Red Hat JBoss Enterprise Application Platform 8-beta

Securing applications and management interfaces using an identity store

Guide to securing JBoss EAP management interfaces and deployed applications by using an identity store such as the filesystem, a database, Lightweight Directory Access Protocol (LDAP), or a custom identity store

Last Updated: 2022-12-16

Red Hat JBoss Enterprise Application Platform 8-beta Securing applications and management interfaces using an identity store

Guide to securing JBoss EAP management interfaces and deployed applications by using an identity store such as the filesystem, a database, Lightweight Directory Access Protocol (LDAP), or a custom identity store

Legal Notice

Abstract

Guide to securing JBoss EAP management interfaces and deployed applications by using an identity store such as the filesystem, a database, Lightweight Directory Access Protocol (LDAP), or a custom identity store.

Table of Contents

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

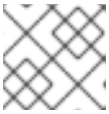
We appreciate your feedback on our documentation. To provide feedback, you can highlight the text in a document and add comments. Follow the steps in the procedure to learn about submitting feedback on Red Hat documentation.

Prerequisites

- Log in to the Red Hat Customer Portal.
- In the Red Hat Customer Portal, view the document in **Multi-page HTML** format.

Procedure

1. Click **Feedback** to see existing reader comments.



NOTE

The feedback feature is enabled only in the **Multi-page HTML** format.

2. Highlight the section of the document where you want to provide feedback.
3. In the prompt menu that displays near the text you selected, click **Add Feedback**.
   A text box opens in the feedback section on the right side of the page.
4. Enter your feedback in the text box and click **Submit**.
   You have created a documentation issue.
5. To view the issue, click the issue tracker link in the feedback view.
6. Highlight the section of the document where you want to provide feedback.
7. In the prompt menu that displays near the text you selected, click **Add Feedback**.
   A text box opens in the feedback section on the right side of the page.
8. Enter your feedback in the text box and click **Submit**.
   You have created a documentation issue.
9. To view the issue, click the issue tracker link in the feedback view.

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

CHAPTER 1. CONFIGURING IDENTITY STORES

1.1. CREATING A FILESYSTEM-REALM

1.1.1. Filesystem realm in Elytron

With a filesystem security realm, **filesystem-realm**, you can use a filesystem-based identity store in Elytron to store user credentials and attributes. Elytron stores each identity along with the associated credentials and attributes in an XML file in the filesystem. The name of the XML file is the name of the identity. You can associate multiple credentials and attributes with each identity.

By default, identities are stored in the filesystem as follows:

- Elytron applies two levels of directory hashing to the directory structure where an identity is stored. For example, an identity named "user1" is stored at the location **u/s/user1.xml**.
  This is done to overcome the limit set by some filesystems on the number of files you can store in a single directory and for performance reasons.

  Use the **levels** attribute to configure the number of levels of directory hashing to apply.

- The identity names are Base32 encoded before they are used as filenames. This is done because some filesystems are case-insensitive or might restrict the set of characters allowed in a filename.
  You can turn off the encoding by setting the attribute **encoded** to **false**.

For information about other attributes and their default values, see **filesystem-realm** attributes.

Encryption
The **filesystem-realm** uses Base64 encoding for clear passwords, hashed passwords, and attributes when storing an identity in an identity file. For added security, you can encrypt the clear passwords, hashed passwords, and attributes using a secret key stored in a credential store. The secret key is used both for encrypting and decrypting the passwords and attributes.

Integrity check
To ensure that the identities created with a **filesystem-realm** are not tampered with, you can enable integrity checking on the **filesystem-realm** by referencing a key pair in the **filesystem-realm** during creation.

Integrity checking works in **filesystem-realm** as follows:

- When you create an identity in the **filesystem-realm** with integrity checking enabled, Elytron creates the identity file and generates a signature for it.

- Whenever the identity file is read, for example when updating the identity or loading the identity for authentication, Elytron verifies the identity file contents against the signature to ensure the file has not been tampered with since the last authorized write.

- When you update an existing identity that has an associated signature, Elytron updates the content and generates a new signature after the original content passes verification.
  If the verification fails, you get the following failure message:

```
{
    "outcome" => "failed",
    "failure-description" => "WFLYCTL0158: Operation handler
failed:java.lang.RuntimeException: WFLYELY01008: Failed to obtain the authorization
```

```
identity.",
    "rolled-back" => true
}
```

Additional resources

- **filesystem-realm** attributes
- Creating a **filesystem-realm** in Elytron
- Creating an encrypted **filesystem-realm** in Elytron
- Creating a **filesystem-realm** with integrity in Elytron

1.1.2. Creating a filesystem-realm in Elytron

Create a **filesystem-realm** and a security domain that references the realm to secure the JBoss EAP server interfaces or the applications deployed on the server.

Prerequisites

- JBoss EAP is running.

Procedure

1. Create a **filesystem-realm** in Elytron.

   Syntax

   ```
   /subsystem=elytron/filesystem-realm=<filesystem_realm_name>:add(path=<file_path>)
   ```

   Example

   ```
   /subsystem=elytron/filesystem-realm=exampleSecurityRealm:add(path=fs-realm-
   users,relative-to=jboss.server.config.dir)
   {"outcome" => "success"}
   ```

2. Add a user to the realm and configure the user's role.

   a. Add a user.

      Syntax

      ```
      /subsystem=elytron/filesystem-realm=<filesystem_realm_name>:add-
      identity(identity=<user_name>)
      ```

      Example

      ```
      /subsystem=elytron/filesystem-realm=exampleSecurityRealm:add-identity(identity=user1)
      {"outcome" => "success"}
      ```

   b. Set a password for the user.

Syntax

```
/subsystem=elytron/filesystem-realm=<filesystem_realm_name>:set-
password(identity=<user_name>, clear={password=<password>})
```

Example

```
/subsystem=elytron/filesystem-realm=exampleSecurityRealm:set-
password(identity=user1, clear={password="passwordUser1"})
{"outcome" => "success"}
```

c. Set roles for the user.

Syntax

```
/subsystem=elytron/filesystem-realm=<filesystem_realm_name>:add-identity-
attribute(identity=<user_name>, name=<roles_attribute_name>, value=
[<role_1>,<role_N>])
```

Example

```
/subsystem=elytron/filesystem-realm=exampleSecurityRealm:add-identity-
attribute(identity=user1, name=Roles, value=["Admin","Guest"])
{"outcome" => "success"}
```

3. Create a security domain that references the **filesystem-realm**.

Syntax

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

Example

```
/subsystem=elytron/security-domain=exampleSecurityDomain:add(default-
realm=exampleSecurityRealm,permission-mapper=default-permission-mapper,realms=
[{realm=exampleSecurityRealm}])
{"outcome" => "success"}
```

Verification

- To verify that Elytron can load an identity from the **filesystem-realm**, use the following command:

Syntax

```
/subsystem=elytron/security-domain=<security_domain_name>:read-
identity(name=<username>)
```

Example

```
/subsystem=elytron/security-domain=exampleSecurityDomain:read-identity(name=user1)
{
    "outcome" => "success",
    "result" => {
        "name" => "user1",
        "attributes" => {"Roles" => [
            "Admin",
            "Guest"
        ]},
        "roles" => [
            "Guest",
            "Admin"
        ]
    }
}
```

You now can use the created security domain to add authentication and authorization to management interfaces and applications. For more information, see Securing management interfaces and applications.

Additional resources

- **filesystem-realm** attributes
- **security-domain** attributes
- **simple-role-decoder** attributes

1.1.3. Creating an encrypted filesystem-realm in Elytron

Create an encrypted **filesystem-realm** to secure JBoss EAP applications or server interfaces and ensure that the user credentials are encrypted and therefore secure.

1.1.3.1. Creating a secret-key-credential-store for a standalone server

Create a **secret-key-credential-store** using the management CLI. When you create a **secret-key-credential-store**, JBoss EAP generates a secret key by default. The name of the generated key is **key** and its size is 256-bit.

Prerequisites

- JBoss EAP is running.
- You have provided at least read/write access to the directory containing the **secret-key-credential-store** for the user account under which JBoss EAP is running.

Procedure

- Use the following command to create a **secret-key-credential-store** using the management CLI:

  Syntax

```
/subsystem=elytron/secret-key-credential-store=<name_of_credential_store>:add(path="<path_to_the_credential_store>", relative-to=<path_to_store_file>)
```

Example

```
/subsystem=elytron/secret-key-credential-store=examplePropertiesCredentialStore:add(path=examplePropertiesCredentialStore.cs, relative-to=jboss.server.config.dir)
{"outcome" => "success"}
```

1.1.3.2. Creating an encrypted filesystem-realm

Create an encrypted **filesystem-realm** and a security domain that references the realm to secure the JBoss EAP server interfaces or the applications deployed on the server.

Prerequisites

- JBoss EAP is running.
- You have created a **secret-key-credential-store**.
  For more information, see Creating a **secret-key-credential-store** for a standalone server.

Procedure

1. Create an encrypted **filesystem-realm** in Elytron.

   Syntax

   ```
   /subsystem=elytron/filesystem-realm=<filesystem_realm_name>:add(path=<file_path>,credential-store=<name_of_credential_store>,secret-key=<key>)
   ```

   Example

   ```
   /subsystem=elytron/filesystem-realm=exampleSecurityRealm:add(path=fs-realm-users,relative-to=jboss.server.config.dir, credential-store=examplePropertiesCredentialStore, secret-key=key)
   {"outcome" => "success"}
   ```

2. Add a user to the realm and configure the user's role.

   a. Add a user.

   Syntax

   ```
   /subsystem=elytron/filesystem-realm=<filesystem_realm_name>:add-identity(identity=<user_name>)
   ```

   Example

```
/subsystem=elytron/filesystem-realm=exampleSecurityRealm:add-identity(identity=user1)
{"outcome" => "success"}
```

b. Set a password for the user.

Syntax

```
/subsystem=elytron/filesystem-realm=<filesystem_realm_name>:set-
password(identity=<user_name>, clear={password=<password>})
```

Example

```
/subsystem=elytron/filesystem-realm=exampleSecurityRealm:set-
password(identity=user1, clear={password="passwordUser1"})
{"outcome" => "success"}
```

c. Set roles for the user.

Syntax

```
/subsystem=elytron/filesystem-realm=<filesystem_realm_name>:add-identity-
attribute(identity=<user_name>, name=<roles_attribute_name>, value=
[<role_1>,<role_N>])
```

Example

```
/subsystem=elytron/filesystem-realm=exampleSecurityRealm:add-identity-
attribute(identity=user1, name=Roles, value=["Admin","Guest"])
{"outcome" => "success"}
```

3. Create a security domain that references the **filesystem-realm**.

Syntax

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

Example

```
/subsystem=elytron/security-domain=exampleSecurityDomain:add(default-
realm=exampleSecurityRealm,permission-mapper=default-permission-mapper,realms=
[{realm=exampleSecurityRealm}])
{"outcome" => "success"}
```

Verification

- To verify that Elytron can load an identity from the encrypted **filesystem-realm**, use the following command:

Syntax

```
/subsystem=elytron/security-domain=<security_domain_name>:read-
identity(name=<username>)
```

Example

```
/subsystem=elytron/security-domain=exampleSecurityDomain:read-identity(name=user1)
{
    "outcome" => "success",
    "result" => {
        "name" => "user1",
        "attributes" => {"Roles" => [
            "Admin",
            "Guest"
        ]},
        "roles" => [
            "Guest",
            "Admin"
        ]
    }
}
```

You can now use the created security domain to add authentication and authorization to management interfaces and applications.

Additional resources

- **filesystem-realm** attributes
- **security-domain** attributes
- **simple-role-decoder** attributes

1.1.4. Creating a filesystem-realm with integrity support in Elytron

Create a **filesystem-realm** with integrity support to secure JBoss EAP applications or server interfaces and ensure that the user credentials are not tampered with.

1.1.4.1. Creating a key pair using the management CLI

Create a key store with a key pair in Elytron.

Prerequisites

- JBoss EAP is running.

Procedure

1. Create a key store.

   Syntax

   ```
   /subsystem=elytron/key-
   store=<key_store_name>:add(path=<path_to_key_store_file>,credential-reference=
   {<password>})
   ```

-

Example

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

2. Create a key pair in the key store.

Syntax

```
/subsystem=elytron/key-store=<key_store_name>:generate-key-
pair(alias=<alias>,algorithm=<key_algorithm>,key-
size=<size_of_key>,validity=<validity_in_days>,distinguished-
name="<distinguished_name>")
```

Example

```
/subsystem=elytron/key-store=exampleKeystore:generate-key-
pair(alias=localhost,algorithm=RSA,key-size=1024,validity=365,distinguished-
name="CN=localhost")
{"outcome" => "success"}
```

3. Persist the key pair to the key store file.

Syntax

```
/subsystem=elytron/key-store=<key_store_name>:store()
```

Example

```
/subsystem=elytron/key-store=exampleKeystore:store()
{
    "outcome" => "success",
    "result" => undefined
}
```

1.1.4.2. Creating a filesystem-realm with integrity support

Create a **filesystem-realm** with integrity support and a security domain that references the realm to secure the JBoss EAP server interfaces or the applications deployed on the server.

Prerequisites

- JBoss EAP is running.
- You have created a **secret-key-credential-store**.
  For more information, see Creating a key pair using the management CLI .

Procedure

1. Create **filesystem-realm** in Elytron.

Syntax

```
/subsystem=elytron/filesystem-realm=<filesystem_realm_name>:add(path=<file_path>,key-store=<key_store_name>,key-store-alias=<key_store_alias>)
```

Example

```
/subsystem=elytron/filesystem-realm=exampleSecurityRealm:add(path=fs-realm-users,relative-to=jboss.server.config.dir, key-store=exampleKeystore, key-store-alias=localhost)
{"outcome" => "success"}
```

2. Add a user to the realm and configure the user's role.

   a. Add a user.

   Syntax

   ```
   /subsystem=elytron/filesystem-realm=<filesystem_realm_name>:add-identity(identity=<user_name>)
   ```

   Example

   ```
   /subsystem=elytron/filesystem-realm=exampleSecurityRealm:add-identity(identity=user1)
   {"outcome" => "success"}
   ```

   b. Set a password for the user.

   Syntax

   ```
   /subsystem=elytron/filesystem-realm=<filesystem_realm_name>:set-password(identity=<user_name>, clear={password=<password>})
   ```

   Example

   ```
   /subsystem=elytron/filesystem-realm=exampleSecurityRealm:set-password(identity=user1, clear={password="passwordUser1"})
   {"outcome" => "success"}
   ```

   c. Set roles for the user.

   Syntax

   ```
   /subsystem=elytron/filesystem-realm=<filesystem_realm_name>:add-identity-attribute(identity=<user_name>, name=<roles_attribute_name>, value=[<role_1>,<role_N>])
   ```

   Example

   ```
   /subsystem=elytron/filesystem-realm=exampleSecurityRealm:add-identity-attribute(identity=user1, name=Roles, value=["Admin","Guest"])
   {"outcome" => "success"}
   ```

-

3. Create a security domain that references the **filesystem-realm**.

   Syntax

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

   Example

   ```
   /subsystem=elytron/security-domain=exampleSecurityDomain:add(default-
   realm=exampleSecurityRealm,permission-mapper=default-permission-mapper,realms=
   [{realm=exampleSecurityRealm}])
   {"outcome" => "success"}
   ```

Verification

- To verify that Elytron can load an identity from the **filesystem-realm**, use the following command:

  Syntax

  ```
  /subsystem=elytron/security-domain=<security_domain_name>:read-
  identity(name=<username>)
  ```

  Example

  ```
  /subsystem=elytron/security-domain=exampleSecurityDomain:read-identity(name=user1)
  {
      "outcome" => "success",
      "result" => {
          "name" => "user1",
          "attributes" => {"Roles" => [
              "Admin",
              "Guest"
          ]},
          "roles" => [
              "Guest",
              "Admin"
          ]
      }
  }
  ```

You now can use the created security domain to add authentication and authorization to management interfaces and applications. For more information, see Securing management interfaces and applications.

Additional resources

- **filesystem-realm** attributes
- **security-domain** attributes

- **simple-role-decoder** attributes

1.1.4.3. Updating the key pair in an existing filesystem-realm with integrity support enabled

You can update the key pair referenced in a **filesystem-realm** with integrity support enabled in the case that the existing key was compromised. Also, it is a good practice to rotate keys.

Prerequisites

- You have generated a key pair.
- You have created a **filesystem-realm** with integrity checking enabled.
  For more information, see Creating a **filesystem-realm** with integrity support.

Procedure

1. Create a key pair in the existing key store.

   Syntax

   ```
   /subsystem=elytron/key-store=<key_store_name>:generate-key-pair(alias=<alias>,algorithm=<key_algorithm>,key-size=<size_of_key>,validity=<validity_in_days>,distinguished-name="<distinguished_name>")
   ```

   Example

   ```
   /subsystem=elytron/key-store=exampleKeystore:generate-key-pair(alias=localhost2,algorithm=RSA,key-size=1024,validity=365,distinguished-name="CN=localhost")
   {"outcome" => "success"}
   ```

2. Persist the key pair to the key store file.

   Syntax

   ```
   /subsystem=elytron/key-store=<key_store_name>:store()
   ```

   Example

   ```
   /subsystem=elytron/key-store=exampleKeystore:store()
   {
       "outcome" => "success",
       "result" => undefined
   }
   ```

3. Update the key store alias to reference a new key pair.

   Syntax

   ```
   /subsystem=elytron/filesystem-realm=<filesystem_realm_name>:write-attribute(name=key-store-alias, value=<key_store_alias>)
   ```

Example

```
/subsystem=elytron/filesystem-realm=exampleSecurityRealm:write-attribute(name=key-store-alias, value=localhost2)
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    }
}
```

4. Reload the server.

```
reload
```

5. Use the new key pair to update the files in **filesystem-realm** with new signatures.

Syntax

```
/subsystem=elytron/filesystem-realm=<filesystem_realm_name>:update-key-pair()
```

Example

```
/subsystem=elytron/filesystem-realm=exampleSecurityRealm:update-key-pair()
{"outcome" => "success"}
```

Verification

- Verify that the key pair referenced in the **filesystem-realm** has been updated using the following management CLI command:

Syntax

```
/subsystem=elytron/filesystem-realm=<filesystem_realm_name>:read-resource()
```

Example

```
/subsystem=elytron/filesystem-realm=exampleSecurityRealm:read-resource()
{
    "outcome" => "success",
    "result" => {
        "credential-store" => undefined,
        "encoded" => true,
        "hash-charset" => "UTF-8",
        "hash-encoding" => "base64",
        "key-store" => "exampleKeystoreFSRealm",
        "key-store-alias" => "localhost2",
        "levels" => 2,
        "secret-key" => undefined,
        "path" => "fs-realm-users",
```

```
        "relative-to" => "jboss.server.config.dir"
    }
}
```

The key pair referenced in the **filesystem-realm** has been updated.

Additional resources

- **filesystem-realm** attributes

1.1.5. Encrypting an unencrypted filesystem-realm

If you have a **filesystem-realm** configured in Elytron, you can add encryption to it using the WildFly Elytron Tool.

1.1.5.1. Creating a secret-key-credential-store for a standalone server

Create a **secret-key-credential-store** using the management CLI. When you create a **secret-key-credential-store**, JBoss EAP generates a secret key by default. The name of the generated key is **key** and its size is 256-bit.

Prerequisites

- JBoss EAP is running.
- You have provided at least read/write access to the directory containing the **secret-key-credential-store** for the user account under which JBoss EAP is running.

Procedure

- Use the following command to create a **secret-key-credential-store** using the management CLI:

  Syntax

  ```
  /subsystem=elytron/secret-key-credential-store=<name_of_credential_store>:add(path="<path_to_the_credential_store>", relative-to=<path_to_store_file>)
  ```

  Example

  ```
  /subsystem=elytron/secret-key-credential-store=examplePropertiesCredentialStore:add(path=examplePropertiesCredentialStore.cs, relative-to=jboss.server.config.dir)
  {"outcome" => "success"}
  ```

1.1.5.2. Converting an unencrypted filesystem-realm to an encrypted filesystem-realm

You can convert an unencrypted **filesystem-realm** into an encrypted one using the WildFly Elytron Tool **filesystem-realm-encrypt**.

Prerequisites

- You have an existing **filesystem-realm**.
  For more information, see Creating a **filesystem-realm** in Elytron.
- You have created a **secret-key-credential-store**.
  For more information, see Creating a **secret-key-credential-store** for a standalone server.
- JBoss EAP is running.

Procedure

1. Convert an unencrypted **filesystem-realm** into an encrypted one.

   Syntax

   ```
   $ JBOSS_HOME/bin/elytron-tool.sh filesystem-realm-encrypt --input-location
   <existing_filesystem_realm_name> --output-location
   JBOSS_HOME/standalone/configuration/<target_filesystem_realm_name> --credential-store
   <path_to_credential_store>/<credential_store>
   ```

   Example

   ```
   $ JBOSS_HOME/bin/elytron-tool.sh filesystem-realm-encrypt --input-location
   JBOSS_HOME/standalone/configuration/fs-realm-users --output-location
   JBOSS_HOME/standalone/configuration/fs-realm-users-enc --credential-store
   JBOSS_HOME/standalone/configuration/examplePropertiesCredentialStore.cs

   Creating encrypted realm for: JBOSS_HOME/standalone/configuration/fs-realm-users
   Found credential store and alias, using pre-existing key
   ```

   The WildFly Elytron command **filesystem-realm-encrypt** creates a filesystem realm specified with the **--output-location** argument. It also creates a CLI script at the root of the filesystem realm that you can use to add the filesystem realm resource in the **elytron** subsystem.

   TIP

   Use the **--summary** option to see a summary of the command execution.

2. Use the generated CLI script to add the filesystem realm resource in the **elytron** subsystem.

   Synax

   ```
   $ JBOSS_HOME/bin/jboss-cli.sh --connect --
   file=<target_filesystem_realm_directory>/<target_filesystem_realm_name>.cli
   ```

   Example

   ```
   $ JBOSS_HOME/bin/jboss-cli.sh --connect --file=JBOSS_HOME/standalone/configuration/fs-
   realm-users-enc/encrypted-filesystem-realm.cli
   {"outcome" => "success"}
   {"outcome" => "success"}
   ```

You can use the encrypted **filesystem-realm** to create a security domain that references the realm to secure the JBoss EAP server interfaces or the applications deployed on the server.

Additional resources

- **filesystem-realm** attributes
- **secret-key-credential-store** attributes
- Creating an encrypted **filesystem-realm** in Elytron
- For more information about the WildFly Elytron tool **filesystem-realm-encrypt** command, run the **filesystem-realm-encrypt --help** command:

```
$ JBOSS_HOME/bin/elytron-tool.sh filesystem-realm-encrypt --help
```

1.2. CREATING A JDBC REALM

1.2.1. Creating a jdbc-realm in Elytron

Create a **jdbc-realm** and a security domain that references the realm to secure the JBoss EAP server interfaces or the applications deployed on the server.

The examples in the procedure use a PostgreSQL database which is configured as follows:

- Database name: postgresdb
- Database login credentials:
  - username: postgres
  - password: postgres
- Table name: example_jboss_eap_users
- example_jboss_eap_users contents:

username	password	roles
user1	passwordUser1	Admin
user2	passwordUser2	Guest

Prerequisites

- You have configured the database containing the users.
- JBoss EAP is running.
- You have downloaded the appropriate JDBC driver.

Procedure

1. Deploy the database driver for the database using the management CLI.

   Syntax

```
deploy <path_to_jdbc_driver>/<jdbc-driver>
```

Example

```
deploy PATH_TO_JDBC_DRIVER/postgresql-42.2.9.jar
```

2. Configure the database as the data source.

Syntax

```
data-source add --name=<data_source_name> --jndi-name=<jndi_name> --driver-
name=<jdbc-driver>  --connection-url=<database_URL> --user-
name=<database_username> --password=<database_username>
```

Example

```
data-source add --name=examplePostgresDS --jndi-name=java:jboss/examplePostgresDS --
driver-name=postgresql-42.2.9.jar  --connection-
url=jdbc:postgresql://localhost:5432/postgresdb --user-name=postgres --password=postgres
```

3. Create a **jdbc-realm** in Elytron.

Syntax

```
/subsystem=elytron/jdbc-realm=<jdbc_realm_name>:add(principal-query=
[<sql_query_to_load_users>])
```

Example

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

NOTE

The 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. Create a security domain that references the **jdbc-realm**.

Syntax

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

Example

```
/subsystem=elytron/security-domain=exampleSecurityDomain:add(default-
realm=exampleSecurityRealm,permission-mapper=default-permission-mapper,realms=
[{realm=exampleSecurityRealm}])
{"outcome" => "success"}
```

Verification

- To verify that Elytron can load data from the database, use the following command:

  Syntax

  ```
  /subsystem=elytron/security-domain=<security_domain_name>:read-
  identity(name=<username>)
  ```

  Example

  ```
  /subsystem=elytron/security-domain=exampleSecurityDomain:read-identity(name=user1)
  {
      "outcome" => "success",
      "result" => {
          "name" => "user1",
          "attributes" => {"Roles" => ["Admin"]},
          "roles" => ["Admin"]
      }
  }
  ```

  The output confirms that Elytron can load data from the database.

You now can use the created security domain to add authentication and authorization to management interfaces and applications. For more information, see Securing management interfaces and applications.

Additional resources

- **jdbc-realm** attributes
- Password Mappers
- **security-domain** attributes

1.3. CREATING AN LDAP REALM

1.3.1. LDAP realm in Elytron

The Lightweight Directory Access Protocol (LDAP) realm, **ldap-realm**, in Elytron is a security realm that you can use to load identities from an LDAP identity store.

The following example illustrates how an identity in LDAP is mapped with an Elytron identity in JBoss EAP.

Example LDAP Data Interchange Format (LDIF) file

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

dn: uid=user1,ou=Users,dc=wildfly,dc=org
objectClass: top
objectClass: person
objectClass: inetOrgPerson
cn: user1
sn: user1
uid: user1
userPassword: userPassword1

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=user1,ou=Users,dc=wildfly,dc=org
```

Example commands to create an LDAP realm

```
/subsystem=elytron/dir-context=exampleDirContext:add(url="ldap://10.88.0.2",principal="cn=admin,dc=wildfly,dc=org",credentia-reference={clear-text="secret"})

/subsystem=elytron/ldap-realm=exampleSecurityRealm:add(dir-context=exampleDirContext,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={1}))",from="cn",to="Roles"}]})
```

The commands result in the following configuration:

```
<ldap-realm name="exampleLDAPRealm" dir-context="exampleDirContext"> 1
    <identity-mapping rdn-identifier="uid" search-base-dn="ou=Users,dc=wildfly,dc=org"> 2
        <attribute-mapping> 3
            <attribute from="cn" to="Roles" filter="(&amp;(objectClass=groupOfNames)(member={1}))" filter-base-dn="ou=Roles,dc=wildfly,dc=org"/> 4
        </attribute-mapping>
        <user-password-mapper from="userPassword"/> 5
    </identity-mapping>
</ldap-realm>
```

1 The realm definition.

- **name** is the **ldap-realm** realm name.
- **dir-context** is the configuration to connect to an LDAP server.

2 Define how identity is mapped.

- **rdn-identifier** is relative distinguished name (RDN) of the principal's distinguished name (DN) to use to obtain the principal's name from an LDAP entry. In the example LDIF, **uid** is configured to represent the principal's name from the base **DN=ou=Users,dc=wildfly,dc=org**.
  search-base-dn is the base DN to search for identities. In the example LDIF, it is defined as **dn: ou=Users,dc=wildfly,dc=org**.

3 Define the LDAP attributes to the identity's attributes mappings.

4 Configure how to map a specific LDAP attribute as an Elytron identity attribute.

- **from** is the LDAP attribute to map. If it is not defined, the DN of the entry is used.
- **to** is the name of the identity's attribute mapped from LDAP attribute. If not provided, the name of the attribute is the same as the one defined in **from**. If **from** is also not defined, the DN of the entry is used.
- **filter** is a filter to use to obtain the values for a specific attribute. String '{0}' is replaced by the username, '{1}' by user identity DN.
  - **objectClass** is the LDAP object class to use. In the example LDIF, the object class to use is defined as **groupOfNames**.
  - **member** is the member to map. **{0}** is replaced by user name, and **{1}** by user identity DN. In this example, **{1}** is used to map **member** to **user1**.
- **filter-base-dn** is the name of the context where the filter should be applied.
  The result of the example filter is that the user **user1** is mapped with the **Admin** role.

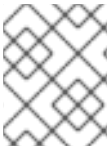
5 **user-password-mapper** defines the LDAP attribute from which an identity's password is obtained. In the example it is configured as **userPassword**, which is defined in the LDIF as **userPassword1**.

Additional resources

- Creating an **ldap-realm** in Elytron
- **ldap-realm** attributes

1.3.2. Creating an ldap-realm in Elytron

Create an Elytron security realm backed by a Lightweight Directory Access Protocol (LDAP) identity store. Use the security realm to create a security domain to add authentication and authorization to management interfaces or the applications deployed on the server.



NOTE

ldap-realm configured as caching realm does not support Active Directory. For more information, see Changing LDAP/AD User Password via JBossEAP CLI for Elytron .

IMPORTANT

In cases where the **elytron** subsystem uses an LDAP server to perform authentication, JBoss EAP will return a **500** error code, or internal server error, if that LDAP server is unreachable.

To ensure that the management interfaces and applications secured using an LDAP realm can be accessed even if the LDAP server becomes available, use a failover realm. For information see Creating a failover realm.

For the examples in this procedure, the following LDAP Data Interchange Format (LDIF) is used:

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

dn: uid=user1,ou=Users,dc=wildfly,dc=org
objectClass: top
objectClass: person
objectClass: inetOrgPerson
cn: user1
sn: user1
uid: user1
userPassword: userPassword1

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=user1,ou=Users,dc=wildfly,dc=org
```

The LDAP connection parameters used for the example are as follows:

- LDAP URL: **ldap://10.88.0.2**
- LDAP admin password: **secret**
  You need this for Elytron to connect with the LDAP server.
- LDAP admin Distinguished Name (DN): **(cn=admin,dc=wildfly,dc=org)**
- LDAP organization: **wildfly**
  If no organization name is specified, it defaults to **Example Inc**.
- LDAP domain: **wildfly.org**
  This is the name that is matched when the platform receives an LDAP search reference.

Prerequisites

- You have configured an LDAP identity store.

- JBoss EAP is running.

Procedure

1. Configure a directory context that provides the URL and the principal used to connect to the LDAP server.

   Syntax

   ```
   /subsystem=elytron/dir-context=<dir_context_name>:add(url="<LDAP_URL>",principal="<principal_distinguished_name>",credential-reference=<credential_reference>)
   ```

   Example

   ```
   /subsystem=elytron/dir-context=exampleDirContext:add(url="ldap://10.88.0.2",principal="cn=admin,dc=wildfly,dc=org",credential-reference={clear-text="secret"})
   ```

2. Create an LDAP realm that references the directory context. Specify the Search Base DN and how users are mapped.

   Syntax

   ```
   /subsystem=elytron/ldap-realm=<ldap_realm_name>add:(dir-context=<dir_context_name>,identity-mapping=search-base-dn="ou=<organization_unit>,dc=<domain_component>",rdn-identifier="<relative_distinguished_name_identifier>",user-password-mapper={from=<password_attribute_name>},attribute-mapping=[{filter-base-dn="ou=<organization_unit>,dc=<domain_component>",filter="<ldap_filter>",from="<ldap_attribute_name>",to="<identity_attribute_name>"}]})
   ```

   Example

   ```
   /subsystem=elytron/ldap-realm=exampleSecurityRealm:add(dir-context=exampleDirContext,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={1}))",from="cn",to="Roles"}]})
   ```

   If you store hashed passwords in the LDIF file, you can specify the following attributes:

   - **hash-encoding**: This attribute specifies the string format for the password if it is not stored in plain text. It is set to **base64** encoding by default, but **hex** is also supported.
   - **hash-charset**: This attribute specifies the character set to use when converting the password string to a byte array. It is set to **UTF-8** by default.

WARNING

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

3. Create a role decoder to map attributes to roles.

Syntax

```
/subsystem=elytron/simple-role-decoder=<role_decoder_name>:add(attribute=<attribute>)
```

Example

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

4. Create a security domain that references the LDAP realm and the role decoder.

Syntax

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

Example

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

You now can use the created security domain to add authentication and authorization to management interfaces and applications. For more information, see Securing management interfaces and applications.

Additional resources

- **ldap-realm** attributes
- **security-domain** attributes

1.4. CREATING A PROPERTIES REALM

1.4.1. Create a security domain referencing a properties-realm in Elytron

Create a **properties-realm** and a security domain that references the realm to secure your JBoss EAP management interfaces or the applications that you deployed on the server.

Prerequisites

- JBoss EAP is running.
- You have an authorized user and an existing legacy properties file with the correct realm written in the commented out line in the **users.properties** file:

  Example $EAP_HOME/standalone/configuration/my-example-users.properties

  ```
  #$REALM_NAME=exampleSecurityRealm$
  user1=078ed9776d4b8e63b6e51135ec45cc75
  ```

  - The password for **user1** is **userPassword1**. The password is hashed to the file as **HEX(MD5(user1:exampleSecurityRealm:userPassword1))**.
- The authorized user listed in your **users.properties** file has a role in the **groups.properties** file:

  Example $EAP_HOME/standalone/configuration/my-example-groups.properties

  ```
  user1=Admin
  ```

Procedure

1. Create a **properties-realm** in Elytron.

   Syntax

   ```
   /subsystem=elytron/properties-realm=<properties_realm_name>:add(users-properties=
   {path=<file_path>},groups-properties={path=<file_path>})
   ```

   Example

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

2. Create a security domain that references the **properties-realm**.

   Syntax

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

   Example

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

Verification

- To verify that Elytron can load data from the properties file, use the following command:

Syntax

```
/subsystem=elytron/security-domain=<security_domain_name>:read-
identity(name=<username>)
```

Example

```
/subsystem=elytron/security-domain=exampleSecurityDomain:read-identity(name=user1)
{
    "outcome" => "success",
    "result" => {
        "name" => "user1",
        "attributes" => {"Roles" => ["Admin"]},
        "roles" => ["Admin"]
    }
}
```

The output confirms that Elytron can load data from the properties file.

You now can use the created security domain to add authentication and authorization to management interfaces and applications. For more information, see Securing management interfaces and applications.

Additional resources

- **properties-realm** attributes
- **security-domain** attributes
- **simple-role-decoder** attributes

1.5. CREATING A CUSTOM REALM

1.5.1. Adding a custom-realm security realm in Elytron

You can use a **custom-realm** to create an Elytron security realm that is tailored to your use case. You can add a **custom-realm** when existing Elytron security realms do not suit your use case.

Prerequisites

- JBoss EAP is installed and running.
- Maven is installed.
- You have an implemented custom realm java class.

Procedure

1. Implement a custom realm java class and package it as a **JAR** file.

```
$ mvn package
```

2. Add a module containing your custom realm implementation.

Syntax

```
module add --name=<name_of_your_wildfly_module>
--resources=<path_to_custom_realm_jar> --dependencies=org.wildfly.security.elytron
```

Example

```
module add --name=com.example.customrealm --resources=EAP_HOME/custom-realm.jar --dependencies=org.wildfly.security.elytron
```

3. Create your **custom-realm**.

Syntax

```
/subsystem=elytron/custom-realm=<name_of_your_custom_realm>:add(module=<name_of_your_wildfly_module>,class-name=<class_name_of_custom_realm_>,configuration={<configuration_option_1>=<configuration_value_1>,<configuration_option_2>=<configuration_value_2>})
```

Example

```
/subsystem=elytron/custom-realm=example-realm:add(module=com.example.customrealm,class-name=com.example.customrealm.ExampleRealm,configuration={exampleConfigOption1=exampleConfigValue1,exampleConfigOption2=exampleConfigValue2})
```

NOTE

This example expects that the implemented custom realm has the class name **com.example.customrealm.ExampleRealm**.

NOTE

You can use the **configuration** attribute to pass **key**/**value** configuration to the **custom-realm**. The **configuration** attribute is optional.

4. Define a security domain based on the realm that you created.

Syntax

```
/subsystem=elytron/security-domain=<your_security_domain_name>:add(realms=[{realm=<your_realm_name>}],default-realm=<your_realm_name>,permission-mapper=<your_permission_mapper_name>)
```

Example

```
/subsystem=elytron/security-domain=exampleSecurityDomain:add(realms=[{realm=example-realm}],default-realm=example-realm,permission-mapper=default-permission-mapper)
```

You now can use the created security domain to add authentication and authorization to management interfaces and applications. For more information, see Securing management interfaces and applications.

Additional resources

- **custom-realm** attributes
- **security-domain** attributes
- To learn more about the **module add** command, you can run the **--help** command in the JBoss EAP management CLI:

```
module add --help
```

CHAPTER 2. SECURING MANAGEMENT INTERFACES AND APPLICATIONS

2.1. ADDING AUTHENTICATION AND AUTHORIZATION TO MANAGEMENT INTERFACES

You can add authentication and authorization for management interfaces to secure them by using a security domain. To access the management interfaces after you add authentication and authorization, users must enter login credentials.

You can secure JBoss EAP management interfaces as follows:

- Management CLI
  By configuring a **sasl-authentication-factory**.
- Management console
  By configuring an **http-authentication-factory**.

Prerequisites

- You have created a security domain referencing a security realm.
- JBoss EAP is running.

Procedure

1. Create an **http-authentication-factory**, or a **sasl-authentication-factory**.
   - Create an **http-authentication-factory**.

     Syntax

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

     Example

     ```
     /subsystem=elytron/http-authentication-factory=exampleAuthenticationFactory:add(http-server-mechanism-factory=global, security-domain=exampleSecurityDomain, mechanism-configurations=[{mechanism-name=BASIC, mechanism-realm-configurations=[{realm-name=exampleSecurityRealm}]}])
     {"outcome" => "success"}
     ```

   - Create a **sasl-authentication-factory**.

     Syntax

     ```
     /subsystem=elytron/sasl-authentication-factory=<sasl_authentication_factory_name>:add(security-domain=<security_domain>,sasl-server-factory=configured,mechanism-configurations=
     ```

```
[{mechanism-name=<mechanism-name>,mechanism-realm-configurations=[{realm-name=<realm_name>}]}])
```

Example

```
/subsystem=elytron/sasl-authentication-factory=exampleSaslAuthenticationFactory:add(security-domain=exampleSecurityDomain,sasl-server-factory=configured,mechanism-configurations=[{mechanism-name=PLAIN,mechanism-realm-configurations=[{realm-name=exampleSecurityRealm}]}])
{"outcome" => "success"}
```

2. Update the management interfaces.

- Use the **http-authentication-factory** to secure the management console.

Syntax

```
/core-service=management/management-interface=http-interface:write-attribute(name=http-authentication-factory, value=<authentication_factory_name>)
```

Example

```
/core-service=management/management-interface=http-interface:write-attribute(name=http-authentication-factory, value=exampleAuthenticationFactory)
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    }
}
```

- Use the **sasl-authentication-factory** to secure the management CLI.

Syntax

```
/core-service=management/management-interface=http-interface:write-attribute(name=http-upgrade,value={enabled=true,sasl-authentication-factory=<sasl_authentication_factory>})
```

Example

```
/core-service=management/management-interface=http-interface:write-attribute(name=http-upgrade,value={enabled=true,sasl-authentication-factory=exampleSaslAuthenticationFactory})
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    }
}
```

3. Reload the server.

```
reload
```

Verification

- To verify that the management console requires authentication and authorization, navigate to the management console at **http://127.0.0.1:9990/console/index.html**.
  You are prompted to enter user name and password.
- To verify that the management CLI requires authentication and authorization, start the management CLI using the following command:

```
$ bin/jboss-cli.sh --connect
```

  You are prompted to enter user name and password.

Additional resources

- **http-authentication-factory** attributes
- **sasl-authentication-factory** attributes

2.2. USING A SECURITY DOMAIN TO AUTHENTICATE AND AUTHORIZE APPLICATION USERS

Use a security domain that references a security realm to authenticate and authorize application users. The procedures for developing an application are provided only as an example.

2.2.1. Developing a simple web application

You can create a simple web application to follow along with the configuring security realms examples.

NOTE

The following procedures are provided as an example only. If you already have an application that you want to secure, you can skip these and go directly to Adding authentication and authorization to applications

2.2.1.1. Creating a Maven project for web-application development

For creating a web-application, create a Maven project with the required dependencies and the directory structure.

Prerequisites

- You have installed Maven. For more information, see Downloading Apache Maven.

Procedure

1. Set up a Maven project using the **mvn** command. The command creates the directory structure for the project and the **pom.xml** configuration file.

Syntax

```
$ mvn archetype:generate \
-DgroupId=${group-to-which-your-application-belongs} \
-DartifactId=${name-of-your-application} \
-DarchetypeGroupId=org.apache.maven.archetypes \
-DarchetypeArtifactId=maven-archetype-webapp \
-DinteractiveMode=false
```

Example

```
$ mvn archetype:generate \
-DgroupId=com.example.app \
-DartifactId=simple-webapp-example \
-DarchetypeGroupId=org.apache.maven.archetypes \
-DarchetypeArtifactId=maven-archetype-webapp \
-DinteractiveMode=false
```

2. Navigate to the application root directory:

Syntax

```
$ cd <name-of-your-application>
```

Example

```
$ cd simple-webapp-example
```

3. Replace the content of the generated **pom.xml** file with the following text:

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example.app</groupId>
  <artifactId>simple-webapp-example</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>

  <name>simple-webapp-example Maven Webapp</name>
  <!-- FIXME change it to the project's website -->
  <url>http://www.example.com</url>

 <properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>11</maven.compiler.source>
  <maven.compiler.target>11</maven.compiler.target>
 </properties>
```

```
<dependencies>
 <dependency>
  <groupId>jakarta.servlet</groupId>
  <artifactId>jakarta.servlet-api</artifactId>
  <version>6.0.0</version>
  <scope>provided</scope>
 </dependency>
</dependencies>

<build>
   <finalName>${project.artifactId}</finalName>
   <plugins>
      <plugin>
         <groupId>org.wildfly.plugins</groupId>
         <artifactId>wildfly-maven-plugin</artifactId>
         <version>2.1.0.Final</version>
      </plugin>
   </plugins>
</build>
</project>
```

Verification

- In the application root directory, enter the following command:

```
$ mvn install
```

You get an output similar to the following:

```
...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 0.795 s
[INFO] Finished at: 2022-04-28T17:39:48+05:30
[INFO] ------------------------------------------------------------------------
```

You can now create a web-application.

2.2.1.2. Creating a web application

Create a web application containing a servlet that returns the user name obtained from the logged-in user's principal. If there is no logged-in user, the servlet returns the text "NO AUTHENTICATED USER".

In this procedure, *<application_home>* refers to the directory that contains the **pom.xml** configuration file for the application.

Prerequisites

- You have created a Maven project.
  For more information, see Creating a Maven project for web-application development .
- JBoss EAP is running.

Procedure

1. Create a directory to store the Java files.

   Syntax

   ```
   $ mkdir -p src/main/java/<path_based_on_artifactID>
   ```

   Example

   ```
   $ mkdir -p src/main/java/com/example/app
   ```

2. Navigate to the new directory.

   Syntax

   ```
   $ cd src/main/java/<path_based_on_artifactID>
   ```

   Example

   ```
   $ cd src/main/java/com/example/app
   ```

3. Create a file **SecuredServlet.java** with the following content:

```
package com.example.app;

import java.io.IOException;
import java.io.PrintWriter;
import java.security.Principal;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

/**
 * A simple secured HTTP servlet. It returns the user name of obtained
 * from the logged-in user's Principal. If there is no logged-in user,
 * it returns the text "NO AUTHENTICATED USER".
 */

@WebServlet("/secured")
public class SecuredServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        try (PrintWriter writer = resp.getWriter()) {
            writer.println("<html>");
            writer.println("  <head><title>Secured Servlet</title></head>");
            writer.println("  <body>");
            writer.println("    <h1>Secured Servlet</h1>");
            writer.println("    <p>");
```

```
            writer.print(" Current Principal '");
            Principal user = req.getUserPrincipal();
            writer.print(user != null ? user.getName() : "NO AUTHENTICATED USER");
            writer.print("'");
            writer.println("    </p>");
            writer.println("  </body>");
            writer.println("</html>");
        }
    }

}
```

4. In the application root directory, compile your application with the following command:

```
$ mvn package
...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.015 s
[INFO] Finished at: 2022-04-28T17:48:53+05:30
[INFO] ------------------------------------------------------------------------
```

5. Deploy the application.

```
$ mvn wildfly:deploy
```

Verification

- In a browser, navigate to **http://localhost:8080/simple-webapp-example/secured**.
  You get the following message:

```
Secured Servlet
Current Principal 'NO AUTHENTICATED USER'
```

  Because no authentication mechanism is added, you can access the application.

You can now secure this application by using a security domain so that only authenticated users can access it.

2.2.2. Adding authentication and authorization to applications

You can add authentication and authorization to web applications to secure them by using a security domain. To access the web applications after you add authentication and authorization, users must enter login credentials.

Prerequisites

- You have created a security domain referencing a security realm.
- You have deployed applications on JBoss EAP.
- JBoss EAP is running.

Procedure

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

   Syntax

   ```
   /subsystem=undertow/application-security-domain=<application_security_domain_name>:add(security-domain=<security_domain_name>)
   ```

   Example

   ```
   /subsystem=undertow/application-security-domain=exampleApplicationSecurityDomain:add(security-domain=exampleSecurityDomain)
   {"outcome" => "success"}
   ```

2. Configure the application's **web.xml** to protect the application resources.

   Syntax

   ```
   <!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd" >

   <web-app>

    <!-- Define the security constraints for the application resources.
        Specify the URL pattern for which a challenge is -->

    <security-constraint>
        <web-resource-collection>
            <web-resource-name><!-- Name of the resources to protect --></web-resource-name>
            <url-pattern> <!-- The URL to protect  --></url-pattern>
        </web-resource-collection>

        <!-- Define the role that can access the protected resource -->
        <auth-constraint>
            <role-name> <!-- Role name as defined in the security domain --></role-name>
            <!-- To disable authentication you can use the wildcard *
              To authenticate but allow any role, use the wildcard **. -->
        </auth-constraint>
    </security-constraint>

    <login-config>
        <auth-method>
         <!-- The authentication method to use. Can be:
          BASIC
          CLIENT-CERT
          DIGEST
          FORM
          SPNEGO
          -->
        </auth-method>
   ```

```
    <realm-name><!-- The name of realm to send in the challenge  --></realm-name>
  </login-config>
</web-app>
```

Example

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>

 <!-- Define the security constraints for the application resources.
     Specify the URL pattern for which a challenge is -->

 <security-constraint>
      <web-resource-collection>
         <web-resource-name>all</web-resource-name>
         <url-pattern>/*</url-pattern>
      </web-resource-collection>

      <!-- Define the role that can access the protected resource -->
      <auth-constraint>
         <role-name>Admin</role-name>
         <!-- To disable authentication you can use the wildcard *
           To authenticate but allow any role, use the wildcard **. -->
      </auth-constraint>
   </security-constraint>

   <login-config>
      <auth-method>BASIC</auth-method>

      <realm-name>exampleSecurityRealm</realm-name>
   </login-config>
</web-app>
```

NOTE

You can use a different **auth-method**.

3. Configure your application to use a security domain by either creating a **jboss-web.xml** file in your application or setting the default security domain in the **undertow** subsystem.

   - Create **jboss-web.xml** file in the your application's **WEB-INF** directory referencing the **application-security-domain**.

     Syntax

     ```
     <jboss-web>
       <security-domain> <!-- The security domain to associate with the application --
     ></security-domain>
     </jboss-web>
     ```

     Example

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

- Set the default security domain in the **undertow** subsystem for applications.

  Syntax

  ```
  /subsystem=undertow:write-attribute(name=default-security-
  domain,value=<application_security_domain_to_use>)
  ```

  Example

  ```
  /subsystem=undertow:write-attribute(name=default-security-
  domain,value=exampleApplicationSecurityDomain)
  {
      "outcome" => "success",
      "response-headers" => {
          "operation-requires-reload" => true,
          "process-state" => "reload-required"
      }
  }
  ```

4. Reload the server.

   ```
   reload
   ```

Verification

1. In the application root directory, compile your application with the following command:

   ```
   $ mvn package
   ...
   [INFO] ------------------------------------------------------------------------
   [INFO] BUILD SUCCESS
   [INFO] ------------------------------------------------------------------------
   [INFO] Total time: 1.015 s
   [INFO] Finished at: 2022-04-28T17:48:53+05:30
   [INFO] ------------------------------------------------------------------------
   ```

2. Deploy the application.

   ```
   $ mvn wildfly:deploy
   ```

3. In a browser, navigate to **http://localhost:8080/simple-webapp-example/secured**. You get a login prompt confirming that authentication is now required to access the application.

Your application is now secured with a security domain and users can log in only after authenticating. Additionally, only users with specified roles can access the application.

CHAPTER 3. CONFIGURING ELYTRON WITH IDENTITY REALM TO ALLOW EASY AUTHENTICATION AND AUTHORIZATION FOR LOCAL USERS

You can use an **identity-realm** provided by Elytron to allow local users to connect to JBoss EAP management interfaces.

The JBoss EAP management CLI is preconfigured to use an **identity-realm** named *local*. This allows local users to connect without having to provide credentials. An identity realm can only be used with the *JBOSS-LOCAL-USER* mechanism.

3.1. SECURING A MANAGEMENT INTERFACE WITH AN IDENTITY REALM

You can secure a management interface by using an **identity-realm** security realm with the *JBOSS-LOCAL-USER* mechanism.

Prerequisites

- JBoss EAP is running.

Procedure

1. Create a local **identity-realm**.

   #### Syntax

   ```
   /subsystem=elytron/identity-realm=
   <local_identity_realm_name>:add(identity="$local",attribute-
   name=<attribute_name>,attribute-values=<attribute_value>)
   ```

   #### Example

   ```
   /subsystem=elytron/identity-
   realm=exampleLocalIdentityRealm:add(identity="$local",attribute-
   name=AttributeName,attribute-values=Value)
   ```

   a. **Optional** If you want to use a name for your local **identity-realm** other than *$local*, change the value of **wildfly.sasl.local-user.default-user** property in the attribute **configurable-sasl-server-factory=*<sasl_server_factory>***.

      #### Syntax

      ```
      /subsystem=elytron/configurable-sasl-server-factory=<sasl_server_factory>:write-
      attribute(name=properties,value={"wildfly.sasl.local-user.default-user" =>
      "<new_local_username>", "wildfly.sasl.local-user.challenge-path" => expression
      "${jboss.server.temp.dir}/auth"})
      ```

      #### Example

```
/subsystem=elytron/configurable-sasl-server-factory=configured:write-
attribute(name=properties,value={"wildfly.sasl.local-user.default-user" => "john",
"wildfly.sasl.local-user.challenge-path" => expression "${jboss.server.temp.dir}/auth"})
```

2. Create a security domain that references the **identity-realm** that you created.

Syntax

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

Example

```
/subsystem=elytron/security-domain=exampleSecurityDomain:add(default-
realm=exampleLocalIdentityRealm,permission-mapper=default-permission-mapper,realms=
[{realm=exampleLocalIdentityRealm}])
```

3. Add SASL Authentication factory.

Syntax

```
/subsystem=elytron/sasl-authentication-factory=<sasl_auth_factory_name>:add(security-
domain=<security_domain_name>,sasl-server-factory=configured,mechanism-
configurations=[{mechanism-name=JBOSS-LOCAL-USER}])
```

Example

```
/subsystem=elytron/sasl-authentication-
factory=exampleSaslAuthenticationFactory:add(security-
domain=exampleSecurityDomain,sasl-server-factory=configured,mechanism-configurations=
[{mechanism-name=JBOSS-LOCAL-USER}])
```

4. Enable SASL Authentication factory for your management interface.

Syntax

```
/core-service=management/management-interface=http-interface:write-attribute(name=http-
upgrade,value={enabled=true,sasl-authentication-factory=<sasl_auth_factory_name>})
```

Example

```
/core-service=management/management-interface=http-interface:write-attribute(name=http-
upgrade,value={enabled=true,sasl-authentication-
factory=exampleSaslAuthenticationFactory})
```

5. Reload your management interface.

```
$ reload
```

Additional resources

- **identity-realm** attributes
- **security-domain** attributes
- **sasl-authentication-factory** attributes

CHAPTER 4. REFERENCE

4.1. CUSTOM-REALM ATTRIBUTES

You can configure your **custom-realm** by setting its attributes.

Table 4.1. custom-realm attributes

Attribute	Description
class-name	Fully qualified class name of the implementation of the custom realm.
configuration	The optional **key**/**value** configuration for the custom realm.
module	Name of the module to use to load the custom realm.

4.2. FILESYSTEM-REALM ATTRIBUTES

You can configure **filesystem-realm** by setting its attributes.

Table 4.2. filesystem-realm attributes

Attribute	Description
credential-store	Reference to the credential store that contains the secret key to encrypt and decrypt the clear passwords, hashed passwords, and attributes in the realm. When you use this attribute, you must also specify the secret key to use by defining it in the **secret-key** attribute.
encoded	The attribute that indicates whether the identity names should be stored encoded (Base32) in file names. The default value is **true**.
hash-charset	The character set to use when converting the password string to a byte array. The default is UTF-8.
hash-encoding	The string format for the password if it is not stored in plain text. It can be one of: • base64 • hex The default is base64.
key-store	Reference to the key store that contains the key pair to use to verify integrity. When you define this attribute, you must also specify the key store alias in the **key-store-alias** attribute.

Attribute	Description
key-store-alias	The alias that identifies the private key entry within the key store to use to verify integrity. Use this attribute if you have added a reference to a key store by defining the **key-store** attribute.
levels	The number of levels of directory hashing to apply. The default value is **2**.
path	The path to the directory containing the realm.
relative-to	The predefined relative path to use with **path**. For example **jboss.server.config.dir**.
secret-key	The alias of the secret key to encrypt and decrypt the clear passwords, hashed passwords, and attributes in the realm. Use this attribute if you have added a reference to a credential store by defining the **credential-store** attribute.

4.3. HTTP-AUTHENTICATION-FACTORY ATTRIBUTES

You can configure **http-authentication-factory** by setting its attributes.

Table 4.3. http-authentication-factory attributes

Attribute	Description
http-server-mechanism-factory	The **HttpServerAuthenticationMechanismFactory** to associate with this resource.
mechanism-configurations	The list of mechanism-specific configurations.
security-domain	The security domain to associate with the resource.

Table 4.4. http-authentication-factory mechanism-configurations attributes

Attribute	Description
credential-security-factory	The security factory to use to obtain a credential as required by the mechanism.
final-principal-transformer	A final principal transformer to apply for this mechanism realm.
host-name	The host name this configuration applies to.
mechanism-name	This configuration will only apply where a mechanism with the name specified is used. If this attribute is omitted then this will match any mechanism name.

Attribute	Description
mechanism-realm-configurations	The list of definitions of the realm names as understood by the mechanism.
pre-realm-principal-transformer	A principal transformer to apply before the realm is selected.
post-realm-principal-transformer	A principal transformer to apply after the realm is selected.
protocol	The protocol this configuration applies to.
realm-mapper	The realm mapper to be used by the mechanism.

Table 4.5. http-authentication-factory mechanism-configurations mechanism-realm-configurations attributes

Attribute	Description
final-principal-transformer	A final principal transformer to apply for this mechanism realm.
post-realm-principal-transformer	A principal transformer to apply after the realm is selected.
pre-realm-principal-transformer	A principal transformer to apply before the realm is selected.
realm-mapper	The realm mapper to be used by the mechanism.
realm-name	The name of the realm to be presented by the mechanism.

4.4. IDENTITY-REALM ATTRIBUTES

You can configure your **identity-realm** by setting its attributes.

Table 4.6. identity-realm attributes

Attribute	Description
attribute-name	The name of the attribute associated with this identity.
attribute-values	The list of values associated with the identity's attribute.
identity	The identity available from the security realm.

4.5. JDBC-REALM ATTRIBUTES

You can configure **jdbc-realm** by setting its attributes.

Table 4.7. jdbc-realm Attributes

Attribute	Description
hash-charset	The character set to use when converting the password string to a byte array. The default is UTF-8.
principal-query	The list of authentication queries used to authenticate users based on specific key types.

Table 4.8. jdbc-realm principal-query Attributes

Attribute	Description
attribute-mapping	The list of attribute mappings defined for this resource.
bcrypt-mapper	A key mapper that maps a column returned from a SQL query to a **Bcrypt** key type.
clear-password-mapper	A key mapper that maps a column returned from a SQL query to a clear password key type. This has a **password-index** child element that is the column index from an authentication query that represents the user password.
data-source	The name of the data source used to connect to the database.
salted-simple-digest-mapper	A key mapper that maps a column returned from a SQL query to a **Salted Simple Digest** key type.
scram-mapper	A key mapper that maps a column returned from a SQL query to a **SCRAM** key type.
simple-digest-mapper	A key mapper that maps a column returned from a SQL query to a **Simple Digest** key type.
sql	The SQL statement used to obtain the keys as table columns for a specific user and map them accordingly with their type.

Table 4.9. jdbc-realm principal-query attribute-mapping Attributes

Attribute	Description
index	The column index from the SQL query that represents the mapped attribute.

Attribute	Description
to	The name of the identity attribute mapped from a column returned from the SQL query.

Additional resources

- Password mapper attributes

4.6. LDAP-REALM ATTRIBUTES

You can configure **ldap-realm** by setting its attributes.

Table 4.10. ldap-realm attributes

Attribute	Description
allow-blank-password	Whether this realm supports blank password direct verification. If this attribute is not set, a blank password attempt is rejected.
dir-context	The name of the **dir-context** which will be used to connect to the LDAP server.
direct-verification	If this attribute is set to **true**, this realm supports verification of credentials by directly connecting to LDAP as the account being authenticated. Otherwise, the password is retrieved from the LDAP server and verified in JBoss EAP. If enabled, the JBoss EAP server must be able to obtain the plain user password from the client, which requires either the **PLAIN** SASL or **BASIC** HTTP mechanism to be used for authentication. Defaults to **false**.
hash-charset	The character set to use when converting the password string to a byte array. The default is UTF-8.
hash-encoding	The string format for the password if it is not stored in plain text. It can be one of: • base64 • hex The default is base64.
identity-mapping	The configuration options that define how principals are mapped to their corresponding entries in the underlying LDAP server.

Table 4.11. ldap-realm identity-mapping attributes

Attribute	Description
attribute-mapping	List of attribute mappings defined for this resource.
filter-name	The LDAP filter for getting identity by name.
iterator-filter	The LDAP filter for iterating over identities of the realm.
new-identity-attributes	The list of attributes of newly created identities. It is required for the modifiability of the realm. This is a list of **name** and **value** pair objects.
new-identity-parent-dn	The DN of the parent of newly created identities. Required for modifiability of the realm.
otp-credential-mapper	The credential mapping for OTP credential.
rdn-identifier	The RDN part of the principal's DN to be used to obtain the principal's name from an LDAP entry. This is also used when creating new identities.
search-base-dn	The base DN to search for identities.
use-recursive-search	If this attribute is set to **true**, identity search queries are recursive. Defaults to **false**.
user-password-mapper	The credential mapping for a credential, similar to userPassword.
x509-credential-mapper	The configuration that enables using LDAP as storage of X509 credentials. If none of the **-from** child attributes are defined, then this configuration will be ignored. If more than one **-from** child attribute is defined, then the user certificate must match all the defined criteria.

Table 4.12. ldap-realm identity-mapping attribute-mapping attributes

Attribute	Description
extract-rdn	The RDN key to use as the value for an attribute, in case the value in its raw form is in X.500 format.
filter	The filter to use to obtain the values for a specific attribute. The string **{0}** will be replaced by username and`{1}` by user identity DN.
filter-base-dn	The name of the context where the filter should be performed.

Attribute	Description
from	The name of the LDAP attribute to map to an identity attribute. If not defined, DN of entry is used.
reference	The name of the LDAP attribute containing DN of entry to obtain value from.
role-recursion	Maximum depth for recursive role assignment. Use **0** to specify no recursion. Defaults to **0**.
role-recursion-name	Determine the LDAP attribute of role entry which will be a substitute for "{0}" in **filter-name** when searching roles of role.
search-recursive	If **true** attribute LDAP search queries are recursive. Defaults to **true**.
to	The name of the identity attribute mapped from a specific LDAP attribute. If not provided, the name of the attribute is the same as defined in **from**. If **from** is not defined too, the **dn** value is used.

Table 4.13. ldap-realm identity-mapping user-password-mapper attributes

Attribute	Description
from	The name of the LDAP attribute to map to an identity attribute. If not defined, DN of entry is used.
verifiable	If **true** password can be used to verify the user. Defaults to **true**.
writable	If **true** password can be changed. Defaults to **false**.

Table 4.14. ldap-realm identity-mapping otp-credential-mapper Attributes

Attribute	Description
algorithm-from	The name of the LDAP attribute of OTP algorithm.
hash-from	The name of the LDAP attribute of OTP hash function.
seed-from	The name of the LDAP attribute of OTP seed.
sequence-from	The name of the LDAP attribute of OTP sequence number.

Table 4.15. ldap-realm identity-mapping x509-credential-mapper attributes

Attribute	Description
certificate-from	The name of the LDAP attribute to map to an encoded user certificate. If not defined, the encoded certificate will not be checked.
digest-algorithm	The digest algorithm, which is the hash function, that is used to compute the digest of the user certificate. It will be used only if **digest-from** has been defined.
digest-from	The name of the LDAP attribute to map to a user certificate digest. If not defined, the certificate digest will not be checked.
serial-number-from	The name of the LDAP attribute to map to a serial number of the user certificate. If not defined, the serial number will not be checked.
subject-dn-from	The name of the LDAP attribute to map to a subject DN of user certificate. If not defined, the subject DN will not be checked.

4.7. PASSWORD MAPPER ATTRIBUTES

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 4.16. 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 supports multiple pieces of information to be encoded in a single string. The information can include the following: • password type • hash or digest • salt • iteration count

4.8. PROPERTIES-REALM ATTRIBUTES

You can configure **properties-realm** by setting its attributes.

Table 4.17. properties-realm attributes

Attribute	Description
groups-attribute	The name of the attribute in the returned **AuthorizationIdentity** that should contain the group membership information for the identity.
groups-properties	The properties file containing the users and their groups.
hash-charset	Specifies the name of the character set to use when converting the client provided password string to a byte array for hashing calculations. Set to **UTF-8** by default.
hash-encoding	Specifies the string format for the hashed password if the password is not being stored in plain text. It may specify one of two: **hex** or **base64**. Set to **hex** by default for **properties-realm**.
users-properties	The properties file containing the users and their passwords.

Table 4.18. properties-realm users-properties attributes

Attribute	Description
digest-realm-name	The default realm name to use for digested passwords if one is not discovered in the properties file.
path	The path to the file containing the users and their passwords. The file should contain realm name declaration.
plain-text	If **true**, the passwords in properties file stored in plain text. If **false**, they are pre-hashed, taking the form of **HEX(MD5(username ':' realm ':' password))**. Defaults to **false**.
relative-to	The predefined path that the path is relative to.

Table 4.19. properties-realm groups-properties attributes

Attribute	Description
path	The path to the file containing the users and their groups.
relative-to	The predefined path that the path is relative to.

4.9. SASL-AUTHENTICATION-FACTORY ATTRIBUTES

You can configure **sasl-authentication-factory** by setting its attributes.

Table 4.20. sasl-authentication-factory attributes

Attribute	Description
mechanism-configurations	The list of mechanism specific configurations.
sasl-server-factory	The SASL server factory to associate with this resource.
security-domain	The security domain to associate with this resource.

Table 4.21. sasl-authentication-factory mechanism-configurations attributes

Attribute	Description
credential-security-factory	The security factory to use to obtain a credential as required by the mechanism.
final-principal-transformer	A final principal transformer to apply for this mechanism realm.
host-name	The host name this configuration applies to.
mechanism-name	This configuration will only apply where a mechanism with the name specified is used. If this attribute is omitted then this will match any mechanism name.
mechanism-realm-configurations	The list of definitions of the realm names as understood by the mechanism.
protocol	The protocol this configuration applies to.
post-realm-principal-transformer	A principal transformer to apply after the realm is selected.
pre-realm-principal-transformer	A principal transformer to apply before the realm is selected.
realm-mapper	The realm mapper to be used by the mechanism.

Table 4.22. sasl-authentication-factory mechanism-configurations mechanism-realm-configurations attributes

Attribute	Description
final-principal-transformer	A final principal transformer to apply for this mechanism realm.
post-realm-principal-transformer	A principal transformer to apply after the realm is selected.

Attribute	Description
pre-realm-principal-transformer	A principal transformer to apply before the realm is selected.
realm-mapper	The realm mapper to be used by the mechanism.
realm-name	The name of the realm to be presented by the mechanism.

4.10. SECRET-KEY-CREDENTIAL-STORE ATTRIBUTES

You can configure **secret-key-credential-store** by setting its attributes.

Table 4.23. secret-key-credential-store Attributes

Attribute	Description
create	Set the value to **false** if you do not want Elytron to create one if it doesn't already exist. Defaults to **true**.
default-alias	The alias name for a key generated by default. The default value is **key**.
key-size	The size of a generated key. The default size is 256 bits. You can set the value to one of the following: • **128** • **192** • **256**
path	The path to the credential store.
populate	If a credential store does not contain a **default-alias**, this attribute indicates whether Elytron should create one. The default is **true**.
relative-to	A reference to a previously defined path that the attribute **path** is relative to.

4.11. SECURITY-DOMAIN ATTRIBUTES

You can configure **security-domain** by setting its attributes.

Attribute	Description
default-realm	The default realm contained by this security domain.

Attribute	Description
evidence-decoder	A reference to an EvidenceDecoder to be used by this domain.
outflow-anonymous	This attribute specifies whether the anonymous identity should be used if outflow to a security domain is not possible. Outflowing anonymous identity has the effect of clearing any identity already established for that domain.
outflow-security-domains	The list of security domains that the security identity from this domain should automatically outflow to.
permission-mapper	A reference to a PermissionMapper to be used by this domain.
post-realm-principal-transformer	A reference to a principal transformer to be applied after the realm has operated on the supplied identity name.
pre-realm-principal-transformer	A reference to a principal transformer to be applied before the realm is selected.
principal-decoder	A reference to a PrincipalDecoder to be used by this domain.
realm-mapper	Reference to the RealmMapper to be used by this domain.
realms	The list of realms contained by this security domain.
role-decoder	Reference to the RoleDecoder to be used by this domain.
role-mapper	Reference to the RoleMapper to be used by this domain.
security-event-listener	Reference to a listener for security events.
trusted-security-domains	The list of security domains that are trusted by this security domain.

4.12. SIMPLE-ROLE-DECODER ATTRIBUTES

You can configure simple role decoder by setting its attribute.

Table 4.24. simple-role-decoder attributes

Attribute	Description
attribute	The name of the attribute from the identity to map directly to roles.