Red Hat JBoss Enterprise Application Platform 7.3

How to Configure Identity Management

Instructions for managing user access to Red Hat JBoss Enterprise Application Platform using LDAP directories and other identity stores.

Last Updated: 2022-02-01

Red Hat JBoss Enterprise Application Platform 7.3 How to Configure Identity Management

Instructions for managing user access to Red Hat JBoss Enterprise Application Platform using LDAP directories and other identity stores.

Legal Notice

Abstract

This guide explores how to use LDAP directories and other identity stores for use with JBoss EAP management interfaces and security domains. This guide expands on the concepts provided in the JBoss EAP Security Architecture guide, and should be reviewed after administrators have basic knowledge of LDAP and a solid understanding of security concepts within JBoss EAP.

Table of Contents

CHAPTER 1. IDENTITY MANAGEMENT OVERVIEW

The basic identity management concepts for securing applications with various identity stores are covered in the Red Hat JBoss Enterprise Application Platform (JBoss EAP) *Security Architecture* guide. This guide shows you how to configure various identity stores, such as a filesystem or LDAP, to secure applications. In some cases you can also use certain identity stores, such as LDAP, as an authorization authority. Various role and access information about principals can be stored in an LDAP directory which can then be used directly by JBoss EAP or mapped to existing JBoss EAP roles.

NOTE

Using identity stores backed by external datastores, such as databases or LDAP directories, can have a performance impact on authentication and authorization due to the data access and transport between the external datastore and the JBoss EAP instance.

CHAPTER 2. ELYTRON SUBSYSTEM

2.1. CONFIGURE AUTHENTICATION WITH A FILESYSTEM-BASED IDENTITY STORE

1. Configure a **filesystem-realm** in JBoss EAP:

```
/subsystem=elytron/filesystem-realm=exampleFsRealm:add(path=fs-realm-users,relative-to=jboss.server.config.dir)
```

   If your directory is located outside of **jboss.server.config.dir**, then you need to change the **path** and **relative-to** values appropriately.

2. Add a user:
   When using the **filesystem-realm**, you can add users using the management CLI.

```
/subsystem=elytron/filesystem-realm=exampleFsRealm:add-identity(identity=user1)
/subsystem=elytron/filesystem-realm=exampleFsRealm:set-password(identity=user1, clear={password="password123"})
/subsystem=elytron/filesystem-realm=exampleFsRealm:add-identity-attribute(identity=user1, name=Roles, value=["Admin","Guest"])
```

3. Add a **simple-role-decoder**:

```
/subsystem=elytron/simple-role-decoder=from-roles-attribute:add(attribute=Roles)
```

   This **simple-role-decoder** decodes a principal's roles from the **Roles** attribute. You can change this value if your roles are in a different attribute.

4. Configure a **security-domain**:

```
/subsystem=elytron/security-domain=exampleFsSD:add(realms=[{realm=exampleFsRealm,role-decoder=from-roles-attribute}],default-realm=exampleFsRealm,permission-mapper=default-permission-mapper)
```

5. Configure an **application-security-domain** in the **undertow** subsystem:

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(security-domain=exampleFsSD)
```

NOTE

An **application-security-domain** in the **undertow** subsystem can be configured using the management console by navigating to **Configuration → Subsystems → Web (Undertow) → Application Security Domain**.

6. Configure your application's **web.xml** and **jboss-web.xml**:
   Your application's **web.xml** and **jboss-web.xml** must be updated to use the **application-security-domain** you configured in JBoss EAP. An example of this is available in Configure Web Applications to Use Elytron or Legacy Security for Authentication.

Your application is now using a file system-based identity store for authentication.

2.2. CONFIGURE AUTHENTICATION WITH A PROPERTIES FILE-BASED IDENTITY STORE

1. Create properties files:
   You must create two properties files: one that maps users to passwords and another that maps users to roles. Usually, these files are located in the **jboss.server.config.dir** directory and follow the naming convention ***-users.properties** and ***-roles.properties**, but other locations and names can be used. The ***-users.properties** file must also contain a reference to the **properties-realm**, which you will create in the next step:
   #$REALM_NAME=*YOUR_PROPERTIES_REALM_NAME*$

   Example user to password file: example-users.properties

   ```
   #$REALM_NAME=examplePropRealm$
   user1=password123
   user2=password123
   ```

   Example user to roles file: example-roles.properties

   ```
   user1=Admin
   user2=Guest
   ```

2. Configure a **properties-realm** in JBoss EAP:

   ```
   /subsystem=elytron/properties-realm=examplePropRealm:add(groups-
   attribute=groups,groups-properties={path=example-roles.properties,relative-
   to=jboss.server.config.dir},users-properties={path=example-users.properties,relative-
   to=jboss.server.config.dir,plain-text=true})
   ```

   The name of the **properties-realm** is **examplePropRealm**, which is used in the previous step in the **example-users.properties** file. Also, if your properties files are located outside of **jboss.server.config.dir**, then you must change the **path** and **relative-to** values appropriately.

3. Configure a **security-domain**:

   ```
   /subsystem=elytron/security-domain=exampleSD:add(realms=
   [{realm=examplePropRealm,role-decoder=groups-to-roles}],default-
   realm=examplePropRealm,permission-mapper=default-permission-mapper)
   ```

4. Configure an **application-security-domain** in the **undertow** subsystem:

   ```
   /subsystem=undertow/application-security-domain=exampleApplicationDomain:add(security-
   domain=exampleSD)
   ```

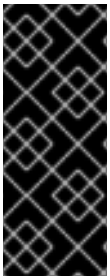
   NOTE

   An **application-security-domain** in the **undertow** subsystem can be configured using the management console by navigating to **Configuration → Subsystems → Web (Undertow) → Application Security Domain**.

5. Configure your application's **web.xml** and **jboss-web.xml**:

Your application's **web.xml** and **jboss-web.xml** must be updated to use the **application-security-domain** you configured in JBoss EAP. An example of this is available in Configure Web Applications to Use Elytron or Legacy Security for Authentication.

Your application is now using a properties file-based identity store for authentication.



IMPORTANT

The properties files are only read when the server starts. Any users added after server startup, either manually or by using an **add-user** script, requires a server reload. This reload is accomplished by running the **reload** command from the management CLI.

```
reload
```

2.3. CONFIGURE AUTHENTICATION WITH A DATABASE-BASED IDENTITY STORE

1. Determine your database format for usernames, passwords, and roles:
   To set up authentication using a database for an identity store, you need to determine how your usernames, passwords, and roles are stored in that database. In this example, we are using a single table with the following sample data:

username	password	roles
user1	password123	Admin
user2	password123	Guest

2. Configure a datasource:
   To connect to a database from JBoss EAP, you must have the appropriate database driver deployed, as well as a datasource configured. This example shows deploying the driver for PostgreSQL and configuring a datasource in JBoss EAP:

```
deploy /path/to/postgresql-9.4.1210.jar

data-source add --name=examplePostgresDS --jndi-name=java:jboss/examplePostgresDS --driver-name=postgresql-9.4.1210.jar  --connection-url=jdbc:postgresql://localhost:5432/postgresdb --user-name=postgresAdmin --password=mysecretpassword
```

3. Configure a **jdbc-realm** in JBoss EAP:

```
/subsystem=elytron/jdbc-realm=exampleDbRealm:add(principal-query=[{sql="SELECT password,roles FROM eap_users WHERE username=?",data-source=examplePostgresDS,clear-password-mapper={password-index=1},attribute-mapping=[{index=2,to=groups}]}])
```

NOTE

The above example shows how to obtain passwords and roles from a single **principal-query**. You can also create additional **principal-query** with **attribute-mapping** attributes if you require multiple queries to obtain roles or additional authentication or authorization information.

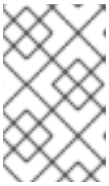
For a list of supported password mappers, see Password Mappers.

4. Configure a **security-domain**:

```
/subsystem=elytron/security-domain=exampleDbSD:add(realms=
[{realm=exampleDbRealm,role-decoder=groups-to-roles}],default-
realm=exampleDbRealm,permission-mapper=default-permission-mapper)
```

5. Configure an **application-security-domain** in the **undertow** subsystem:

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(security-
domain=exampleDbSD)
```



NOTE

An **application-security-domain** in the **undertow** subsystem can be configured using the management console by navigating to **Configuration → Subsystems → Web (Undertow) → Application Security Domain**.

6. Configure your application's **web.xml** and **jboss-web.xml**:
Your application's **web.xml** and **jboss-web.xml** must be updated to use the **application-security-domain** you configured in JBoss EAP. An example of this is available in Configure Web Applications to Use Elytron or Legacy Security for Authentication.

2.4. CONFIGURE AUTHENTICATION WITH AN LDAP-BASED IDENTITY STORE

1. Determine your LDAP format for usernames, passwords, and roles:
To set up authentication using an LDAP server for an identity store, you need to determine how your usernames, passwords, and roles are stored. In this example, we are using the following structure:

```
dn: dc=wildfly,dc=org
dc: wildfly
objectClass: top
objectClass: domain

dn: ou=Users,dc=wildfly,dc=org
objectClass: organizationalUnit
objectClass: top
ou: Users

dn: uid=jsmith,ou=Users,dc=wildfly,dc=org
objectClass: top
objectClass: person
objectClass: inetOrgPerson
```

```
cn: John Smith
sn: smith
uid: jsmith
userPassword: password123

dn: ou=Roles,dc=wildfly,dc=org
objectclass: top
objectclass: organizationalUnit
ou: Roles

dn: cn=Admin,ou=Roles,dc=wildfly,dc=org
objectClass: top
objectClass: groupOfNames
cn: Admin
member: uid=jsmith,ou=Users,dc=wildfly,dc=org
```

2. Configure a **dir-context**:
   To connect to the LDAP server from JBoss EAP, you need to configure a **dir-context** that provides the URL as well as the principal used to connect to the server.

```
/subsystem=elytron/dir-context=exampleDC:add(url="ldap://127.0.0.1:10389",principal="uid=admin,ou=system",credential-reference={clear-text="secret"})
```

NOTE

It is not possible to use a JMX **ObjectName** to decrypt the LDAP credentials. Instead, credentials can be secured by using a Credential Store as discussed in *How to Configure Server Security* for JBoss EAP.

3. Configure an **ldap-realm** in JBoss EAP:

```
/subsystem=elytron/ldap-realm=exampleLR:add(dir-context=exampleDC,identity-mapping={search-base-dn="ou=Users,dc=wildfly,dc=org",rdn-identifier="uid",user-password-mapper={from="userPassword"},attribute-mapping=[{filter-base-dn="ou=Roles,dc=wildfly,dc=org",filter="(&(objectClass=groupOfNames)(member={0}))",from="cn",to="Roles"}]})
```

WARNING

If any referenced LDAP servers contain a loop in referrals, it can result in a **java.lang.OutOfMemoryError** error on the JBoss EAP server.

4. Add a **simple-role-decoder**:

```
/subsystem=elytron/simple-role-decoder=from-roles-attribute:add(attribute=Roles)
```

5. Configure a **security-domain**:

```
/subsystem=elytron/security-domain=exampleLdapSD:add(realms=[{realm=exampleLR,role-decoder=from-roles-attribute}],default-realm=exampleLR,permission-mapper=default-permission-mapper)
```

6. Configure an **application-security-domain** in the **undertow** subsystem:

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(security-domain=exampleLdapSD)
```

NOTE

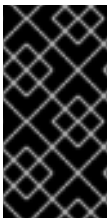
An **application-security-domain** in the **undertow** subsystem can be configured using the management console by navigating to **Configuration → Subsystems → Web (Undertow) → Application Security Domain**.

7. Configure your application's **web.xml** and **jboss-web.xml**:
Your application's **web.xml** and **jboss-web.xml** must be updated to use the **application-security-domain** you configured in JBoss EAP. An example of this is available in Configure Web Applications to Use Elytron or Legacy Security for Authentication.

IMPORTANT

In cases where the **elytron** subsystem uses an LDAP server to perform authentication, JBoss EAP will return a **500**, or internal server error, error code if that LDAP server is unreachable. This behavior differs from previous versions of JBoss EAP using the legacy **security** subsystem, which returned a **401**, or unauthorized, error code under the same conditions.

2.5. CONFIGURE AUTHENTICATION WITH CERTIFICATES



IMPORTANT

Before you can set up certificate-based authentication, you must have two-way SSL configured. More details on configuring two-way SSL can be found in the Enable Two-way SSL/TLS for Applications using the Elytron Subsystem section of the *How to Configure Server Security* guide.

1. Configure a **key-store-realm**.

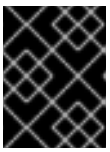
```
/subsystem=elytron/key-store-realm=ksRealm:add(key-store=twoWayTS)
```

You must configure this realm with a truststore that contains the client's certificate. The authentication process uses the same certificate presented by the client during the two-way SSL handshake.

2. Create a decoder.
You need to create a **x500-attribute-principal-decoder** to decode the principal you get from your certificate. The below example will decode the principal based on the first **CN** value.

```
/subsystem=elytron/x500-attribute-principal-decoder=CNDecoder:add(oid="2.5.4.3",maximum-segments=1)
```

For example, if the full **DN** was **CN=client,CN=client-certificate,DC=example,DC=jboss,DC=org**, **CNDecoder** would decode the principal as **client**. This decoded principal is used as the **alias** value to lookup a certificate in the truststore configured in **ksRealm**.



IMPORTANT

The decoded principal **MUST** be the **alias** value you set in your server's truststore for the client's certificate.

- Optionally, you can configure an evidence decoder using a subject alternative name extension to use a subject alternative name as the principal. For more information, see Configuring Evidence Decoder for X.509 Certificate with Subject Alternative Name Extension in the *How to Configure Server Security* guide.

3. Add a **constant-role-mapper** for assigning roles.
This is example uses a **constant-role-mapper** to assign roles to a principal from **ksRealm**, but you can also use other approaches.

```
/subsystem=elytron/constant-role-mapper=constantClientCertRole:add(roles=[Admin,Guest])
```

4. Configure a **security-domain**.

```
/subsystem=elytron/security-domain=exampleCertSD:add(realms=[{realm=ksRealm}],default-realm=ksRealm,permission-mapper=default-permission-mapper,principal-decoder=CNDecoder,role-mapper=constantClientCertRole)
```

5. Configure an **application-security-domain** in the **undertow** subsystem.

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:add(security-domain=exampleCertSD)
```

NOTE

An **application-security-domain** in the **undertow** subsystem can be configured using the management console by navigating to **Configuration → Subsystems → Web (Undertow) → Application Security Domain**.

6. Update **server-ssl-context**.

```
/subsystem=elytron/server-ssl-context=twoWaySSC:write-attribute(name=security-domain,value=exampleCertSD)
/subsystem=elytron/server-ssl-context=twoWaySSC:write-attribute(name=authentication-optional, value=true)
reload
```

7. Configure your application's **web.xml** and **jboss-web.xml**.
Your application's **web.xml** and **jboss-web.xml** must be updated to use the **application-security-domain** you configured in JBoss EAP. An example of this is available in Configure Web Applications to Use Elytron or Legacy Security for Authentication.

In addition, you need to update your **web.xml** to use **CLIENT-CERT** as its authentication method.

```
<login-config>
  <auth-method>CLIENT-CERT</auth-method>
  <realm-name>exampleApplicationDomain</realm-name>
</login-config>
```

2.6. CONFIGURE AUTHENTICATION AND AUTHORIZATION USING MULTIPLE IDENTITY STORES

If you store attributes of an identity across different identity stores, then use an **aggregate-realm** to load the identity attributes into a single security realm for authentication and authorization.

2.6.1. Aggregate Realm in Elytron

With an **aggregate-realm**, you can use one security realm for authentication and another security realm, or an aggregation of multiple security realms, for authorization in Elytron. For example, you can configure an aggregate realm to use a properties realm for authentication and a JDBC realm for authorization.

In an aggregate realm configured to aggregate multiple authorization realms, an identity is created as follows:

- Attribute values from each security realm configured for authorization are loaded.
- If an attribute is defined in more than one authorization realm, the value of the first occurrence of the attribute is used.

The following example illustrates how an identity is created when multiple authorization realms contain definitions for the same identity attribute.

Example

Aggregate realm configuration:

```
authentication-realm=properties-realm,
authorization-realms=[jdbc-realm,ldap-realm]
```

- Attribute values obtained from the JDBC realm:

```
e-mail: user@example.com
groups: Supervisor, User
```

- Attribute values obtained from the ldap realm:

```
e-mail: administrator@example.com
phone: 0000 0000 0000
```

Resulting identity obtained from the aggregate realm:

```
e-mail: user@example.com
groups: Supervisor, User
phone: 0000 0000 0000
```

In the example, the attribute **e-mail** is defined in both the authorization realms. The value defined in

JDBC realm gets used for the attribute **e-mail** in the resulting aggregate realm because the aggregate realm was configured to aggregate the authorization realms as: **authorization-realms=[jdbc-realm,ldap-realm]**.

2.6.2. Configuring Authentication and Authorization Using an Aggregate Realm

To configure authentication and authorization using an aggregate realm, create an aggregate realm, and configure a security domain and an application security domain to use the aggregate realm.

Prerequisites

- The security realms to be aggregated are configured.
  For information about configuring security realms, see Elytron Subsystem in the *How to Configure Identity Management* guide.
- A role decoder to be used in the security domain is configured.
  For information about role decoders, see Create an Elytron Role Decoder in the *How to Configure Server Security* guide.

Procedure

1. Create an aggregate realm:
   - To create an aggregate realm with one authorization realm:

     ```
     /subsystem=elytron/aggregate-realm=exampleAggregateRealm:add(authentication-realm=__SECURITY_REALM_FOR_AUTHENTICATION__, authorization-realm=__SECURITY_REALM_FOR_AUTHORIZATION__)
     ```

   - To create an aggregate realm with multiple authorization realms:

     ```
     /subsystem=elytron/aggregate-realm=exampleAggregateRealm:add(authentication-realm=__SECURITY_REALM_FOR_AUTHENTICATION__, authorization-realms=[__SECURITY_REALM_FOR_AUTHORIZATION_1__,__SECURITY_REALM_FOR_AUTHORIZATION_2__,...,__SECURITY_REALM_FOR_AUTHORIZATION_N__])
     ```

2. Configure a **security-domain**:

   ```
   /subsystem=elytron/security-domain=exampleAggregateRealmSD:add(realms=[{realm=exampleAggregateRealm,role-decoder=__ROLE-DECODER__}],default-realm=exampleAggregateRealm,permission-mapper=default-permission-mapper)
   ```

3. Configure an **application-security-domain** in the **undertow** subsystem:

   ```
   /subsystem=undertow/application-security-domain=exampleAggregareRealmApplicationDomain:add(security-domain=exampleAggregateRealmSD)
   ```

4. Configure your application's **web.xml** and **jboss-web.xml**:
   Your application's **web.xml** and **jboss-web.xml** must be updated to use the **application-security-domain** you configured in JBoss EAP. An example of this is available in Configure Web Applications to Use Elytron or Legacy Security for Authentication.

2.6.3. Example Aggregate Realms

Example aggregate realm with a single authorizarion realm

In this example, a **properties-realm** is used for authentication and a **jdbc-realm** is used for authorization.

You must preconfigure the following realms:

- **properties-realm** named examplPropertiesRealm
- **jdbc-realm** named exampleJdbcRealm

Issuing the following command creates an aggregate realm:

```
/subsystem=elytron/aggregate-realm:exampleSimpleAggregateRealm:add(authentication-realm=examplPropertiesRealm,authorization-realm=exampleJdbcRealm)
```

Example aggregate realm with two authorization realms

In this example, **properties-realm** is used for authentication and an aggregation of **ldap-realm** and **jdbc-realm** is used for authorization.

You must preconfigure the following realms:

- **properties-realm** named examplPropertiesRealm
- **jdbc-realm** named exampleJdbcRealm
- **ldap-realm** named exampleLdapRealm

Issuing the following command creates an aggregate realm:

```
/subsystem=elytron/aggregate-realm:exampleSimpleAggregateRealm:add(authentication-realm=examplPropertiesRealm,authorization-realms=[exampleJdbcRealm,exampleLdapRealm])
```

2.7. OVERRIDE AN APPLICATION'S AUTHENTICATION CONFIGURATION

You can override the authentication configuration of an application with one configured in JBoss EAP. To do this, use the **override-deployment-configuration** property in the **application-security-domain** section of the **undertow** subsystem:

```
/subsystem=undertow/application-security-domain=exampleApplicationDomain:write-attribute(name=override-deployment-config,value=true)
```

NOTE

An **application-security-domain** in the **undertow** subsystem can be configured using the management console by navigating to **Configuration → Subsystems → Web (Undertow) → Application Security Domain**.

For example, an application is configured to use **FORM** authentication with the **exampleApplicationDomain** in its **jboss-web.xml**.

Example jboss-web.xml

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>exampleApplicationDomain</realm-name>
</login-config>
```

By enabling **override-deployment-configuration**, you can create a new **http-authentication-factory** that specifies a different authentication mechanism, such as **BASIC** or **DIGEST**.

Example http-authentication-factory

```
/subsystem=elytron/http-authentication-factory=exampleHttpAuth:read-resource()
{
    "outcome" => "success",
    "result" => {
        "http-server-mechanism-factory" => "global",
        "mechanism-configurations" => [{
            "mechanism-name" => "BASIC",
            "mechanism-realm-configurations" => [{"realm-name" => "exampleApplicationDomain"}]
        }],
        "security-domain" => "exampleSD"
    }
}
```

This will override the authentication mechanism defined in the application's **jboss-web.xml** and attempt to authenticate a user using **BASIC** instead of **FORM**.

2.8. SET UP CACHING FOR SECURITY REALMS

Elytron provides a **caching-realm** which allows you to cache the results of a credential lookup from a security realm. For example, you could use this to configure a cache for credentials coming from LDAP or a database to increase performance for frequently queried users.

The **caching-realm** caches the **PasswordCredential** credential using a *LRU* or *Least Recently Used* caching strategy, in which the least accessed entries are discarded when maximum number of entries is reached.

You can use a **caching-realm** with the following security realms:

- **filesystem-realm**
- **jdbc-realm**
- **ldap-realm**
- a custom security realm

If you make changes to your credential source outside of JBoss EAP, those changes are only propagated to a JBoss EAP caching realm if the underlying security realm supports listening. In particular, an **ldap-realm** supports listening, however filtered attributes, such as **roles**, inside the **ldap-realm** do not.

To ensure that your caching realm has a correct cache of user data, it is recommended that you modify your user attributes through the caching realm rather than at your credential source. Alternatively, you can clear the cache.

IMPORTANT

Making user changes through a caching realm is provided as Technology Preview only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs), might not be functionally complete, and Red Hat does not recommend to use them for production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

See Technology Preview Features Support Scope on the Red Hat Customer Portal for information about the support scope for Technology Preview features.

To configure and use a **caching-realm**:

1. Create an existing security realm.
   You need an existing security realm to use with a **caching-realm**. For example, you could create a **filesystem-realm** similar to the steps in Configure Authentication with a Filesystem-Based Identity Store.

   Example filesystem-realm

   ```
   /subsystem=elytron/filesystem-realm=exampleFsRealm:add(path=fs-realm-users, relative-to=jboss.server.config.dir)

   /subsystem=elytron/filesystem-realm=exampleFsRealm:add-identity(identity=user1)

   /subsystem=elytron/filesystem-realm=exampleFsRealm:set-password(identity=user1, clear={password="password123"})

   /subsystem=elytron/filesystem-realm=exampleFsRealm:add-identity-attribute(identity=user1,name=Roles,value=["Admin","Guest"])

   /subsystem=elytron/simple-role-decoder=from-roles-attribute:add(attribute=Roles)
   ```

2. Create a **caching-realm**.
   Once you have an existing realm you want to cache, create a **caching-realm** that references it.

   Example caching-realm that Uses exampleFsRealm

   ```
   /subsystem=elytron/caching-realm=exampleCacheRealm:add(realm=exampleFsRealm)
   ```

3. Use the **caching-realm**.
   After you create the **caching-realm**, you can then use it in your security configuration just as you would any other security realm. For example, you could use it in the same place you would use a **filesystem-realm** in Configure Authentication with a Filesystem-Based Identity Store .

   Example Configuration Using the caching-realm

   ```
   /subsystem=elytron/security-domain=exampleFsSD:add(realms=[{realm=exampleCacheRealm, role-decoder=from-roles-attribute}], default-
   ```

```
realm=exampleCacheRealm, permission-mapper=default-permission-mapper)

/subsystem=elytron/http-authentication-factory=example-fs-http-auth:add(http-server-
mechanism-factory=global, security-domain=exampleFsSD, mechanism-configurations=
[{mechanism-name=BASIC, mechanism-realm-configurations=[{realm-
name=exampleApplicationDomain}]}])
```

You can control the cache size as well as item expiration by using **maximum-entries** and **maximum-age** attributes of the **caching-realm**. For more details on those attributes, see the Elytron Subsystem Components Reference section in *How to Configure Server Security*.

Clear a caching-realm Cache

You can clear an existing cache by using the **clear-cache** command. Clearing a cache forces it to repopulate using the latest data from the security realm.

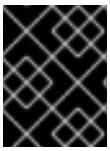
```
/subsystem=elytron/caching-realm=exampleCacheRealm:clear-cache
```

2.9. CONFIGURE APPLICATIONS TO USE CONTAINER-MANAGED SINGLE SIGN-ON

You can configure JBoss EAP to use container-managed single sign-on for applications using the Elytron **FORM** authentication method. This allows users to authenticate once and access other resources secured by the **FORM** authentication method without having to reauthenticate.

The related single sign-on session is invalidated when:

- there are no active local sessions left.
- logging out from an application.



IMPORTANT

You can use single sign-on across applications deployed on different JBoss EAP instances as long as these instances are in a cluster.

1. Create a **key-store**.
   A **key-store** is necessary in order to configure a secure communication channel between the different servers participating in the SSO. This channel is used to exchange messages about events that occur when single sign-on sessions are created or destroyed, during log in and log out respectively.

   To create a **key-store** in the **elytron** subsystem, first create a Java KeyStore as follows:

   ```
   keytool -genkeypair -alias localhost -keyalg RSA -keysize 1024 -validity 365 -keystore
   keystore.jks -dname "CN=localhost" -keypass secret -storepass secret
   ```

   Once the **keystore.jks** file is created, execute the following management CLI command to create a **key-store** definition in Elytron:

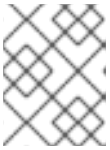
   ```
   /subsystem=elytron/key-store=example-keystore:add(path=keystore.jks, relative-
   to=jboss.server.config.dir, credential-reference={clear-text=secret}, type=JKS)
   ```

2. Add the security realm.
   Create a **FileSystem** realm, an identity store where users are stored in the local file system, using the following management CLI command:

```
/subsystem=elytron/filesystem-realm=example-realm:add(path=/tmp/example-realm)
```

3. Use the following management CLI command to create a **security-domain**:

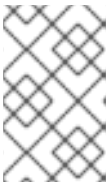
```
/subsystem=elytron/security-domain=example-domain:add(default-realm=example-realm,permission-mapper=default-permission-mapper,realms=[{realm=example-realm,role-decoder=groups-to-roles}]
```



NOTE

Applications using SSO should use **HTTP FORM** authentication as they usually need to provide a login page for the users.

4. Create an application security domain in the **undertow** subsystem.



NOTE

If you already have a **application-security-domain** defined in the **undertow** subsystem and just want to use it to enable single sign-on to your applications, you can skip this step.

```
/subsystem=undertow/application-security-domain=other:add(security-domain=example-domain)
```

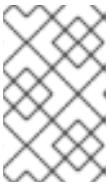
NOTE

By default, if your application does not define any specific security-domain in the **jboss-web.xml** file, the application server will choose one with a name **other**.

5. Update the **undertow** subsystem to enable single sign-on and use the keystore.
   Single sign-on is enabled to a specific **application-security-domain** definition in the **undertow** subsystem. It is important that the servers you are using to deploy the applications are using the same configuration.

   To enable single sign-on, just change an existing **application-security-domain** in the **undertow** subsystem as follows:

```
/subsystem=undertow/application-security-domain=other/setting=single-sign-on:add(key-store=example-keystore, key-alias=localhost, domain=localhost, credential-reference={clear-text=secret})
```



NOTE

An **application-security-domain** in the **undertow** subsystem can be configured using the management console by navigating to **Configuration → Subsystems → Web (Undertow) → Application Security Domain**.

For more information on the SSO attributes and their definitions, see Reference for Single Sign-on Attributes.

6. Configure your application's **web.xml** and **jboss-web.xml** files.
   Your application's **web.xml** and **jboss-web.xml** must be updated to use the **application-security-domain** you configured in JBoss EAP. An example of this is available in Configure Web Applications to Use Elytron or Legacy Security for Authentication.

JBoss EAP provides out-of-the-box support for clustered and non-clustered SSO using the **undertow** and **infinispan** subsystems.

2.10. CONFIGURE AUTHENTICATION AND AUTHORIZATION WITH BEARER TOKENS

2.10.1. Bearer token authentication

You can use **BEARER_TOKEN** authentication mechanism to authorize HTTP requests sent to your application. After a client, such as a web browser, sends an HTTP request to your application, the **BEARER_TOKEN** mechanism verifies the presence of a bearer token in the **Authorization** HTTP header of the request.

Elytron supports authentication by using bearer tokens in JWT format, such as OpenID Connect ID tokens, or by using opaque tokens issued by the OAuth2 compliant authorization server. See the additional resources section.

The following example shows that the **Authorization** HTTP header contains the **mF_9.B5f-4.1JqM** bearer token:

```
GET /resource HTTP/1.1
Host: server.example.com
Authorization: Bearer mF_9.B5f-4.1JqM
```

The **BEARER_TOKEN** mechanism can now extract the bearer token string and pass it to the **token-realm** implementation for validation. If the implementation successfully validates the bearer token, Elytron creates a security context based on the information represented by the token. The application can use this security context to obtain information about the requester. It can then decide whether to fulfill the request by providing the requester access to the HTTP resource.

The **BEARER_TOKEN** mechanism returns a **401 HTTP** status code if the requester does not provide a bearer token. For example:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="example"
```

The **BEARER_TOKEN** mechanism returns a **403 HTTP** status code if a requester is not authorized to access a resource.

```
HTTP/1.1 403 Forbidden
```

Additional resources

- For information about the JWT validation, see Configuring JSON Web Tokens (JWTs) authentication.

- For information about the OAuth2 validation, see Configuring authentication with tokens issued by an OAuth2 compliant authorization server.
- For more information about attributes that you can use with the **token-realm**, see Table A.93.**token-realm** Attributes in the *How to Configure Server Security* guide.

2.10.2. Configuring JSON Web Tokens (JWTs) authentication

You can enable support for JWTs by specifying a **token-realm** in the **elytron** subsystem.

Within the **token-realm**, you can specify attributes and a **jwt** token validator. Elytron completes the following checks:

- Automatic expiration checks on the values specified in the **exp** claim and **nbf** claim.
- Optional: Signature checks based on a public key that is provided by one of the following methods:
  - Using a **public-key** or a **certificate** attribute.
  - Using a key map with named public keys.
  - Using the **client-ssl-context** attribute to retrieve a remote JSON Web Key (JWK) set from the URL specified in **jku** claim.
- Optional: Checks on a JWT to ensure it contains only supported values in **iss** and **aud** claims. You can use **issuer** and **audience** attributes to perform these checks.

The following example shows **token-realm** as the security realm and **principal-claim** as the attribute. The **principal-claim** attribute defines the name of a claim that **elytron** uses to obtain a principal's name. The **jwt** element specifies that the token must be validated as a JWT.

Example of a configured token-realm

```
<token-realm name="${token_realm_name}" principal-claim="${principal_claim_key}">
   <jwt issuer="${issuer_name}"
        audience="${audience_name}"
        <public-key="${public_key_in_PEM_format}"/>
 </token-realm>
```

You can define a key map for your **token-realm**. You can then use different key pairs for signature verification and easily rotate these key pairs. Elytron takes a **kid** claim from the token and uses the corresponding public key for verification.

- If a **kid** claim is not present in JWT, the **token-realm** uses the value specified in the **public-key** attribute of **jwt** to verify the signature.
- If a **kid** claim is not present in JWT and you have not configured the **public-key** then the **token-realm** invalidates the token.

Example of a configured key map for a token-realm.

```
<token-realm name="${token_realm_name}" principal-claim="${principal_claim_key}">
   <jwt issuer="${issuer_name}" audience="${audience_name}">
      <key kid="${key_ID_from_kid_claim}" public-key="${public_key_in_PEM_format}"/>
```

```
        <key kid="${another_key_ID_from_kid_claim}" public-key="${public_key_in_PEM_format}"/>
    </jwt>
</token-realm>
```

Procedure

1. Create a **key-store** by using the **keytool**.

   Example of creating a key-store by using keytool.

   ```
   keytool -genkeypair -alias <alias_name> -keyalg <key_algorithm> -keysize <key_size> -
   validity <key_validity_in_days> -keystore <key_store_path> -dname <distinguished_name> -
   keypass <key_password> -storepass  <key_store_password>
   ```

   Next, add the **key-store** definition in the **elytron** subsystem.

   Example of adding a key-store definition in the elytron subsystem.

   ```
   /subsystem=elytron/key-store=<key_store_name>:add(path=<key_store_path> , credential-
   reference={clear-text=<key_store_password>}, type=<keystore_type>)
   ```

2. Create your **token-realm** in the **elytron** subsystem, and specify attributes and a **jwt** token validator for your **token-realm**.

   Example of creating a token-realm in the elytron subsystem.

   ```
   /subsystem=elytron/token-realm=<token_realm_name>:add(jwt={issuer=
   [<issuer_name>],audience=[<audience_name>],key-
   store=<key_store_name>,certificate=<alias_name>},principal-claim=<principal_claim_key>)
   ```

Next steps

- To configure authentication for your application, see Configuring authentication for an application.

Additional resources

- For more information about **token-realm jwt** attributes, see Table A.94. token-realm jwt Attributes in the *How to Configure Server Security* guide.

2.10.3. Configuring authentication with tokens issued by an OAuth2 compliant authorization server

Elytron supports bearer tokens issued by an OAuth2-compliant authorization server. You can configure a token realm to validate tokens against the predefined **oauth2-introspection** endpoint.

Procedure

1. Create a token realm.

   Example of creating a token realm by using the elytron subsystem:

```
/subsystem=elytron/token-realm=<token_realm_name>:add(principal-
claim=<principal_claim_key>, oauth2-introspection={client-id=<client_id>, client-
secret=<client_secret>, introspection-url=<introspection_URL>})
```

The following example shows an **oauth2-introspection** element specified in the **token-realm** element. This token realm is configured to validate tokens against the predefined **oauth2-introspection** endpoint. The **oauth2-introspection** endpoint uses the values specified in the **client-id** and **client-secret** attributes to identify the client.

Example of an oauth2-introspection element inside the token-realm element:

```
<token-realm name="${token_realm_name}" principal-claim="${principal_claim_key}">
   <oauth2-introspection client-id="${client_id}"
                         client-secret="${client_secret}"
                         introspection-url="${introspection_URL}"
                         host-name-verification-policy="${hostname_verification_policy_value}"/>
</token-realm>
```

Next steps

- To configure authentication for your application, see Configuring authentication for an application.

Additional resources

- For more information about the token introspection endpoint, see Table A.95. **token-realm oauth2-introspection** Attributes.

2.10.4. Configuring bearer token authentication for an application

You can configure authentication for an application by using bearer tokens in JWT format, such as OpenID Connect ID tokens, or by using opaque tokens issued by the OAuth2 compliant authorization server.

Prerequisites

- Depending on the authentication method you require, create a **token-realm** by completing one of the following procedures:
  - Configuring JSON Web Tokens (JWTs) authentication.
  - Configuring authentication with tokens issued by an OAuth2 compliant authorization server .

Procedure

1. Create a security domain in the **elytron** subsystem. Ensure you specify your token security realm in the security domain.

Example of creating a security domain in the elytron subsystem.

```
/subsystem=elytron/security-domain=<security_domain_name>:add(realms=
[{realm=<token_realm_name>,role-decoder=<role_decoder_name>}],permission-
mapper=<permission_mapper_name>,default-realm=<token_realm_name>)
```

2. Create a **http-authentication-factory** that uses the **BEARER_TOKEN** mechanism.

   Example of creating an http-authentication-factory.

   ```
   /subsystem=elytron/http-authentication-factory=<authentication_factory_name>:add(security-domain=<security_domain_name>,http-server-mechanism-factory=global,mechanism-configurations=[{mechanism-name=BEARER_TOKEN,mechanism-realm-configurations=[{realm-name=<token_realm_name>}]}])
   ```

3. Configure an **application-security-domain** in the **undertow** subsystem.

   Example of configuring an application-security-domain in the undertow subsystem.

   ```
   /subsystem=undertow/application-security-domain=<application_security_domain_name>:add(http-authentication-factory=<authentication_factory_name>)
   ```

4. Configure your application's **web.xml** and **jboss-web.xml** files. You must ensure your application's **web.xml** specifies the **BEARER_TOKEN** authentication method. Additionally, ensure **jboss-web.xml** uses the **application-security-domain** you configured in JBoss EAP.

Additional resources

- For more information about the **BEARER_TOKEN** authentication mechanism, see Bearer token authentication.
- For more information about **token-realm jwt** attributes, see Table A.94. token-realm jwt Attributes in the *How to Configure Server Security* guide.
- For more information about attributes that you can use with the OAuth2 token endpoint, see Table A.95. **token-realm oauth2-introspection** Attributes.
- For more information about configuring your application's **web.xml** and **jboss-web.xml**, see Configure Web Applications to Use Elytron or Legacy Security for Authentication in the *How to Configure Identity Management* guide.

CHAPTER 3. LEGACY SECURITY SUBSYSTEM

3.1. CONFIGURE A SECURITY DOMAIN TO USE LDAP

Security domains can be configured to use an LDAP server for authentication and authorization by using a login module. The basics of security domains and login modules are covered in the JBoss EAP *Security Architecture* guide. *LdapExtended* is the preferred login module for integrating with LDAP servers (including Active Directory), but there are several other LDAP login modules that can be used as well. Specifically, the *Ldap*, *AdvancedLdap*, and *AdvancedAdLdap* login modules can also be used to configure a security domain to use LDAP. This section uses the *LdapExtended* login module to illustrate how to create a security domain that uses LDAP for authentication and authorization, but the other LDAP login modules can be used as well. For more details on the other LDAP login modules, see the JBoss EAP *Login Module Reference*.

IMPORTANT

In cases where the legacy **security** subsystem uses an LDAP server to perform authentication, JBoss EAP will return a **500**, or internal server error, error code if that LDAP server is unreachable. This behavior differs from previous versions of JBoss EAP which returned a **401**, or unauthorized, error code under the same conditions.

3.1.1. LdapExtended Login Module

LdapExtended (**org.jboss.security.auth.spi.LdapExtLoginModule**) is a login module implementation that uses searches to locate the bind user and associated roles on an LDAP server. The roles query recursively follows DNs to navigate a hierarchical role structure. For the vast majority of cases when using LDAP with security domains, the LdapExtended login module should be used, especially with LDAP implementations that are not Active Directory. For a full list of configuration options for the LdapExtended login module, see the LdapExtended login module section in the JBoss EAP *Login Module Reference*.

The authentication happens as follows:

1. An initial bind to the LDAP server is done using the **bindDN** and **bindCredential** options. The **bindDN** is a LDAP user with the ability to search both the **baseCtxDN** and **rolesCtxDN** trees for the user and roles. The user DN to authenticate against is queried using the filter specified by the **baseFilter** attribute.

2. The resulting user DN is authenticated by binding to the LDAP server using the user DN as the **InitialLdapContext** environment **Context.SECURITY_PRINCIPAL**. The **Context.SECURITY_CREDENTIALS** property is set to the **String** password obtained by the callback handler.

3.1.1.1. Configure a Security Domain to use the LdapExtended Login Module

Example Data (LDIF format)

```
dn: uid=jduke,ou=Users,dc=jboss,dc=org
objectClass: inetOrgPerson
objectClass: person
objectClass: top
cn: Java Duke
sn: duke
uid: jduke
```

```
userPassword: theduke
# ============================
dn: uid=hnelson,ou=Users,dc=jboss,dc=org
objectClass: inetOrgPerson
objectClass: person
objectClass: top
cn: Horatio Nelson
sn: Nelson
uid: hnelson
userPassword: secret
# ============================
dn: ou=groups,dc=jboss,dc=org
objectClass: top
objectClass: organizationalUnit
ou: groups
# ============================
dn: uid=ldap,ou=Users,dc=jboss,dc=org
objectClass: inetOrgPerson
objectClass: person
objectClass: top
cn: LDAP
sn: Service
uid: ldap
userPassword: randall
# ============================
dn: ou=Users,dc=jboss,dc=org
objectClass: top
objectClass: organizationalUnit
ou: Users
# ============================
dn: dc=jboss,dc=org
objectclass: top
objectclass: domain
dc: jboss
# ============================
dn: uid=GroupTwo,ou=groups,dc=jboss,dc=org
objectClass: top
objectClass: groupOfNames
objectClass: uidObject
cn: GroupTwo
member: uid=jduke,ou=Users,dc=jboss,dc=org
uid: GroupTwo
# ============================
dn: uid=GroupThree,ou=groups,dc=jboss,dc=org
objectClass: top
objectClass: groupOfUniqueNames
objectClass: uidObject
cn: GroupThree
uid: GroupThree
uniqueMember: uid=GroupOne,ou=groups,dc=jboss,dc=org
# ============================
dn: uid=HTTP,ou=Users,dc=jboss,dc=org
objectClass: inetOrgPerson
objectClass: person
objectClass: top
cn: HTTP
```

```
sn: Service
uid: HTTP
userPassword: httppwd
# ============================
dn: uid=GroupOne,ou=groups,dc=jboss,dc=org
objectClass: top
objectClass: groupOfUniqueNames
objectClass: uidObject
cn: GroupOne
uid: GroupOne
uniqueMember: uid=jduke,ou=Users,dc=jboss,dc=org
uniqueMember: uid=hnelson,ou=Users,dc=jboss,dc=org
```

CLI Commands for Adding the LdapExtended Login Module

```
/subsystem=security/security-domain=testLdapExtendedExample:add(cache-type=default)

/subsystem=security/security-domain=testLdapExtendedExample/authentication=classic:add

/subsystem=security/security-domain=testLdapExtendedExample/authentication=classic/login-
module=LdapExtended:add(code=LdapExtended, flag=required, module-options=[
("java.naming.factory.initial"=>"com.sun.jndi.ldap.LdapCtxFactory"),
("java.naming.provider.url"=>"ldap://localhost:10389"),
("java.naming.security.authentication"=>"simple"),
("bindDN"=>"uid=ldap,ou=Users,dc=jboss,dc=org"), ("bindCredential"=>"randall"),
("baseCtxDN"=>"ou=Users,dc=jboss,dc=org"), ("baseFilter"=>"(uid={0})"),
("rolesCtxDN"=>"ou=groups,dc=jboss,dc=org"),     ("roleFilter"=>"(uniqueMember={1})"),
("roleAttributeID"=>"uid")])

reload
```

NOTE

The management CLI commands shown assume that you are running a JBoss EAP standalone server. For more details on using the management CLI for a JBoss EAP managed domain, see the JBoss EAP *Management CLI Guide*.

3.1.1.1.1. Configure a Security Domain to use the LdapExtended Login Module for Active Directory

For Microsoft Active Directory, the LdapExtended login module can be used.

The example below represents the configuration for a default Active Directory configuration. Some Active Directory configurations may require searching against the Global Catalog on port **3268** instead of the usual port **389**. This is most likely when the Active Directory forest includes multiple domains.

Example Configuration for the LdapExtended Login Module for a Default AD Configuration

```
/subsystem=security/security-domain=AD_Default:add(cache-type=default)

/subsystem=security/security-domain=AD_Default/authentication=classic:add

/subsystem=security/security-domain=AD_Default/authentication=classic/login-
module=LdapExtended:add(code=LdapExtended,flag=required,module-options=[
("java.naming.provider.url"=>"ldap://ldaphost.jboss.org"),("bindDN"=>"JBOSSsearchuser"),
```

```
("bindCredential"=>"password"), ("baseCtxDN"=>"CN=Users,DC=jboss,DC=org"), ("baseFilter"=>"
(sAMAccountName={0})"), ("rolesCtxDN"=>"CN=Users,DC=jboss,DC=org"), ("roleFilter"=>"
(sAMAccountName={0})"), ("roleAttributeID"=>"memberOf"), ("roleAttributeIsDN"=>"true"),
("roleNameAttributeID"=>"cn"), ("searchScope"=>"ONELEVEL_SCOPE"),
("allowEmptyPasswords"=>"false")])

reload
```

The example below implements a recursive role search within Active Directory. The key difference between this example and the default Active Directory example is that the role search has been replaced to search the member attribute using the DN of the user. The login module then uses the DN of the role to find groups of which the group is a member.

Example Configuration for the LdapExtended Login Module for a Default AD Configuration with Recursive Search

```
/subsystem=security/security-domain=AD_Recursive:add(cache-type=default)

/subsystem=security/security-domain=AD_Recursive/authentication=classic:add

/subsystem=security/security-domain=AD_Recursive/authentication=classic/login-
module=LdapExtended:add(code=LdapExtended,flag=required,module-options=
[("java.naming.provider.url"=>"ldap://ldaphost.jboss.org"), ("java.naming.referral"=>"follow"),
("bindDN"=>"JBOSSsearchuser"), ("bindCredential"=>"password"),
("baseCtxDN"=>"CN=Users,DC=jboss,DC=org"), ("baseFilter"=>"(sAMAccountName={0})"),
("rolesCtxDN"=>"CN=Users,DC=jboss,DC=org"), ("roleFilter"=>"(member={1})"),
("roleAttributeID"=>"cn"), ("roleAttributeIsDN"=>"false"), ("roleRecursion"=>"2"),
("searchScope"=>"ONELEVEL_SCOPE"), ("allowEmptyPasswords"=>"false")])

reload
```

3.2. CONFIGURE A SECURITY DOMAIN TO USE A DATABASE

Similar to LDAP, security domains can be configured to use a database for authentication and authorization by using a login module.

3.2.1. Database Login Module

The Database login module is a Java Database Connectivity-based (JDBC) login module that supports authentication and role mapping. This login module is used if username, password, and role information are stored in a relational database.

This works by providing a reference to logical tables containing principals and roles in the expected format. For example:

```
Table Principals(PrincipalID text, Password text) Table Roles(PrincipalID text, Role text, RoleGroup
text)
```

The **Principals** table associates the user **PrincipalID** with the valid password, and the **Roles** table associates the user **PrincipalID** with its role sets. The roles used for user permissions must be contained in rows with a **RoleGroup** column value of **Roles**.

The tables are logical in that users can specify the SQL query that the login module uses. The only requirement is that the **java.sql.ResultSet** has the same logical structure as the **Principals** and **Roles**

tables described previously. The actual names of the tables and columns are not relevant as the results are accessed based on the column index.

To clarify this concept, consider a database with two tables, **Principals** and **Roles**, as already declared. The following statements populate the tables with the following data:

- **PrincipalID java** with a password of **echoman** in the **Principals** table
- **PrincipalID java** with a role named **Echo** in the **RolesRoleGroup** in the **Roles** table
- **PrincipalID java** with a role named **caller-java** in the **CallerPrincipalRoleGroup** in the **Roles** table

For a full list of configuration options for the Database login module, see the Database login module section in the JBoss EAP *Login Module Reference*.

3.2.1.1. Configure a Security Domain to use the Database Login Module

Before configuring a security domain to use the Database login module, a datasource must be properly configured.

For more information on creating and configuring datasources in JBoss EAP, see the Datasource Management section of the JBoss EAP *Configuration Guide*.

Once a datasource has been properly configured, a security domain can be configured to use the Database login module. The below example assumes a datasource named **MyDatabaseDS** has been created and properly configured with a database that is constructed with the following:

```
CREATE TABLE Users(username VARCHAR(64) PRIMARY KEY, passwd VARCHAR(64))
CREATE TABLE UserRoles(username VARCHAR(64), role VARCHAR(32))
```

CLI Commands for Adding the Database Login Module

```
/subsystem=security/security-domain=testDB:add

/subsystem=security/security-domain=testDB/authentication=classic:add

/subsystem=security/security-domain=testDB/authentication=classic/login-
module=Database:add(code=Database,flag=required,module-options=
[("dsJndiName"=>"java:/MyDatabaseDS"),("principalsQuery"=>"select passwd from Users where
username=?"),("rolesQuery"=>"select role, 'Roles' from UserRoles where username=?")])

reload
```

3.3. CONFIGURE A SECURITY DOMAIN TO USE A PROPERTIES FILE

Security domains can also be configured to use a filesystem as an identity store for authentication and authorization by using a login module.

3.3.1. UsersRoles Login Module

UsersRoles is a simple login module that supports multiple users and user roles loaded from Java properties files. The primary purpose of this login module is to easily test the security settings of multiple users and roles using properties files deployed with the application. The default username-to-

password mapping filename is **users.properties** and the default username-to-roles mapping filename is **roles.properties**.

NOTE

This login module supports password stacking, password hashing, and unauthenticated identity.

The properties files are loaded during initialization using the initialize method thread context class loader. This means that these files can be placed on the classpath of the Jakarta EE deployment (for example, into the **WEB-INF/classes** folder in the WAR archive), or into any directory on the server classpath.

For a full list of configuration options for the UsersRoles login module, see the UsersRoles login module section in the JBoss EAP *Login Module Reference*.

3.3.1.1. Configure a Security Domain to use the UsersRoles Login Module

The below example assumes the following files have been created and are available on the application's classpath:

- **sampleapp-users.properties**
- **sampleapp-roles.properties**

CLI Commands for Adding the UserRoles Login Module

```
/subsystem=security/security-domain=sampleapp:add

/subsystem=security/security-domain=sampleapp/authentication=classic:add

/subsystem=security/security-domain=sampleapp/authentication=classic/login-
module=UsersRoles:add(code=UsersRoles,flag=required,module-options=
[("usersProperties"=>"sampleapp-users.properties"),("rolesProperties"=>"sampleapp-
roles.properties")])

reload
```

3.4. CONFIGURE A SECURITY DOMAIN TO USE CERTIFICATE-BASED AUTHENTICATION

JBoss EAP provides you with the ability to use certificate-based authentication with security domains to secure web applications or EJBs.

IMPORTANT

Before you can configure certificate-based authentication, you need to have Two-Way SSL/TLS for Applications enabled and configured, which requires X509 certificates configured for both the JBoss EAP instance as well as any clients accessing the web application or EJB secured by the security domain.

Once the certificates, truststores, and two-way SSL/TLS are configured, you then can proceed with configuring a security domain that uses certificate-based authentication, configuring an application to use that security domain, and configuring your client to use the client certificate.

3.4.1. Creating a Security Domain with Certificate-Based Authentication

To create a security domain that uses certificate-based authentication, you need to specify a truststore as well as a Certificate login module or one of its subclasses.

The truststore must contain any trusted client certificates used for authentication, or it must contain the certificate of the certificate authority used to sign the client's certificate. The login module is used to authenticate the certificate presented by the client using the configured truststore. The security domain as a whole also must provide a way to map a role to the principal once it is authenticated. The Certificate login module itself will not map any role information to the principal, but it may be combined with another login module to do so. Alternatively, two subclasses of the Certificate login module, CertificateRoles and DatabaseCertificate, do provide a way to map roles to a principal after it is authenticated. The below example shows how to configure a security domain with certificate-based authentication using the CertificateRoles login module.

WARNING

When performing authentication, the security domain will use the same certificate presented by the client when establishing two-way SSL/TLS. As a result, the client must use the same certificate for **BOTH** two-way SSL/TLS and the certificate-based authentication with the application or EJB.

Example Security Domain with Certificate-Based Authentication

```
/subsystem=security/security-domain=cert-roles-domain:add

/subsystem=security/security-domain=cert-roles-domain/jsse=classic:add(truststore=
{password=secret, url="/path/to/server.truststore.jks"}, keystore={password=secret,
url="/path/to/server.keystore.jks"}, client-auth=true)

/subsystem=security/security-domain=cert-roles-domain/authentication=classic:add

/subsystem=security/security-domain=cert-roles-domain/authentication=classic/login-
module=CertificateRoles:add(code=CertificateRoles, flag=required, module-options=[
securityDomain="cert-roles-domain", rolesProperties="${jboss.server.config.dir}/cert-
roles.properties",password-stacking="useFirstPass",
verifier="org.jboss.security.auth.certs.AnyCertVerifier"])
```

NOTE

The above example uses the CertificateRoles login module to handle authentication and map roles to authenticated principals. It does so by referencing a properties file using the **rolesProperties** attribute. This file lists usernames and roles using the following format:

```
user1=roleA
user2=roleB,roleC
user3=
```

Since usernames are presented as the DN from the provided certificate, for example **CN=valid-client, OU=JBoss, O=Red Hat, L=Raleigh, ST=NC, C=US**, you have to escape special characters such as **=** and spaces when using a properties file:

Example Roles Properties File

```
CN\=valid-client,\ OU\=JBoss,\ O\=Red\ Hat,\ L\=Raleigh,\ ST\=NC,\ C\=US=Admin
```

To view, the DN of certificate:

```
$ keytool -printcert -file valid-client.crt
Owner: CN=valid-client, OU=JBoss, O=Red Hat, L=Raleigh, ST=NC, C=US
...
```

3.4.2. Configure an Application to use a Security Domain with Certificate-Based Authentication

Similar to configuring an application to use a security domain with other forms of authentication, you need to configure both the **jboss-web.xml** and **web.xml** files appropriately.

For **jboss-web.xml**, add a reference to the security domain you configured for certificate-based authentication.

Example jboss-web.xml

```
<jboss-web>
  <security-domain>cert-roles-domain</security-domain>
</jboss-web>
```

For **web.xml**, set the **<auth-method>** attribute in **<login-config>** to **CLIENT-CERT**. You also need define **<security-constraint>** as well as **<security-roles>**.

Example web.xml

```
<web-app>
  <!-- URL for secured portion of application-->
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>secure</web-resource-name>
      <url-pattern>/secure/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>All</role-name>
```

```
  </auth-constraint>
</security-constraint>

<!-- Security roles referenced by this web application -->
<security-role>
  <description>The role that is required to log in to the application</description>
  <role-name>All</role-name>
</security-role>

<login-config>
  <auth-method>CLIENT-CERT</auth-method>
  <realm-name>cert-roles-domain</realm-name>
</login-config>
</web-app>
```

3.4.3. Configure the Client

For a client to authenticate against an application secured with certificate-based authentication, the client needs access to a client certificate that is contained in the JBoss EAP instance's truststore. For example, if accessing the application using a browser, the client will need to import the trusted certificate into the browser's truststore.

3.5. CONFIGURE CACHING FOR A SECURITY DOMAIN

You can specify a cache for a security domain to speed up authentication checks. By default, a security domain uses a simple map as the cache. This default cache is a Least Recently Used (LRU) cache with a maximum of 1000 entries. Alternatively, you can set a security domain to use an Infinispan cache, or disable caching altogether.

3.5.1. Setting the Cache Type for a Security Domain

Prerequisites

- If you are configuring a security domain to use an Infinispan cache, you must first create an Infinispan cache container named **security** that contains a default cache that the security domain will use.

IMPORTANT

You can only define one Infinispan cache configuration for use with security domains. Although you can have multiple security domains that use an Infinispan cache, each security domain creates its own cache instance from the one Infinispan cache configuration.

See the JBoss EAP *Configuration Guide* for more information on creating a cache container.

You can use either the management console or management CLI to set a security domain's cache type.

- To use the management console:
  1. Navigate to **Configuration → Subsystems → Security (Legacy)**.
  2. Select the security domain from the list and click **View**.

3. Click **Edit**, and for the **Cache Type** field, select either **default** or **infinspan**.
4. Click **Save**.

- To use the management CLI, use the following command:

```
/subsystem=security/security-domain=SECURITY_DOMAIN_NAME:write-
attribute(name=cache-type,value=CACHE_TYPE)
```

For example, to set the **other** security domain to use an Infinispan cache:

```
/subsystem=security/security-domain=other:write-attribute(name=cache-
type,value=infinispan)
```

3.5.2. Listing and Flushing Principals

Listing Principals in the Cache

You can see the principals that are stored in a security domain's cache using the following management CLI command:

```
/subsystem=security/security-domain=SECURITY_DOMAIN_NAME:list-cached-principals
```

Flushing Principals from the Cache

If required, you can flush principals from a security domain's cache.

- To flush a specific principal, use the following management CLI command:

```
/subsystem=security/security-domain=SECURITY_DOMAIN_NAME:flush-
cache(principal=USERNAME)
```

- To flush all principals from the cache, use the following management CLI command:

```
/subsystem=security/security-domain=SECURITY_DOMAIN_NAME:flush-cache
```

3.5.3. Disabling Caching for a Security Domain

You can use either the management console or management CLI to disable caching for a security domain.

- To use the management console:
  1. Navigate to **Configuration → Subsystems → Security (Legacy)**.
  2. Select the security domain from the list and click **View**.
  3. Click **Edit** and select the blank value for the **Cache Type**.
  4. Click **Save**.
- To use the management CLI, use the following command:

```
/subsystem=security/security-domain=SECURITY_DOMAIN_NAME:undefine-
attribute(name=cache-type)
```

CHAPTER 4. APPLICATION CONFIGURATION

4.1. CONFIGURE WEB APPLICATIONS TO USE ELYTRON OR LEGACY SECURITY FOR AUTHENTICATION

After you have configured the **elytron** or legacy **security** subsystem for authentication, you need to configure your application to use it.

1. Configure your application's **web.xml**.
   Your application's **web.xml** needs to be configured to use the appropriate authentication method. When using the **elytron** subsystem, this is defined in the **http-authentication-factory** you created. When using the legacy **security** subsystem, this depends on your login module and the type of authentication you want to configure.

   Example web.xml with BASIC Authentication

```
<web-app>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>secure</web-resource-name>
      <url-pattern>/secure/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>Admin</role-name>
    </auth-constraint>
  </security-constraint>
  <security-role>
    <description>The role that is required to log in to /secure/*</description>
    <role-name>Admin</role-name>
  </security-role>
  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>exampleApplicationDomain</realm-name>
  </login-config>
</web-app>
```

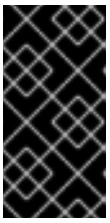
2. Configure your application to use a security domain.
   You can configure your application's **jboss-web.xml** to specify the security domain you want to use for authentication. When using the **elytron** subsystem, this is defined when you created the **application-security-domain**. When using the legacy **security** subsystem, this is the name of the legacy security domain.

   Example jboss-web.xml

```
<jboss-web>
  <security-domain>exampleApplicationDomain</security-domain>
</jboss-web>
```

   Using **jboss-web.xml** allows you to configure the security domain for a single application only. Alternatively, you can specify a default security domain for all applications using the **undertow** subsystem. This allows you to omit using **jboss-web.xml** to configure a security domain for an individual application.

```
/subsystem=undertow:write-attribute(name=default-security-domain,
value="exampleApplicationDomain")
```



IMPORTANT

Setting **default-security-domain** in the **undertow** subsystem will apply to **ALL** applications. If **default-security-domain** is set and an application specifies a security domain in a **jboss-web.xml** file, the configuration in **jboss-web.xml** will override the **default-security-domain** in the **undertow** subsystem.

NOTE

The security domain for EJBs is defined in the EJB configuration, either in the **ejb3** subsystem, the descriptor for EJBs in the **jboss-ejb3.xml** file, or by using the **@SecurityDomain** annotation.

For more information, see EJB Application Security in the *Developing EJB Applications* guide.

Silent BASIC Authentication

You can configure **elytron** to perform a silent **BASIC** authentication. When the silent authentication is enabled, a user is not prompted to log in for accessing the web application. An alternative authentication mechanism is used instead. If the user's request contains an Authorization header, then the **BASIC** authentication mechanism is used.

To enable the silent **BASIC** authentication, set the value of **auth-method** attribute as the following:

```
<auth-method>BASIC?silent=true</auth-method>
```

Using Elytron and Legacy Security Subsystems in Parallel

You can define authentication in both the **elytron** and legacy **security** subsystems and use them in parallel. If you use both **jboss-web.xml** and **default-security-domain** in the **undertow** subsystem, JBoss EAP will first try to match the configured security domain in the **elytron** subsystem. If a match is not found, then JBoss EAP will attempt to match the security domain with one configured in the legacy **security** subsystem. If the **elytron** and legacy **security** subsystem each have a security domain with the same name, the **elytron** security domain is used.

NOTE

If you have a web servlet defined using one security domain and you are calling EJB from another EAR module, which uses EJB specific security domain, one of the following might happen:

- If the WAR and the EJB are mapped to different Elytron security domains, you need to configure the outflow or the trusted security domains so that their identities propagate from one deployment domain to the next one. Unless this is done, once the call reaches the EJB, the identity becomes anonymous. For more information on how to configure security identities for authentication, see Configuring Trusted Security Domain Outflows .
- If the WAR and the EJB references different security domain names but they are mapped to the same Elytron security domain, their identity will propagate without requiring any additional steps.

When migrating, it is best to migrate the entire application. Migrating the EJB and WAR separately and using both **elytron** and legacy **security** subsystems in parallel is not suggested. For more information on how to migrate your application to use Elytron, see Migrating to Elytron in the JBoss EAP Migration Guide.

4.2. CONFIGURE CLIENT AUTHENTICATION WITH ELYTRON CLIENT

Clients connecting to JBoss EAP, such as EJBs, can authenticate using Elytron Client. Elytron Client is a client-side framework that enables remote clients to authenticate using Elytron. Elytron Client has the following components:

Authentication Configuration

The authentication configuration contains authentication information such as usernames, passwords, allowed SASL mechanisms, as well as which security realm to use during digest authentication. The connection information specified in the authentication configuration overrides any values that are specified in the **PROVIDER_URL** of the initial context.

MatchRule

A rule used for deciding which authentication configuration to use.

Authentication Context

A set of rules and authentication configurations to use with a client for establishing a connection.

When a connection is established, the client makes use of an authentication context. This authentication context contains rules to choose which authentication configuration to use for each outbound connection. For example, you could have rules that use one authentication configuration when connecting to **server1** and another authentication configuration when connecting with **server2**. The authentication context is comprised of a set of authentication configurations and a set of rules that define how they are selected when establishing a connection. An authentication context can also reference **ssl-context** and can be matched with rules.

To create a client that uses security information when establishing a connection:

- Create one or more authentication configurations.
- Create an authentication context by creating rule and authentication configuration pairs.
- Create a runnable for establishing your connection.
- Use your authentication context to run your runnable.

When you establish your connection, Elytron Client will use the set of rules provided by the authentication context to match the correct authentication configuration to use during authentication.

You can use one of the following approaches to use security information when establishing a client connection.

IMPORTANT

When using Elytron Client to make EJB calls, any hard-coded programmatic authentication information, such as setting **Context.SECURITY_PRINCIPAL** in the **javax.naming.InitialContext**, will override the Elytron Client configuration.

4.2.1. The Configuration File Approach

The configuration file approach involves creating an XML file with your authentication configuration, authentication context, and match rules.

Example: custom-config.xml

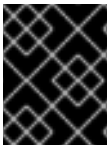
```
<configuration>
    <authentication-client xmlns="urn:elytron:client:1.2">
        <authentication-rules>
            <rule use-configuration="monitor">
                <match-host name="127.0.0.1" />
            </rule>
            <rule use-configuration="administrator">
                <match-host name="localhost" />
            </rule>
        </authentication-rules>
        <authentication-configurations>
            <configuration name="monitor">
                <sasl-mechanism-selector selector="DIGEST-MD5" />
                <providers>
                  <use-service-loader />
                </providers>
                <set-user-name name="monitor" />
                <credentials>
                    <clear-password password="password1!" />
                </credentials>
                <set-mechanism-realm name="ManagementRealm" />
            </configuration>

            <configuration name="administrator">
                <sasl-mechanism-selector selector="DIGEST-MD5" />
                <providers>
                  <use-service-loader />
                </providers>
                <set-user-name name="administrator" />
                <credentials>
                    <clear-password password="password1!" />
                </credentials>
                <set-mechanism-realm name="ManagementRealm" />
            </configuration>
```

```
        </authentication-configurations>
    </authentication-client>
</configuration>
```

You can then reference that file in your client's code by setting a system property when running your client.

```
$ java -Dwildfly.config.url=/path/to/custom-config.xml ...
```



IMPORTANT

If you use the programmatic approach, it will override any provided configuration files even if the **wildfly.config.url** system property is set.

When creating rules, you can look for matches on various parameters, such as **hostname**, **port**, **protocol**, or **user-name**. A full list of options for **MatchRule** are available in the Javadocs. Rules are evaluated in the order in which they are configured.

When no match settings are included in a rule, then the whole rule matches and the authentication configuration is chosen. If more than one match setting is included in a rule, then all must match for the authentication configuration to be chosen.

Table 4.1. Common Rules

Attribute	Description
match-local-security-domain	Takes a single **name** attribute specifying the local security domain to match against.
match-host	Takes a single **name** attribute specifying the hostname to match against. For example, the host **127.0.0.1** would match on **http://127.0.0.1:9990/my/path**.
match-no-user	Matches against URIs with no user.
match-path	Takes a single **name** attribute specifying the path to match against. For example, the path **/my/path/** would match on **http://127.0.0.1:9990/my/path**.
match-port	Takes a single **name** attribute specifying the port to match against. For example, the port **9990** would match on **http://127.0.0.1:9990/my/path**.
match-protocol	Takes a single **name** attribute specifying the protocol to match against. For example, the protocol **http** would match on **http://127.0.0.1:9990/my/path**.
match-urn	Takes a single **name** attribute specifying the URN to match against.

Attribute	Description
match-user	Takes a single **name** attribute specifying the **user** to match against.

An example **wildfly-config.xml** file can be found in Example **wildfly-config.xml**. For more information about how to configure the **wildfly-config.xml** file, see Client Configuration Using the **wildfly-config.xml** File in the *Development Guide* for JBoss EAP.

4.2.2. The Programmatic Approach

The programmatic approach configures all Elytron Client configuration in the client's code:

```
//create your authentication configuration
AuthenticationConfiguration adminConfig =
    AuthenticationConfiguration.empty()
      .useProviders(() -> new Provider[] { new WildFlyElytronProvider() })
      .setSaslMechanismSelector(SaslMechanismSelector.NONE.addMechanism("DIGEST-MD5"))
      .useRealm("ManagementRealm")
      .useName("administrator")
      .usePassword("password1!");

//create your authentication context
AuthenticationContext context = AuthenticationContext.empty();
context = context.with(MatchRule.ALL.matchHost("127.0.0.1"), adminConfig);


//create your runnable for establishing a connection
Runnable runnable =
    new Runnable() {
      public void run() {
        try {
          //Establish your connection and do some work
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    };

//use your authentication context to run your client
context.run(runnable);
```

When adding configuration details to **AuthenticationConfiguration** and **AuthenticationContext**, each method call returns a new instance of that object. For example, if you wanted separate configurations when connecting over different hostnames, you could do the following:

```
//create your authentication configuration
AuthenticationConfiguration commonConfig =
    AuthenticationConfiguration.empty()
      .useProviders(() -> new Provider[] { new WildFlyElytronProvider() })
      .setSaslMechanismSelector(SaslMechanismSelector.NONE.addMechanism("DIGEST-MD5"))
      .useRealm("ManagementRealm");
```

```
AuthenticationConfiguration administrator =
    commonConfig
      .useName("administrator")
      .usePassword("password1!");


AuthenticationConfiguration monitor =
    commonConfig
      .useName("monitor")
      .usePassword("password1!");


//create your authentication context
AuthenticationContext context = AuthenticationContext.empty();
context = context.with(MatchRule.ALL.matchHost("127.0.0.1"), administrator);
context = context.with(MatchRule.ALL.matchHost("localhost"), monitor);
```

Table 4.2. Common Rules

Rule	Description
matchLocalSecurityDomain(String name)	This is the same as **match-domain** in the configuration file approach.
matchNoUser()	This is the same as **match-no-user** in the configuration file approach.
matchPath(String pathSpec)	This is the same as **match-path** in the configuration file approach.
matchPort(int port)	This is the same as **match-port** in the configuration file approach.
matchProtocol(String protoName)	This is the same as **match-port** in the configuration file approach.
matchPurpose(String purpose)	Create a new rule which is the same as this rule, but also matches the given purpose name.
matchUrnName(String name)	This is the same as **match-urn** in the configuration file approach.
matchUser(String userSpec)	This is the same as **match-userinfo** in the configuration file approach.

Also, instead of starting with an empty authentication configuration, you can start with the currently configured one by using **captureCurrent()**.

```
//create your authentication configuration
AuthenticationConfiguration commonConfig = AuthenticationConfiguration.captureCurrent();
```

Using **captureCurrent()** will capture any previously established authentication context and use it as your new base configuration. An authentication context is established once it has been activated by calling **run()**. If **captureCurrent()** is called and no context is currently active, it will try and use the default authentication if available. You can find more details about this in the following sections:

- The Configuration File Approach
- The Default Configuration Approach
- Using Elytron Client with Clients Deployed to JBoss EAP

AuthenticationConfiguration.empty() should only be used as a base to build a configuration on top of, and should not be used on its own. It provides a configuration that uses the JVM-wide registered providers and enables anonymous authentication.

When specifying the providers on top of the **AuthenticationConfiguration.empty()** configuration, you can specify a custom list, but most users should use **WildFlyElytronProvider()** providers.

When creating an authentication context, using the **context.with(...)** will create a new context that merges the rules and authentication configuration from the current context with the provided rule and authentication configuration. The provided rule and authentication configuration will appear after the ones in the current context.

4.2.3. The Default Configuration Approach

The default configuration approach relies completely on the configuration provided by Elytron Client:

```
//create your runnable for establishing a connection
Runnable runnable =
    new Runnable() {
      public void run() {
        try {
          //Establish your connection and do some work
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    };

// run runnable directly
runnable.run();
```

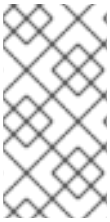
To provide a default configuration, Elytron Client tries to auto-discover a **wildfly-config.xml** file on the filesystem. It looks in the following locations:

- The location specified by the **wildfly.config.url** system property set outside of the client code.
- The classpath root directory.
- The **META-INF** directory on the classpath.
- The current user's home directory.
- The current working directory.

You can use the following example as the basic configuration for your client **wildfly-config.xml** file.

Basic wildfly-config.xml

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <authentication-rules>
      <rule use-configuration="default" />
    </authentication-rules>
    <authentication-configurations>
      <configuration name="default">
        <sasl-mechanism-selector selector="#ALL" />
        <set-mechanism-properties>
          <property key="wildfly.sasl.local-user.quiet-auth" value="true" />
        </set-mechanism-properties>
        <providers>
          <use-service-loader/>
        </providers>
      </configuration>
    </authentication-configurations>
  </authentication-client>
</configuration>
```



NOTE

The **ANONYMOUS** mechanism does not support authorization as a **non-anonymous** user. This means that **set-authorization-name** does not work with **set-anonymous** in the Elytron client configuration file. Instead, if you configure the **set-authorization-name**, you must also specify a **set-user-name** for the authorized identity.

4.2.4. Using Elytron Client with Clients Deployed to JBoss EAP

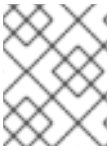
Clients deployed to JBoss EAP can also make use of Elytron Client. The **AuthenticationContext** is automatically parsed and created from the **default-authentication-context** setting in the JBoss EAP configuration. If the **default-authentication-context** is not configured, but you have a **wildfly-config.xml** file included with your deployment or set using the **wildfly.config.url** system property, the **AuthenticationContext** is automatically parsed and created from that file.

Example: Set the Default Authentication Context

```
/subsystem=elytron/authentication-context=AUTH_CONTEXT:add
/subsystem=elytron:write-attribute(name=default-authentication-context,value=AUTH_CONTEXT)
```

To load a configuration file outside of the deployment, you can use the **parseAuthenticationClientConfiguration(URI)** method. This method returns an **AuthenticationContext** that you can then use in your client code using the programmatic approach.

Additionally, clients will also automatically parse and create an **AuthenticationContext** from the client configuration provided by the **elytron** subsystem. The client configuration in the **elytron** subsystem can also take advantage of other components defined in the **elytron** subsystem, such as credential stores. If the client configuration is provided by both the deployment and the **elytron** subsystem, the **elytron** subsystem's configuration is used.



NOTE

The **AuthenticationContext** from the **elytron** subsystem can only be used when this **authentication-context** is set as the default for the **elytron** subsystem.

4.2.5. Configuring a JMX Client Using the wildfly-config.xml File

Starting in JBoss EAP 7.1, JMX clients, including JConsole, can be configured using the **wildfly-config.xml** file. You specify the file path to the configuration file using the **-Dwildfly.config.url** system property when starting the JMX client.

```
-Dwildfly.config.url=path/to/wildfly-config.xml
```

NOTE

When using JConsole, the **-Dwildfly.config.url** system property must be prefixed with **-J**, for example:

```
-J-Dwildfly.config.url=path/to/wildfly-config.xml
```

For more information, see Client Configuration Using the **wildfly-config.xml** File in the JBoss EAP *Development Guide*.

4.2.6. Using the ElytronAuthenticator to Propagate Identities

WARNING

Using the **ElytronAuthenticator** in JBoss EAP is not supported or recommended due to known credential limitations in Java 8. Be aware of the following limitations when using this class to propagate identities.

- Security identity propagation does not work for calls to protected servlets due to Java 8 design limitations.
- Do not use the **ElytronAuthenticator** on the server, for example, in EJBs.
- Credentials caching can impact its use in a standalone client JVM.

JBoss EAP 7.1 introduced the **ElytronAuthenticator** class, which uses the current security context to perform the authentication. The org.wildfly.security.auth.util.ElytronAuthenticator class is an implementation of java.net.Authenticator.

- It has one constructor, **ElytronAuthenticator()**, that constructs a new instance.
- It has one method, **getPasswordAuthentication()**, that returns the **PasswordAuthentication** instance.

The following is an example of client code that creates and uses the **ElytronAuthenticator** class to propagate an identity to the server.

Example: Code Using the ElytronAuthenticator

```
// Create the authentication configuration
AuthenticationConfiguration httpConfig = AuthenticationConfiguration.empty().useName("bob");

// Create the authentication context
AuthenticationContext context = AuthenticationContext.captureCurrent().with(MatchRule.ALL,
httpConfig.usePassword(createPassword(httpConfig, "secret")));

String response = context.run((PrivilegedExceptionAction<String>) () -> {
    Authenticator.setDefault(new ElytronAuthenticator());
    HttpURLConnection connection = HttpURLConnection.class.cast(new URL("http://localhost:" +
SERVER_PORT).openConnection());
    try (InputStream inputStream = connection.getInputStream()) {
        return new BufferedReader(new InputStreamReader(inputStream)).lines().findFirst().orElse(null);
    }
});
```

4.3. CONFIGURING TRUSTED SECURITY DOMAIN OUTFLOWS

For any security invocation, a security identity is established for the security domain. As the invocation is handled, the **SecurityIdentity** is associated with the current thread. For subsequent calls to **getCurrentSecurityIdentity()** on the same security domain, the associated identity is returned.

Within the application server, there can be multiple **SecurityDomain** instances for a single invocation or thread. Each **SecurityDomain** instance can be associated with a different **SecurityIdentity**. The correct security identity is returned when you call that security domain's **getCurrentSecurityIdentity()** method. Deployments can invoke other deployments during request handling. Each deployment is associated with a single security domain. If the invoked deployments use the same security domain, then the notion of a single security domain with a current security identity remains. However, each deployment can reference its own security domain.

It is possible to import a security identity that is associated with a security domain into another security domain, as described in the next section.

Importing a Security Identity

To import a security identity from a security domain into another security domain to obtain a security identity for this domain, there are predominantly three processing flows.

Same Security Domain

A security domain can always import its own security identities. In this case, the security domain always trusts itself.

Common Security Realm

During the import process, the security domain takes the principal from the security identity being imported, passes it through its configured principal transformers and realm mappers, and maps it to an identity within that security domain. If the same security realm is used within the security domain as was used in the security domain that created the identity, both are backed by the same underlying identity and the import is accepted.

Trusted Security Domain

If the identity is successfully mapped but there is no common security realm, the security domain handling the import is tested to see if it trusts the original security domain. If it does, the import is accepted.

NOTE

The identity must exist in the security domain handling the import. The security identity is never trusted in its entirety.

Outflow

A security domain can be configured to automatically outflow its security identities to a different security domain.

In the security domain, if the security identity is established and used for the current invocation, the list of outflow security domains is iterated and the security identity is imported for each of them.

This model is more appropriate where multiple invocations to a deployment using a different security domain are likely to occur, for example, when a web application calls five different EJBs using a common security domain.

CHAPTER 5. SECURING THE MANAGEMENT INTERFACES WITH LDAP

The management interfaces can authenticate against an LDAP server (including Microsoft Active Directory). This is accomplished by using an LDAP authenticator. An LDAP authenticator operates by first establishing a connection (using an outbound LDAP connection) to the remote directory server. It then performs a search using the username which the user passed to the authentication system, to find the fully-qualified distinguished name (DN) of the LDAP record. If successful, a new connection is established, using the DN of the user as the credential, and password supplied by the user. If this second connection and authentication to the LDAP server is successful, the DN is verified to be valid and authentication has succeeded.

NOTE

Securing the management interfaces with LDAP changes the authentication from digest to BASIC/Plain, which by default, will cause usernames and passwords to be sent unencrypted over the network. SSL/TLS can be enabled on the outbound connection to encrypt this traffic and avoid sending this information in the clear.

IMPORTANT

In cases where a legacy security realm uses an LDAP server to perform authentication, such as securing the management interfaces using LDAP, JBoss EAP will return a **500**, or internal server error, error code if that LDAP server is unreachable. This behavior differs from previous versions of JBoss EAP which returned a **401**, or unauthorized, error code under the same conditions.

5.1. USING ELYTRON

You can secure the management interfaces using LDAP with the **elytron** subsystem in the same way as using any identity store. Information on using identity stores for security with the **elytron** subsystem can be found in the Secure the Management Interfaces with a New Identity Store section of *How to Configure Server Security*. For example, to secure the management console with LDAP:

NOTE

If the JBoss EAP server does not have permissions to read the password, such as when an Active Directory LDAP server is used, it is necessary to set **direct-verification** to **true** on the defined LDAP realm. This attribute allows verification to be directly performed on the LDAP server instead of the JBoss EAP server.

Example LDAP Identity Store

```
/subsystem=elytron/dir-context=exampleDC:add(url="ldap://127.0.0.1:10389",principal="uid=admin,ou=system",credential-reference={clear-text="secret"})

/subsystem=elytron/ldap-realm=exampleLR:add(dir-context=exampleDC,identity-mapping={search-base-dn="ou=Users,dc=wildfly,dc=org",rdn-identifier="uid",user-password-mapper={from="userPassword"},attribute-mapping=[{filter-base-dn="ou=Roles,dc=wildfly,dc=org",filter="(&(objectClass=groupOfNames)(member={0}))",from="cn",to="Roles"}]})

/subsystem=elytron/simple-role-decoder=from-roles-attribute:add(attribute=Roles)
```

```
/subsystem=elytron/security-domain=exampleLdapSD:add(realms=[{realm=exampleLR,role-decoder=from-roles-attribute}],default-realm=exampleLR,permission-mapper=default-permission-mapper)

/subsystem=elytron/http-authentication-factory=example-ldap-http-auth:add(http-server-mechanism-factory=global,security-domain=exampleLdapSD,mechanism-configurations=[{mechanism-name=BASIC,mechanism-realm-configurations=[{realm-name=exampleApplicationDomain}]}])

/core-service=management/management-interface=http-interface:write-attribute(name=http-authentication-factory, value=example-ldap-http-auth)

reload
```

5.1.1. Using Elytron for Two-way SSL/TLS for the Outbound LDAP Connection

When using LDAP to secure the management interfaces, you can configure the outbound LDAP connection to use two-way SSL/TLS. To do this, create an **ssl-context** and add it to the **dir-context** used by your **ldap-realm**. Creating a two-way SSL/TLS **ssl-context** is covered in the Enable Two-way SSL/TLS for Applications using the Elytron Subsystem section of *How to Configure Server Security*.

WARNING

Red Hat recommends that SSLv2, SSLv3, and TLSv1.0 be explicitly disabled in favor of TLSv1.1 or TLSv1.2 in all affected packages.

5.2. USING LEGACY CORE MANAGEMENT AUTHENTICATION

To use an LDAP directory server as the authentication source for the management interfaces using the legacy **security** subsystem, the following steps must be performed:

1. Create an outbound connection to the LDAP server.
   The purpose of creating an outbound LDAP connection is to allow the security realm (and the JBoss EAP instance) to establish a connection to the LDAP server. This is similar to the case of creating a datasource for use with the **Database** login module in a security domain.

   The LDAP outbound connection allows the following attributes:

Attribute	Required	Description
url	yes	The URL address of the directory server.
search-dn	no	The fully distinguished name (DN) of the user authorized to perform searches.

Attribute	Required	Description
search-credential	no	The password of the user authorized to perform searches. The attributes supported by this element are: • **store** - Reference to the credential store to obtain the search credential from. • **alias** - The alias of the credential in the referenced store. • **type** - The fully qualified class name of the credential type to obtain from the credential store. • **clear-text** - Instead of referencing a credential store, this attribute can be used to specify a clear text password.
initial-context-factory	no	The initial context factory to use when establishing the connection. Defaults to **com.sun.jndi.ldap.LdapCtxFactory**.
security-realm	no	The security realm to reference to obtain a configured **SSLContext** to use when establishing the connection.

Attribute	Required	Description
referrals	no	Specifies the behavior when encountering a referral when doing a search. Valid options are **IGNORE**, **FOLLOW**, and **THROW**. • **IGNORE**: The default option. Ignores the referral. • **FOLLOW**: When referrals are encountered during a search, the **DirContext** being used will attempt to follow that referral. This assumes the same connection settings can be used to connect to the second server and the name used in the referral is reachable. • **THROW**: The **DirContext** will throw an exception, **LdapReferralException**, to indicate that a referral is required. The security realm will handle and attempt to identify an alternative connection to use for the referral.
always-send-client-cert	no	By default the server's client certificate is not sent while verifying the users credential. If this is set to **true** it will always be sent.
handles-referrals-for	no	Specifies the referrals a connection can handle. If specifying list of URIs, they should be separated by spaces. This enables a connection with connection properties to be defined and used when different credentials are needed to follow a referral. This is useful in situations where different credentials are needed to authenticate against the second server, or for situations where the server returns a name in the referral that is not reachable from the JBoss EAP installation and an alternative address can be substituted.

NOTE

search-dn and **search-credential** are different from the username and password provided by the user. The information provided here is specifically for establishing an initial connection between the JBoss EAP instance and the LDAP server. This connection allows JBoss EAP to perform a subsequent search for the DN of the user trying to authenticate. The DN of the user, which is a result of the search, that is trying to authenticate and the password they provided are used to establish a separate second connection for completing the authentication process.

Given the following example LDAP server, below are the management CLI commands for configuring an outbound LDAP connection:

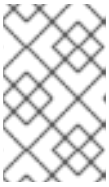
Table 5.1. Example LDAP Server

Attribute	Value
url	127.0.0.1:389
search-credential	myPass
search-dn	cn=search,dc=acme,dc=com

CLI for Adding the Outbound Connection

```
/core-service=management/ldap-connection=ldap-connection/:add(search-
credential=myPass,url=ldap://127.0.0.1:389,search-dn="cn=search,dc=acme,dc=com")

reload
```



NOTE

This creates an unencrypted connection between the JBoss EAP instance and the LDAP server. For more details on setting up an encrypted connection using SSL/TLS, see Using SSL/TLS for the Outbound LDAP Connection .

2. Create a new LDAP-enabled security realm.
Once the outbound LDAP connection has been created, a new LDAP-enabled security realm must be created to use it.

   The LDAP security realm has the following configuration attributes:

Attribute	Description
connection	The name of the connection defined in outbound-connections to use to connect to the LDAP directory.

Attribute	Description
base-dn	The DN of the context to begin searching for the user.
recursive	Whether the search should be recursive throughout the LDAP directory tree, or only search the specified context. Defaults to **false**.
user-dn	The attribute of the user that holds the DN. This is subsequently used to test authentication as the user can complete. Defaults to **dn**.
allow-empty-passwords	This attribute determines whether an empty password is accepted. The default value is **false**.
username-attribute	The name of the attribute to search for the user. This filter performs a simple search where the user name entered by the user matches the specified attribute.
advanced-filter	The fully defined filter used to search for a user based on the supplied user ID. This attribute contains a filter query in standard LDAP syntax. The filter must contain a variable in the following format: **{0}**. This is later replaced with the user name supplied by the user. More details and **advanced-filter** examples can be found in the Combining LDAP and RBAC for Authorization section.

WARNING

It is important to ensure that empty LDAP passwords are not allowed since it is a serious security concern. Unless this behavior is specifically desired in the environment, ensure empty passwords are not allowed and *allow-empty-passwords* remains false.

Below are the management CLI commands for configuring an LDAP-enabled security realm using the **ldap-connection** outbound LDAP connection.

```
/core-service=management/security-realm=ldap-security-realm:add

/core-service=management/security-realm=ldap-security-
realm/authentication=ldap:add(connection="ldap-connection", base-
```

```
dn="cn=users,dc=acme,dc=com",username-attribute="sambaAccountName")

reload
```

3. Reference the new security realm in the management interface. Once a security realm has been created and is using the outbound LDAP connection, that new security realm must be referenced by the management interfaces.

```
/core-service=management/management-interface=http-interface/:write-
attribute(name=security-realm,value="ldap-security-realm")
```

NOTE

The management CLI commands shown assume that you are running a JBoss EAP standalone server. For more details on using the management CLI for a JBoss EAP managed domain, see the JBoss EAP *Management CLI Guide*.

5.2.1. Using Two-way SSL/TLS for the Outbound LDAP Connection

Follow these steps to create an outbound LDAP connection secured by SSL/TLS:

WARNING

Red Hat recommends that SSLv2, SSLv3, and TLSv1.0 be explicitly disabled in favor of TLSv1.1 or TLSv1.2 in all affected packages.

1. Configure a security realm for the outbound LDAP connection to use.
   The security realm must contain a keystore configured with the key that the JBoss EAP server will use to decrypt/encrypt communications between itself and the LDAP server. This keystore will also allow the JBoss EAP instance to verify itself against the LDAP server. The security realm must also contain a truststore that contains the LDAP server's certificate, or the certificate of the certificate authority used to sign the LDAP server's certificate. See Setting up Two-Way SSL/TLS for the Management Interfaces in the JBoss EAP *How to Configure Server Security* guide for instructions on configuring keystores and truststores and creating a security realm that uses them.

2. Create an outbound LDAP connection with the SSL/TLS URL and security realm.
   Similar to the process defined in Using Legacy Core Management Authentication, an outbound LDAP connection should be created, but using the SSL/TLS URL for the LDAP server and the SSL/TLS security realm.

   Once the outbound LDAP connection and SSL/TLS security realm for the LDAP server have been created, the outbound LDAP connection needs to be updated with that information.

   Example CLI for Adding the Outbound Connection with an SSL/TLS URL

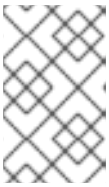
```
/core-service=management/ldap-connection=ldap-connection/:add(search-
credential=myPass, url=ldaps://LDAP_HOST:LDAP_PORT, search-
dn="cn=search,dc=acme,dc=com")
```

Adding the security realm with the SSL/TLS certificates

```
/core-service=management/ldap-connection=ldap-connection:write-attribute(name=security-realm,value="CertificateRealm")

reload
```

3. Create a new security realm that uses the outbound LDAP connection for use by the management interfaces.
   Follow the steps *Create a new LDAP-Enabled Security Realm* and *Reference the new security realm in the Management Interface* from the procedure in Using Legacy Core Management Authentication.



NOTE

The management CLI commands shown assume that you are running a JBoss EAP standalone server. For more details on using the management CLI for a JBoss EAP managed domain, see the JBoss EAP *Management CLI Guide*.

5.3. LDAP AND RBAC

RBAC (Role-Based Access Control) is a mechanism for specifying a set of permissions (roles) for management users. This allows users to be granted different management responsibilities without giving them full, unrestricted access. For more details on RBAC, see the Role-Based Access Control section of the JBoss EAP *Security Architecture* guide.

RBAC is used only for authorization, with authentication being handled separately. Since LDAP can be used for authentication as well as authorization, JBoss EAP can be configured in the following ways:

- Use RBAC for authorization only, and use LDAP, or another mechanism, only for authentication.
- Use RBAC combined with LDAP for making authorization decisions in the management interfaces.

5.3.1. Using LDAP and RBAC Independently

JBoss EAP allows for authentication and authorization to be configured independently in security realms. This enables LDAP to be configured as an authentication mechanism and RBAC to be configured as an authorization mechanism. If configured in this manner, when a user attempts to access a management interface, they will first be authenticated using the configured LDAP server. If successful, the user's role, and configured permissions of that role, will be determined using only RBAC, independently of any group information found in the LDAP server.

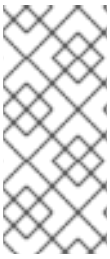
For more details on using just RBAC as an authorization mechanism for the management interfaces, see *How to Configure Server Security* for JBoss EAP. For more details on configuring LDAP for authentication with the management interfaces, see the previous section.

5.3.2. Combining LDAP and RBAC for Authorization

Users who have authenticated using an LDAP server or using a properties file can be members of user groups. A user group is simply an arbitrary label that can be assigned to one or more users. RBAC can be configured to use this group information to automatically assign a role to a user or exclude a user from a role.

An LDAP directory contains entries for user accounts and groups, cross referenced by attributes. Depending on the LDAP server configuration, a user entity can map the groups the user belongs to through **memberOf** attributes; a group entity can map which users belong to it through **uniqueMember** attributes; or a combination of the two. Once a user is successfully authenticated to the LDAP server, a group search is performed to load that user's group information. Depending on the directory server in use, group searches can be performed using their SN, which is usually the username used in authentication, or by using the DN of the user's entry in the directory. Group searches (**group-search**) as well as mapping between a username and a distinguished name (**username-to-dn**) are configured when setting up LDAP as an authorization mechanism in a security realm.

Once a user's group membership information is determined from the LDAP server, a mapping within the RBAC configuration is used to determine what roles a user has. This mapping is configured to explicitly include or exclude groups as well as individual users.



NOTE

The authentication step of a user connecting to the server always happens first. Once the user is successfully authenticated the server loads the user's groups. The authentication step and the authorization step each require a connection to the LDAP server. The security realm optimizes this process by reusing the authentication connection for the group loading step.

5.3.2.1. Using group-search

There are two different styles that can be used when searching for group membership information: *Principal to Group* and *Group to Principal*. Principal to Group has the user's entry containing references to the groups it is a member of, using the **memberOf** attribute. Group to Principal has the group's entry contain the references to the users who are members of it, using the **uniqueMember** attribute.

NOTE

JBoss EAP supports both Principal to Group as well as Group to Principal searches, but Principal to Group is recommended over Group to Principal. If Principal to Group is used, group information can be loaded directly by reading attributes of known distinguished names without having to perform any searches. Group to Principal requires extensive searches to identify the all groups that reference a user.

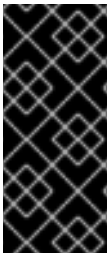
Both Principal to Group and Group to Principal use **group-search** which contains the following attributes:

Attribute	Description
group-name	This attribute is used to specify the form that should be used for the group name returned as the list of groups of which the user is a member. This can either be the simple form of the group name or the group's distinguished name. If the distinguished name is required this attribute can be set to **DISTINGUISHED_NAME**. Defaults to **SIMPLE**.

Attribute	Description
iterative	This attribute is used to indicate if, after identifying the groups a user is a member of, it should also iteratively search based on the groups to identify which groups the groups are a member of. If iterative searching is enabled, it keeps going until either it reaches a group that is not a member if any other groups or a cycle is detected. Defaults to **false**.
group-dn-attribute	On an entry for a group which attribute is its distinguished name. Defaults to **dn**.
group-name-attribute	On an entry for a group which attribute is its simple name. Defaults to **uid**.

NOTE

Cyclic group membership is not a problem. A record of each search is kept to prevent groups that have already been searched from being searched again.



IMPORTANT

For iterative searching to work, the group entries need to look the same as user entries. The same approach used to identify the groups a user is a member of is then used to identify the groups of which the group is a member. This would not be possible if, for group to group membership, the name of the attribute used for the cross reference changes, or if the direction of the reference changes.

Principal to Group (memberOf) for Group Search

Consider an example where a user **TestUserOne** who is a member of **GroupOne**, and **GroupOne** is in turn a member of **GroupFive**. The group membership would be shown by the use of a **memberOf** attribute at the member level. This means, **TestUserOne** would have a **memberOf** attribute set to the **dn** of **GroupOne**. **GroupOne** in turn would have a **memberOf** attribute set to the **dn** of **GroupFive**.

To use this type of searching, the **principal-to-group** element is added to the **group-search** element:

Principal to Group, memberOf, Configuration

```
/core-service=management/security-realm=ldap-security-realm:add

batch

/core-service=management/security-realm=ldap-security-
realm/authorization=ldap:add(connection=ldap-connection)

/core-service=management/security-realm=ldap-security-realm/authorization=ldap/group-
search=principal-to-group:add(group-attribute="memberOf",iterative=true,group-dn-attribute="dn",
group-name="SIMPLE",group-name-attribute="cn")

run-batch
```

IMPORTANT

The above example assumes you already have **ldap-connection** defined. You also need to configure the authentication mechanism which is covered earlier in this section .

Notice that the **group-attribute** attribute is used with the **group-search=principal-to-group**. For reference:

Table 5.2. principal-to-group

Attribute	Description
group-attribute	The name of the attribute on the user entry that matches the distinguished name of the group the user is a member of. Defaults to **memberOf**.
prefer-original-connection	This value is used to indicate which group information to prefer when following a referral. Each time a principal is loaded, attributes from each of their group memberships are subsequently loaded. Each time attributes are loaded, either the original connection or connection from the last referral can be used. Defaults to **true**.

Group to Principal, uniqueMember, Group Search

Consider the same example as Principal to Group where a user **TestUserOne** who is a member of **GroupOne**, and **GroupOne** is in turn a member of **GroupFive**. However, in this case the group membership would be shown by the use of the **uniqueMember** attribute set at the group level. This means that **GroupFive** would have a **uniqueMember** set to the **dn** of **GroupOne**. **GroupOne** in turn would have a **uniqueMember** set to the **dn** of **TestUserOne**.

To use this type of searching, the **group-to-principal** element is added to the **group-search** element:

Group to Principal, uniqueMember, Configuration

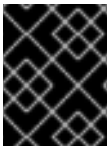
```
/core-service=management/security-realm=ldap-security-realm:add

batch

/core-service=management/security-realm=ldap-security-
realm/authorization=ldap:add(connection=ldap-connection)

/core-service=management/security-realm=ldap-security-realm/authorization=ldap/group-
search=group-to-principal:add(iterative=true, group-dn-attribute="dn", group-name="SIMPLE", group-
name-attribute="uid", base-dn="ou=groups,dc=group-to-principal,dc=example,dc=org", principal-
attribute="uniqueMember", search-by="DISTINGUISHED_NAME")

run-batch
```



IMPORTANT

The above example assumes you already have **ldap-connection** defined. You also need to configure the authentication mechanism which is covered earlier in this section .

Notice that the **principal-attribute** attribute is used with **group-search=group-to-principal**. **group-to-principal** is used to define how searches for groups that reference the user entry will be performed, and **principal-attribute** is used to define the group entry that references the principal.

For reference:

Table 5.3. group-to-principal

Attribute	Description
base-dn	The distinguished name of the context to use to begin the search.
recursive	Whether sub-contexts also be searched. Defaults to **false**.
search-by	The form of the role name used in searches. Valid values are **SIMPLE** and **DISTINGUISHED_NAME**. Defaults to **DISTINGUISHED_NAME**.
prefer-original-connection	This value is used to indicate which group information to prefer when following a referral. Each time a principal is loaded, attributes from each of their group memberships are subsequently loaded. Each time attributes are loaded, either the original connection or connection from the last referral can be used.

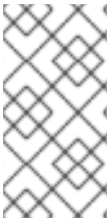
Table 5.4. membership-filter

Attribute	Description
principal-attribute	The name of the attribute on the group entry that references the user entry. Defaults to **member**.

5.3.2.2. Using username-to-dn

It is possible to define rules within the authorization section to convert a user's simple user name to their distinguished name. The **username-to-dn** element specifies how to map the user name to the distinguished name of their entry in the LDAP directory. This element is **optional** and only required when both of the following are true:

- The authentication and authorization steps are against different LDAP servers.
- The group search uses the distinguished name.



NOTE

This could also be applicable in instances where the security realm supports both LDAP and Kerberos authentication and a conversion is needed for Kerberos, if LDAP authentication has been performed the DN discovered during authentication can be used.

It contains the following attributes:

Table 5.5. username-to-dn

Attribute	Description
force	The result of a user name to distinguished name mapping search during authentication is cached and reused during the authorization query when the force attribute is set to **false**. When force is **true**, the search is performed again during authorization while loading groups. This is typically done when different servers perform authentication and authorization.

username-to-dn can be configured with one of the following:

username-is-dn

This specifies that the user name entered by the remote user is the user's distinguished name.

username-is-dn Example

```
/core-service=management/security-realm=ldap-security-realm:add

batch

/core-service=management/security-realm=ldap-security-
realm/authorization=ldap:add(connection=ldap-connection)

/core-service=management/security-realm=ldap-security-realm/authorization=ldap/group-
search=group-to-principal:add(iterative=true, group-dn-attribute="dn", group-name="SIMPLE",
group-name-attribute="uid", base-dn="ou=groups,dc=group-to-principal,dc=example,dc=org",
principal-attribute="uniqueMember", search-by="DISTINGUISHED_NAME")

/core-service=management/security-realm=ldap-security-realm/authorization=ldap/username-to-
dn=username-is-dn:add(force=false)

run-batch
```

This defines a 1:1 mapping and there is no additional configuration.

username-filter

A specified attribute is searched for a match against the supplied user name.

username-filter Example

```
/core-service=management/security-realm=ldap-security-realm:add
```

```
batch

/core-service=management/security-realm=ldap-security-realm/authorization=ldap:add(connection=ldap-connection)

/core-service=management/security-realm=ldap-security-realm/authorization=ldap/group-search=group-to-principal:add(iterative=true, group-dn-attribute="dn", group-name="SIMPLE", group-name-attribute="uid", base-dn="ou=groups,dc=group-to-principal,dc=example,dc=org", principal-attribute="uniqueMember", search-by="DISTINGUISHED_NAME")

/core-service=management/security-realm=ldap-security-realm/authorization=ldap/username-to-dn=username-filter:add(force=false, base-dn="dc=people,dc=harold,dc=example,dc=com", recursive="false", attribute="sn", user-dn-attribute="dn")

run-batch
```

Attribute	Description
base-dn	The distinguished name of the context to begin the search.
recursive	Whether the search will extend to sub contexts. Defaults to **false**.
attribute	The attribute of the user's entry to try and match against the supplied user name. Defaults to **uid**.
user-dn-attribute	The attribute to read to obtain the user's distinguished name. Defaults to **dn**.

advanced-filter

This option uses a custom filter to locate the user's distinguished name.

advanced-filter Example

```
/core-service=management/security-realm=ldap-security-realm:add

batch

/core-service=management/security-realm=ldap-security-realm/authorization=ldap:add(connection=ldap-connection)

/core-service=management/security-realm=ldap-security-realm/authorization=ldap/group-search=group-to-principal:add(iterative=true, group-dn-attribute="dn", group-name="SIMPLE", group-name-attribute="uid", base-dn="ou=groups,dc=group-to-principal,dc=example,dc=org", principal-attribute="uniqueMember", search-by="DISTINGUISHED_NAME")

/core-service=management/security-realm=ldap-security-realm/authorization=ldap/username-to-dn=advanced-filter:add(force=true, base-dn="dc=people,dc=harold,dc=example,dc=com",
```

```
recursive="false", user-dn-attribute="dn",filter="sAMAccountName={0}")

run-batch
```

For the attributes that match those in the **username-filter** example, the meaning and default values are the same. There is one additional attribute:

Attribute	Description
filter	Custom filter used to search for a user's entry where the user name will be substituted in the **{0}** placeholder.

IMPORTANT

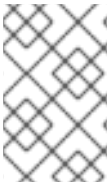
This must remain valid after the filter is defined so if any special characters are used (such as **&**) ensure the proper form is used. For example **&** for the **&** character.

5.3.2.3. Mapping LDAP Group Information to RBAC Roles

Once the connection to the LDAP server has been created and the group searching has been properly configured, a mapping needs to be created between the LDAP groups and RBAC roles. This mapping can be both inclusive as well as exclusive, and enables users to be automatically assigned one or more roles based on their group membership.

WARNING

If RBAC is not already configured, pay close attention when doing so, especially if switching to a newly-created LDAP-enabled realm. Enabling RBAC without having users and roles properly configured could result in administrators being unable to login to the JBoss EAP management interfaces.



NOTE

The management CLI commands shown assume that you are running a JBoss EAP standalone server. For more details on using the management CLI for a JBoss EAP managed domain, see the JBoss EAP *Management CLI Guide*.

Ensure RBAC is Enabled and Configured

Before mappings between LDAP and RBAC Roles can be used, RBAC must enabled and initially configured.

```
/core-service=management/access=authorization:read-attribute(name=provider)
```

It should yield the following result:

```
{ "outcome" => "success", "result" => "rbac" }
```

For more information on enabling and configuring RBAC, see Enabling Role-Based Access Control in *How to Configure Server Security* for JBoss EAP.

Verify Existing List of Roles

Use the **read-children-names** operation to get a complete list of the configured roles:

```
/core-service=management/access=authorization:read-children-names(child-type=role-mapping)
```

Which should yield a list of roles:

```
{
  "outcome" => "success",
  "result" =>
    [ "Administrator", "Deployer", "Maintainer", "Monitor", "Operator", "SuperUser" ]
}
```

In addition, all existing mappings for a role can be checked:

```
/core-service=management/access=authorization/role-mapping=Administrator:read-
resource(recursive=true)
```

```
{
  "outcome" => "success",
  "result" =>
  {
    "include-all" => false,
    "exclude" => undefined,
    "include" => {
      "user-theboss" => {
        "name" => "theboss",
        "realm" => undefined,
        "type" => "USER"
      },
      "user-harold" => {
        "name" => "harold",
        "realm" => undefined,
        "type" => "USER"
      },
      "group-SysOps" => {
        "name" => "SysOps",
        "realm" => undefined,
        "type" => "GROUP"
      }
    }
  }
}
```

Configure a Role-Mapping entry

If a role does not already have a **Role-Mapping** entry, one needs to be created. For instance:

```
/core-service=management/access=authorization/role-mapping=Auditor:read-resource()
```

```
{
  "outcome" => "failed",
  "failure-description" => "WFLYCTL0216: Management resource '[ (\"core-service\" =>
\"management\"), (\"access\" => \"authorization\"), (\"role-mapping\" => \"Auditor\") ]' not found"
}
```

To add a role mapping:

```
/core-service=management/access=authorization/role-mapping=Auditor:add()
```

```
{
  "outcome" => "success"
}
```

To verify:

```
/core-service=management/access=authorization/role-mapping=Auditor:read-resource()
```

```
{
  "outcome" => "success",
  "result" => {
    "include-all" => false,
    "exclude" => undefined,
    "include" => undefined
  }
}
```

Add Groups to the Role for Inclusion and Exclusion

Groups can be added for inclusion or exclusion from a role.

NOTE

The exclusion mapping takes precedence or the inclusion mapping.

To add a group for inclusion:

```
/core-service=management/access=authorization/role-mapping=Auditor/include=group-
GroupToInclude:add(name=GroupToInclude, type=GROUP)
```

To add a group for exclusion:

```
/core-service=management/access=authorization/role-mapping=Auditor/exclude=group-
GroupToExclude:add(name=GroupToExclude, type=GROUP)
```

To check the result:

```
/core-service=management/access=authorization/role-mapping=Auditor:read-
resource(recursive=true)
```

```
{
  "outcome" => "success",
  "result" => {
    "include-all" => false,
    "exclude" => {
      "group-GroupToExclude" => {
        "name" => "GroupToExclude",
        "realm" => undefined,
        "type" => "GROUP"
      }
    },
    "include" => {
      "group-GroupToInclude" => {
        "name" => "GroupToInclude",
        "realm" => undefined,
        "type" => "GROUP"
      }
    }
  }
}
```

Removing a Group from exclusion or inclusion in an RBAC Roles Groups

To remove a group from inclusion:

```
/core-service=management/access=authorization/role-mapping=Auditor/include=group-GroupToInclude:remove
```

To remove a group from exclusion:

```
/core-service=management/access=authorization/role-mapping=Auditor/exclude=group-GroupToExclude:remove
```

5.4. ENABLING CACHING

Security Realms also offer the ability to cache the results of LDAP queries for both authentication as well as group loading. This enables the results of different queries to be reused across multiple searches by different users in certain circumstances, for example iteratively querying the group membership information of groups. There are three different caches available, each of which are configured separately and operate independently:

- authentication
- group-to-principal
- username-to-dn

5.4.1. Cache Configuration

Even though the caches are independent of one another, all three are configured in the same manner. Each cache offers the following configuration options:

Attribute	Description
type	This defines the eviction strategy that the cache will adhere to. Options are **by-access-time** and **by-search-time**. **by-access-time** evicts items from the cache after a certain period of time has elapsed since their last access. **by-search-time** evicts items based on how long they have been in the cache regardless of their last access.
eviction-time	This defines the time (in seconds) used for evictions depending on the strategy.
cache-failures	This is a boolean that enables/disables the caching of failed searches. This has the potential for preventing an LDAP server from being repeatedly accessed by the same failed search, but it also has the potential to fill up the cache with searches for users that do not exist. This setting is particularly important for the authentication cache.
max-cache-size	This defines maximum size (number of items) of the cache, which in-turn dictates when items will begin getting evicted. Old items are evicted from the cache to make room for new authentication and searches as needed, meaning **max-cache-size** will not prevent new authentication attempts or searches from occurring.

5.4.2. Example

NOTE

This example assumes a security realm, named **LDAPRealm**, has been created. It connects to an existing LDAP server and is configured for authentication and authorization. The commands to display the current configuration are detailed in Reading the Current Cache Configuration. More details on creating a security realm that uses LDAP can be found in Using Legacy Core Management Authentication .

Example Base Configuration

```
"core-service" : {
  "management" : {
    "security-realm" : {
      "LDAPRealm" : {
        "authentication" : {
          "ldap" : {
            "allow-empty-passwords" : false,
            "base-dn" : "...",
            "connection" : "MyLdapConnection",
            "recursive" : false,
```

```
            "user-dn" : "dn",
            "username-attribute" : "uid",
            "cache" : null
          }
        },
        "authorization" : {
          "ldap" : {
            "connection" : "MyLdapConnection",
            "group-search" : {
              "group-to-principal" : {
                "base-dn" : "...",
                "group-dn-attribute" : "dn",
                "group-name" : "SIMPLE",
                "group-name-attribute" : "uid",
                "iterative" : true,
                "principal-attribute" : "uniqueMember",
                "search-by" : "DISTINGUISHED_NAME",
                "cache" : null
              }
            },
            "username-to-dn" : {
              "username-filter" : {
                "attribute" : "uid",
                "base-dn" : "...",
                "force" : false,
                "recursive" : false,
                "user-dn-attribute" : "dn",
                "cache" : null
              }
            }
          },
        }
      }
    }
}
```

In all areas where **"cache" : null** appear, a cache may be configured:

Authentication

During authentication, the user's distinguished name is discovered using this definition and an attempt to connect to the LDAP server and verify their identity is made using these credentials.

A group-search definition

There is the group search definition. In this case it is an iterative search because **iterative** is set to **true** in the sample configuration above. First, a search will be performed to find all groups the user is a direct member of. After that, a search will be performed for each of those groups to identify if they have membership to other groups. This process continues until either a cyclic reference is detected or the final groups are not members of any further groups.

A username-to-dn definition in group search

Group searching relies on the availability of the user's distinguished name. This section is not used in all situations, but it can be used as a second attempt to discover a user's distinguished name. This can be useful, or even required, when a second form of authentication is supported, for example local authentication.

5.4.2.1. Reading the Current Cache Configuration

NOTE

The CLI commands used in this and subsequent sections use **LDAPRealm** for the name of the security realm. This should be substituted for the name of the actual realm being configured.

CLI Command to Read the Current Cache Configuration

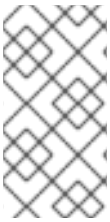
```
/core-service=management/security-realm=LDAPRealm:read-resource(recursive=true)
```

Output

```
{
  "outcome" => "success",
  "result" => {
    "map-groups-to-roles" => true,
    "authentication" => {
      "ldap" => {
        "advanced-filter" => undefined,
        "allow-empty-passwords" => false,
        "base-dn" => "dc=example,dc=com",
         "connection" => "ldapConnection",
         "recursive" => true,
         "user-dn" => "dn",
         "username-attribute" => "uid",
         "cache" => undefined
       }
      },
      "authorization" => {
        "ldap" => {
          "connection" => "ldapConnection",
          "group-search" => {
            "principal-to-group" => {
              "group-attribute" => "description",
              "group-dn-attribute" => "dn",
              "group-name" => "SIMPLE",
              "group-name-attribute" => "cn",
              "iterative" => false,
              "prefer-original-connection" => true,
              "skip-missing-groups" => false,
              "cache" => undefined
            }
          },
          "username-to-dn" => {
            "username-filter" => {
              "attribute" => "uid",
              "base-dn" => "ou=Users,dc=jboss,dc=org",
              "force" => true,
              "recursive" => false,
              "user-dn-attribute" => "dn",
              "cache" => undefined
            }
```

```
        }
       }
      },
      "plug-in" => undefined,
      "server-identity" => undefined
    }
  }
```

5.4.2.2. Enabling a Cache



NOTE

The management CLI commands used in this and subsequent sections configure the cache in the authentication section of the security realm, in other words **authentication=ldap**/. Caches in the authorization section can also be configured in a similar manner by updating the path of the command.

Management CLI Command for Enabling a Cache

```
/core-service=management/security-realm=LDAPRealm/authentication=ldap/cache=by-access-time:add(eviction-time=300, cache-failures=true, max-cache-size=100)
```

This commands adds a **by-access-time** cache for authentication with an eviction time of 300 seconds (5 minutes) and a maximum cache size of 100 items. In addition, failed searches will be cached. Alternatively, a **by-search-time** cache could also be configured:

```
/core-service=management/security-realm=LDAPRealm/authentication=ldap/cache=by-search-time:add(eviction-time=300, cache-failures=true, max-cache-size=100)
```

5.4.2.3. Inspecting an Existing Cache

Management CLI Command for Inspecting an Existing Cache

```
/core-service=management/security-realm=LDAPRealm/authentication=ldap/cache=by-access-time:read-resource(include-runtime=true)

{
  "outcome" => "success",
  "result" => {
    "cache-failures" => true,
    "cache-size" => 1,
    "eviction-time" => 300,
    "max-cache-size" => 100
  }
}
```

The **include-runtime** attribute adds **cache-size**, which displays the current number of items in the cache. It is **1** in the above output.

5.4.2.4. Testing an Existing Cache's Contents

Management CLI Command for Testing an Existing Cache's Contents

```
/core-service=management/security-realm=LDAPRealm/authentication=ldap/cache=by-access-time:contains(name=TestUserOne)

{
  "outcome" => "success",
  "result" => true
}
```

This shows that an entry for **TestUserOne** exists the in the cache.

5.4.2.5. Flushing a Cache

You can flushing a single item from a cache, or flush the entire cache.

Management CLI Command for Flushing a Single Item

```
/core-service=management/security-realm=LDAPRealm/authentication=ldap/cache=by-access-time:flush-cache(name=TestUserOne)
```

Management CLI Command for Flushing an Entire Cache

```
/core-service=management/security-realm=LDAPRealm/authentication=ldap/cache=by-access-time:flush-cache()
```

5.4.2.6. Removing a Cache

Management CLI Command for Removing a Cache

```
/core-service=management/security-realm=LDAPRealm/authentication=ldap/cache=by-access-time:remove()

reload
```

CHAPTER 6. CONFIGURE A SECURITY DOMAIN TO USE A SECURITY MAPPING

Adding a security mapping to a security domain allows for authentication and authorization information to be combined after the authentication or authorization happens, but before the information is passed to the application. For more information on security mapping, see the Security Mapping section of the JBoss EAP *Security Architecture* guide.

To add a security mapping to an existing security domain, a **code**, **type**, and relevant module options must be configured. The **code** field is the short name, for example **SimpleRoles**, **PropertiesRoles**, **DatabaseRoles**, or class name of the security mapping module. The **type** field refers to the type of mapping this module performs, and the allowed values are **principal**, **role**, **attribute**, or **credential**. For a full list of the available security mapping modules and their module options, see the Security Mapping Modules section of the JBoss EAP *Login Module Reference*.

Example: Management CLI Commands for Adding a SimpleRoles Security Mapping to an Existing Security Domain

```
/subsystem=security/security-domain=sampleapp/mapping=classic:add

/subsystem=security/security-domain=sampleapp/mapping=classic/mapping-
module=SimpleRoles:add(code=SimpleRoles,type=role,module-options=[("user1"=>"specialRole")])

reload
```

CHAPTER 7. STANDALONE SERVER VS. MANAGED DOMAIN CONSIDERATIONS

Setting up identity management with an LDAP server, including Microsoft Active Directory, is essentially the same regardless or whether it is used in a standalone server or a managed domain. In general, this also applies to setting up most identity stores with both security realms and security domains. Just as with any other configuration setting, the standalone configuration resides in the **standalone.xml** file and the configuration for a managed domain resides in the **domain.xml** and **host.xml** files.

APPENDIX A. REFERENCE MATERIAL

A.1. EXAMPLE WILDFLY-CONFIG.XML

The **wildlfly-config.xml** file is one way for clients to use Elytron Client, which allows clients to use security information when making connections to JBoss EAP. For more details on using Elytron Client, see Configure Client Authentication with Elytron Client.

Example: custom-config.xml

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <authentication-rules>
      <rule use-configuration="monitor">
        <match-host name="127.0.0.1" />
      </rule>
      <rule use-configuration="administrator">
        <match-host name="localhost" />
      </rule>
    </authentication-rules>
    <authentication-configurations>
      <configuration name="monitor">
        <sasl-mechanism-selector selector="DIGEST-MD5" />
        <providers>
          <use-service-loader />
        </providers>
        <set-user-name name="monitor" />
        <credentials>
          <clear-password password="password1!" />
        </credentials>
        <set-mechanism-realm name="ManagementRealm" />
      </configuration>

      <configuration name="administrator">
        <sasl-mechanism-selector selector="DIGEST-MD5" />
        <providers>
          <use-service-loader />
        </providers>
        <set-user-name name="administrator" />
        <credentials>
          <clear-password password="password1!" />
        </credentials>
        <set-mechanism-realm name="ManagementRealm" />
      </configuration>
    </authentication-configurations>

    <net-authenticator/>

    <!-- This decides which SSL context configuration to use -->
    <ssl-context-rules>
      <rule use-ssl-context="mycorp-client">
        <match-host name="mycorp.com"/>
      </rule>
    </ssl-context-rules>
    <ssl-contexts>
```

```
      <default-ssl-context name="mycorp-context"/>
      <ssl-context name="mycorp-context">
        <key-store-ssl-certificate key-store-name="store1" alias="mycorp-client-certificate"/>
        <!-- This is an OpenSSL-style cipher suite selection string; this example is the expanded form of
DEFAULT to illustrate the format -->
        <cipher-suite selector="ALL:!EXPORT:!LOW:!aNULL:!eNULL:!SSLv2"/>
        <protocol names="TLSv1.2"/>
      </ssl-context>
    </ssl-contexts>
  </authentication-client>
</configuration>
```

For more information about how to configure clients using the **wildfly-config.xml** file, see Client Configuration Using the wildfly-config.xml File in the *Development Guide* for JBoss EAP.

A.2. REFERENCE FOR SINGLE SIGN-ON ATTRIBUTES

An SSO authentication mechanism configuration.

This is the reference for the **setting=single-sign-on** resource of the **application-security-domain** in the **undertow** subsystem.

A.2.1. Single Sign-on

Table A.1. single-sign-on Attributes

Attribute	Description
domain	The cookie domain to be used.
path	The cookie path.
http-only	For setting cookie's **httpOnly** attribute.
secure	For setting cookie's **secure** attribute.
cookie-name	The name of the cookie.
key-store	The reference to keystore containing a private key entry.
key-alias	The alias of the private key entry used for signing and verifying back-channel logout connection.

Attribute	Description
credential-reference	The credential reference to decrypt the private key entry. **credential-reference** has the following attributes: • **store** : The name of the credential store holding the alias to credential. • **alias** : The alias which denotes stored secret or credential in the store. • **type** : The type of credential this reference is denoting. • **clear-text** : The secret specified using clear text. Checks the credential store way of supplying credential or secrets to services.
client-ssl-context	The reference to the SSL context used to secure back-channel logout connection.

A.3. PASSWORD MAPPERS

A password mapper constructs a password from multiple fields in a database using one of the following algorithm types:

- Clear text
- Simple digest
- Salted simple digest
- bcrypt
- SCRAM
- Modular crypt

A password mapper has the following attributes:

NOTE

The index of the first column is **1** for all the mappers.

Table A.2. password mapper attributes

Mapper name	Attributes	Encryption method
clear-password-mapper	• **password-index** The index of the column containing the clear text password.	No encryption.

Mapper name	Attributes	Encryption method
simple-digest	• **password-index** The index of the column containing the password hash. • **algorithm** The hashing algorithm used. The following values are supported: ◦ **simple-digest-md2** ◦ **simple-digest-md5** ◦ **simple-digest-sha-1** ◦ **simple-digest-sha-256** ◦ **simple-digest-sha-384** ◦ **simple-digest-sha-512** • **hash-encoding** Specify the representation hash. Permitted values: ◦ **base64** (default) ◦ **hex**	A simple hashing mechanism is used.

Mapper name	Attributes	Encryption method
salted-simple-digest	• **password-index** The index of the column containing the password hash. • **algorithm** The hashing algorithm used. The following values are supported: ◦ **password-salt-digest-md5** ◦ **password-salt-digest-sha-1** ◦ **password-salt-digest-sha-256** ◦ **password-salt-digest-sha-384** ◦ **password-salt-digest-sha-512** ◦ **salt-password-digest-md5** ◦ **salt-password-digest-sha-1** ◦ **salt-password-digest-sha-256** ◦ **salt-password-digest-sha-384** ◦ **salt-password-digest-sha-512** • **salt-index** Index of the column containing the salt used for hashing. • **hash-encoding** Specify the representation for the hash. Permitted values: ◦ **base64** (default) ◦ **hex** • **salt-encoding** Specify the representation for the salt. Permitted values: ◦ **base64** (default) ◦ **hex**	A simple hashing mechanism is used with a salt.

Mapper name	Attributes	Encryption method
bcrypt-password-mapper	• **password-index** The index of the column containing the password hash. • **salt-index** Index of the column containing the salt used for hashing. • **iteration-count-index** Index of the column containing the number of iterations used. • **hash-encoding** Specify the representation for the hash. Permitted values: ◦ **base64** (default) ◦ **hex** • **salt-encoding** Specify the representation for the salt. Permitted values: ◦ **base64** (default) ◦ **hex**	Blowfish algorithm used for hashing.

Mapper name	Attributes	Encryption method

scram-mapper	• **password-index**
	The index of the column containing the password hash.
	• **algorithm**
	The hashing algorithm used. The following values are supported:
	  ◦ **scram-sha-1**
	  ◦ **scram-sha-256**
	  ◦ **scram-sha-384**
	  ◦ **scram-sha-512**
	• **salt-index**
	Index of the column containing the salt is used for hashing.
	• **iteration-count-index**
	Index of the column containing the number of iterations used.
	• **hash-encoding**
	Specify the representation for the hash. Permitted values:
	  ◦ **base64** (default)
	  ◦ **hex**
	• **salt-encoding**
	Specify the representation for the salt. Permitted values:
	  ◦ **base64** (default)
	  ◦ **hex**
		Salted Challenge Response Authentication mechanism is used for hashing.

modular-crypt-mapper	• **password-index** The index of the column containing the encrypted password.	The modular-crypt encoding allows for multiple pieces of information to be encoded in single string such as the password type, the hash or digest, the salt, and the iteration count.

Revised on 2022-02-01 13:03:04 UTC