checkPasswords="false";
};
```

.NET example using forward slashes

```
Endeca {
  com.endeca.navigation.FileLoginModule required
  passwordFile="c:/Endeca/PlatformServices/workspace/etc/passwd"
  checkPasswords="false";
};
```

.NET example using backslashes

```
Endeca {
  com.endeca.navigation.FileLoginModule required
  passwordFile="c:\\Endeca\\PlatformServices\\workspace\\etc\\passwd"
  checkPasswords="true";
};
```

Chapter 5

Using Record Permissions

This section describes the Endeca Access Control System, which allows application developers to control which records can be seen by different users.

Using ACLs for document access control

The Endeca Access Control System allows sensitive information to be indexed and presented in a MDEX Engine in such a way that only authorized personnel can search and navigate those records.

The Access Control System controls document access by matching Access Control Lists (ACLs) properties on records to group information associated with a user's entitlement filter. If you are using one of the LoginModule plug-ins for user authentication, this entitlement filter is automatically created during the authentication process.

ACLs can be added to Endeca records like any other property during data processing. The Access Rules component, which you can create with Developer Studio, lets you configure rules from which Endeca ACLs can be created.

The Endeca Content Acquisition System can extract native file-system ACLs from crawled files and directories. These extracted ACLs can be used directly or transformed by Forge pipeline components to restrict access to records.

Each entry in an ACL extracted by the Content Acquisition System is represented by a property attached to the Endeca record. The format of these properties depend on the type of machine (UNIX or Windows) from which the crawl is run. An overview of the ACL formats is given in the next two sections. For a complete list of the properties extracted by the Content Acquisition System, see the *Endeca CAS Developer's Guide*, which is available on EDeN.

Windows ACLs

Crawls run from a Windows machine return one of the following name/value forms for each source document:

```
Endeca.FileSystem.ACL.AllowRead = DOMAIN\principal
Endeca.FileSystem.ACL.DenyRead = DOMAIN\principal
```

where *principal* is the name of a user, group, or other principal who either has the right to read the record (the `Endeca.FileSystem.ACL.AllowRead` property) or is denied that right (the `Endeca.FileSystem.ACL.DenyRead` property). The name of the principal is prepended with the domain to which the name belongs.

For example, a Windows file could result in this property being set on a record:

```
<PROP NAME="Endeca.FileSystem.ACL.AllowRead">
   <PVAL>SALES\jbrown</PVAL>
</PROP>
```

This property indicates that the user named `jbrown` from the SALES group is granted the privilege to read that record.

Keep in mind that the `Endeca.FileSystem.ACL.AllowRead` source property must be mapped to the `Endeca.ACL.Allow.Read` property by a Forge pipeline component (such as with a record manipulator or with the property mapper.

UNIX ACLs

Crawls run from a UNIX machine return the following properties on the record:

Endeca Property Name	Property Value
`Endeca.FileSystem.Owner`	The name of a user or other principal who is the owner of the document.
`Endeca.FileSystem.IsOwn¬erReadable`	A Boolean that indicates whether the file owner (the `Ende¬ca.FileSystem.Owner` value) has read rights to the document.
`Endeca.FileSystem.Group`	The name of a group for which permissions have been set for the document.
`Endeca.FileSystem.Is¬GroupReadable`	A Boolean that indicates whether the group (the `Ende¬ca.FileSystem.Group` value) has read rights to the document.
`Endeca.FileSystem.IsWorl¬dReadable`	A Boolean that indicates whether everyone on the system (`world`) has read rights to the document.

With UNIX ACLs, a component (such as a record manipulator or Perl manipulator) can test the value of the Readable properties and, if true, assign the principal to the `Endeca.ACL.Allow.Read` property. For example, the pseudo-code would be:

```
If Endeca.FileSystem.IsOwnerReadable is true
Then assign Endeca.ACL.Allow.Read to the value of Endeca.FileSystem.Owner
```

Refinements and spelling with Access Control

Besides access to records, the Access Control System also limits access to refinements and spelling suggestions based upon the records that a user is allowed to access.

The MDEX Engine enforces this access as follows:

- For spelling corrections, the MDEX Engine will only suggest a word if it appears in at least one record that the user is allowed to see.
- For refinements, the MDEX Engine will only suggest a refinement if each of its constituent dimension values are tagged to at least one record that the user is allowed to see.

Creating the crawler pipeline

This section describes two pipeline components for records returned by the Endeca File System Crawler.

The section also describes how to create the `Endeca.ACL.Allow.Read` property and configure the property mapper.

Configuring a Binary or XML record adapter

The File System Crawler creates Endeca records in a format (XML or binary) ready for processing by Forge.

To read in the output file, you can add an input record adapter with a format of either XML or binary (depending on how you configure the output format). The URL field of the record adapter will point to the location of the output file.

To configure the record adapter:

1. In Developer Studio, specify the following basic settings in the General tab of the Record Adapter editor:

Field	Value
Direction	Must be `Input`.
Format	Must be either `Binary` or `XML`.
URL	Enter an input URL as a path, using a wildcard a pattern for the filename. For example, a URL pattern of `../incoming/*.bin.gz` means that Forge will read any file in the incoming directory that has the `bin.gz` suffix. Each file that matches the pattern will be read in strict lexicographic order of their filenames.
Multi File	Check this box to specify that Forge can read data from more than one input file and that the input URL is to be interpreted as a pattern.

2. You can leave the other tabs (Sources, Record Index, and so on) in their default state.
3. Click **OK** to add the component.

Adding a record manipulator

Parsing, duplicate detection, and a number of other small tasks related to crawling are performed by expressions in a record manipulator.

The expressions in a record manipulator are evaluated against each record as it flows through the pipeline. When an expression is evaluated, it may change the current record.

For example, one way to rename the Windows `Endeca.FileSystem.ACL.AllowRead` source property to the `Endeca.ACL.Allow.Read` property is to add a record manipulator with a RENAME expression. The code for the expression would be similar to this:

```
<EXPRESSION LABEL="" NAME="RENAME" TYPE="VOID" URL="">
  <EXPRNODE NAME="OLD_NAME"
    VALUE="Endeca.FileSystem.ACL.AllowRead"/>
  <EXPRNODE NAME="NEW_NAME" VALUE="Endeca.ACL.Allow.Read"/>
</EXPRESSION>
```

You would add the record manipulator after the record adapter.

When the pipeline runs, the record manipulator changes the name of this source property:

```
<PROP NAME="Endeca.FileSystem.ACL.AllowRead">
   <PVAL>SALES\jbrown</PVAL>
</PROP>
```

to this property name in the Endeca record that is output by Forge:

```
<PROP NAME="Endeca.ACL.Allow.Read">
   <PVAL>SALES\jbrown</PVAL>
</PROP>
```

Keep in mind that if you use a record manipulator to change the property name, you do not have to use the property mapper to rename the property. However, the property mapper still has to map the source property to an Endeca property, even though the two names are the same.

Creating the Endeca.ACL.Allow.Read property

While the Endeca File System crawler can create many different properties from native file system ACL information, the MDEX Engine only uses the `Endeca.ACL.Allow.Read` permission to determine record access.

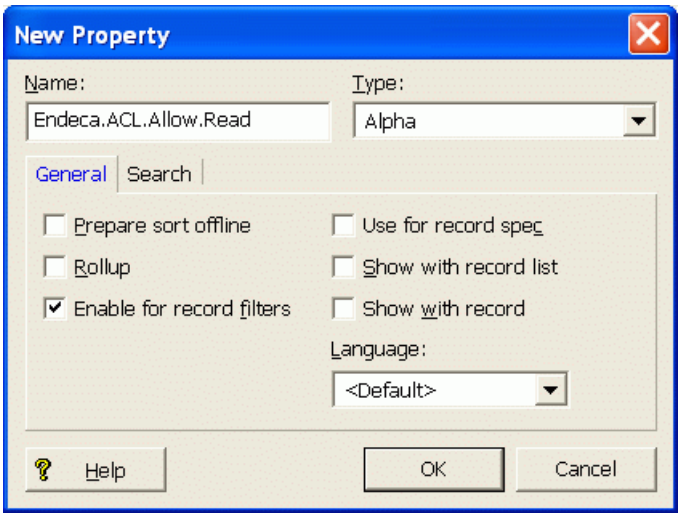
This requirement means that if an application's access control policy depends upon other native ACL information (for example, Deny information), the pipeline will need to transform these other properties into a form that is usable by the MDEX Engine. For access control to function, the data set must contain an `Endeca.ACL.Allow.Read` property that is enabled for user entitlement filters. Because this property is not automatically created during the data processing stage, you must explicitly create it with Developer Studio.

To create the `Endeca.ACL.Allow.Read` property:

1. In the Project tab of Developer Studio, double-click Properties to open the Properties view.
2. Click **New**. The New Property editor is displayed.
3. Configure the property as follows:
   a) Enter `Endeca.ACL.Allow.Read` in the Name text box.
   b) Select **Alpha** as the property type.
   c) Check the **Enable for Record Filters** option.

   These three attributes are mandatory; however, you can add other attributes if you wish.
4. Click **OK**. The Properties view is redisplayed with the new property listed.
5. From the File menu, choose Save.

After step 3, the New Property editor should look like this:



Configuring the property mapper

The property mapper maps a record's source properties to Endeca properties or dimensions.

The configuration information that follows assumes that you have already created the Endeca.ACL.Allow.Read property and have also added a property mapper to the pipeline.

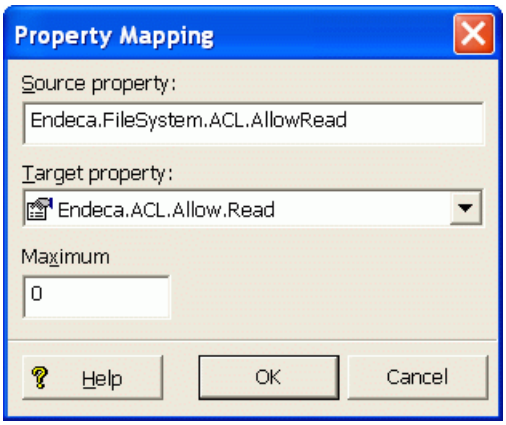
To map the `Endeca.ACL.Allow.Read` property:

1. In the Pipeline Diagram, double-click the property mapper to open it in the Property Mapper editor.
2. In the Property Mapper editor, click **Mappings**.
3. In the Mappings editor, select **New > Property Mapping**.
4. In the Property Mapping editor, configure the mapping as follows:

Field	Value
Source property	Enter the name of the source property on the record. For example, enter `Endeca.FileSystem.ACL.AllowRead` for Windows ACLs.
Target property	Select **Endeca.ACL.Allow.Read** as the target. Note that you must have previously created the target property before its name appears in the drop-down list.
Maximum	Enter 0 (zero) to set no limit on the length of the property value.

5. Click **OK**.

After step 4, the Property Mapping editor should look similar to this example:



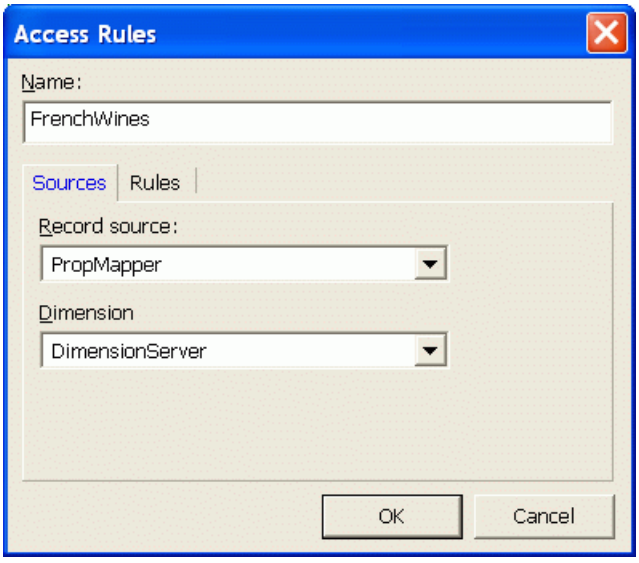
Creating the Access Rules component

Another method of setting Endeca ACLs on your records is to add an Access Rules component to your pipeline.

With an Access Rules component, you can restrict the Endeca records that a user can see by creating a set of access rules for the Endeca properties in your implementation. Each rule states that if the value of a property on a record is equal to a value that you specified, then that record can be seen by any member of a group that you also have specified.

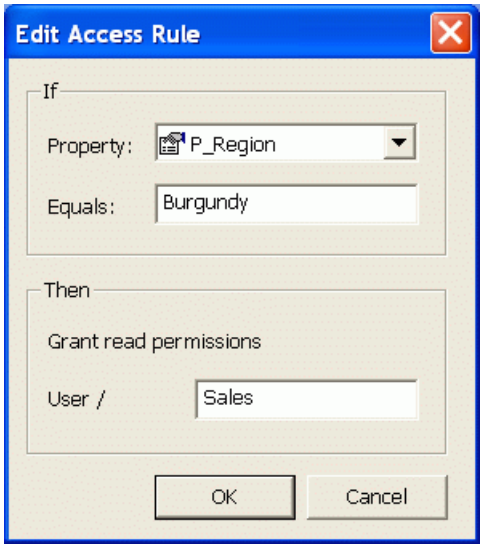
To create an Access component in Developer Studio:

1. Make sure that you have created an `Endeca.ACL.Allow.Read` property in your implementation. Developer Studio will automatically add the property to the property mapper and map the property to itself.
2. In the Pipeline Diagram, select **New > Access Rules**. The Access Rules editor will be displayed.
3. After naming the Access Rules component, use the Sources tab to specify its Record source as a property mapper (or another Access Rules component) and its Dimension source as a Dimension Server:

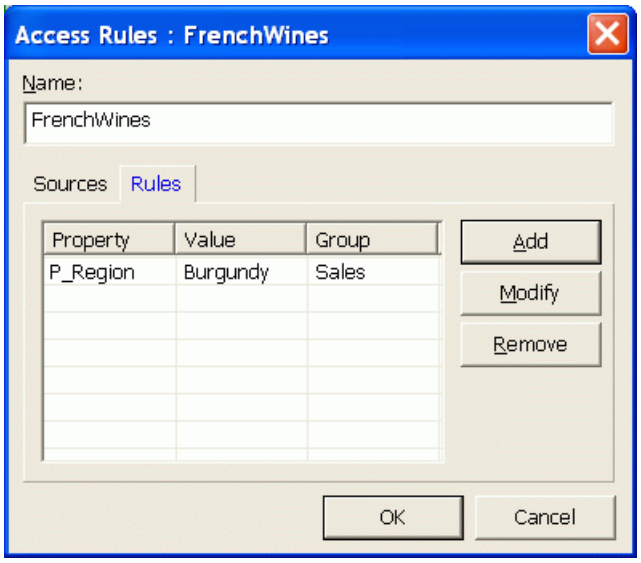


4. On the Rules tab, add an access rule by clicking the **Add** button.
5. In the Edit Access Rule editor, define the rule by specifying an If-Then statement:
   a) Specify the If condition clause by selecting a property (to identify records for access) and then entering a string in the Equals text box. The string is the value of the property against which the rule will be tested. Note that the string must be an exact match with the value of the property on the record.
   b) In the Then panel, enter the name of a group. This group is allowed access to any records that have the property and value that you specified in the Then clause.

   In this example, if the value of the P_Region property on a record equals the string value "Burgundy", then members of the Users group can read the record:

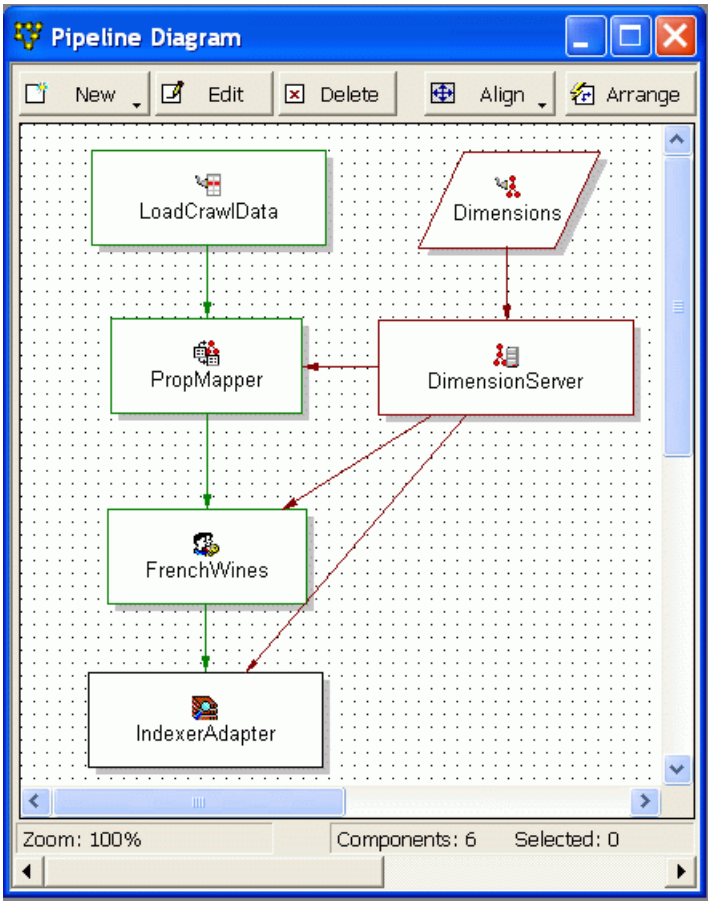


6. When you finished adding the rules, click **OK**. The Rules tab should look like this example:



7. To add additional access rules, repeat steps 4 through 6.
8. Click **OK** to add the Access Rules component.

The Pipeline Diagram should look like this example, which has an Access Rules component named FrenchWines:



Making MDEX Engine queries

After you have configured the LoginModule plug-in and have extracted the document ACLs, you can make MDEX Engine queries.

MDEX Engine URL query parameters

There are no URL parameters that are relevant to the Endeca Access Control System. All access control information is managed by the `AuthHttpENEConnection` object via the user's entitlement filter and cannot be modified from the URL. This prevents the Access Control System from being broken by attackers modifying URLs.

Access Control objects and method calls

To use the Access Control System, an `AuthHttpENEConnection` instance must be used to connect to the MDEX Engine. (If an `HttpENEConnection` is used, it will not restrict user access, regardless of whether ACLs have been defined for the MDEX Engine's records.) Use the `AuthHttpENEConnection` methods to ensure that all interfaces accessing sensitive information are protected by the Access Control System.

The `AuthHttpENEConnection` class is identical in use to the `HttpENEConnection` class, except that it has these additional methods:

- The `login()` method in Java and the `Login()` method in .NET.

- The `logout()` method in Java and the `Logout()` method in .NET.

After constructing the `ENEConnection` and setting the hostname and port as usual, the appropriate login method must be called to authenticate the user and gather authorization information.

Performance impact

There is some overhead for the initial authentication process for a user. The length of this delay depends upon the LoginModule plug-in that is used and on any other systems accessed by this module (for example, if a heavily loaded LDAP system is used for authentication via the `LDAPLoginModule`, a substantial login cost might be introduced).

The `Endeca.ACL.Allow.Read` property introduces a small index space (RAM) overhead for each record.

Response times for the MDEX Engine requests that include Access Control are relatively unaffected.

Chapter 6

User Authentication with LDAP

This section explains how to authenticate users via the Endeca `LDAPLoginModule` plug-in.

Overview of the LDAP user authentication process

The `LDAPLoginModule` plug-in handles logins that are authenticated against an LDAP directory.

To ensure a high level of security, this section assumes that X.509 certificates are required from users to identify them.

The procedures below use SSL to illustrate how you can combine the LDAP authentication and SSL features. Keep in mind, however, that using SSL during LDAP-based user authentication is completely independent from the user authentication procedure itself. Of course, you have to use SSL if the `useSSL` parameter has been set in the Access Control configuration file.

The general flow of the user authentication process via an LDAP directory is as follows:

1. Obtain the User Information: The user establishes connection to the application server and gives an X.509 certificate to the server. The application extracts the user identity from the user's X.509 certificate. (Note that JSP front-ends can use the Java `X509Certificate` class methods.) Up to this point, no Endeca software has been used.
2. Instantiate an MDEX Engine Connection Object: The Presentation API `AuthHttpENEConnection` constructor is used to instantiate an `AuthHttpENEConnection` object that will be used to connect to the MDEX Engine.
3. Query the LDAP Server: The `AuthHttpENEConnection` class has a login method that is used to connect to and query an external LDAP server to authenticate the user. If the LDAP directory has so been configured, the results can also provide the user's group information. The Endeca Access Control System automatically creates an entitlement filter for the user based on this group information.
4. Make a Secure MDEX Engine Query: The Presentation API `AuthHttpENEConnection` class has a query method that is used to make a query to the MDEX Engine that limits the user's access to what is specified in the entitlement filter.

These steps are described in detail in the following sections.

Note: User authentication via an LDAP directory is supported by the Java and .NET versions of the Endeca Presentation API. All procedures apply to both versions, unless otherwise noted.

Obtaining the user information

The first step is for the application to extract the user identity from the user's X.509 certificate.

Requiring X.509 certificates from your users is one way to provide secure user authentication. Although using X.509 certificates is not a requirement for any type of authentication, it is a often used in Web browsers that support the SSL protocol.

Java implementation

To extract the contents of the certificate, use the methods in the Java Certificate API, which is in the `java.security.cert` package available from Sun. In particular, the `X509Certificate` class provides a standard way to access all the attributes of an X.509 certificate.

The following JSP code fragment shows how to extract the user's name from an X.509 certificate.

```
<%@ page import="java.security.cert.X509Certificate" %>
<%@ page import="java.security.Principal" %>
<%
// Later in the page...
// Get the client SSL certificates associated with the request
X509Certificate[] certs = (X509Certificate[])
request.getAttribute("javax.servlet.request.X509Certificate");
// Check that a certificate was obtained
if (certs.length < 1) {
  System.err.println("SSL not client authenticated");
  return;
}
// The base of the certificate chain contains the client's info
X509Certificate principalCert = certs[0];

// Get the Distinguished Name from the certificate
// Ex/ "E=joeuser@endeca.com, CN=joeuser, O=Endeca,
//     "L=Cambridge, S=MA, C=US"
Principal principal = principalCert.getSubjectDN();

// Extract the common name (CN)
int start = principal.getName().indexOf("CN");
String tmpName, name = "";
if (start > 0) {
  tmpName = principal.getName().substring(start+3);
  int end = tmpName.indexOf(",");
  if (end > 0) {
    name = tmpName.substring(0, end);
  }
  else {
    name = tmpName;
  }
}
// Now query the LDAP server for authentication
...
%>
```

.NET implementation

The ASPX front end can also extract user information from X.509 certificates. The .NET Framework includes the `System.Security.Cryptography.X509Certificates` namespace that contains the `X509Certificate` class. For details on its usage, refer to the Microsoft .NET Framework documentation.

Instantiating an MDEX Engine connection object

An `AuthHttpENEConnection` object is used to connect to the MDEX Engine.

An `AuthHttpENEConnection` connection functions as a repository for the hostname and port configuration for the MDEX Engine you want to query. The class methods are briefly described in the following sections. For more information on the methods, see the *Endeca API Javadocs* or the *Endeca API Guide for .NET*.

Java implementation

The signature for an `AuthHttpENEConnection` constructor looks like this:

```
//Instantiate a connection object for the MDEX Engine
AuthHttpENEConnection nec = new AuthHttpENEConnection(emeHost, emePort);
```

In the instantiation, *emeHost* is the host name or IP address of the Endeca MDEX Engine and *emePort* is its port number.

Use the `AuthHttpENEConnection.enableSSL()` method if you want to enable SSL for the connection:

```
//Instantiate a connection object for the MDEX Engine
AuthHttpENEConnection nec = new AuthHttpENEConnection(emeHost, emePort);
// Enable the SSL connection with our NullHostnameVerifier class
nec.setHostnameVerifier(new NullHostnameVerifier());
nec.enableSSL();
```

Note that at this time, an actual connection has not been opened to the MDEX Engine.

.NET implementation

For .NET, the ASPX code to instantiate an `AuthHttpENEConnection` object looks like this:

```
//Instantiate a connection object for the MDEX Engine
AuthHttpENEConnection nec = new AuthHttpENEConnection(emeHost, emePort);
```

In the instantiation, *emeHost* is the host name or IP address of the Endeca MDEX Engine and *emePort* is its port number.

If you want to enable SSL for the connection for a .NET application, use the `AuthHttpENEConnection.EnableSSL()` method:

```
// Set the MDEX Engine connection
AuthHttpENEConnection nec = new AuthHttpENEConnection(ENEHost, ENEPort);
// Create the X509 certificate from the DER version
X509Certificate privateCert =
   X509Certificate.CreateFromCertFile(@"C:\Endeca\MyCerts\eneCert.der");
// Enable SSL for the connection, using the new X509 certificate.
nec.EnableSSL(privateCert);
// Now update the certificate validation with a custom policy.
// Required because Endeca certificates throw Host Not Found exceptions.
ServicePointManager.CertificatePolicy =
   new AcceptAllCertificatePolicy(privateCert);
```

At this time, an actual connection has not been opened to the MDEX Engine.

Querying the LDAP server

The `AuthHttpENEConnection` class has a login method to query the LDAP directory for authentication.

Use this `AuthHttpENEConnection` method to establish a connection with an external LDAP server and query the LDAP directory to authenticate the user:

- Java: the `login()` method
- .NET: the `Login()` method

These methods use the `LDAPLoginModule` plug-in to connect to and query the LDAP server.

The LDAP server's URL is obtained from the `serverInfo` parameter (or parameters) in the Access Control configuration file.

If the user is not in the LDAP directory, the authentication will fail and the user will not be allowed access to the Endeca implementation.

Java implementation

If the user's name and password have been extracted from a certificate or obtained by prompting the user, the call to the LDAP server would be:

```
//Create a CallbackHandler
CallbackHandler cbh = new StaticCallbackHandler(name,pass);
// Query the LDAP server to authenticate this user
try {
   nec.login(cbh);
} catch (ENEAuthenticationException exp) {
   System.err.println(exp);
   exp.printStackTrace();
   System.err.println("LDAP Authentication failed");
   System.exit(1);
}
```

The `StaticCallbackHandler` object provides the user name and password. If the login attempt fails, an `ENEAuthenticationException` exception is thrown.

.NET implementation

The .NET version of the Login method has this signature:

```
Login(IAuthCredentialRequestHandler handler)
```

where *handler* is an `IAuthCredentialRequestHandler` object instance that provides the mechanism to obtain credentials for authentication purposes.

To use an example, if the user has supplied a username and password, the call to authenticate the user would be:

```
//Authenticate the user
nec.Login(new StaticCredentialRequestHandler(user, pass));
```

The `StaticCredentialRequestHandler` object provides the user name and password. If the login attempt fails, an ENEAuthenticationException exception is thrown.

User entitlement filter

The Endeca Access Control System automatically creates an entitlement filter for a user.

A user's name and group membership are used to define the access rights for that user in an Endeca implementation. After an initial query for the user's entry in the LDAP directory, the Endeca LDAP plug-in queries the directory again for the user's group information.

The group information is automatically transformed into a user entitlement filter that defines the user's access rights to the data in the Endeca MDEX Engine. This entitlement filter is essentially a Boolean expression describing the user's rights.

The user entitlement filter is stored in the `AuthHttpENEConnection` object and therefore exists for the lifetime of this object. As the application services the user's requests, the entitlement filter is automatically added to every `AuthHttpENEConnection.query()` Java call made to the MDEX Engine (`AuthHttpENEConnection.Query()` for .NET).

The MDEX Engine uses the filter to restrict its results to only those to which the user has rights to read. To do this, it uses the `Endeca.ACL.Allow.Read` properties that were previously tagged onto the Endeca records by Forge.

Making a secure MDEX Engine query

The Presentation API `AuthHttpENEConnection` class methods are used to for secure queries to the MDEX Engine.

The *Endeca Basic Development Guide* describes how to create an MDEX Engine query by using the `ENEQuery` class and its `UrlENEQuery` subclass. It also describes how to execute the query with the Java `ENEConnection.query()` method (`ENEConnection.Query()` for .NET).

When you create an MDEX Engine query, you still use the `ENEQuery` or UrlENEQuery class methods. However, you use the Java `AuthHttpENEConnection.query()` method to make the query to the MDEX Engine (the .NET version is the `Query()` method).

What makes the query secure is the presence of the user entitlement filter, which limits the query results to records that the user is authorized to view.

The following examples show how to make the query to the MDEX Engine (the examples assume that the user has been successfully authenticated). Note that if you use an SSL connection between the Endeca components, the URL query parameter string will be encrypted as it is passed between the Presentation API and the MDEX Engine.

Java example

```
//Create a query from the browser request query string
ENEQuery usq = new UrlENEQuery(request.getQueryString(),"UTF-8");

// Set query so that only explicitly requested refinements
// are returned
usq.setNavAllRefinements(false);

// Make the query request to the MDEX Engine over
// the SSL connection
ENEQueryResults qresults = nec.query(usq);

//Use additional calls to process the query results
...
```

.NET example

```
//Create a query from the browser request query string
String queryString = Request.Url.Query.Substring(1);
ENEQuery usq = new UrlENEQuery(queryString, "UTF-8");

// Set query so only explicitly requested refinements are returned
usq.NavAllRefinements = false;

//Make the query request to the MDEX Engine
ENEQueryResults qresults = nec.Query(usq);

//Use additional calls to process the query results
...
```

Using stacked authentication

The Java version of the Endeca Access Control System also supports the notion of stacked authentication.

With stacked authentication, an Endeca implementation may be configured to use both the `LDAPLoginModule` and the `FileLoginModule` plug-ins.

How stacked authentication works

Stacking allows a subject to authenticate to multiple services at the same time, in cases where the integrated use of these services is necessary or desired.

For example, assume that an LDAP directory is used to store user names and passwords, but does not maintain information about a user's groups. The user group information is provided in a locally configurable password file (analogous to `/etc/passwd` on a UNIX system). This password file (which is used by the `FileLoginModule` plug-in) does not replace the LDAP authentication, but rather augments it. As the Presentation API constructs a user's corresponding Principal based on his LDAP entry, it will fill in any missing fields based on the values stored in the password file. Note that in this example, no passwords are needed in the `FileLoginModule` file because only group information is stored there.

The ability to use separate sources to gather the information for a user offers important flexibility. For example, it makes it easy to deploy an application to a small group of users supported by a single administrator within a larger organization that maintains a master LDAP directory. The administrator can easily support the application using the local password file to define group membership. Meanwhile, the central LDAP directory continues to be the basis for more fundamental user information.

Configuration file for stacking

To enable stacked authentication, you must put both LoginModule entries in the Access Control System configuration file. The order in which you place the entries is the order in which they are used for authentication.

The following is an example of a configuration file with two stacked LoginModules:

```
Endeca {
   //First use the LDAP module for user name authentication
   com.endeca.navigation.LDAPLoginModule required
   ldapBindAuthentication="false"
   serviceUsername="cn=Manager,dc=endeca,dc=com"
```

```
    servicePassword="nosecret"
    checkPasswords="false"
    groupTemplate="%{cn}"
    useSSL="true"
    serverInfo.0="ldap://web01.qa.endeca.com:1234"
    serverInfo.1="ldap://corona.dev.endeca.com:1234"
    serverInfo.2="ldap://web02.qa.endeca.com:1234"
    userPath="/ou=People,dc=endeca,dc=com??sub?(cn=%{#username:1})"
    groupPath="/ou=Groups,dc=endeca,dc=com?sub?(uniqueMember=%{#dn})";

    //Now get the group info from the password file
    com.endeca.navigation.FileLoginModule required
    passwordFile="c:/Endeca/PlatformServices/workspace/etc/passwd"
    checkPasswords="false";
};
```

Authenticating users

You authenticate users in a stacked configuration in the same way as you would if you were using the `LDAPLoginModule` plug-in alone. That is, you use only one `AuthHttpENEConnection.login()` method.

Related Links

File-based User Authentication on page 73

This section explains how to authenticate users via the Endeca `FileLoginModule` plug-in. This plug-in handles logins authenticated against a password file.

Chapter 7

File-based User Authentication

This section explains how to authenticate users via the Endeca `FileLoginModule` plug-in. This plug-in handles logins authenticated against a password file.

FileLoginModule configuration

To use file-based authentication, the Access Control configuration file must specify the `FileLogin¬Module` plug-in.

In addition, you must have set up the password file, which includes the names of all valid users, their passwords, and their groups.

Additional information for Java implementations

If you want to use stacked authentication for your implementation, you must put both the `LDAPLoginModule` and `FileLoginModule` plug-in entries in the JAAS configuration file.

Note that the application used as an example in this chapter is based on the JSP version of the Endeca reference implementation. Tomcat is used as the application server, with the JSP implementation being located in the Tomcat webapps directory.

File-based user authentication process

The `FileLoginModule` plug-in handles logins that are authenticated against a password file.

The general flow of the user authentication process via a password file is as follows:

1. Obtain the user information: The user establishes a connection to the application server and supplies his or her user identity.
2. Instantiate an MDEX Engine connection object: The Presentation API `AuthHttpENEConnection` constructor is used to instantiate an `AuthHttpENEConnection` object that will be used to connect to the MDEX Engine.
3. Authenticate the user against the password file: The `AuthHttpENEConnection` class has a login method that is used to read the password file to authenticate the user. The Endeca Access Control System automatically creates an entitlement filter for the user based on this information.
4. Make a Secure MDEX Engine Query: The Presentation API `AuthHttpENEConnection` class has a query method that is used to make a query to the MDEX Engine that limits the user's access to what is specified in the entitlement filter.

These steps are described in detail in the following sections.

Note: User authentication via a password file is supported by the Java and .NET versions of the Endeca Presentation API. All procedures apply to both versions, unless otherwise noted.

Obtaining the user identity

To use the `FileLoginModule` plug-in, your front-end application must obtain a user name from the person using the Endeca implementation.

This user information can be supplied in a number of ways, such as:

- Requiring X.509 certificates from your users. The contents of the certificate are extracted by using the `java.security.cert` methods (for Java implementations) or the `System.Security.Cryptography.X509Certificates` methods (for .NET implementations).
- Having the user enter the name in an HTML form.

The supplied name will be used to authenticate the user against the password file.

Instantiating an MDEX Engine connection object

The Presentation API `AuthHttpENEConnection` class is used to instantiate connection objects to the Endeca MDEX Engine.

An `AuthHttpENEConnection` connection functions as a repository for the hostname and port configuration for the MDEX Engine you want to query. The class methods are briefly described in the following sections. For more information on the methods, see the *Endeca API Javadocs* or the *Endeca API Guide for .NET*.

Java implementation

The signature for an `AuthHttpENEConnection` constructor looks like this:

```
//Instantiate a connection object for the MDEX Engine
AuthHttpENEConnection nec = new AuthHttpENEConnection(emeHost, emePort);
```

In the instantiation, *emeHost* is the host name or IP address of the Endeca MDEX Engine and *emePort* is its port number.

.NET implementation

For .NET, the ASPX code to instantiate an `AuthHttpENEConnection` object looks like this:

```
//Instantiate a connection object for the MDEX Engine
AuthHttpENEConnection nec = new AuthHttpENEConnection(emeHost, emePort);
```

In the instantiation, *emeHost* is the host name or IP address of the Endeca MDEX Engine and *emePort* is its port number.

Authenticating the user against the password file

The `AuthHttpENEConnection` class has a login method to read the password file for authentication.

Use this `AuthHttpENEConnection` method to authenticate the user against the password file:

- Java: the `login()` method
- .NET: the `Login()` method

These methods use the `FileLoginModule` plug-in to locate the password file (as specified by the `passwordFile` parameter in the Access Control configuration file) and read its contents.

If the user is not in the password file, the authentication will fail (if the `FileLoginModule` is marked as `required` in the Access Control configuration file) and the user will not be allowed access to the Endeca implementation.

Java implementation

If the user has supplied a username and password, an example of the call to authenticate the user would be:

```
// Authenticate the user via a local password file.
nec.login(new StaticCallbackHandler(name, password));
```

The `StaticCallbackHandler` object provides the user name and password. If the login attempt fails, an `ENEAuthenticationException` exception is thrown.

.NET implementation

The .NET version of the Login method has this signature:

```
Login(IAuthCredentialRequestHandler handler)
```

where *handler* is an `IAuthCredentialRequestHandler` object instance that provides the mechanism to obtain credentials for authentication purposes.

To use an example, if the user has supplied a username and password, the call to authenticate the user would be:

```
//Authenticate the user
nec.Login(new StaticCredentialRequestHandler(user, pass));
```

The `StaticCredentialRequestHandler` object provides the user name and password. If the login attempt fails, an ENEAuthenticationException exception is thrown.

User entitlement filter

The Endeca Access Control System automatically creates an entitlement filter for a user.

The `FileLoginModule` gets a user's group information from the local password file and automatically uses it to create a user entitlement filter. This filter defines the user's access rights to the data in the Endeca MDEX Engine.

The user entitlement filter is stored in the `AuthHttpENEConnection` object and therefore exists for the lifetime of this object. As the application services the user's requests, the entitlement filter is automatically added to every `AuthHttpENEConnection.query()` Java call made to the MDEX Engine (`AuthHttpENEConnection.Query()` for .NET).

The MDEX Engine uses the filter to restrict its results to only those to which the user has rights to read. To do this, it uses the `Endeca.ACL.Allow.Read` properties that were previously tagged onto the Endeca records by Forge.

Making a secure MDEX Engine query

The Presentation API `AuthHttpENEConnection` class methods are used to for secure queries to the MDEX Engine.

The *Endeca Basic Development Guide* describes how to create an MDEX Engine query by using the `ENEQuery` class and its `UrlENEQuery` subclass. It also describes how to execute the query with the Java `ENEConnection.query()` method (`ENEConnection.Query()` for .NET).

When you create an MDEX Engine query, you still use the `ENEQuery` or UrlENEQuery class methods. However, you use the Java `AuthHttpENEConnection.query()` method to make the query to the MDEX Engine (the .NET version is the `Query()` method).

What makes the query secure is the presence of the user entitlement filter, which limits the query results to records that the user is authorized to view.

The following examples show how to make the query to the MDEX Engine (the examples assume that the user has been successfully authenticated). Note that if you use an SSL connection between the Endeca components, the URL query parameter string will be encrypted as it is passed between the Presentation API and the MDEX Engine.

Java example

```
//Create a query from the browser request query string
ENEQuery usq = new UrlENEQuery(request.getQueryString(),"UTF-8");

// Set query so that only explicitly requested refinements
// are returned
usq.setNavAllRefinements(false);

// Make the query request to the MDEX Engine over
// the SSL connection
ENEQueryResults qresults = nec.query(usq);

//Use additional calls to process the query results
...
```

.NET example

```
//Create a query from the browser request query string
String queryString = Request.Url.Query.Substring(1);
ENEQuery usq = new UrlENEQuery(queryString, "UTF-8");

// Set query so only explicitly requested refinements are returned
usq.NavAllRefinements = false;

//Make the query request to the MDEX Engine
ENEQueryResults qresults = nec.Query(usq);

//Use additional calls to process the query results
...
```

Index

G

H

I

J

K

L

M

P

R

S

T

U

X