



Red Hat JBoss Enterprise Application Platform 6.4

Development Guide

For Use with Red Hat JBoss Enterprise Application Platform 6

Red Hat JBoss Enterprise Application Platform 6.4 Development Guide

For Use with Red Hat JBoss Enterprise Application Platform 6

Legal Notice

Copyright © 2017 Red Hat, Inc..

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](https://creativecommons.org/licenses/by-sa/3.0/). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This book provides references and examples for Java EE 6 developers using Red Hat JBoss Enterprise Application Platform 6 and its patch releases.

Table of Contents

CHAPTER 1. GET STARTED DEVELOPING APPLICATIONS	6
1.1. INTRODUCTION	6
1.2. PREREQUISITES	6
1.3. SET UP THE DEVELOPMENT ENVIRONMENT	9
1.4. RUN YOUR FIRST APPLICATION	16
CHAPTER 2. MAVEN GUIDE	36
2.1. LEARN ABOUT MAVEN	36
2.2. INSTALL MAVEN AND THE JBOSS MAVEN REPOSITORY	38
2.3. USE THE MAVEN REPOSITORY	42
2.4. UPGRADE THE MAVEN REPOSITORY	58
CHAPTER 3. CLASS LOADING AND MODULES	60
3.1. INTRODUCTION	60
3.2. ADD AN EXPLICIT MODULE DEPENDENCY TO A DEPLOYMENT	64
3.3. GENERATE MANIFEST.MF ENTRIES USING MAVEN	67
3.4. PREVENT A MODULE BEING IMPLICITLY LOADED	69
3.5. EXCLUDE A SUBSYSTEM FROM A DEPLOYMENT	70
3.6. USE THE CLASS LOADER PROGRAMMATICALLY IN A DEPLOYMENT	71
3.7. CLASS LOADING AND SUBDEPLOYMENTS	76
3.8. DEPLOY TAG LIBRARY DESCRIPTORS (TLDS) IN A CUSTOM MODULE	77
3.9. REFERENCE	78
CHAPTER 4. VALVES	84
4.1. ABOUT VALVES	84
4.2. ABOUT GLOBAL VALVES	84
4.3. ABOUT AUTHENTICATOR VALVES	84
4.4. CONFIGURE A WEB APPLICATION TO USE A VALVE	84
4.5. CONFIGURE A WEB APPLICATION TO USE AN AUTHENTICATOR VALVE	86
4.6. CREATE A CUSTOM VALVE	87
CHAPTER 5. LOGGING FOR DEVELOPERS	90
5.1. INTRODUCTION	90
5.2. LOGGING WITH THE JBOSS LOGGING FRAMEWORK	92
5.3. PER-DEPLOYMENT LOGGING	95
5.4. LOGGING PROFILES	96
CHAPTER 6. INTERNATIONALIZATION AND LOCALIZATION	98
6.1. INTRODUCTION	98
6.2. JBOSS LOGGING TOOLS	98
CHAPTER 7. REMOTE JNDI LOOKUP	117
7.1. REGISTERING OBJECTS TO JNDI	117
7.2. CONFIGURING A REMOTE JNDI CLIENT	117
CHAPTER 8. ENTERPRISE JAVABEANS	119
8.1. INTRODUCTION	119
8.2. CREATING ENTERPRISE BEAN PROJECTS	122
8.3. SESSION BEANS	132
8.4. MESSAGE-DRIVEN BEANS	136
8.5. INVOKING SESSION BEANS	143
8.6. CONTAINER INTERCEPTORS	154
8.7. CLUSTERED ENTERPRISE JAVABEANS	162

8.8. REFERENCE	171
CHAPTER 9. JBOSS MBEAN SERVICES	177
9.1. WRITING JBOSS MBEAN SERVICES	177
9.2. A STANDARD MBEAN EXAMPLE	177
9.3. DEPLOYING JBOSS MBEAN SERVICES	179
CHAPTER 10. CLUSTERING IN WEB APPLICATIONS	181
10.1. SESSION REPLICATION	181
10.2. HTTPSESSION PASSIVATION AND ACTIVATION	185
10.3. IMPLEMENT AN HA SINGLETON	188
10.4. APACHE MOD_CLUSTER-MANAGER APPLICATION	193
CHAPTER 11. CDI	196
11.1. OVERVIEW OF CDI	196
11.2. USE CDI	197
CHAPTER 12. JAVA TRANSACTION API (JTA)	221
12.1. OVERVIEW	221
12.2. TRANSACTION CONCEPTS	221
12.3. TRANSACTION OPTIMIZATIONS	228
12.4. TRANSACTION OUTCOMES	232
12.5. OVERVIEW OF JTA TRANSACTIONS	234
12.6. TRANSACTION SUBSYSTEM CONFIGURATION	235
12.7. USE JTA TRANSACTIONS	255
12.8. ORB CONFIGURATION	266
12.9. TRANSACTION REFERENCES	268
CHAPTER 13. HIBERNATE	272
13.1. ABOUT HIBERNATE CORE	272
13.2. JAVA PERSISTENCE API (JPA)	272
13.3. HIBERNATE ANNOTATIONS	288
13.4. HIBERNATE QUERY LANGUAGE	293
13.5. HIBERNATE SERVICES	307
13.6. BEAN VALIDATION	313
13.7. ENVERS	318
13.8. PERFORMANCE TUNING	329
CHAPTER 14. HIBERNATE SEARCH	332
14.1. GETTING STARTED WITH HIBERNATE SEARCH	332
14.2. MAPPING ENTITIES TO THE INDEX STRUCTURE	338
14.3. QUERYING	367
14.4. MANUAL INDEX CHANGES	397
14.5. INDEX OPTIMIZATION	401
14.6. ADVANCED FEATURES	403
CHAPTER 15. JAX-RS WEB SERVICES	410
15.1. ABOUT JAX-RS	410
15.2. ABOUT RESTEASY	410
15.3. ABOUT RESTFUL WEB SERVICES	410
15.4. RESTEASY DEFINED ANNOTATIONS	410
15.5. RESTEASY CONFIGURATION	413
15.6. JAX-RS WEB SERVICE SECURITY	415
15.7. EXCEPTION HANDLING	417
15.8. RESTEASY INTERCEPTORS	420

15.9. STRING BASED ANNOTATIONS	426
15.10. CONFIGURE FILE EXTENSIONS	429
15.11. RESTEASY JAVASCRIPT API	431
15.12. RESTEASY ASYNCHRONOUS JOB SERVICE	435
15.13. RESTEASY JAXB	438
15.14. RESTEASY ATOM SUPPORT	442
15.15. YAML PROVIDER	444
15.16. EJB INTEGRATION	445
15.17. JSON SUPPORT VIA JACKSON	445
15.18. RESTEASY/SPRING INTEGRATION	446
CHAPTER 16. JAX-WS WEB SERVICES	448
16.1. ABOUT JAX-WS WEB SERVICES	448
16.2. CONFIGURE THE WEBSERVICES SUBSYSTEM	449
16.3. CONFIGURE THE HTTP TIMEOUT PER APPLICATION	452
16.4. JAX-WS WEB SERVICE ENDPOINTS	453
16.5. JAX-WS WEB SERVICE CLIENTS	457
16.6. JAX-WS DEVELOPMENT REFERENCE	467
CHAPTER 17. WEBSOCKETS	472
17.1. ABOUT WEBSOCKETS	472
17.2. CREATE A WEBSOCKET APPLICATION	472
CHAPTER 18. APPLICATION SECURITY	478
18.1. FOUNDATIONAL CONCEPTS	478
18.2. ROLE-BASED SECURITY IN APPLICATIONS	479
18.3. LOGIN MODULES	495
18.4. EJB APPLICATION SECURITY	529
18.5. JAX-RS APPLICATION SECURITY	548
18.6. PASSWORD VAULTS FOR SENSITIVE STRINGS	550
18.7. JAVA AUTHORIZATION CONTRACT FOR CONTAINERS (JACC)	571
18.8. JAVA AUTHENTICATION SPI FOR CONTAINERS (JASPI)	573
CHAPTER 19. SINGLE SIGN ON (SSO)	574
19.1. ABOUT SINGLE SIGN ON (SSO) FOR WEB APPLICATIONS	574
19.2. ABOUT CLUSTERED SINGLE SIGN ON (SSO) FOR WEB APPLICATIONS	574
19.3. CHOOSE THE RIGHT SSO IMPLEMENTATION	574
19.4. USE SINGLE SIGN ON (SSO) IN A WEB APPLICATION	575
19.5. ABOUT KERBEROS	577
19.6. ABOUT SPNEGO	577
19.7. ABOUT MICROSOFT ACTIVE DIRECTORY	578
19.8. CONFIGURE KERBEROS OR MICROSOFT ACTIVE DIRECTORY DESKTOP SSO FOR WEB APPLICATIONS	578
19.9. CONFIGURE SPNEGO FALL BACK TO FORM AUTHENTICATION	582
19.10. ABOUT SAML WEB BROWSER BASED SSO	583
19.11. COOKIE DOMAIN	583
CHAPTER 20. DEVELOPMENT SECURITY REFERENCES	586
20.1. EJB SECURITY PARAMETER REFERENCE	586
CHAPTER 21. CONFIGURATION REFERENCES	588
21.1. JBOSS-WEB.XML CONFIGURATION REFERENCE	588
CHAPTER 22. SUPPLEMENTAL REFERENCES	592
22.1. TYPES OF JAVA ARCHIVES	592

APPENDIX A. REVISION HISTORY 594

CHAPTER 1. GET STARTED DEVELOPING APPLICATIONS

1.1. INTRODUCTION

1.1.1. About Red Hat JBoss Enterprise Application Platform 6

Red Hat JBoss Enterprise Application Platform 6 (JBoss EAP 6) is a middleware platform built on open standards and compliant with the Java Enterprise Edition 6 specification. It integrates JBoss Application Server 7 with high-availability clustering, messaging, distributed caching, and other technologies.

JBoss EAP 6 includes a new, modular structure that allows service enabling only when required, improving startup speed.

The Management Console and Management Command Line Interface make editing XML configuration files unnecessary and add the ability to script and automate tasks.

In addition, JBoss EAP 6 includes APIs and development frameworks for quickly developing secure and scalable Java EE applications.

[Report a bug](#)

1.2. PREREQUISITES

1.2.1. Become Familiar with Java Enterprise Edition 6

1.2.1.1. Overview of EE 6 Profiles

Java Enterprise Edition 6 (EE 6) includes support for multiple profiles, or subsets of APIs. The only two profiles that the EE 6 specification defines are the *Full Profile* and the *Web Profile*.

EE 6 Full Profile includes all APIs and specifications included in the EE 6 specification. EE 6 Web Profile includes a subset of APIs which are useful to web developers.

JBoss EAP 6 is a certified implementation of the Java Enterprise Edition 6 Full Profile and Web Profile specifications.

- [Section 1.2.1.2, “Java Enterprise Edition 6 Web Profile”](#)
- [Section 1.2.1.3, “Java Enterprise Edition 6 Full Profile”](#)

[Report a bug](#)

1.2.1.2. Java Enterprise Edition 6 Web Profile

The Web Profile is one of two profiles defined by the Java Enterprise Edition 6 specification. It is designed for web application development. The other profile defined by the Java Enterprise Edition 6 specification is the Full Profile. See [Section 1.2.1.3, “Java Enterprise Edition 6 Full Profile”](#) for more details.

Java EE 6 Web Profile Requirements

- Java Platform, Enterprise Edition 6

- **Java Web Technologies**

- Servlet 3.0 (JSR 315)
- JSP 2.2 and Expression Language (EL) 1.2
- JavaServer Faces (JSF) 2.1 (JSR 314)
- Java Standard Tag Library (JSTL) for JSP 1.2
- Debugging Support for Other Languages 1.0 (JSR 45)

- **Enterprise Application Technologies**

- Contexts and Dependency Injection (CDI) (JSR 299)
- Dependency Injection for Java (JSR 330)
- Enterprise JavaBeans 3.1 Lite (JSR 318)
- Java Persistence API 2.0 (JSR 317)
- Common Annotations for the Java Platform 1.1 (JSR 250)
- Java Transaction API (JTA) 1.1 (JSR 907)
- Bean Validation (JSR 303)

[Report a bug](#)

1.2.1.3. Java Enterprise Edition 6 Full Profile

The Java Enterprise Edition 6 (EE 6) specification defines a concept of profiles, and defines two of them as part of the specification. Besides the items supported in the Java Enterprise Edition 6 Web Profile ([Section 1.2.1.2, “Java Enterprise Edition 6 Web Profile”](#)), the Full Profile supports the following APIs.

Items Included in the EE 6 Full Profile

- EJB 3.1 (not Lite) (JSR 318)
- Java EE Connector Architecture 1.6 (JSR 322)
- Java Message Service (JMS) API 1.1 (JSR 914)
- JavaMail 1.4 (JSR 919)
- **Web Service Technologies**
 - Jax-RS RESTful Web Services 1.1 (JSR 311)
 - Implementing Enterprise Web Services 1.3 (JSR 109)
 - JAX-WS Java API for XML-Based Web Services 2.2 (JSR 224)
 - Java Architecture for XML Binding (JAXB) 2.2 (JSR 222)
 - Web Services Metadata for the Java Platform (JSR 181)

- Java APIs for XML-based RPC 1.1 (JSR 101)
- Java APIs for XML Messaging 1.3 (JSR 67)
- Java API for XML Registries (JAXR) 1.0 (JSR 93)
- **Management and Security Technologies**
 - Java Authentication Service Provider Interface for Containers 1.0 (JSR 196)
 - Java Authentication Contract for Containers 1.3 (JSR 115)
 - Java EE Application Deployment 1.2 (JSR 88)
 - J2EE Management 1.1 (JSR 77)

[Report a bug](#)

1.2.2. About Modules and the New Modular Class Loading System used in JBoss EAP 6

1.2.2.1. Modules

A Module is a logical grouping of classes used for class loading and dependency management. JBoss EAP 6 identifies two different types of modules, sometimes called static and dynamic modules. However the only difference between the two is how they are packaged.

Static Modules

Static Modules are predefined in the **EAP_HOME/modules/** directory of the application server. Each sub-directory represents one module and defines a **main/** subdirectory that contains a configuration file (**module.xml**) and any required JAR files. The name of the module is defined in the **module.xml** file. All the application server provided APIs are provided as static modules, including the Java EE APIs as well as other APIs such as JBoss Logging.

Example 1.1. Example module.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.0" name="com.mysql">
  <resources>
    <resource-root path="mysql-connector-java-5.1.15.jar"/>
  </resources>
  <dependencies>
    <module name="javax.api"/>
    <module name="javax.transaction.api"/>
  </dependencies>
</module>
```

The module name, **com.mysql**, should match the directory structure for the module, excluding the **main/** subdirectory name.

The modules provided in JBoss EAP distributions are located in a **system** directory within the **EAP_HOME/modules** directory. This keeps them separate from any modules provided by third parties.

Any Red Hat provided layered products that layer on top of JBoss EAP 6.1 or later will also install their modules within the **system** directory.

Creating custom static modules can be useful if many applications are deployed on the same server that use the same third-party libraries. Instead of bundling those libraries with each application, a module containing these libraries can be created and installed by the JBoss administrator. The applications can then declare an explicit dependency on the custom static modules.

Users must ensure that custom modules are installed into the **EAP_HOME/modules** directory, using a one directory per module layout. This ensures that custom versions of modules that already exist in the **system** directory are loaded instead of the shipped versions. In this way, user provided modules will take precedence over system modules.

If you use the **JBOSS_MODULEPATH** environment variable to change the locations in which JBoss EAP searches for modules, then the product will look for a **system** subdirectory structure within one of the locations specified. A **system** structure must exist somewhere in the locations specified with **JBOSS_MODULEPATH**.

Dynamic Modules

Dynamic Modules are created and loaded by the application server for each JAR or WAR deployment (or subdeployment in an EAR). The name of a dynamic module is derived from the name of the deployed archive. Because deployments are loaded as modules, they can configure dependencies and be used as dependencies by other deployments.

Modules are only loaded when required. This usually only occurs when an application is deployed that has explicit or implicit dependencies.

[Report a bug](#)

1.3. SET UP THE DEVELOPMENT ENVIRONMENT

1.3.1. Download and Install Red Hat JBoss Developer Studio

1.3.1.1. Setup Red Hat JBoss Developer Studio

1. [Section 1.3.1.2, “Download Red Hat JBoss Developer Studio”](#)
2. [Section 1.3.1.3, “Install Red Hat JBoss Developer Studio”](#)
3. [Section 1.3.1.4, “Start Red Hat JBoss Developer Studio”](#)
4. [Section 1.3.1.5, “Add the JBoss EAP Server Using Define New Server”](#)

[Report a bug](#)

1.3.1.2. Download Red Hat JBoss Developer Studio

1. Go to <https://access.redhat.com/>.
2. Select **Downloads** from the menu at the top of the page.
3. Find **Red Hat JBoss Developer Studio** in the list and click on it.

4. Select the appropriate version and click **Download**.

[Report a bug](#)

1.3.1.3. Install Red Hat JBoss Developer Studio

Prerequisites:

[Section 1.3.1.2, “Download Red Hat JBoss Developer Studio”](#)

Procedure 1.1. Install Red Hat JBoss Developer Studio

1. Open a terminal.
2. Move into the directory containing the downloaded `.jar` file.
3. Run the following command to launch the GUI installer:

```
java -jar jbdevstudio-build_version.jar
```

4. Click **Next** to start the installation process.
5. Select **I accept the terms of this license agreement** and click **Next**.
6. Adjust the installation path and click **Next**.



NOTE

If the installation path folder does not exist, a prompt will appear. Click **Ok** to create the folder.

7. Choose a JVM, or leave the default JVM selected, and click **Next**.
8. Add any application platforms available, and click **Next**.
9. Review the installation details, and click **Next**.
10. Click **Next** when the installation process is complete.
11. Configure the desktop shortcuts for Red Hat JBoss Developer Studio, and click **Next**.
12. Click **Done**.

[Report a bug](#)

1.3.1.4. Start Red Hat JBoss Developer Studio

Prerequisites:

[Section 1.3.1.3, “Install Red Hat JBoss Developer Studio”](#)

Procedure 1.2. Command to start Red Hat JBoss Developer Studio

1. Open a terminal.
2. Change into the installation directory.
3. Run the following command to start Red Hat JBoss Developer Studio:

```
[localhost]$ ./jbdevstudio
```

[Report a bug](#)

1.3.1.5. Add the JBoss EAP Server Using Define New Server

These instructions assume this is your first introduction to Red Hat JBoss Developer Studio and you have not yet added any Red Hat JBoss Enterprise Application Platform servers. The procedure below adds the JBoss EAP server using the **Define New Server** wizard.

Procedure 1.3. Add the server

1. Open the **Servers** tab. If there is no **Servers** tab, add it to the panel as follows:
 - a. Click **Window** → **Show View** → **Other...**
 - b. Select **Servers** from the **Server** folder and click **OK**.
2. Click on **No servers are available. Click this link to create a new server...** or, if you prefer, right-click within the blank Server panel and select **New** → **Server**.

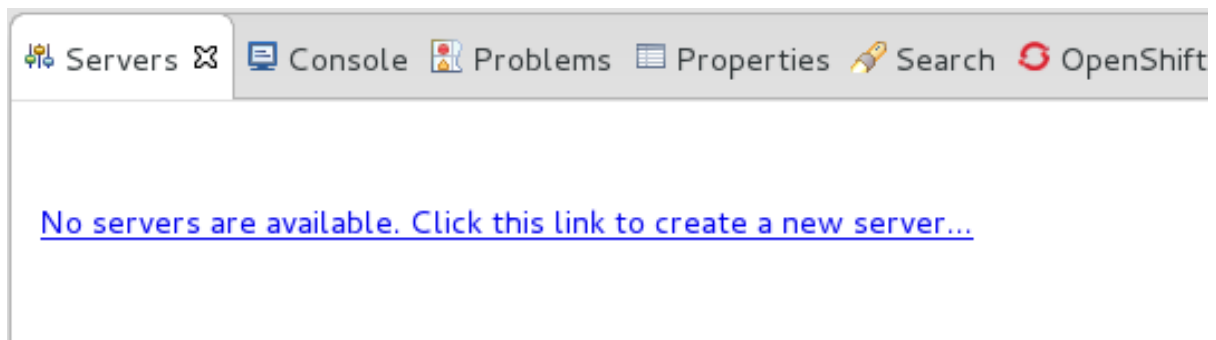


Figure 1.1. Add a new server - No servers available

3. Expand **JBoss Enterprise Middleware** and choose **JBoss Enterprise Application Platform 6.1+**. Enter a server name, for example, "JBoss Enterprise Application Platform 6.4", then click **Next** to create the JBoss runtime and define the server. The next time you define a new server, this dialog displays a **Server runtime environment** selection with the new runtime definition.

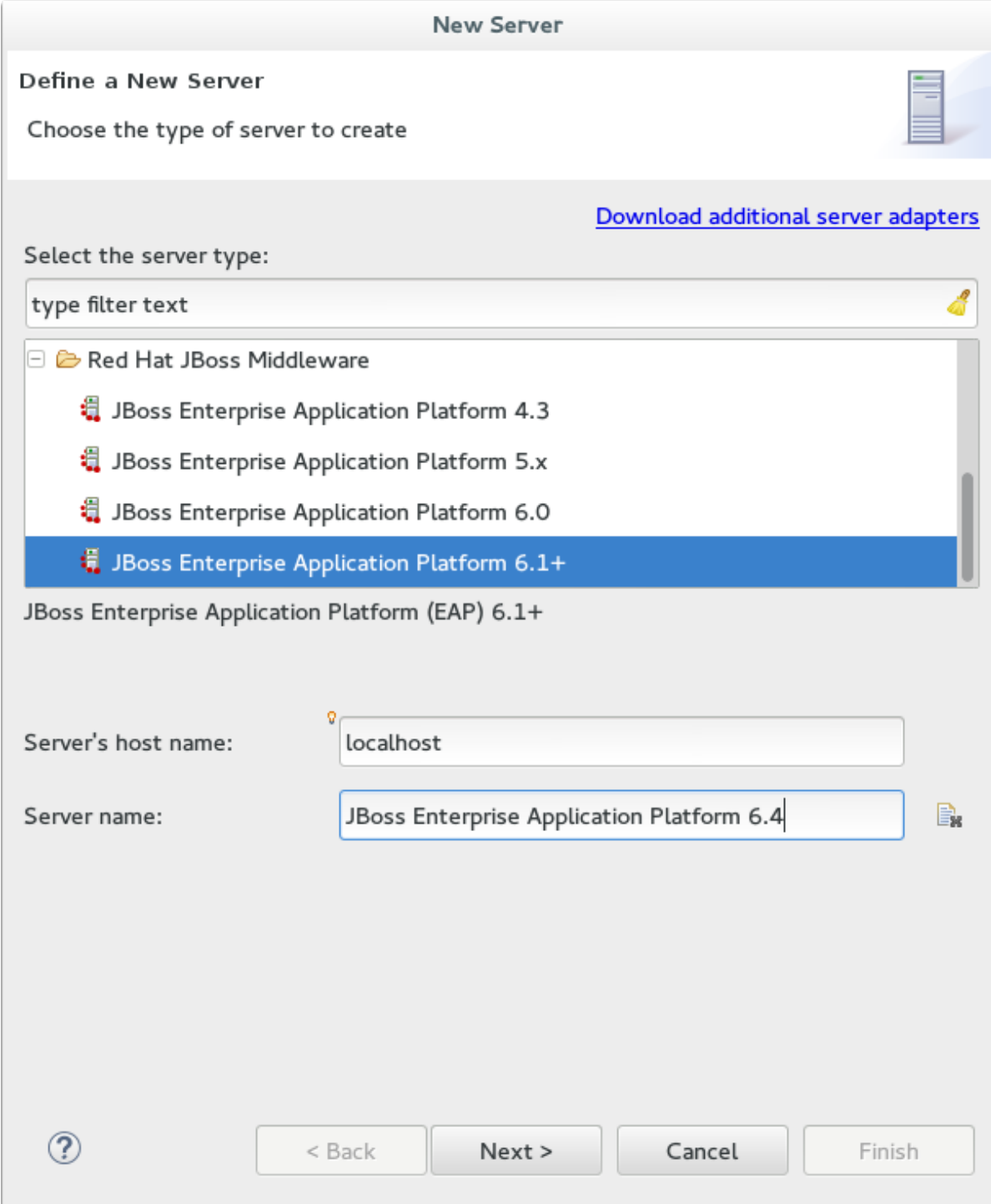

The image shows a 'New Server' dialog box with a title bar. Inside, there's a section 'Define a New Server' with a sub-instruction 'Choose the type of server to create' and a server rack icon. A link 'Download additional server adapters' is on the right. Below, 'Select the server type:' is followed by a search bar 'type filter text' with a bell icon. A list box shows 'Red Hat JBoss Middleware' expanded, listing 'JBoss Enterprise Application Platform 4.3', '5.x', '6.0', and '6.1+' (which is selected and highlighted in blue). Below the list, the text 'JBoss Enterprise Application Platform (EAP) 6.1+' is displayed. At the bottom, there are two text fields: 'Server's host name:' with 'localhost' and 'Server name:' with 'JBoss Enterprise Application Platform 6.4'. A help icon is next to the 'Server name' field. At the very bottom, there's a question mark icon and four buttons: '< Back', 'Next >', 'Cancel', and 'Finish'.

Figure 1.2. Define a New Server

4. Create a Server Adapter to manage starting and stopping the server. Keep the defaults and click **Next**.



New Server

Create a new Server Adapter

JBoss Enterprise Application Platform (EAP) 6.1+

A Server Adapter manages starting and stopping instances of your server. It manages command line arguments and keeps track of which modules have been deployed.

The server is: ☒ Local
☐ Remote

Controlled by: ☒ Filesystem and shell operations
☐ Management Operations

The selected profile requires a runtime.

☒ Assign a runtime to this server

Create new runtime (next page) ▼


Runtime Details

JRE:
Home Directory:
Base Directory:
Configuration File:

? < Back Next > Cancel Finish

Figure 1.3. Create a New Server Adapter

5. Enter a name, for example "JBoss EAP 6.4 Runtime". Under **Home Directory**, click **Browse** and navigate to your JBoss EAP install location. Then click **Next**.



New Server

JBoss Runtime

JBoss Enterprise Application Platform (EAP) 6.1+

A JBoss Server runtime references a JBoss installation directory. It can be used to set up classpaths for projects which depend on this runtime, as well as by a "server" which will be able to start and stop instances of JBoss.

Name

Home Directory [Download and install runtime...](#)

Runtime JRE

☒ Execution Environment:

☐ Alternate JRE:

Configuration base directory:

Configuration file:

Figure 1.4. Add New Server Runtime Environment



NOTE

Some quickstarts require that you run the server with a different profile or additional arguments. To deploy a quickstart that requires the **full** profile, you must define a new server and add a **Server Runtime Environment** that specifies **standalone-full.xml** for the **Configuration file**. Be sure to give the new server a descriptive name.

- Configure existing projects for the new server. Because you do not have any projects at this point, click **Finish**.

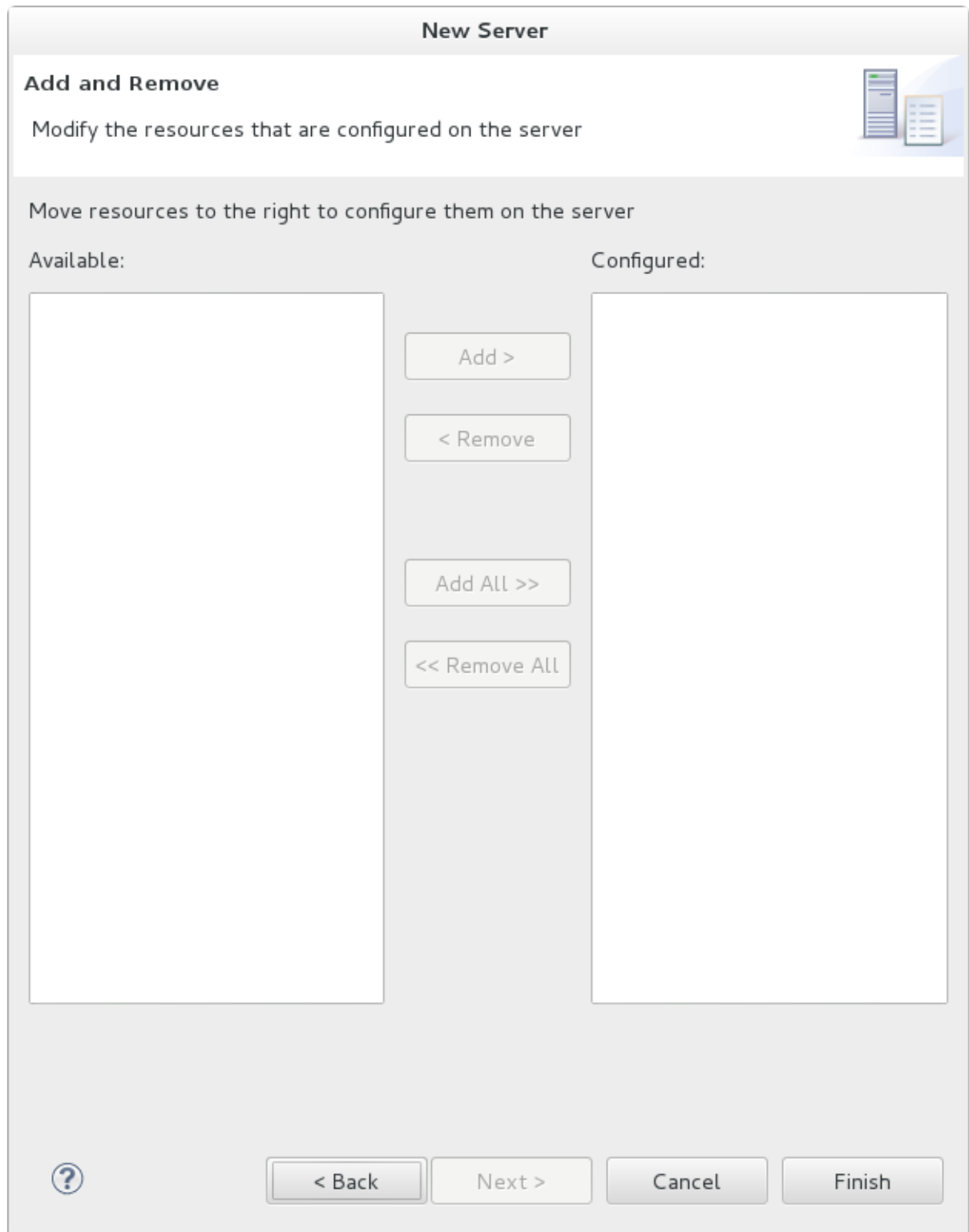


Figure 1.5. Modify resources for the new JBoss server

Result

The JBoss EAP Runtime Server is listed in the **Servers** tab.

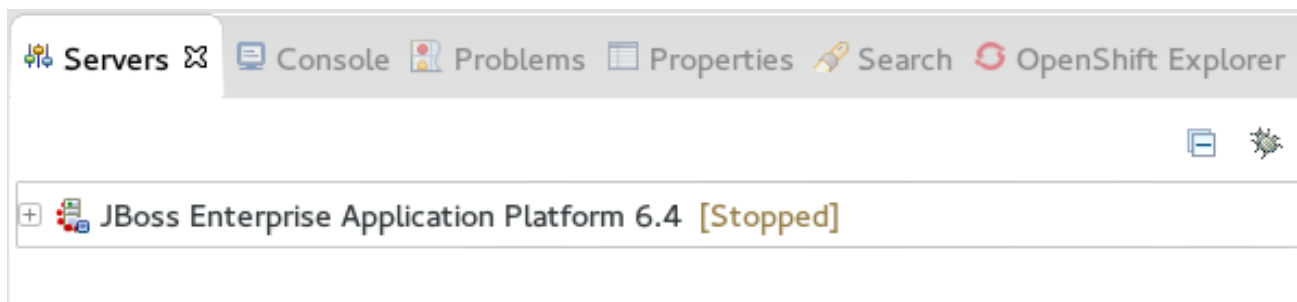


Figure 1.6. Server appears in the server list

[Report a bug](#)

1.4. RUN YOUR FIRST APPLICATION

1.4.1. Download the Quickstart Code Examples

1.4.1.1. Access the Quickstarts

Summary

JBoss EAP 6 comes with a series of quickstart examples designed to help users begin writing applications using the Java EE 6 technologies.

Prerequisites

- Maven 3.0.0 or higher. For more information on installing Maven, refer to <http://maven.apache.org/download.html>.
- [Section 2.1.1, “About the Maven Repository”](#)
- The JBoss EAP 6 Maven repository is available online, so it is not necessary to download and install it locally. If you plan to use the online repository, you can skip to the next step. If you prefer to install a local repository, see: [Section 2.2.3, “Install the JBoss EAP 6 Maven Repository Locally”](#).
- [Section 2.3.2, “Configure the JBoss EAP 6 Maven Repository Using the Maven Settings”](#)

Procedure 1.4. Download the Quickstarts

1. Open a web browser and access this URL:
<https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?product=appplatform>.
2. Find "Quickstarts" in the list.
3. Click the **Download** button to download a Zip archive containing the examples.
4. Unzip the archive in a directory of your choosing.

Result

The JBoss EAP Quickstarts have been downloaded and unzipped. Refer to the **README.md** file in the top-level directory of the Quickstart archive for instructions about deploying each quickstart.

[Report a bug](#)

1.4.2. Run the Quickstarts

1.4.2.1. Run the Quickstarts in Red Hat JBoss Developer Studio

This section describes how to use Red Hat JBoss Developer Studio to deploy the quickstarts and run the Arquillian tests.

Procedure 1.5. Import the quickstarts into Red Hat JBoss Developer Studio

Each quickstart ships with a POM (Project Object Model) file that contains project and configuration information for the quickstart. Using this POM file, you can easily import the quickstart into Red Hat JBoss Developer Studio.



IMPORTANT

If your quickstart project folder is located within the IDE workspace when you import it into Red Hat JBoss Developer Studio, the IDE generates an invalid project name and WAR archive name. Be sure your quickstart project folder is located outside the IDE workspace before you begin!

1. If you have not yet done so, [Section 2.3.2, “Configure the JBoss EAP 6 Maven Repository Using the Maven Settings”](#).
2. Start Red Hat JBoss Developer Studio.
3. From the menu, select **File** → **Import**.
4. In the selection list, choose **Maven** → **Existing Maven Projects**, then click **Next**.

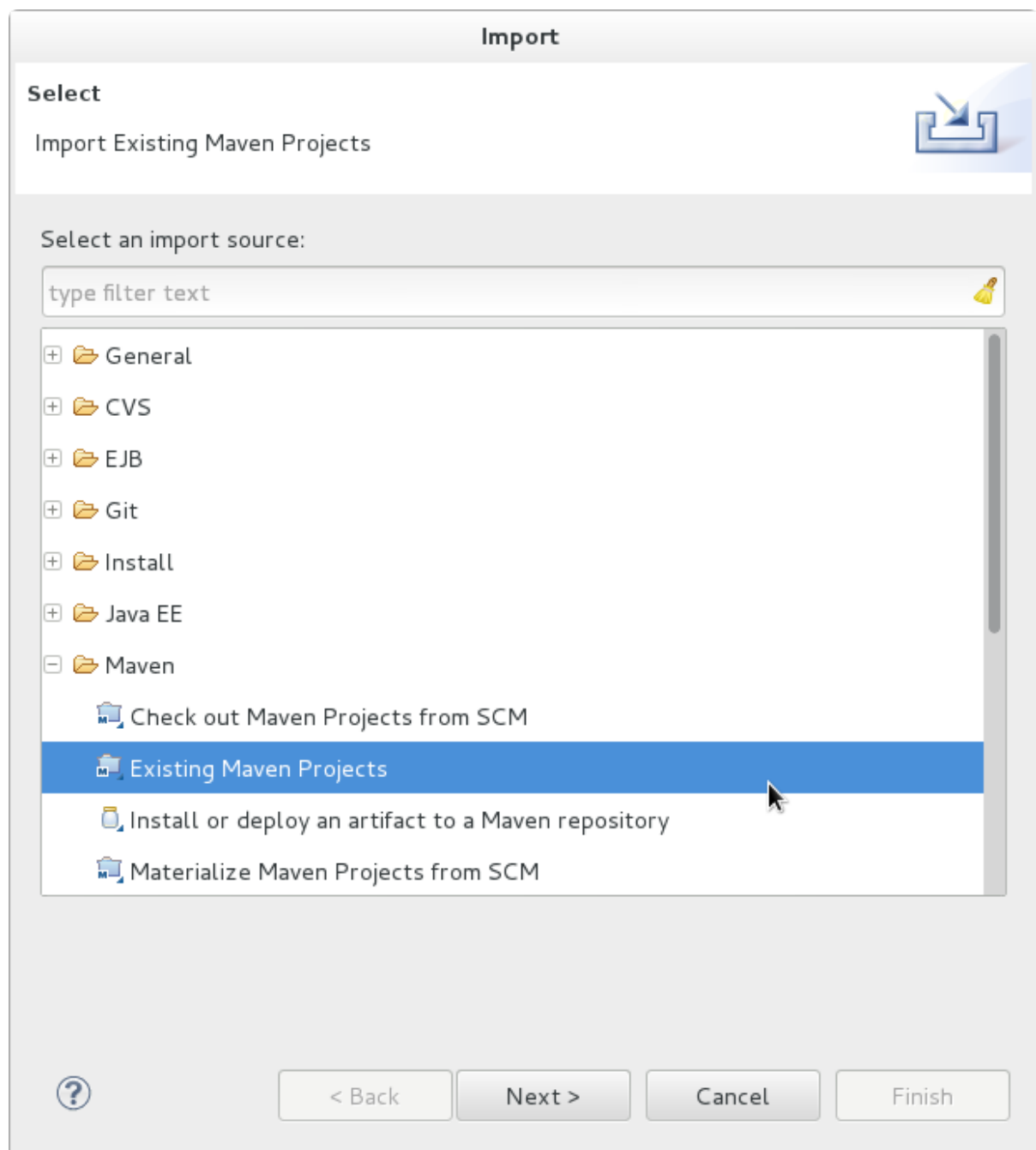


Figure 1.7. Import Existing Maven Projects

5. Browse to the directory of the quickstart you plan to test, for example the **helloworld** quickstart, and click **OK**. The **Projects** list box is populated with the **pom.xml** file of the selected quickstart project.

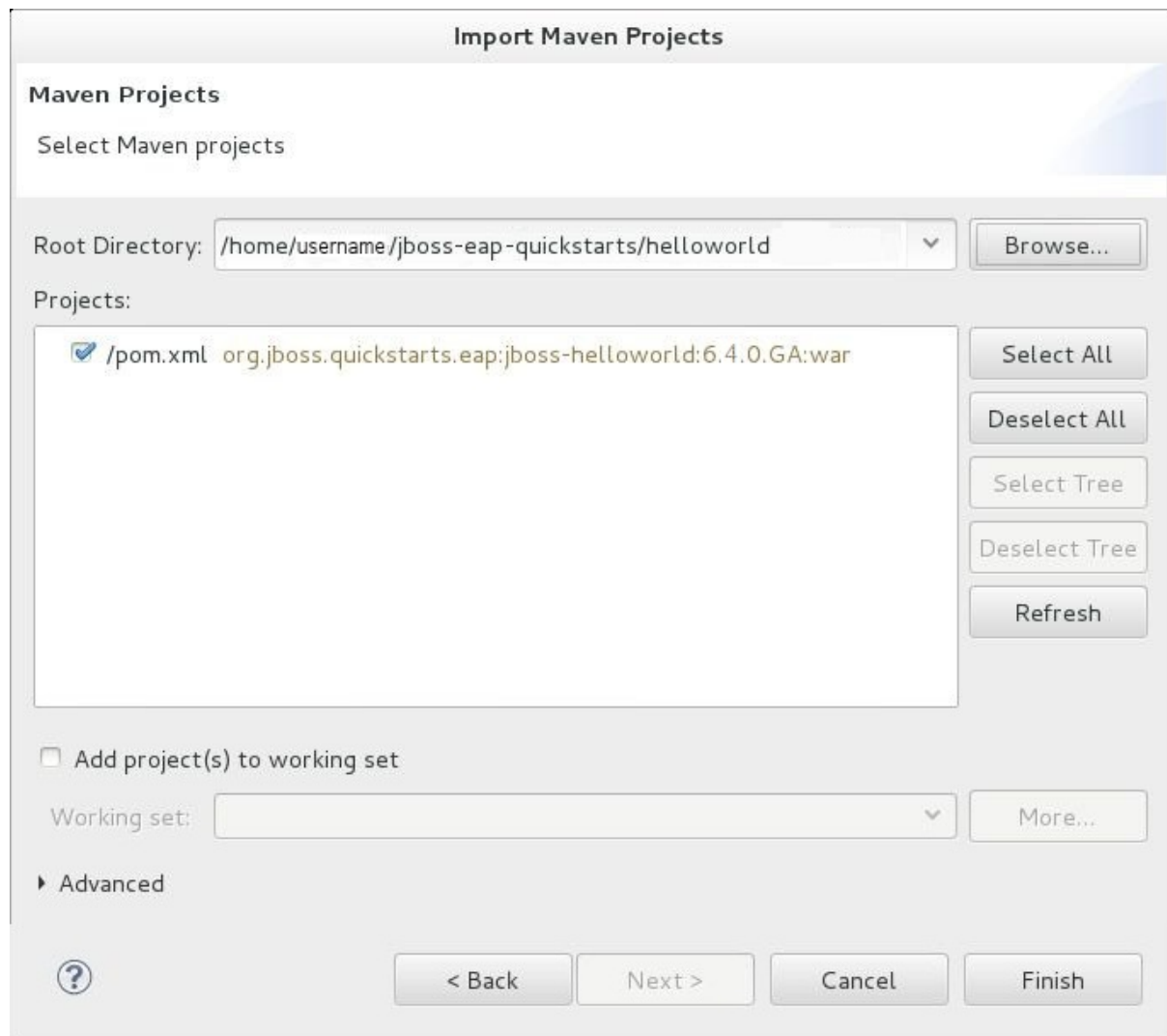


Figure 1.8. Select Maven Projects

6. Click **Finish**.

Procedure 1.6. Build and Deploy the helloworld quickstart

The **helloworld** quickstart is one of the simplest quickstarts and is a good way to verify that the JBoss server is configured and running correctly.

1. If you do not see a **Servers** tab or have not yet defined a server, follow the instructions here: [Section 1.3.1.5, “Add the JBoss EAP Server Using Define New Server”](#). If you plan to deploy a quickstart that requires the **full** profile or additional startup arguments, be sure to create the server runtime environment as noted in the quickstart instructions.
2. Right-click on the **jboss-helloworld** project in the **Project Explorer** tab and select **Run As**. You are provided with a list of choices. Select **Run on Server**.



Figure 1.9. Run As - Run on Server

3. Select **JBoss EAP 6.1+ Runtime Server** from the server list and click **Next**.

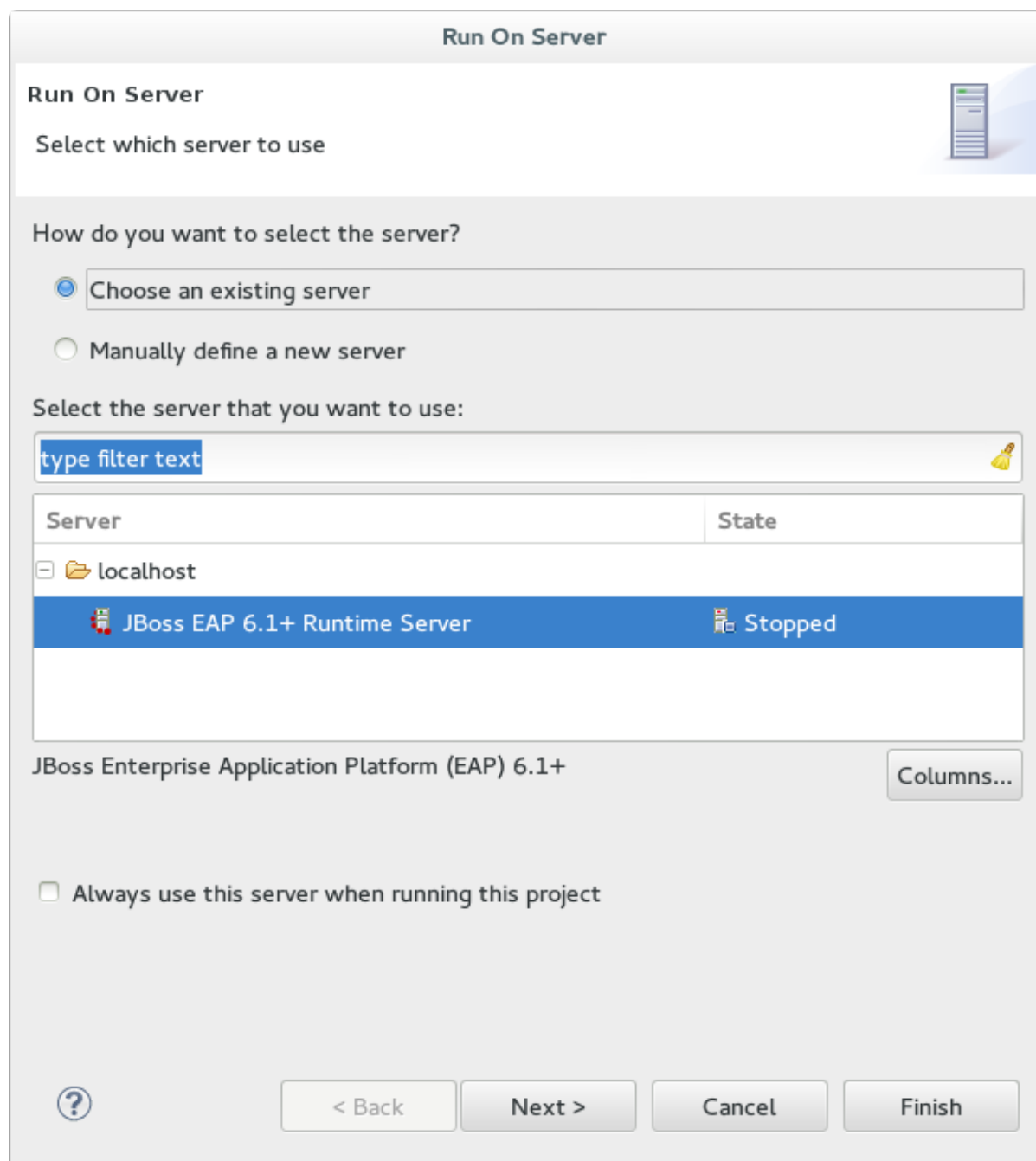


Figure 1.10. Run on Server

- The next screen displays the resources that are configured on the server. The **jboss-helloworld** quickstart is configured for you. Click **Finish** to deploy the quickstart.

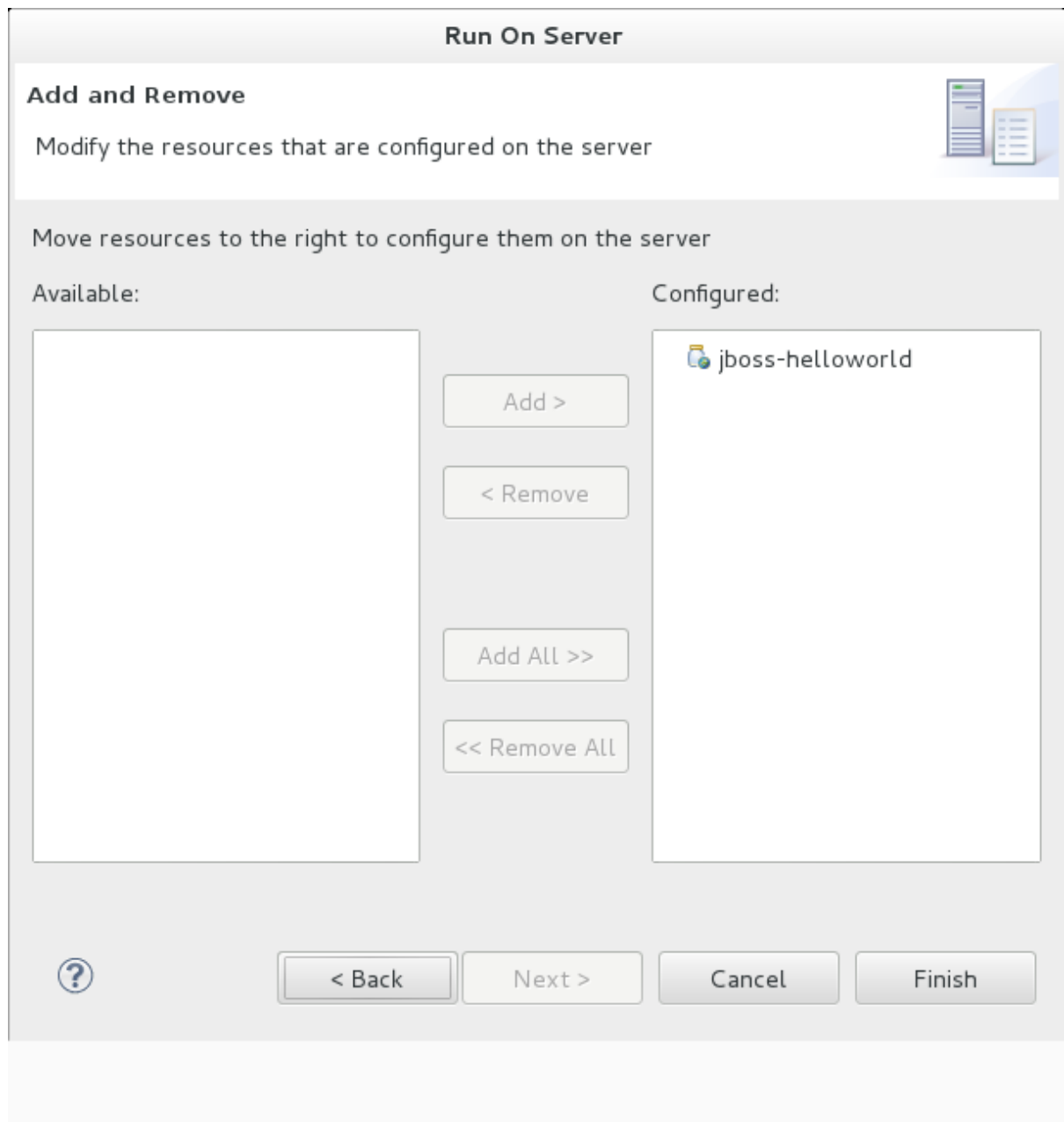


Figure 1.11. Modify Resources Configured on the Server

5. Review the results.

- In the **Server** tab, the JBoss EAP 6.x Runtime Server status changes to [**Started, Republish**] .
- The server **Console** tab shows messages detailing the JBoss EAP 6.x server start and the helloworld quickstart deployment.
- A **helloworld** tab appears displaying the URL <http://localhost:8080/jboss-helloworld/HelloWorld> and the text "Hello World!".
- The following messages in the **Console** confirm deployment of the **jboss-helloworld.war** file:

```
JBAS018210: Register web context: /jboss-helloworld
JBAS018559: Deployed "jboss-helloworld.war" (runtime-name :
"jboss-helloworld.war")
```

The registered web context is appended to **http://localhost:8080** to provide the URL used to access the deployed application.

6. To verify the **helloworld** quickstart deployed successfully to the JBoss server, open a web browser and access the application at this URL: <http://localhost:8080/jboss-helloworld>

Procedure 1.7. Run the bean-validation quickstart Arquillian tests

Some quickstarts do not provide a user interface layer and instead provide Arquillian tests to demonstrate the code examples. The **bean-validation** quickstart is an example of a quickstart that provides Arquillian tests.

1. Follow the procedure above to import the **bean-validation** quickstart into Red Hat JBoss Developer Studio.
2. In the **Servers** tab, right-click on the server and choose **Start** to start the JBoss EAP server. If you do not see a **Servers** tab or have not yet defined a server, follow the instructions here: [Section 1.3.1.5, “Add the JBoss EAP Server Using Define New Server”](#).
3. Right-click on the **jboss-bean-validation** project in the **Project Explorer** tab and select **Run As**. You are provided with a list of choices. Select **Maven Build**.
4. In the **Goals** input field of the **Edit Configuration** dialog, type: **clean test -Parq-jbossas-remote**

Then click **Run**.

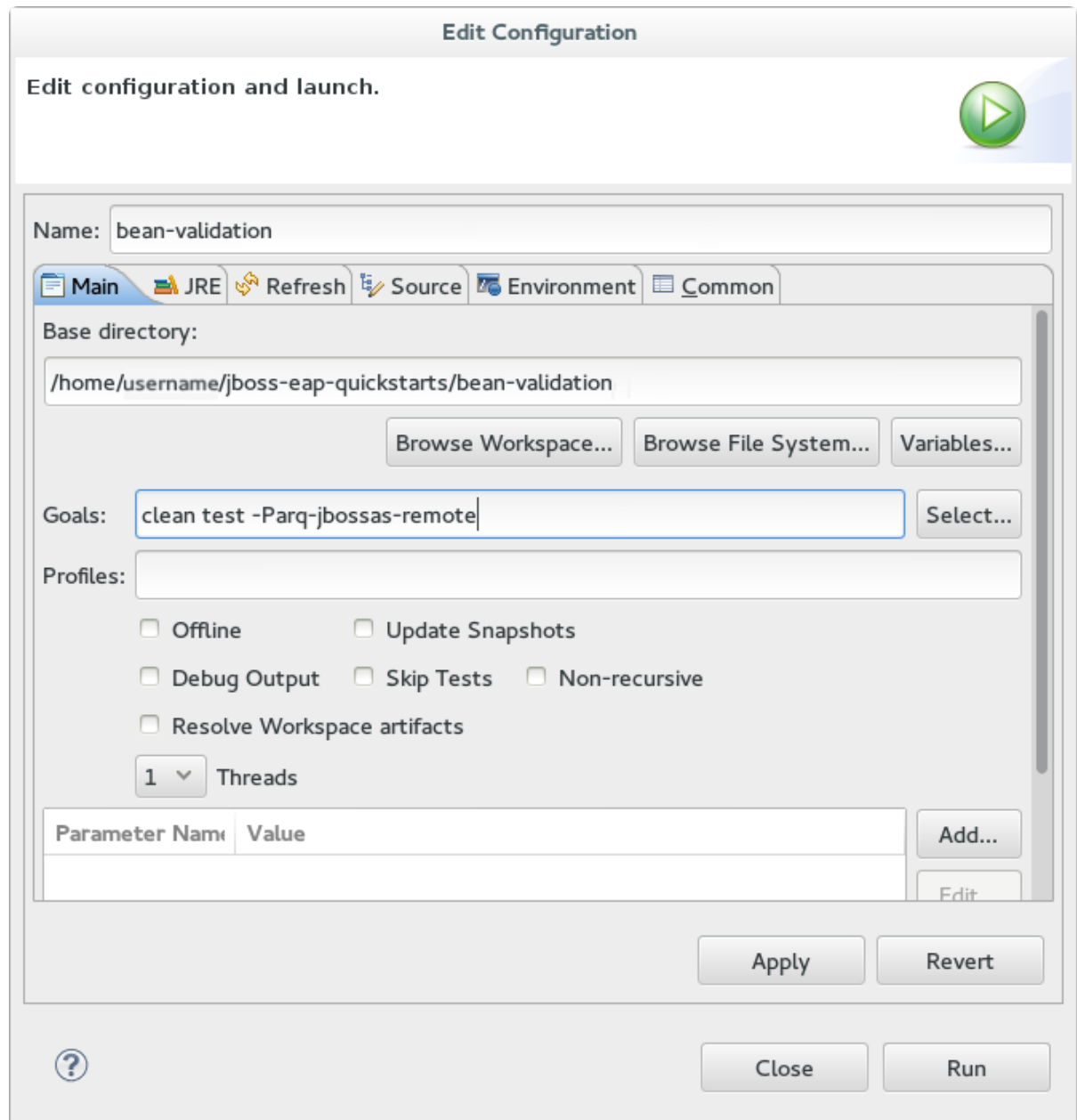


Figure 1.12. Edit Configuration

5. Review the results.

The server **Console** tab shows messages detailing the JBoss EAP server start and the output of the **bean-validation** quickstart Arquillian tests.

```

-----
T E S T S
-----
Running
org.jboss.as.quickstarts.bean_validation.test.MemberValidationTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
2.189 sec

Results :

Tests run: 5, Failures: 0, Errors: 0, Skipped: 0

[INFO] -----

```

```
-----  
[INFO] BUILD SUCCESS
```

```
[INFO] -----  
-----
```

[Report a bug](#)

1.4.2.2. Run the Quickstarts Using a Command Line

Procedure 1.8. Build and Deploy the Quickstarts Using a Command Line

You can easily build and deploy the quickstarts using a command line. Be aware that, when using a command line, you are responsible for starting the JBoss server if it is required.

1. If you have not yet done so, [Section 2.3.2, “Configure the JBoss EAP 6 Maven Repository Using the Maven Settings”](#).
2. Review the **README.html** file in the root directory of the quickstarts.

This file contains general information about system requirements, how to configure Maven, how to add users, and how to run the Quickstarts. Be sure to read through it before you get started.

It also contains a table listing the available quickstarts. The table lists each quickstart name and the technologies it demonstrates. It gives a brief description of each quickstart and the level of experience required to set it up. For more detailed information about a quickstart, click on the quickstart name.

Some quickstarts are designed to enhance or extend other quickstarts. These are noted in the **Prerequisites** column. If a quickstart lists prerequisites, you must install them first before working with the quickstart.

Some quickstarts require the installation and configuration of optional components. Do not install these components unless the quickstart requires them.

3. Run the **helloworld** quickstart.

The **helloworld** quickstart is one of the simplest quickstarts and is a good way to verify that the JBoss server is configured and running correctly. Open the **README.html** file in the root of the **helloworld** quickstart. It contains detailed instructions on how to build and deploy the quickstart and access the running application

4. Run the other quickstarts.

Follow the instructions in the **README.html** file located in the root folder of each quickstart to run the example.

[Report a bug](#)

1.4.3. Review the Quickstart Tutorials

1.4.3.1. Explore the helloworld Quickstart

Summary

The **helloworld** quickstart shows you how to deploy a simple Servlet to JBoss EAP 6. The business

logic is encapsulated in a service which is provided as a CDI (Contexts and Dependency Injection) bean and injected into the Servlet. This quickstart is very simple. All it does is print "Hello World" onto a web page. It is a good starting point to be sure you have configured and started your server properly.

Detailed instructions to build and deploy this quickstart using a command line can be found in the README.html file at the root of the **helloworld** quickstart directory. Here we show you how to use Red Hat JBoss Developer Studio to run the quickstart. This topic assumes you have installed Red Hat JBoss Developer Studio, configured Maven, and imported and successfully run the **helloworld** quickstart.

Prerequisites

- Install Red Hat JBoss Developer Studio following the procedure here: [Section 1.3.1.3, "Install Red Hat JBoss Developer Studio"](#).
- Configure Maven for use with Red Hat JBoss Developer Studio following the procedure here: [Section 2.3.3, "Configure Maven for Use with Red Hat JBoss Developer Studio"](#).
- Follow the procedures here to import, build, and deploy the **helloworld** quickstart in Red Hat JBoss Developer Studio: [Section 1.4.2.1, "Run the Quickstarts in Red Hat JBoss Developer Studio"](#)
- Verify the **helloworld** quickstart was deployed successfully to JBoss EAP by opening a web browser and accessing the application at this URL: <http://localhost:8080/jboss-helloworld>

Procedure 1.9. Examine the Directory Structure

The code for the **helloworld** quickstart can be found in the **QUICKSTART_HOME/helloworld** directory. The **helloworld** quickstart is comprised of a Servlet and a CDI bean. It also includes an empty beans.xml file which tells JBoss EAP 6 to look for beans in this application and to activate the CDI.

1. The **beans.xml** file is located in the **WEB-INF/** folder in the **src/main/webapp/** directory of the quickstart.
2. The **src/main/webapp/** directory also includes an **index.html** file which uses a simple meta refresh to redirect the user's browser to the Servlet, which is located at <http://localhost:8080/jboss-helloworld/HelloWorld>.
3. All the configuration files for this example are located in **WEB-INF/**, which can be found in the **src/main/webapp/** directory of the example.
4. Notice that the quickstart doesn't even need a **web.xml** file!

Procedure 1.10. Examine the Code

The package declaration and imports have been excluded from these listings. The complete listing is available in the quickstart source code.

1. Review the HelloWorldServlet code

The **HelloWorldServlet.java** file is located in the **src/main/java/org/jboss/as/quickstarts/helloworld/** directory. This Servlet sends the information to the browser.

```
42. @SuppressWarnings("serial")
```

```

43. @WebServlet("/HelloWorld")
44. public class HelloWorldServlet extends HttpServlet {
45.
46.     static String PAGE_HEADER = "<html><head>
<title>helloworld</title></head><body>";
47.
48.     static String PAGE_FOOTER = "</body></html>";
49.
50.     @Inject
51.     HelloService helloService;
52.
53.     @Override
54.     protected void doGet(HttpServletRequest req,
HttpServletRequestResponse resp) throws ServletException, IOException {
55.         resp.setContentType("text/html");
56.         PrintWriter writer = resp.getWriter();
57.         writer.println(PAGE_HEADER);
58.         writer.println("<h1>" +
helloService.createHelloMessage("World") + "</h1>");
59.         writer.println(PAGE_FOOTER);
60.         writer.close();
61.     }
62.
63. }

```

Table 1.1. HelloWorldServlet Details

Line	Note
43	Before Java EE 6, an XML file was used to register Servlets. It is now much cleaner. All you need to do is add the @WebServlet annotation and provide a mapping to a URL used to access the servlet.
46-48	Every web page needs correctly formed HTML. This quickstart uses static Strings to write the minimum header and footer output.
50-51	These lines inject the HelloService CDI bean which generates the actual message. As long as we don't alter the API of HelloService, this approach allows us to alter the implementation of HelloService at a later date without changing the view layer.
58	This line calls into the service to generate the message "Hello World", and write it out to the HTTP request.

2. Review the HelloService code

The **HelloService.java** file is located in the **src/main/java/org/jboss/as/quickstarts/helloworld/** directory. This service is very simple. It returns a message. No XML or annotation registration is required.

```

public class HelloService {

    String createHelloMessage(String name) {

```

```

    }
    return "Hello " + name + "!";
}

```

[Report a bug](#)

1.4.3.2. Explore the numberguess Quickstart

Summary

This quickstart shows you how to create and deploy a simple application to JBoss EAP 6. This application does not persist any information. Information is displayed using a JSF view, and business logic is encapsulated in two CDI (Contexts and Dependency Injection) beans. In the **numberguess** quickstart, you get 10 attempts to guess a number between 1 and 100. After each attempt, you're told whether your guess was too high or too low.

The code for the **numberguess** quickstart can be found in the **QUICKSTART_HOME/numberguess** directory. The **numberguess** quickstart is comprised of a number of beans, configuration files and Facelets (JSF) views, packaged as a WAR module.

Detailed instructions to build and deploy this quickstart using a command line can be found in the README.html file at the root of the **numberguess** quickstart directory. Here we show you how to use Red Hat JBoss Developer Studio to run the quickstart. This topic assumes you have installed Red Hat JBoss Developer Studio, configured Maven, and imported and successfully run the **numberguess** quickstart.

Prerequisites

- Install Red Hat JBoss Developer Studio following the procedure here: [Section 1.3.1.3, “Install Red Hat JBoss Developer Studio”](#).
- Configure Maven for use with Red Hat JBoss Developer Studio following the procedure here: [Section 2.3.3, “Configure Maven for Use with Red Hat JBoss Developer Studio”](#).
- Follow the procedures here to import, build, and deploy the **numberguess** quickstart in Red Hat JBoss Developer Studio: [Section 1.4.2.1, “Run the Quickstarts in Red Hat JBoss Developer Studio”](#)
- Verify the **numberguess** quickstart was deployed successfully to JBoss EAP by opening a web browser and accessing the application at this URL: <http://localhost:8080/jboss-numberguess>

Procedure 1.11. Examine the Configuration Files

All the configuration files for this example are located in **WEB-INF/** directory which can be found in the **src/main/webapp/** directory of the quickstart.

1. Examine the **faces-config.xml** file.

This quickstart uses the JSF 2.0 version of **faces-config.xml** filename. A standardized version of Facelets is the default view handler in JSF 2.0, so there's really nothing that you have to configure. JBoss EAP 6 goes above and beyond Java EE here. It will automatically configure the JSF for you if you include this configuration file. As a result, the configuration consists of only the root element:

```

19. <faces-config version="2.0"
20.     xmlns="http://java.sun.com/xml/ns/javaee"

```

```

21.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
22.     xsi:schemaLocation="
23.         http://java.sun.com/xml/ns/javaee>
24.         http://java.sun.com/xml/ns/javaee/web-
facesconfig_2_0.xsd">
25.
26. </faces-config>

```

2. Examine the **beans.xml** file.

There's also an empty **beans.xml** file, which tells JBoss EAP 6 to look for beans in this application and to activate the CDI.

3. There is no **web.xml** file

Notice that the quickstart doesn't even need a **web.xml** file!

Procedure 1.12. Examine the JSF Code

JSF uses the **.xhtml** file extension for source files, but serves up the rendered views with the **.jsf** extension.

- Examine the **home.xhtml** code.

The **home.xhtml** file is located in the **src/main/webapp/** directory.

```

19. <html xmlns="http://www.w3.org/1999/xhtml"
20.     xmlns:ui="http://java.sun.com/jsf/facelets"
21.     xmlns:h="http://java.sun.com/jsf/html"
22.     xmlns:f="http://java.sun.com/jsf/core">
23.
24. <head>
25. <meta http-equiv="Content-Type" content="text/html; charset=iso-
8859-1" />
26. <title>Numberguess</title>
27. </head>
28.
29. <body>
30.     <div id="content">
31.         <h1>Guess a number...</h1>
32.         <h:form id="numberGuess">
33.
34.             <!-- Feedback for the user on their guess -->
35.             <div style="color: red">
36.                 <h:messages id="messages" globalOnly="false" />
37.                 <h:outputText id="Higher" value="Higher!"
38.                     rendered="#{game.number gt game.guess and
game.guess ne 0}" />
39.                 <h:outputText id="Lower" value="Lower!"
40.                     rendered="#{game.number lt game.guess and
game.guess ne 0}" />
41.             </div>
42.
43.             <!-- Instructions for the user -->
44.             <div>

```



```

45.         I'm thinking of a number between <span
46.             id="numberGuess:smallest">#
{game.smallest}</span> and <span
47.             id="numberGuess:biggest">#{game.biggest}</span>.
You have
48.         #{game.remainingGuesses} guesses remaining.
49.     </div>
50.
51.     <!-- Input box for the users guess, plus a button to
submit, and reset -->
52.     <!-- These are bound using EL to our CDI beans -->
53.     <div>
54.         Your guess:
55.         <h:inputText id="inputGuess" value="#{game.guess}"
56.             required="true" size="3"
57.             disabled="#{game.number eq game.guess}"
58.             validator="#{game.validateNumberRange}" />
59.         <h:commandButton id="guessButton" value="Guess"
60.             action="#{game.check}"
61.             disabled="#{game.number eq game.guess}" />
62.     </div>
63.     <div>
64.         <h:commandButton id="restartButton" value="Reset"
65.             action="#{game.reset}" immediate="true" />
66.     </div>
67. </h:form>
68.
69. </div>
70.
71. <br style="clear: both" />
72.
73. </body>
74. </html>

```

Table 1.2. JSF Details

Line	Note
36-40	These are the messages which can be sent to the user: "Higher!" and "Lower!"
45-48	As the user guesses, the range of numbers they can guess gets smaller. This sentence changes to make sure they know the number range of a valid guess.
55-58	This input field is bound to a bean property using a value expression.
58	A validator binding is used to make sure the user does not accidentally input a number outside of the range in which they can guess. If the validator was not here, the user might use up a guess on an out of bounds number.
59-61	There must be a way for the user to send their guess to the server. Here we bind to an action method on the bean.

Procedure 1.13. Examine the Class Files

All of the **numberguess** quickstart source files can be found in the **src/main/java/org/jboss/as/quickstarts/numberguess/** directory. The package declaration and imports have been excluded from these listings. The complete listing is available in the quickstart source code.

1. Review the **Random.java** qualifier code.

A qualifier is used to remove ambiguity between two beans, both of which are eligible for injection based on their type. For more information on qualifiers, refer to [Section 11.2.3.3, “Use a Qualifier to Resolve an Ambiguous Injection”](#)

The **@Random** qualifier is used for injecting a random number.

```
@Target({ TYPE, METHOD, PARAMETER, FIELD })
@Retention(RUNTIME)
@Documented
@Qualifier
public @interface Random {

}
```

2. Review the **MaxNumber.java** qualifier code.

The **@MaxNumberQualifier** is used for injecting the maximum number allowed.

```
@Target({ TYPE, METHOD, PARAMETER, FIELD })
@Retention(RUNTIME)
@Documented
@Qualifier
public @interface MaxNumber {

}
```

3. Review the **Generator.java** code.

The **Generator** class is responsible for creating the random number via a producer method. It also exposes the maximum possible number via a producer method. This class is application scoped so you don't get a different random each time.

```
@SuppressWarnings("serial")
@ApplicationScoped
public class Generator implements Serializable {

    private java.util.Random random = new
java.util.Random(System.currentTimeMillis());

    private int maxNumber = 100;

    java.util.Random getRandom() {
        return random;
    }

    @Produces
    @Random
    int next() {
```

```

        // a number between 1 and 100
        return getRandom().nextInt(maxNumber - 1) + 1;
    }

    @Produces
    @MaxNumber
    int getMaxNumber() {
        return maxNumber;
    }
}

```

4. Review the **Game.java** code.

The session scoped class **Game** is the primary entry point of the application. It is responsible for setting up or resetting the game, capturing and validating the user's guess, and providing feedback to the user with a **FacesMessage**. It uses the post-construct lifecycle method to initialize the game by retrieving a random number from the **@Random Instance<Integer>** bean.

Notice the **@Named** annotation in the class. This annotation is only required when you want to make the bean accessible to a JSF view via Expression Language (EL), in this case **#{game}**.

```

@SuppressWarnings("serial")
@Named
@SessionScoped
public class Game implements Serializable {

    /**
     * The number that the user needs to guess
     */
    private int number;

    /**
     * The users latest guess
     */
    private int guess;

    /**
     * The smallest number guessed so far (so we can track the valid
     guess range).
     */
    private int smallest;

    /**
     * The largest number guessed so far
     */
    private int biggest;

    /**
     * The number of guesses remaining
     */
    private int remainingGuesses;

    /**
     * The maximum number we should ask them to guess
     */
}

```

```

@Inject
@MaxNumber
private int maxNumber;

/**
 * The random number to guess
 */
@Inject
@Random
Instance<Integer> randomNumber;

public Game() {
}

public int getNumber() {
    return number;
}

public int getGuess() {
    return guess;
}

public void setGuess(int guess) {
    this.guess = guess;
}

public int getSmallest() {
    return smallest;
}

public int getBiggest() {
    return biggest;
}

public int getRemainingGuesses() {
    return remainingGuesses;
}

/**
 * Check whether the current guess is correct, and update the
 * biggest/smallest guesses as needed. Give feedback to the user
 * if they are correct.
 */
public void check() {
    if (guess > number) {
        biggest = guess - 1;
    } else if (guess < number) {
        smallest = guess + 1;
    } else if (guess == number) {
        FacesContext.getCurrentInstance().addMessage(null, new
FacesMessage("Correct!"));
    }
    remainingGuesses--;
}

/**

```

```

        * Reset the game, by putting all values back to their defaults,
        and getting a new random number. We also call this method
        * when the user starts playing for the first time using
        {@linkplain PostConstruct @PostConstruct} to set the initial
        * values.
        */
    @PostConstruct
    public void reset() {
        this.smallest = 0;
        this.guess = 0;
        this.remainingGuesses = 10;
        this.biggest = maxNumber;
        this.number = randomNumber.get();
    }

    /**
     * A JSF validation method which checks whether the guess is
     * valid. It might not be valid because there are no guesses left,
     * or because the guess is not in range.
     */
    public void validateNumberRange(FacesContext context,
    UIComponent toValidate, Object value) {
        if (remainingGuesses <= 0) {
            FacesMessage message = new FacesMessage("No guesses
left!");
            context.addMessage(toValidate.getClientId(context),
message);
            ((UIInput) toValidate).setValid(false);
            return;
        }
        int input = (Integer) value;

        if (input < smallest || input > biggest) {
            ((UIInput) toValidate).setValid(false);

            FacesMessage message = new FacesMessage("Invalid
guess");
            context.addMessage(toValidate.getClientId(context),
message);
        }
    }
}

```

[Report a bug](#)

1.4.4. Replace the Default Welcome Web Application

JBoss EAP 6 includes a Welcome application, which displays when you open the URL of the server at port 8080. You can replace this application with your own web application by following this procedure.

Procedure 1.14. Replace the Default Welcome Web Application With Your Own Web Application

1. Disable the Welcome application.

Use the Management CLI script **`EAP_HOME/bin/jboss-cli.sh`** to run the following command. You may need to change the profile to modify a different managed domain profile, or remove the **`/profile=default`** portion of the command for a standalone server.

```
/profile=default/subsystem=web/virtual-server=default-host:write-attribute(name=enable-welcome-root,value=false)
```

2. Configure your Web application to use the root context.

To configure your web application to use the root context (/) as its URL address, modify its **`jboss-web.xml`**, which is located in the **`META-INF/`** or **`WEB-INF/`** directory. Replace its **`<context-root>`** directive with one that looks like the following.

```
<jboss-web>
  <context-root>/</context-root>
</jboss-web>
```

3. Deploy your application.

Deploy your application to the server group or server you modified in the first step. The application is now available on **`http://SERVER_URL:PORT/`**.

[Report a bug](#)

1.4.5. Using WS-AtomicTransaction

The **`wsat-simple`** quickstart demonstrates the deployment of a WS-AT (WS-AtomicTransaction) enabled JAX-WS Web Service bundled in a WAR archive for deployment to Red Hat JBoss Enterprise Application Platform.

The Web service is offered by a Restaurant for making bookings. The Service allows bookings to be made within an Atomic Transaction. This example demonstrates the basics of implementing a WS-AT enabled Web service. It is beyond the scope of this quick start to demonstrate more advanced features. In particular:

- The Service does not implement the required hooks to support recovery in the presence of failures.
- It also does not utilize a transactional back end resource.
- Only one Web service participates in the protocol. As WS-AT is a 2PC coordination protocol, it is best suited to multi-participant scenarios.

For a more complete example, refer the XTS demonstrator application that ships with the Narayana project: <http://www.jboss.org/narayana>.

It is also assumed that you have an understanding of WS-AtomicTransaction. For more details, read the XTS documentation that ships with the Narayana project, which can be downloaded here: http://www.jboss.org/narayana/documentation/4174_Final.

The application consists of a single JAX-WS web service that is deployed within a WAR archive. It is tested with a JBoss Arquillian enabled JUnit test.

When running the **`org.jboss.as.quickstarts.wsat.simple.ClientTest#testCommit()`** method, the following steps occur:

1. A new Atomic Transaction (AT) is created by the client.

2. An operation on a WS-AT enabled Web service is invoked by the client.
3. The JaxWSHeaderContextProcessor in the WS Client handler chain inserts the WS-AT context into the outgoing SOAP message.
4. When the service receives the SOAP request, the JaxWSHeaderContextProcessor in its handler chain inspects the WS-AT context and associates the request with this AT.
5. The Web service operation is invoked.
6. A participant is enlisted in this AT. This allows the Web Service logic to respond to protocol events, such as Commit and Rollback.
7. The service invokes the business logic. In this case, a booking is made with the restaurant.
8. The backend resource is prepared. This ensures that the Backend resource can undo or make permanent the change when told to do so by the coordinator.
9. The client can then decide to commit or rollback the AT. If the client decides to commit, the coordinator will begin the 2PC protocol. If the participant decides to rollback, all participants will be told to rollback.

There is another test that shows what happens if the client decides to rollback the AT.

[Report a bug](#)

CHAPTER 2. MAVEN GUIDE

2.1. LEARN ABOUT MAVEN

2.1.1. About the Maven Repository

Apache Maven is a distributed build automation tool used in Java application development to create, manage, and build software projects. Maven uses standard configuration files called Project Object Model, or POM, files to define projects and manage the build process. POMs describe the module and component dependencies, build order, and targets for the resulting project packaging and output using an XML file. This ensures that the project is built in a correct and uniform manner.

Maven achieves this by using a repository. A Maven repository stores Java libraries, plug-ins, and other build artifacts. The default public repository is the [Maven 2 Central Repository](#), but repositories can be private and internal within a company with a goal to share common artifacts among development teams. Repositories are also available from third-parties. JBoss EAP 6 includes a Maven repository that contains many of the requirements that Java EE developers typically use to build applications on JBoss EAP 6. To configure your project to use this repository, see [Section 2.3.1, “Configure the JBoss EAP Maven 6 Repository”](#).

Remote repositories are accessed using common protocols such as **http://** for a repository on an HTTP server or **file://** for a repository on a file server.

For more information about Maven, see [Welcome to Apache Maven](#).

For more information about Maven repositories, see [Apache Maven Project - Introduction to Repositories](#).

For more information about Maven POM files, see the [Apache Maven Project POM Reference](#) and [Section 2.1.2, “About the Maven POM File”](#).

[Report a bug](#)

2.1.2. About the Maven POM File

The Project Object Model, or POM, file is a configuration file used by Maven to build projects. It is an XML file that contains information about the project and how to build it, including the location of the source, test, and target directories, the project dependencies, plug-in repositories, and goals it can execute. It can also include additional details about the project including the version, description, developers, mailing list, license, and more. A **pom.xml** file requires some configuration options and will default all others. See [Section 2.1.3, “Minimum Requirements of a Maven POM File”](#) for details.

The schema for the **pom.xml** file can be found at http://maven.apache.org/maven-v4_0_0.xsd.

For more information about POM files, see the [Apache Maven Project POM Reference](#).

[Report a bug](#)

2.1.3. Minimum Requirements of a Maven POM File

Minimum requirements

The minimum requirements of a **pom.xml** file are as follows:

- project root

- `modelVersion`
- `groupId` - the id of the project's group
- `artifactId` - the id of the artifact (project)
- `version` - the version of the artifact under the specified group

Sample `pom.xml` file

A basic `pom.xml` file might look like this:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jboss.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

[Report a bug](#)

2.1.4. About the Maven Settings File

The Maven `settings.xml` file contains user-specific configuration information for Maven. It contains information that must not be distributed with the `pom.xml` file, such as developer identity, proxy information, local repository location, and other settings specific to a user.

There are two locations where the `settings.xml` can be found.

In the Maven installation

The settings file can be found in the `M2_HOME/conf/` directory. These settings are referred to as **global** settings. The default Maven settings file is a template that can be copied and used as a starting point for the user settings file.

In the user's installation

The settings file can be found in the `USER_HOME/.m2/` directory. If both the Maven and user `settings.xml` files exist, the contents are merged. Where there are overlaps, the user's `settings.xml` file takes precedence.

The following is an example of a Maven `settings.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <profiles>
    <!-- Configure the JBoss EAP Maven repository -->
    <profile>
      <id>jboss-eap-maven-repository</id>
      <repositories>
        <repository>
          <id>jboss-eap</id>
          <url>file:///path/to/repo/jboss-eap-6.4-maven-repository</url>
```

```

        <releases>
          <enabled>true</enabled>
        </releases>
        <snapshots>
          <enabled>>false</enabled>
        </snapshots>
      </repository>
    </repositories>
    <pluginRepositories>
      <pluginRepository>
        <id>jboss-eap-maven-plugin-repository</id>
        <url>file:///path/to/repo/jboss-eap-6.4-maven-repository</url>
        <releases>
          <enabled>true</enabled>
        </releases>
        <snapshots>
          <enabled>>false</enabled>
        </snapshots>
      </pluginRepository>
    </pluginRepositories>
  </profile>
</profiles>
<activeProfiles>
  <!-- Optionally, make the repository active by default -->
  <activeProfile>jboss-eap-maven-repository</activeProfile>
</activeProfiles>
</settings>

```

The schema for the `settings.xml` file can be found at <http://maven.apache.org/xsd/settings-1.0.0.xsd>.

[Report a bug](#)

2.2. INSTALL MAVEN AND THE JBOSS MAVEN REPOSITORY

2.2.1. Download and Install Maven

If you plan to use Maven command line to build and deploy your applications to JBoss EAP, you must download and install Maven. If you plan to use Red Hat JBoss Developer Studio to build and deploy your applications, you can skip this procedure as Maven is distributed with Red Hat JBoss Developer Studio.

1. Go to [Apache Maven Project - Download Maven](#) and download the latest distribution for your operating system.
2. See the Maven documentation for information on how to download and install Apache Maven for your operating system.

[Report a bug](#)

2.2.2. Install the JBoss EAP 6 Maven Repository

There are three ways to install the repository; on your local file system, on Apache Web Server, or with a Maven repository manager.

- [Section 2.2.3, “Install the JBoss EAP 6 Maven Repository Locally”](#)

- [Section 2.2.4, “Install the JBoss EAP 6 Maven Repository for Use with Apache httpd”](#)
- [Section 2.2.5, “Install the JBoss EAP 6 Maven Repository Using Nexus Maven Repository Manager”](#)

[Report a bug](#)

2.2.3. Install the JBoss EAP 6 Maven Repository Locally

Summary

The JBoss EAP 6 Maven repository is available online, so it is not necessary to download and install it locally. However, if you prefer to install the JBoss EAP Maven repository locally, there are three ways to do it: on your local file system, on Apache Web Server, or with a Maven repository manager. This example covers the steps to download the JBoss EAP 6 Maven Repository to the local file system. This option is easy to configure and allows you to get up and running quickly on your local machine. It can help you become familiar with using Maven for development but is not recommended for team production environments.

Procedure 2.1. Download and Install the JBoss EAP 6 Maven Repository to the Local File System

1. Open a web browser and access this URL:
<https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?product=appplatform>.
2. Find "Red Hat JBoss Enterprise Application Platform *VERSION* Maven Repository" in the list.
3. Click the **Download** button to download a **.zip** file containing the repository.
4. Unzip the file on the local file system into a directory of your choosing.
5. [Section 2.3.2, “Configure the JBoss EAP 6 Maven Repository Using the Maven Settings”](#).

Result

This creates a Maven repository directory called **jboss-eap-version-maven-repository**.



IMPORTANT

If you want to continue to use an older local repository, you must configure it separately in the Maven **settings.xml** configuration file. Each local repository must be configured within its own **<repository>** tag.



IMPORTANT

When downloading a new Maven repository, remove the cached **repository/** subdirectory located under the **.m2/** directory before attempting to use the new Maven repository.

[Report a bug](#)

2.2.4. Install the JBoss EAP 6 Maven Repository for Use with Apache httpd

There are three ways to install the repository; on your local file system, on Apache Web Server, or with a Maven repository manager. This example will cover the steps to download the JBoss EAP 6 Maven Repository for use with Apache httpd. This option is good for multi-user and cross-team development

environments because any developer that can access the web server can also access the Maven repository.

Prerequisites

You must configure Apache httpd. See [Apache HTTP Server Project](#) documentation for instructions.

Procedure 2.2. Download the JBoss EAP 6 Maven Repository ZIP archive

1. Open a web browser and access this URL:
<https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?product=appplatform>.
2. Find "Red Hat JBoss Enterprise Application Platform <VERSION> Maven Repository" in the list.
3. Click the **Download** button to download a **.zip** file containing the repository.
4. Unzip the files in a directory that is web accessible on the Apache server.
5. Configure Apache to allow read access and directory browsing in the created directory.
6. [Section 2.3.2, "Configure the JBoss EAP 6 Maven Repository Using the Maven Settings"](#).

Result

This allows a multi-user environment to access the Maven repository on Apache httpd.



NOTE

If you're upgrading from a previous version of the repository, note that JBoss EAP Maven Repository artifacts can be extracted into an existing JBoss product Maven repository (such as JBoss EAP 6.1.0) without any conflicts. After the repository archive has been extracted, the artifacts can be used with the existing Maven settings for this repository.

[Report a bug](#)

2.2.5. Install the JBoss EAP 6 Maven Repository Using Nexus Maven Repository Manager

There are three ways to install the repository; on your local file system, on Apache Web Server, or with a Maven repository manager. This option is best if you have a license and already use a repository manager because you can host the JBoss repository alongside your existing repositories. For more information about Maven repository managers, see [Section 2.2.6, "About Maven Repository Managers"](#).

This example will cover the steps to install the JBoss EAP 6 Maven Repository using Sonatype Nexus Maven Repository Manager. For more complete instructions, see [Sonatype Nexus: Manage Artifacts](#).

Procedure 2.3. Download the JBoss EAP 6 Maven Repository ZIP archive

1. Open a web browser and access this URL:
<https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?product=appplatform>.
2. Find "Red Hat JBoss Enterprise Application Platform <VERSION> Maven Repository" in the list.
3. Click the **Download** button to download a **.zip** file containing the repository.
4. Unzip the files into a directory of your choosing on the server hosting Nexus.

Procedure 2.4. Add the JBoss EAP 6 Maven Repository using Nexus Maven Repository Manager

1. Log into Nexus as an Administrator.
2. Select the **Repositories** section from the **Views** → **Repositories** menu to the left of your repository manager.
3. Click the **Add...** dropdown, then select **Hosted Repository**.
4. Give the new repository a name and ID.
5. Enter the path on disk to the unzipped repository in the field **Override Local Storage Location**.
6. Continue if you want the artifact to be available in a repository group. Do not continue with this procedure if this is not what you want.
7. Select the repository group.
8. Click on the **Configure** tab.
9. Drag the new JBoss Maven repository from the **Available Repositories** list to the **Ordered Group Repositories** list on the left.

**NOTE**

Note that the order of this list determines the priority for searching Maven artifacts.

10. [Section 2.3.2, “Configure the JBoss EAP 6 Maven Repository Using the Maven Settings”](#).

Result

The repository is configured using Nexus Maven Repository Manager.

[Report a bug](#)

2.2.6. About Maven Repository Managers

A repository manager is a tool that allows you to easily manage Maven repositories. Repository managers are useful in multiple ways:

- They provide the ability to configure proxies between your organization and remote Maven repositories. This provides a number of benefits, including faster and more efficient deployments and a better level of control over what is downloaded by Maven.
- They provide deployment destinations for your own generated artifacts, allowing collaboration between different development teams across an organization.

For more information about Maven repository managers, see [Apache Maven Project - The List of Repository Managers](#).

Commonly used Maven repository managers**Sonatype Nexus**

See [Sonatype Nexus: Manage Artifacts](#) for more information about Nexus.

Artifactory

See [Artifactory Open Source](#) for more information about Artifactory.

Apache Archiva

See [Apache Archiva: The Build Artifact Repository Manager](#) for more information about Apache Archiva.

[Report a bug](#)

2.3. USE THE MAVEN REPOSITORY

2.3.1. Configure the JBoss EAP Maven 6 Repository

Overview

There are two approaches to direct Maven to use the JBoss EAP 6 Maven Repository in your project:

- You can configure the repositories in the Maven global or user settings.
- You can configure the repositories in the project's POM file.

Procedure 2.5. Configure Maven Settings to Use the JBoss EAP 6 Maven Repository

1. Configure the Maven repository using Maven settings

This is the recommended approach. Maven settings used with a repository manager or repository on a shared server provide better control and manageability of projects. Settings also provide the ability to use an alternative mirror to redirect all lookup requests for a specific repository to your repository manager without changing the project files. For more information about mirrors, see <http://maven.apache.org/guides/mini/guide-mirror-settings.html>.

This method of configuration applies across all Maven projects, as long as the project POM file does not contain repository configuration.

[Section 2.3.2, “Configure the JBoss EAP 6 Maven Repository Using the Maven Settings”](#)

2. Configure the Maven repository using the project POM

This method of configuration is generally not recommended. If you decide to configure repositories in your project POM file, plan carefully and be aware that it can slow down your build and you may even end up with artifacts that are not from the expected repository.



NOTE

In an Enterprise environment, where a repository manager is usually used, Maven should query all artifacts for all projects using this manager. Because Maven uses all declared repositories to find missing artifacts, if it can't find what it's looking for, it will try and look for it in the repository central (defined in the built-in parent POM). To override this central location, you can add a definition with **central** so that the default repository central is now your repository manager as well. This works well for established projects, but for clean or 'new' projects it causes a problem as it creates a cyclic dependency.

Transitively included POMs are also an issue with this type of configuration. Maven has to query these external repositories for missing artifacts. This not only slows down your build, it also causes you to lose control over where your artifacts are coming from and likely to cause broken builds.

This method of configuration overrides the global and user Maven settings for the configured project.

[Section 2.3.4, “Configure the JBoss EAP 6 Maven Repository Using the Project POM”.](#)

[Report a bug](#)

2.3.2. Configure the JBoss EAP 6 Maven Repository Using the Maven Settings

There are two approaches to direct Maven to use the JBoss EAP 6 Maven Repository in your project:

- You can modify the Maven settings. This directs Maven to use the configuration across all projects.
- You can configure the project's POM file. This limits the configuration to the specific project.

This topic shows you how to direct Maven to use the JBoss EAP 6 Maven Repository across all projects using the Maven settings. This is the recommended approach.

You can configure Maven to use either the online or a locally installed JBoss EAP 6 repository. If you choose to use the online repository, you can use a preconfigured settings file or add the JBoss EAP 6 Maven profiles to the existing settings file. To use a local repository, you must download the repository and configure the settings to point to your locally installed repository. The following procedures describe how to configure Maven for JBoss EAP 6.



NOTE

The URL of the repository will depend on where the repository is located; on the filesystem, or web server. For information on how to install the repository, see [Section 2.2.2, “Install the JBoss EAP 6 Maven Repository”](#). The following are examples for each of the installation options:

File System

`file:///path/to/repo/jboss-eap-6.x-maven-repository`

Apache Web Server

`http://intranet.acme.com/jboss-eap-6.x-maven-repository/`

Nexus Repository Manager

`https://intranet.acme.com/nexus/content/repositories/jboss-eap-6.x-maven-repository`

You can configure Maven using either the Maven install global settings or the user install settings. These instructions configure the user install settings as this is the most common configuration.

Procedure 2.6. Configure Maven Using the Settings Shipped with the Quickstart Examples

The JBoss EAP 6 Quickstarts ship with a **settings.xml** file that is configured to use the online JBoss EAP 6 Maven repository. This is the simplest approach.

1. This procedure overwrites the existing Maven settings file, so you must back up the existing Maven **settings.xml** file.
 - a. Locate the Maven install directory for your operating system. It is usually installed in **USER_HOME/.m2/** directory.
 - For Linux or Mac, this is: `~/ .m2/`
 - For Windows, this is: `\Documents and Settings\USER_NAME\.m2\` or `\Users\USER_NAME\.m2\`
 - b. If you have an existing **USER_HOME/.m2/settings.xml** file, rename it or make a backup copy so you can restore it later.
2. Download and unzip the quickstart examples that ship with JBoss EAP 6. For more information, see [Section 1.4.1.1, “Access the Quickstarts”](#)
3. Copy the **QUICKSTART_HOME/settings.xml** file to the **USER_HOME/.m2/** directory.
4. If you modify the **settings.xml** file while Red Hat JBoss Developer Studio is running, follow the procedure below entitled [Procedure 2.9, “Refresh the Red Hat JBoss Developer Studio User Settings”](#).

Procedure 2.7. Manually Edit and Configure the Maven Settings To Use the Online JBoss EAP 6 Maven Repository

You can manually add the JBoss EAP 6 profiles to an existing Maven settings file.

1. Locate the Maven install directory for your operating system. It is usually installed in **USER_HOME/.m2/** directory.

- o For Linux or Mac, this is `~/.m2/`
 - o For Windows, this is `\Documents and Settings\USER_NAME\.m2\` or `\Users\USER_NAME\.m2\`
2. If you do not find a **settings.xml** file, copy the **settings.xml** file from the **USER_HOME/.m2/conf/** directory into the **USER_HOME/.m2/** directory.
 3. Copy the following XML into the **<profiles>** element of the file.

```

<!-- Configure the JBoss GA Maven repository -->
<profile>
  <id>jboss-ga-repository</id>
  <repositories>
    <repository>
      <id>jboss-ga-repository</id>
      <url>http://maven.repository.redhat.com/techpreview/all</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>>false</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>jboss-ga-plugin-repository</id>
      <url>http://maven.repository.redhat.com/techpreview/all</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>>false</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
</profile>
<!-- Configure the JBoss Early Access Maven repository -->
<profile>
  <id>jboss-earlyaccess-repository</id>
  <repositories>
    <repository>
      <id>jboss-earlyaccess-repository</id>
      <url>http://maven.repository.redhat.com/earlyaccess/all/</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>>false</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>jboss-earlyaccess-plugin-repository</id>

```

```

        <url>http://maven.repository.redhat.com/earlyaccess/all/</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>>false</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
</profile>

```

Copy the following XML into the **<activeProfiles>** element of the **settings.xml** file.

```

<activeProfile>jboss-ga-repository</activeProfile>
<activeProfile>jboss-earlyaccess-repository</activeProfile>

```

4. If you modify the **settings.xml** file while Red Hat JBoss Developer Studio is running, follow the procedure below entitled [Procedure 2.9, “Refresh the Red Hat JBoss Developer Studio User Settings”](#).

Procedure 2.8. Configure the Settings to Use a Locally Installed JBoss EAP Repository

You can modify the settings to use the JBoss EAP 6 repository installed on the local file system.

1. Locate the Maven install directory for your operating system. It is usually installed in **USER_HOME/.m2/** directory.
 - o For Linux or Mac, this is **~/ .m2/**
 - o For Windows, this is **\Documents and Settings\USER_NAME\.m2** or **\Users\USER_NAME\.m2**
2. If you do not find a **settings.xml** file, copy the **settings.xml** file from the **USER_HOME/.m2/conf/** directory into the **USER_HOME/.m2/** directory.
3. Copy the following XML into the **<profiles>** element of the **settings.xml** file. Be sure to change the **<url>** to the actual repository location.

```

<profile>
  <id>jboss-eap-repository</id>
  <repositories>
    <repository>
      <id>jboss-eap-repository</id>
      <name>JBoss EAP Maven Repository</name>
      <url>file:///path/to/repo/jboss-eap-6.x-maven-repository</url>
      <layout>default</layout>
      <releases>
        <enabled>true</enabled>
        <updatePolicy>never</updatePolicy>
      </releases>
      <snapshots>
        <enabled>>false</enabled>
        <updatePolicy>never</updatePolicy>
      </snapshots>
    </repository>
  </repositories>
</profile>

```

```

</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>jboss-eap-repository-group</id>
    <name>JBoss EAP Maven Repository</name>
    <url>
      file:///path/to/repo/jboss-eap-6.x-maven-repository
    </url>
    <layout>default</layout>
    <releases>
      <enabled>true</enabled>
      <updatePolicy>never</updatePolicy>
    </releases>
    <snapshots>
      <enabled>false</enabled>
      <updatePolicy>never</updatePolicy>
    </snapshots>
  </pluginRepository>
</pluginRepositories>
</profile>

```

Copy the following XML into the **<activeProfiles>** element of the **settings.xml** file.

```
<activeProfile>jboss-eap-repository</activeProfile>
```

4. If you modify the **settings.xml** file while Red Hat JBoss Developer Studio is running, follow the procedure below entitled [Procedure 2.9, “Refresh the Red Hat JBoss Developer Studio User Settings”](#).

Procedure 2.9. Refresh the Red Hat JBoss Developer Studio User Settings

If you modify the **settings.xml** file while Red Hat JBoss Developer Studio is running, you must refresh the user settings.

1. From the menu, choose **Window** → **Preferences**.
2. In the **Preferences** Window, expand **Maven** and choose **User Settings**.
3. Click the **Update Settings** button to refresh the Maven user settings in Red Hat JBoss Developer Studio.

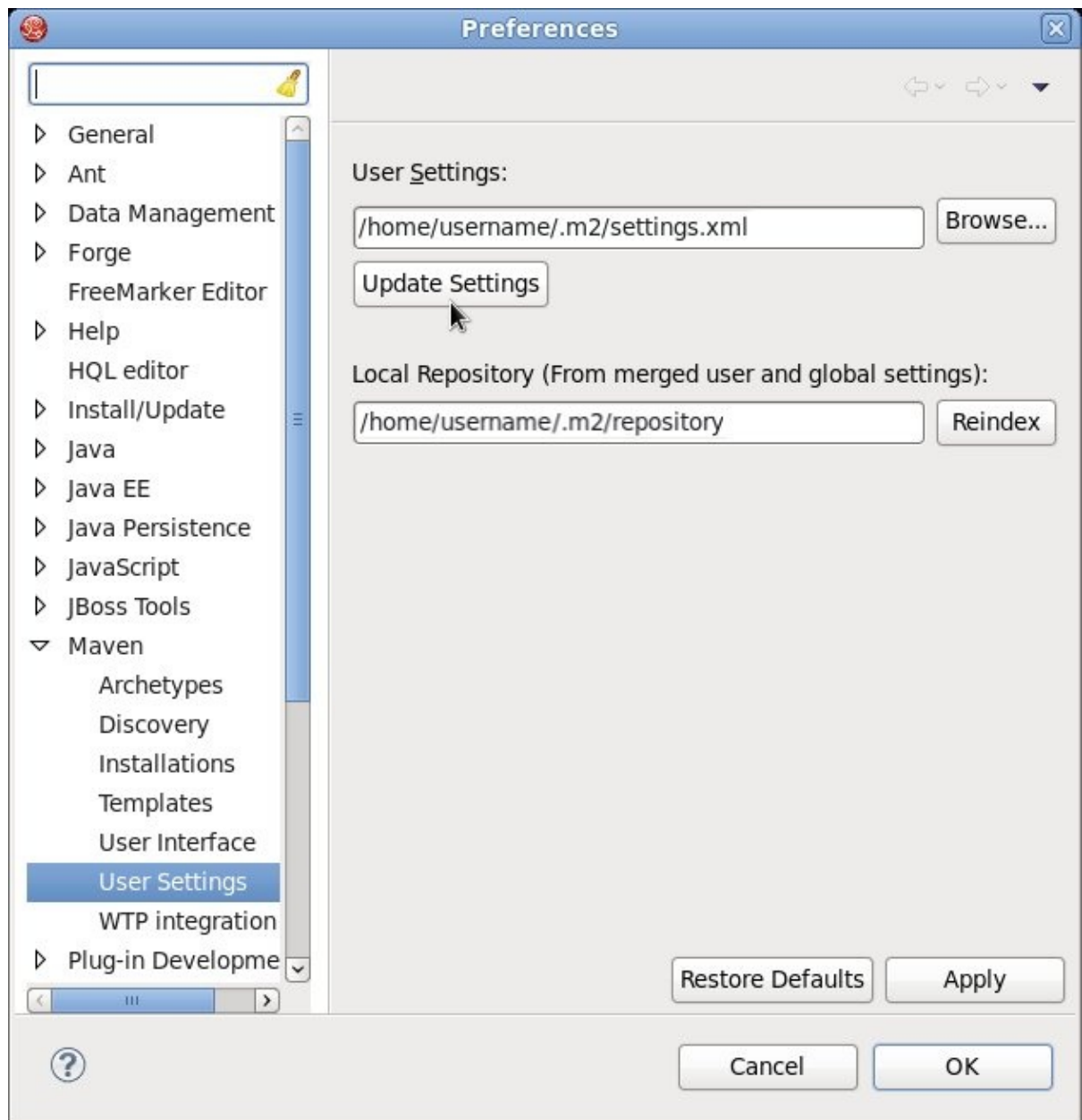


Figure 2.1. Update Maven User Settings

IMPORTANT

If your Maven repository contains outdated artifacts, you may encounter one of the following Maven error messages when you build or deploy your project:

- Missing artifact *ARTIFACT_NAME*
- [ERROR] Failed to execute goal on project *PROJECT_NAME*; Could not resolve dependencies for *PROJECT_NAME*

To resolve the issue, delete the cached version of your local repository to force a download of the latest Maven artifacts. The cached repository is located in your `~/.m2/repository/` subdirectory on Linux, or the `%SystemDrive%\Users\USERNAME\.m2\repository\` subdirectory on Windows.

[Report a bug](#)

2.3.3. Configure Maven for Use with Red Hat JBoss Developer Studio

The artifacts and dependencies needed to build and deploy applications to Red Hat JBoss Enterprise Application Platform are hosted on a public repository. You must direct Maven to use this repository when you build your applications. This topic covers the steps to configure Maven if you plan to build and deploy applications using Red Hat JBoss Developer Studio.

Maven is distributed with Red Hat JBoss Developer Studio, so it is not necessary to install it separately. However, you must configure Maven for use by the Java EE Web Project wizard for deployments to JBoss EAP. The procedure below demonstrates how to configure Maven for use with JBoss EAP by editing the Maven configuration file from within Red Hat JBoss Developer Studio.

Procedure 2.10. Configure Maven in Red Hat JBoss Developer Studio

1. Click **Window** → **Preferences**, expand **JBoss Tools** and select **JBoss Maven Integration**.

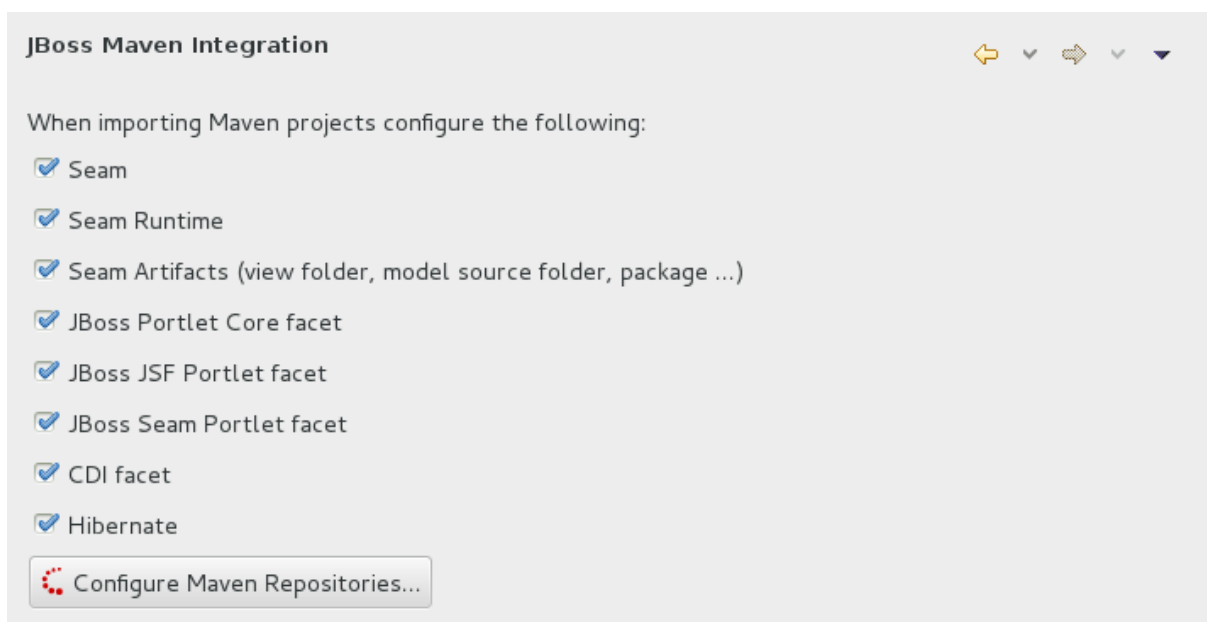
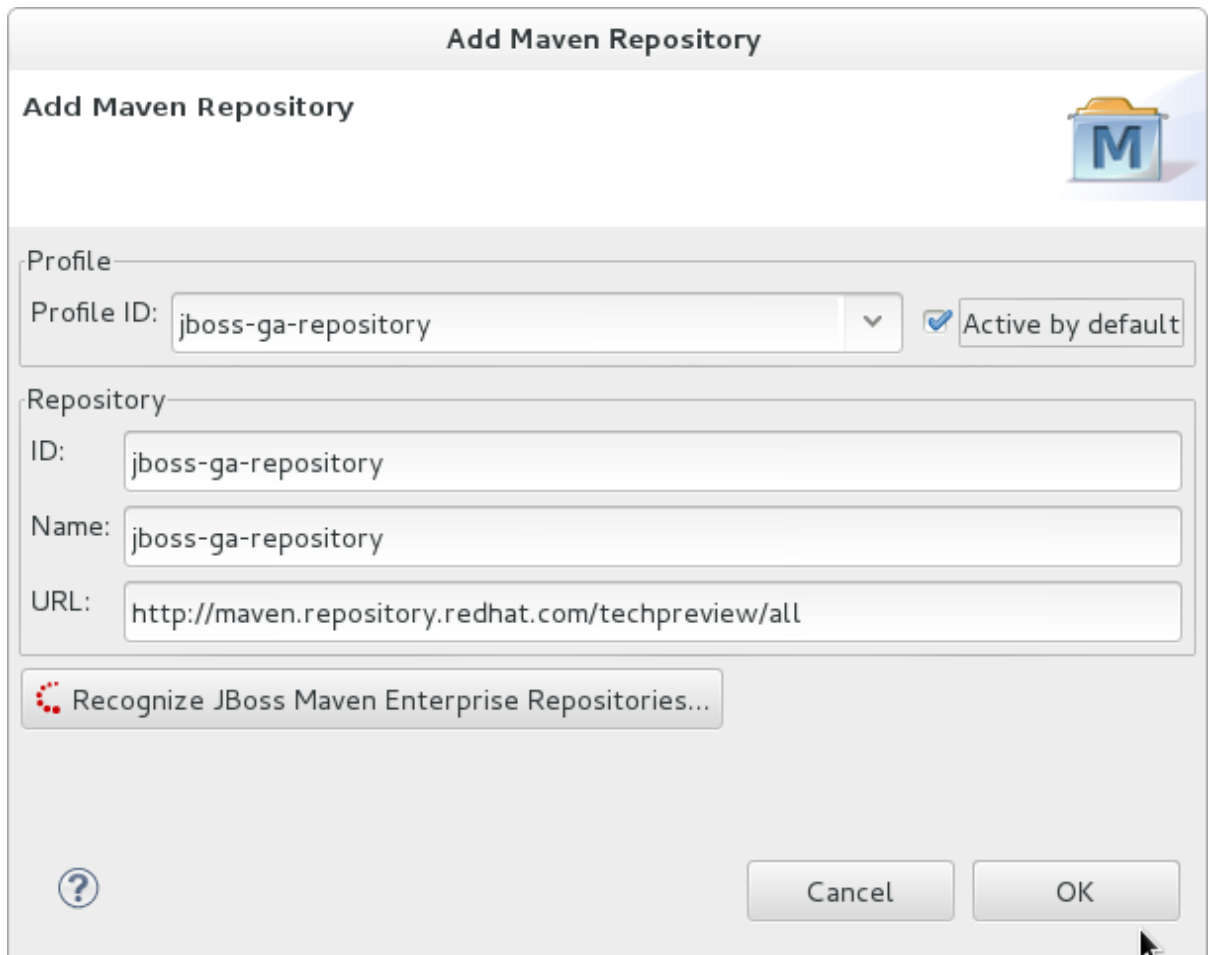


Figure 2.2. JBoss Maven Integration Pane in the Preferences Window

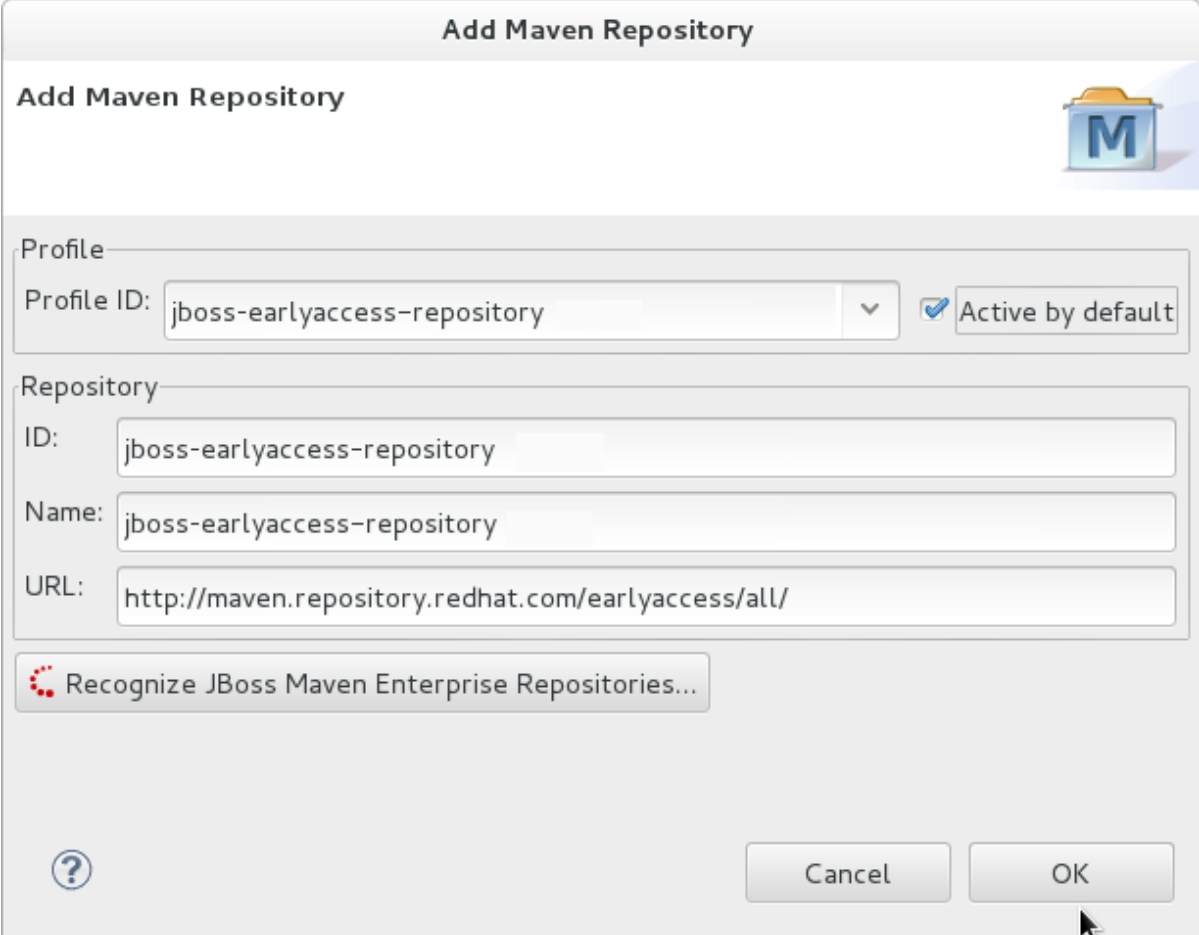
2. Click **Configure Maven Repositories**.
3. Click **Add Repository** to configure the JBoss GA Tech Preview Maven repository. Complete the **Add Maven Repository** dialog as follows:
 - a. Set the **Profile ID**, **Repository ID**, and **Repository Name** values to **jboss-ga-repository**.
 - b. Set the **Repository URL** value to **`http://maven.repository.redhat.com/techpreview/all`**.
 - c. Click the **Active by default** checkbox to enable the Maven repository.
 - d. Click **OK**



The image shows a dialog box titled "Add Maven Repository". It has a header bar with the title and a small icon of a folder with an 'M'. Below the header, there are two main sections: "Profile" and "Repository". In the "Profile" section, there is a "Profile ID:" label followed by a text box containing "jboss-ga-repository" and a dropdown arrow. To the right of this is a checkbox labeled "Active by default" which is checked. In the "Repository" section, there are three labels: "ID:", "Name:", and "URL:". Each has a corresponding text box. The "ID:" and "Name:" boxes both contain "jboss-ga-repository". The "URL:" box contains "http://maven.repository.redhat.com/techpreview/all". Below these text boxes is a button with a red circular icon and the text "Recognize JBoss Maven Enterprise Repositories...". At the bottom left is a question mark icon. At the bottom right are two buttons: "Cancel" and "OK". A mouse cursor is pointing at the "OK" button.

Figure 2.3. Add Maven Repository - JBoss Tech Preview

4. Click **Add Repository** to configure the JBoss Early Access Maven repository. Complete the **Add Maven Repository** dialog as follows:
 - a. Set the **Profile ID**, **Repository ID**, and **Repository Name** values to **jboss-earlyaccess-repository**.
 - b. Set the **Repository URL** value to **http://maven.repository.redhat.com/earlyaccess/all/**.
 - c. Click the **Active by default** checkbox to enable the Maven repository.
 - d. Click **OK**



The image shows a Java Swing dialog box titled "Add Maven Repository". The dialog has a title bar with the text "Add Maven Repository". Inside the dialog, there is a header area with the text "Add Maven Repository" and a small icon of a blue box with a yellow 'M' on it. Below the header, there are two main sections: "Profile" and "Repository". The "Profile" section contains a "Profile ID:" label followed by a text field containing "jboss-earlyaccess-repository" and a dropdown arrow. To the right of the text field is a checkbox labeled "Active by default" which is checked. The "Repository" section contains three labels: "ID:", "Name:", and "URL:". Each label is followed by a text field. The "ID:" field contains "jboss-earlyaccess-repository", the "Name:" field contains "jboss-earlyaccess-repository", and the "URL:" field contains "http://maven.repository.redhat.com/earlyaccess/all/". Below the "Repository" section, there is a button with a red circular icon and the text "Recognize JBoss Maven Enterprise Repositories...". At the bottom left of the dialog is a question mark icon. At the bottom right are two buttons: "Cancel" and "OK". A mouse cursor is pointing at the "OK" button.

Add Maven Repository

Add Maven Repository

Profile


Profile ID: jboss-earlyaccess-repository ☐ Active by default

Repository

ID: jboss-earlyaccess-repository

Name: jboss-earlyaccess-repository

URL: http://maven.repository.redhat.com/earlyaccess/all/

 Recognize JBoss Maven Enterprise Repositories...


 Cancel OK

Figure 2.4. Add Maven Repository - JBoss Early Access

5. Review the repositories and click **Finish**.

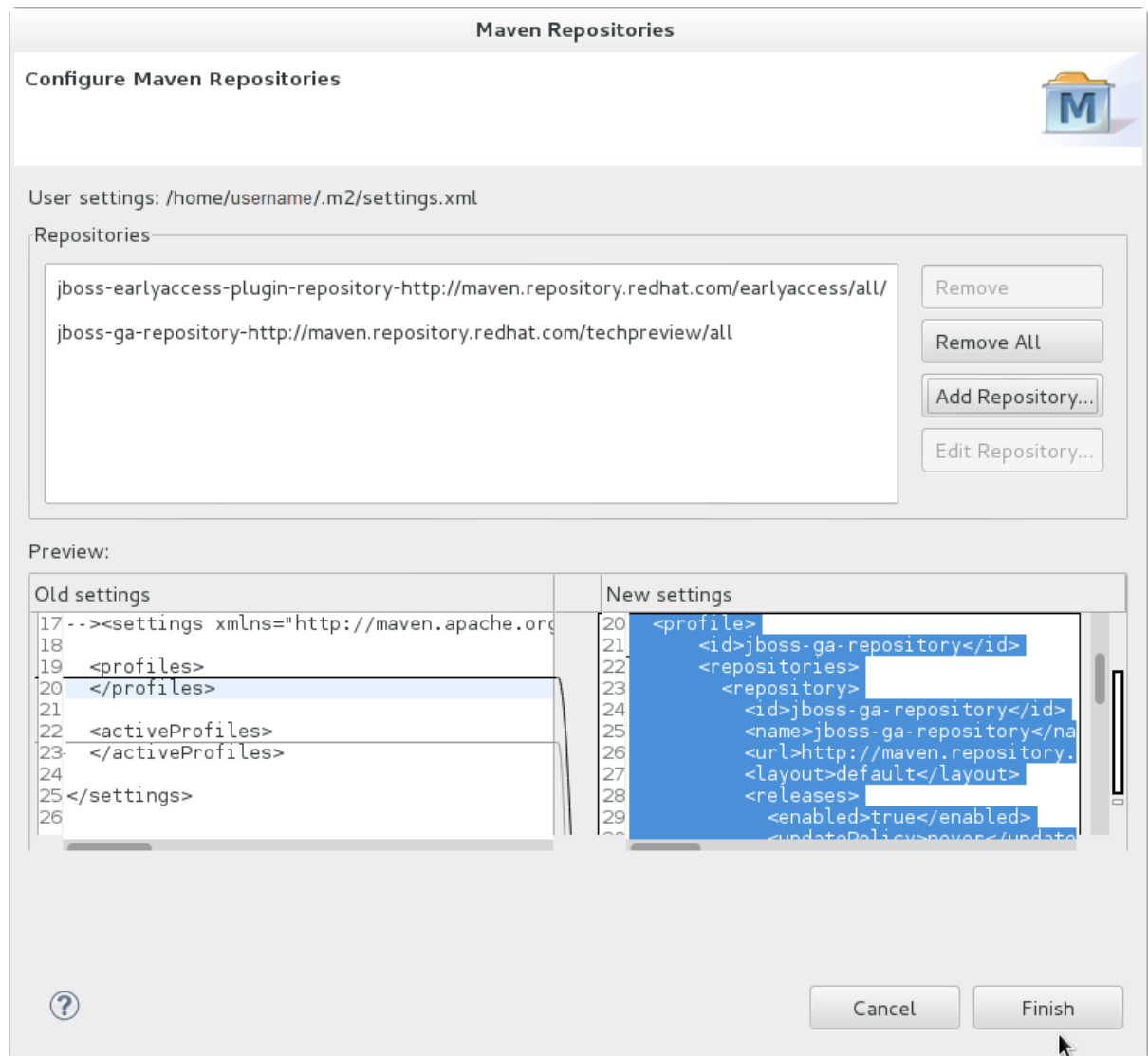


Figure 2.5. Review Maven Repositories

6. You are prompted with the message "Are you sure you want to update the file 'MAVEN_HOME/settings.xml'?". Click **Yes** to update the settings. Click **OK** to close the dialog.

The JBoss EAP Maven repository is now configured for use with Red Hat JBoss Developer Studio.

[Report a bug](#)

2.3.4. Configure the JBoss EAP 6 Maven Repository Using the Project POM

There are two approaches to direct Maven to use the JBoss EAP 6 Maven Repository in your project:

- You can modify the Maven settings.
- You can configure the project's POM file.

This task shows you how to configure a specific project to use the JBoss EAP 6 Maven Repository by adding repository information to the project **pom.xml**. This configuration method supercedes and overrides the global and user settings configurations.

This method of configuration is generally not recommended. If you decide to configure repositories in your project POM file, plan carefully and be aware that it can slow down your build and you may even end up with artifacts that are not from the expected repository.

NOTE

In an Enterprise environment, where a repository manager is usually used, Maven should query all artifacts for all projects using this manager. Because Maven uses all declared repositories to find missing artifacts, if it can't find what it's looking for, it will try and look for it in the repository central (defined in the built-in parent POM). To override this central location, you can add a definition with **central** so that the default repository central is now your repository manager as well. This works well for established projects, but for clean or 'new' projects it causes a problem as it creates a cyclic dependency.

Transitively included POMs are also an issue with this type of configuration. Maven has to query these external repositories for missing artifacts. This not only slows down your build, it also causes you to lose control over where your artifacts are coming from and likely to cause broken builds.

NOTE

The URL of the repository will depend on where the repository is located; on the filesystem, or web server. For information on how to install the repository, see: [Section 2.2.2, "Install the JBoss EAP 6 Maven Repository"](#). The following are examples for each of the installation options:

File System

file:///path/to/repo/jboss-eap-6.x-maven-repository

Apache Web Server

http://intranet.acme.com/jboss-eap-6.x-maven-repository/

Nexus Repository Manager

https://intranet.acme.com/nexus/content/repositories/jboss-eap-6.x-maven-repository

1. Open your project's **pom.xml** file in a text editor.
2. Add the following repository configuration. If there is already a **<repositories>** configuration in the file, then add the **<repository>** element to it. Be sure to change the **<url>** to the actual repository location.

```
<repositories>
  <repository>
    <id>jboss-eap-repository-group</id>
    <name>JBoss EAP Maven Repository</name>
    <url>file:///path/to/repo/jboss-eap-6.x.0-maven-
repository/</url>
    <layout>default</layout>
    <releases>
      <enabled>true</enabled>
      <updatePolicy>never</updatePolicy>
    </releases>
    <snapshots>
      <enabled>true</enabled>
      <updatePolicy>never</updatePolicy>
```

```

    </snapshots>
  </repository>
</repositories>

```

3. Add the following plug-in repository configuration. If there is already a **<pluginRepositories>** configuration in the file, then add the **<pluginRepository>** element to it.

```

<pluginRepositories>
  <pluginRepository>
    <id>jboss-eap-repository-group</id>
    <name>JBoss EAP Maven Repository</name>
    <url>file:///path/to/repo/jboss-eap-6.x.0-maven-
repository/</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </pluginRepository>
</pluginRepositories>

```

[Report a bug](#)

2.3.5. Manage Project Dependencies

This topic describes the usage of Bill of Materials (BOM) POMs for Red Hat JBoss Enterprise Application Platform 6.

A BOM is a Maven **pom.xml** (POM) file that specifies the versions of all runtime dependencies for a given module. Version dependencies are listed in the dependency management section of the file.

A project uses a BOM by adding its **groupId:artifactId:version** (GAV) to the dependency management section of the project **pom.xml** file and specifying the **<scope>import</scope>** and **<type>pom</type>** element values.



NOTE

In many cases, dependencies in project POM files use the **provided** scope. This is because these classes are provided by the application server at runtime and it is not necessary to package them with the user application.

Supported Maven Artifacts

As part of the product build process, all runtime components of JBoss EAP are built from source in a controlled environment. This helps to ensure that the binary artifacts do not contain any malicious code, and that they can be supported for the life of the product. These artifacts can be easily identified by the **-redhat** version qualifier, for example **1.0.0-redhat-1**.

Adding a supported artifact to the build configuration **pom.xml** file ensures that the build is using the correct binary artifact for local building and testing. Note that an artifact with a **-redhat** version is not necessarily part of the supported public API, and may change in future revisions. For information about the public supported API, see the JavaDoc documentation included in the release.

For example, to use the supported version of hibernate, add something similar to the following to your build configuration.

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>4.2.16.Final-redhat-1</version>
  <scope>provided</scope>
</dependency>
```

Notice that the above example includes a value for the **<version/>** field. However, it is recommended to use Maven dependency management for configuring dependency versions.

Dependency Management

Maven includes a mechanism for managing the versions of direct and transitive dependencies throughout the build. For general information about using dependency management, see the Apache Maven Project [Introduction to the Dependency Mechanism](#).

Using one or more supported JBoss dependencies directly in your build does not guarantee that all transitive dependencies of the build will be fully supported JBoss artifacts. It is common for Maven builds to use a mix of artifact sources from the Maven central repository, the JBoss.org Maven repository, and other Maven repositories.

Included with the JBoss EAP Maven repository is a dependency management BOM, which specifies all supported JBoss EAP binary artifacts. This BOM can be used in a build to ensure that Maven will prioritize supported JBoss EAP dependencies for all direct and transitive dependencies in the build. In other words, transitive dependencies will be managed to the correct supported dependency version where applicable. The version of this BOM matches the version of the JBoss EAP release.

```
<dependencyManagement>
  <dependencies>
    ...
    <dependency>
      <groupId>org.jboss.bom</groupId>
      <artifactId>eap6-supported-artifacts</artifactId>
      <version>6.4.0.GA</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>
```

JBoss JavaEE Specs Bom

The **jboss-javaee-6.0** BOM contains the Java EE Specification API JARs used by JBoss EAP.

To use this BOM in a project, add a dependency for the GAV that contains the version of the JSP and Servlet API JARs needed to build and deploy the application.

The following example uses the **3.0.2.Final-redhat-x** version of the **jboss-javaee-6.0** BOM.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.spec</groupId>
```

```

        <artifactId>jboss-javaee-6.0</artifactId>
        <version>3.0.2.Final-redhat-x</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
    ...
</dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
        <groupId>org.jboss.spec.javax.servlet</groupId>
        <artifactId>jboss-servlet-api_3.0_spec</artifactId>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.jboss.spec.javax.servlet.jsp</groupId>
        <artifactId>jboss-jsp-api_2.2_spec</artifactId>
        <scope>provided</scope>
    </dependency>
    ...
</dependencies>

```

JBoss EAP BOMs and Quickstarts

The JBoss BOMs are located in the jboss-bom project at <https://github.com/jboss-developer/jboss-eap-boms>.

The quickstarts provide the primary use case examples for the Maven repository. The following table lists the Maven BOMs used by the quickstarts.

Table 2.1. JBoss BOMs Used by the Quickstarts

Maven artifactId	Description
jboss-javaee-6.0-with-hibernate	This BOM builds on the Java EE full profile BOM, adding Hibernate Community projects including Hibernate ORM, Hibernate Search and Hibernate Validator. It also provides tool projects such as Hibernate JPA Model Gen and Hibernate Validator Annotation Processor.
jboss-javaee-6.0-with-hibernate3	This BOM builds on the Java EE full profile BOM, adding Hibernate Community projects including Hibernate 3 ORM, Hibernate Entity Manager (JPA 1.0) and Hibernate Validator.
jboss-javaee-6.0-with-logging	This BOM builds on the Java EE full profile BOM, adding the JBoss Logging Tools and Log4j framework.
jboss-javaee-6.0-with-osgi	This BOM builds on the Java EE full profile BOM, adding OSGI.
jboss-javaee-6.0-with-resteasy	This BOM builds on the Java EE full profile BOM, adding RESTEasy
jboss-javaee-6.0-with-security	This BOM builds on the Java EE full profile BOM, adding Picketlink.

Maven artifactId	Description
jboss-javaee-6.0-with-tools	This BOM builds on the Java EE full profile BOM, adding Arquillian to the mix. It also provides a version of JUnit and TestNG recommended for use with Arquillian.
jboss-javaee-6.0-with-transactions	This BOM includes a world class transaction manager. Use the JBossTS APIs to access its full capabilities.

The following example uses the **6.4.0.GA** version of the **jboss-javaee-6.0-with-hibernate** BOM.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.bom.eap</groupId>
      <artifactId>jboss-javaee-6.0-with-hibernate</artifactId>
      <version>6.4.0.GA</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <scope>provided</scope>
  </dependency>
  ...
</dependencies>
```

JBoss Client BOMs

The JBoss EAP server build includes two client BOMs: **jboss-as-ejb-client-bom** and **jboss-as-jms-client-bom**.

The client BOMs do not create a dependency management section or define dependencies. Instead, they are an aggregate of other BOMs and are used to package the set of dependencies necessary for a remote client use case.

The following example uses the **7.4.0.Final-redhat-x** version of the **jboss-as-ejb-client-bom** client BOM.

```
<dependencies>
  <dependency>
    <groupId>org.jboss.as</groupId>
    <artifactId>jboss-as-ejb-client-bom</artifactId>
    <version>7.5.0.Final-redhat-x</version>
    <type>pom</type>
```

```
</dependency>
...1
</dependencies>
```

This example uses the **7.4.0.Final-redhat-x** version of the **jboss-as-jms-client-bom** client BOM.

```
<dependencies>
  <dependency>
    <groupId>org.jboss.as</groupId>
    <artifactId>jboss-as-jms-client-bom</artifactId>
    <version>7.4.0.Final-redhat-x</version>
    <type>pom</type>
  </dependency>
  ...
</dependencies>
```

For more information about Maven Dependencies and BOM POM files, see [Apache Maven Project - Introduction to the Dependency Mechanism](#).

[Report a bug](#)

2.4. UPGRADE THE MAVEN REPOSITORY

2.4.1. Apply a Patch to the Local Maven Repository

Summary

A Maven repository stores Java libraries, plug-ins, and other artifacts required to build and deploy applications to JBoss EAP. The JBoss EAP repository is available online or as a downloaded ZIP file. If you use the publicly hosted repository, updates are applied automatically for you. However, if you download and install the Maven repository locally, you are responsible for applying any updates. Whenever a patch is available for JBoss EAP, a corresponding patch is provided for the JBoss EAP Maven repository. This patch is available in the form of an incremental ZIP file that is unzipped into the existing local repository. The ZIP file contains new JAR and POM files. It does not overwrite any existing JARs nor does it remove JARs, so there is no rollback requirement.

For more information about the JBoss EAP patching process, see the chapter entitled *Patching and Upgrading JBoss EAP 6* in the *Installation Guide* for JBoss Enterprise Application Platform 6 located on the Customer Portal at https://access.redhat.com/documentation/en-us/red_hat_jboss_enterprise_application_platform/?version=6.4.

This task describes how to apply Maven updates to your locally installed Maven repository using the **unzip** command.

Prerequisites

- Valid access and subscription to the Red Hat Customer Portal.
- The Red Hat JBoss Enterprise Application Platform <VERSION> Maven Repository ZIP file, downloaded and installed locally.

Procedure 2.11. Update the Maven Repository

1. Open a browser and log into <https://access.redhat.com>.
2. Select **Downloads** from the menu at the top of the page.
3. Find **Red Hat JBoss Enterprise Application Platform** in the list and click on it.
4. Select the correct version of JBoss EAP from the **Version** drop-down menu that appears on this screen, then click on **Patches**.
5. Find **Red Hat JBoss Enterprise Application Platform <VERSION> CPx Incremental Maven Repository** in the list and click **Download**.
6. You are prompted to save the ZIP file to a directory of your choice. Choose a directory and save the file.
7. Locate the path to JBoss EAP Maven repository, referred to in the commands below as *EAP_MAVEN_REPOSITORY_PATH*, for your operating system. For more information about how to install the Maven repository on the local file system, see [Section 2.2.3, “Install the JBoss EAP 6 Maven Repository Locally”](#).
8. Unzip the Maven patch file directly into the installation directory of the JBoss EAP <VERSION>.x Maven repository.
 - For Linux, open a terminal and type the following command:


```
[standalone@localhost:9999 /] unzip -o jboss-eap-<VERSION>.x-incremental-maven-repository.zip -d EAP_MAVEN_REPOSITORY_PATH
```
 - For Windows, use the Windows extraction utility to extract the ZIP file into the root of the *EAP_MAVEN_REPOSITORY_PATH* directory.

Result

The locally installed Maven repository is updated with the latest patch.

[Report a bug](#)

CHAPTER 3. CLASS LOADING AND MODULES

3.1. INTRODUCTION

3.1.1. Overview of Class Loading and Modules

JBoss EAP 6 uses a new modular class loading system for controlling the class paths of deployed applications. This system provides more flexibility and control than the traditional system of hierarchical class loaders. Developers have fine-grained control of the classes available to their applications, and can configure a deployment to ignore classes provided by the application server in favor of their own.

The modular class loader separates all Java classes into logical groups called modules. Each module can define dependencies on other modules in order to have the classes from that module added to its own class path. Because each deployed JAR and WAR file is treated as a module, developers can control the contents of their application's class path by adding module configuration to their application.

[Report a bug](#)

3.1.2. Class Loading

Class Loading is the mechanism by which Java classes and resources are loaded into the Java Runtime Environment.

[Report a bug](#)

3.1.3. Modules

A Module is a logical grouping of classes used for class loading and dependency management. JBoss EAP 6 identifies two different types of modules, sometimes called static and dynamic modules. However the only difference between the two is how they are packaged.

Static Modules

Static Modules are predefined in the **EAP_HOME/modules/** directory of the application server. Each sub-directory represents one module and defines a **main/** subdirectory that contains a configuration file (**module.xml**) and any required JAR files. The name of the module is defined in the **module.xml** file. All the application server provided APIs are provided as static modules, including the Java EE APIs as well as other APIs such as JBoss Logging.

Example 3.1. Example module.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.0" name="com.mysql">
  <resources>
    <resource-root path="mysql-connector-java-5.1.15.jar"/>
  </resources>
  <dependencies>
    <module name="javax.api"/>
    <module name="javax.transaction.api"/>
  </dependencies>
</module>
```

The module name, **com.mysql**, should match the directory structure for the module, excluding the **main/** subdirectory name.



The modules provided in JBoss EAP distributions are located in a **system** directory within the **EAP_HOME/modules** directory. This keeps them separate from any modules provided by third parties.

Any Red Hat provided layered products that layer on top of JBoss EAP 6.1 or later will also install their modules within the **system** directory.

Creating custom static modules can be useful if many applications are deployed on the same server that use the same third-party libraries. Instead of bundling those libraries with each application, a module containing these libraries can be created and installed by the JBoss administrator. The applications can then declare an explicit dependency on the custom static modules.

Users must ensure that custom modules are installed into the **EAP_HOME/modules** directory, using a one directory per module layout. This ensures that custom versions of modules that already exist in the **system** directory are loaded instead of the shipped versions. In this way, user provided modules will take precedence over system modules.

If you use the **JBOSS_MODULEPATH** environment variable to change the locations in which JBoss EAP searches for modules, then the product will look for a **system** subdirectory structure within one of the locations specified. A **system** structure must exist somewhere in the locations specified with **JBOSS_MODULEPATH**.

Dynamic Modules

Dynamic Modules are created and loaded by the application server for each JAR or WAR deployment (or subdeployment in an EAR). The name of a dynamic module is derived from the name of the deployed archive. Because deployments are loaded as modules, they can configure dependencies and be used as dependencies by other deployments.

Modules are only loaded when required. This usually only occurs when an application is deployed that has explicit or implicit dependencies.

[Report a bug](#)

3.1.4. Module Dependencies

A module dependency is a declaration that one module requires the classes of another module in order to function. Modules can declare dependencies on any number of other modules. When the application server loads a module, the modular class loader parses the dependencies of that module and adds the classes from each dependency to its class path. If a specified dependency cannot be found, the module will fail to load.

Deployed applications (JAR and WAR) are loaded as dynamic modules and make use of dependencies to access the APIs provided by JBoss EAP 6.

There are two types of dependencies: explicit and implicit.

Explicit Dependencies

Explicit dependencies are declared by the developer in the configuration file. Static modules can declare dependencies in the **module.xml** file. Dynamic modules can have dependencies declared in the **MANIFEST.MF** or **jboss-deployment-structure.xml** deployment descriptors of the deployment.

Explicit dependencies can be specified as optional. Failure to load an optional dependency will not cause a module to fail to load. However if the dependency becomes available later it will NOT be added to the module's class path. Dependencies must be available when the module is loaded.

Implicit Dependencies

Implicit dependencies are added automatically by the application server when certain conditions or meta-data are found in a deployment. The Java EE 6 APIs supplied with JBoss EAP 6 are examples of modules that are added by detection of implicit dependencies in deployments.

Deployments can also be configured to exclude specific implicit dependencies. This is done with the **jboss-deployment-structure.xml** deployment descriptor file. This is commonly done when an application bundles a specific version of a library that the application server will attempt to add as an implicit dependency.

A module's class path contains only its own classes and that of its immediate dependencies. A module is not able to access the classes of the dependencies of one of its dependencies. However a module can specify that an explicit dependency is exported. An exported dependency is provided to any module that depends on the module that exports it.

Example 3.2. Module dependencies

Module A depends on Module B and Module B depends on Module C. Module A can access the classes of Module B, and Module B can access the classes of Module C. Module A cannot access the classes of Module C unless:

- Module A declares an explicit dependency on Module C, or
- Module B exports its dependency on Module C.

[Report a bug](#)

3.1.5. Class Loading in Deployments

For the purposes of class loading, all deployments are treated as modules by JBoss EAP 6. These are called dynamic modules. Class loading behavior varies according to the deployment type.

WAR Deployment

A WAR deployment is considered to be a single module. Classes in the **WEB-INF/lib** directory are treated the same as classes in **WEB-INF/classes** directory. All classes packaged in the WAR will be loaded with the same class loader.

EAR Deployment

EAR deployments are made up of more than one module. The definition of these modules follows these rules:

1. The **lib/** directory of the EAR is a single module called the parent module.
2. Each WAR deployment within the EAR is a single module.
3. Each EJB JAR deployment within the EAR is a single module.

Subdeployment modules (the WAR and JAR deployments within the EAR) have an automatic dependency on the parent module. However they do not have automatic dependencies on each

other. This is called subdeployment isolation and can be disabled on a per deployment basis or for the entire application server.

Explicit dependencies between subdeployment modules can be added by the same means as any other module.

[Report a bug](#)

3.1.6. Class Loading Precedence

The JBoss EAP 6 modular class loader uses a precedence system to prevent class loading conflicts.

During deployment a complete list of packages and classes is created for each deployment and each of its dependencies. The list is ordered according to the class loading precedence rules. When loading classes at runtime, the class loader searches this list, and loads the first match. This prevents multiple copies of the same classes and packages within the deployments class path from conflicting with each other.

The class loader loads classes in the following order, from highest to lowest:

1. Implicit dependencies.

These are the dependencies that are added automatically by JBoss EAP 6, such as the JAVA EE APIs. These dependencies have the highest class loader precedence because they contain common functionality and APIs that are supplied by JBoss EAP 6.

Refer to [Section 3.9.1, “Implicit Module Dependencies”](#) for complete details about each implicit dependency.

2. Explicit dependencies.

These are dependencies that are manually added in the application configuration. This can be done using the application's **MANIFEST.MF** file or the new optional JBoss deployment descriptor **jboss-deployment-structure.xml** file.

Refer to [Section 3.2, “Add an Explicit Module Dependency to a Deployment”](#) to learn how to add explicit dependencies.

3. Local resources.

Class files packaged up inside the deployment itself, e.g. from the **WEB-INF/classes** or **WEB-INF/lib** directories of a WAR file.

4. Inter-deployment dependencies.

These are dependencies on other deployments in a EAR deployment. This can include classes in the **lib** directory of the EAR or classes defined in other EJB jars.

[Report a bug](#)

3.1.7. Dynamic Module Naming

All deployments are loaded as modules by JBoss EAP 6 and named according to the following conventions.

- Deployments of WAR and JAR files are named with the following format:

```
deployment.DEPLOYMENT_NAME
```

For example, **inventory.war** and **store.jar** will have the module names of **deployment.inventory.war** and **deployment.store.jar** respectively.

- Subdeployments within an Enterprise Archive are named with the following format:

```
deployment.EAR_NAME.SUBDEPLOYMENT_NAME
```

For example, the subdeployment of **reports.war** within the enterprise archive **accounts.ear** will have the module name of **deployment.accounts.ear.reports.war**.

[Report a bug](#)

3.1.8. jboss-deployment-structure.xml

jboss-deployment-structure.xml is a new optional deployment descriptor for JBoss EAP 6. This deployment descriptor provides control over class loading in the deployment.

The XML schema for this deployment descriptor is in ***EAP_HOME/docs/schema/jboss-deployment-structure-1_2.xsd***

[Report a bug](#)

3.2. ADD AN EXPLICIT MODULE DEPENDENCY TO A DEPLOYMENT

This task shows how to add an explicit dependency to an application. Explicit module dependencies can be added to applications to add the classes of those modules to the class path of the application at deployment.

Some dependencies are automatically added to deployments by JBoss EAP 6. See [Section 3.9.1, “Implicit Module Dependencies”](#) for details.

Prerequisites

1. You must already have a working software project that you want to add a module dependency to.
2. You must know the name of the module being added as a dependency. See [Section 3.9.2, “Included Modules”](#) for the list of static modules included with JBoss EAP 6. If the module is another deployment then see [Section 3.1.7, “Dynamic Module Naming”](#) to determine the module name.

Dependencies can be configured using two different methods:

1. Adding entries to the **MANIFEST.MF** file of the deployment.
2. Adding entries to the **jboss-deployment-structure.xml** deployment descriptor.

Procedure 3.1. Add dependency configuration to MANIFEST.MF

Maven projects can be configured to create the required dependency entries in the **MANIFEST.MF** file. See [Section 3.3, “Generate MANIFEST.MF entries using Maven”](#).

1. Add **MANIFEST.MF** file

If the project has no **MANIFEST.MF** file, create a file called **MANIFEST.MF**. For a web application (WAR) add this file to the **META-INF** directory. For an EJB archive (JAR) add it to the **META-INF** directory.

2. Add dependencies entry

Add a dependencies entry to the **MANIFEST.MF** file with a comma-separated list of dependency module names.

```
Dependencies: org.javassist, org.apache.velocity
```

3. Optional: Make a dependency optional

A dependency can be made optional by appending **optional** to the module name in the dependency entry.

```
Dependencies: org.javassist optional, org.apache.velocity
```

4. Optional: Export a dependency

A dependency can be exported by appending **export** to the module name in the dependency entry.

```
Dependencies: org.javassist, org.apache.velocity export
```

5. Optional: Dependencies using annotations

This flag is needed when the module dependency contains annotations which need to be processed during annotation scanning, such as when declaring EJB Interceptors. If this is not done, an EJB interceptor declared in a module cannot be used in a deployment. There are other situations involving annotation scanning when this is needed too.

Using this flag requires that the module contain a Jandex index. Instructions for creating and using a Jandex index are included at the end of this topic.

Procedure 3.2. Add dependency configuration to **jboss-deployment-structure.xml**

1. Add **jboss-deployment-structure.xml**

If the application has no **jboss-deployment-structure.xml** file then create a new file called **jboss-deployment-structure.xml** and add it to the project. This file is an XML file with the root element of **<jboss-deployment-structure>**.

```
<jboss-deployment-structure>
</jboss-deployment-structure>
```

For a web application (WAR) add this file to the **WEB-INF** directory. For an EJB archive (JAR) add it to the **META-INF** directory.

2. Add dependencies section

Create a **<deployment>** element within the document root and a **<dependencies>** element within that.

3. Add module elements

Within the dependencies node, add a module element for each module dependency. Set the **name** attribute to the name of the module.

```
<module name="org.javassist" />
```

4. Optional: Make a dependency optional

A dependency can be made optional by adding the **optional** attribute to the module entry with the value of **true**. The default value for this attribute is **false**.

```
<module name="org.javassist" optional="true" />
```

5. Optional: Export a dependency

A dependency can be exported by adding the **export** attribute to the module entry with the value of **true**. The default value for this attribute is **false**.

```
<module name="org.javassist" export="true" />
```

Example 3.3. jboss-deployment-structure.xml with two dependencies

```
<jboss-deployment-structure>
  <deployment>
    <dependencies>
      <module name="org.javassist" />
      <module name="org.apache.velocity" export="true" />
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

JBoss EAP 6 will add the classes from the specified modules to the class path of the application when it is deployed.

Creating a Jandex index

The **annotations** flag requires that the module contain a Jandex index. You can create a new "index JAR" to add to the module. Use the Jandex JAR to build the index, and then insert it into a new JAR file:

Procedure 3.3.

1. Create the index

```
java -jar EAP_HOME/modules/org/jboss/jandex/main/jandex-1.0.3.Final-
redhat-1.jar $JAR_FILE
```

2. Create a temporary working space

```
mkdir /tmp/META-INF
```

3. Move the index file to the working directory

```
mv $JAR_FILE.ifx /tmp/META-INF/jandex.idx
```

4. Option 1: Include the index in a new JAR file

```
jar cf index.jar -C /tmp META-INF/jandex.idx
```

Then place the JAR in the module directory and edit **module.xml** to add it to the resource roots.

Option 2: Add the index to an existing JAR

```
java -jar EAP_HOME/modules/org/jboss/jandex/main/jandex-1.0.3.Final-redhat-1.jar -m $JAR_FILE
```

5. Tell the module import to utilize the annotation index

Tell the module import to utilize the annotation index, so that annotation scanning can find the annotations.

Choose one of the methods below based on your situation:

- If you are adding a module dependency using **MANIFEST.MF**, add **annotations** after the module name.

For example change:

```
Dependencies: test.module, other.module
```

to

```
Dependencies: test.module annotations, other.module
```

- If you are adding a module dependency using **jboss-deployment-structure.xml** add **annotations="true"** on the module dependency.

[Report a bug](#)

3.3. GENERATE MANIFEST.MF ENTRIES USING MAVEN

Maven projects that use the Maven JAR, EJB or WAR packaging plug-ins can generate a **MANIFEST.MF** file with a **Dependencies** entry. This does not automatically generate the list of dependencies, this process only creates the **MANIFEST.MF** file with the details specified in the **pom.xml**.

Prerequisites

1. You must already have a working Maven project.
2. The Maven project must be using one of the JAR, EJB, or WAR plug-ins (**maven-jar-plugin**, **maven-ejb-plugin**, **maven-war-plugin**).
3. You must know the name of the project's module dependencies. Refer to [Section 3.9.2, "Included Modules"](#) for the list of static modules included with JBoss EAP 6. If the module is

another deployment , then refer to [Section 3.1.7, “Dynamic Module Naming”](#) to determine the module name.

Procedure 3.4. Generate a MANIFEST.MF file containing module dependencies

1. Add Configuration

Add the following configuration to the packaging plug-in configuration in the project's `pom.xml` file.

```
<configuration>
  <archive>
    <manifestEntries>
      <Dependencies></Dependencies>
    </manifestEntries>
  </archive>
</configuration>
```

2. List Dependencies

Add the list of the module dependencies in the `<Dependencies>` element. Use the same format that is used when adding the dependencies to the `MANIFEST.MF`. Refer to [Section 3.2, “Add an Explicit Module Dependency to a Deployment”](#) for details about that format.

```
<Dependencies>org.javassist, org.apache.velocity</Dependencies>
```

The `optional` and `export` attributes can also be used here.

```
<Dependencies>org.javassist optional, org.apache.velocity
export</Dependencies>
```

3. Build the Project

Build the project using the Maven assembly goal.

```
[Localhost ]$ mvn assembly:assembly
```

When the project is built using the assembly goal, the final archive contains a `MANIFEST.MF` file with the specified module dependencies.

Example 3.4. Configured Module Dependencies in pom.xml

The example here shows the WAR plug-in but it also works with the JAR and EJB plug-ins (maven-jar-plugin and maven-ejb-plugin).

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-war-plugin</artifactId>
    <configuration>
      <archive>
        <manifestEntries>
          <Dependencies>org.javassist,
org.apache.velocity</Dependencies>
        </manifestEntries>
      </archive>
```



```

        </configuration>
    </plugin>
</plugins>

```

[Report a bug](#)

3.4. PREVENT A MODULE BEING IMPLICITLY LOADED

This task describes how to configure your application to exclude a list of module dependencies.

You can configure a deployable application to prevent implicit dependencies from being loaded. This is commonly done when the application includes a different version of a library or framework than the one that will be provided by the application server as an implicit dependency.

Prerequisites

1. You must already have a working software project that you want to exclude an implicit dependency from.
2. You must know the name of the module to exclude. Refer to [Section 3.9.1, “Implicit Module Dependencies”](#) for a list of implicit dependencies and their conditions.

Procedure 3.5. Add dependency exclusion configuration to `jboss-deployment-structure.xml`

1. If the application has no `jboss-deployment-structure.xml` file, create a new file called `jboss-deployment-structure.xml` and add it to the project. This file is an XML file with the root element of `<jboss-deployment-structure>`.

```

<jboss-deployment-structure>

</jboss-deployment-structure>

```

For a web application (WAR) add this file to the **WEB-INF** directory. For an EJB archive (JAR) add it to the **META-INF** directory.

2. Create a `<deployment>` element within the document root and an `<exclusions>` element within that.

```

<deployment>
    <exclusions>

    </exclusions>
</deployment>

```

3. Within the exclusions element, add a `<module>` element for each module to be excluded. Set the **name** attribute to the name of the module.

```

<module name="org.javassist" />

```

Example 3.5. Excluding two modules

```

<jboss-deployment-structure>
  <deployment>
    <exclusions>
      <module name="org.javassist" />
      <module name="org.dom4j" />
    </exclusions>
  </deployment>
</jboss-deployment-structure>

```

[Report a bug](#)

3.5. EXCLUDE A SUBSYSTEM FROM A DEPLOYMENT

Summary

This topic covers the steps required to exclude a subsystem from a deployment. This is done by editing the **jboss-deployment-structure.xml** configuration file. Excluding a subsystem provides the same effect as removing the subsystem, but it applies only to a single deployment.

Procedure 3.6. Exclude a Subsystem

1. Open the **jboss-deployment-structure.xml** file in a text editor.
2. Add the following XML inside the `<deployment>` tags:

```

<exclude-subsystems>
  <subsystem name="SUBSYSTEM_NAME" />
</exclude-subsystems>

```

3. Save the **jboss-deployment-structure.xml** file.

Result

The subsystem has been successfully excluded. The subsystem's deployment unit processors will no longer run on the deployment.

Example 3.6. Example jboss-deployment-structure.xml file.

```

<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.2">
  <ear-subdeployments-isolated>true</ear-subdeployments-isolated>
  <deployment>
    <exclude-subsystems>
      <subsystem name="jaxrs" />
    </exclude-subsystems>
    <exclusions>
      <module name="org.javassist" />
    </exclusions>
    <dependencies>
      <module name="deployment.javassist.proxy" />
      <module name="deployment.myjavassist" />
      <module name="myservicemodule" services="import"/>
    </dependencies>
    <resources>

```

```

        <resource-root path="my-library.jar" />
    </resources>
</deployment>
<sub-deployment name="myapp.war">
    <dependencies>
        <module name="deployment.myear.ear.myejbjar.jar" />
    </dependencies>
    <local-last value="true" />
</sub-deployment>
<module name="deployment.myjavassist" >
    <resources>
        <resource-root path="javassist.jar" >
            <filter>
                <exclude path="javassist/util/proxy" />
            </filter>
        </resource-root>
    </resources>
</module>
<module name="deployment.javassist.proxy" >
    <dependencies>
        <module name="org.javassist" >
            <imports>
                <include path="javassist/util/proxy" />
                <exclude path="/*" />
            </imports>
        </module>
    </dependencies>
</module>
</jboss-deployment-structure>

```

[Report a bug](#)

3.6. USE THE CLASS LOADER PROGRAMMATICALLY IN A DEPLOYMENT

3.6.1. Programmatically Load Classes and Resources in a Deployment

You can programmatically find or load classes and resources in your application code. The method you choose will depend on a number of factors. This topic describes the methods available and provides guidelines for when to use them.

Load a Class Using the `Class.forName()` Method

You can use the `Class.forName()` method to programmatically load and initialize classes. This method has two signatures.

`Class.forName(String className)`

This signature takes only one parameter, the name of the class you need to load. With this method signature, the class is loaded by the class loader of the current class and initializes the newly loaded class by default.

`Class.forName(String className, boolean initialize, ClassLoader loader)`

This signature expects three parameters: the class name, a boolean value that specifies whether to initialize the class, and the `ClassLoader` that should load the class.

The three argument signature is the recommended way to programmatically load a class. This signature allows you to control whether you want the target class to be initialized upon load. It is also more efficient to obtain and provide the class loader because the JVM does not need to examine the call stack to determine which class loader to use. Assuming the class containing the code is named **CurrentClass**, you can obtain the class's class loader using **CurrentClass.class.getClassLoader()** method.

The following example provides the class loader to load and initialize the **TargetClass** class:

Example 3.7. Provide a class loader to load and initialize the TargetClass.

```
Class<?> targetClass = Class.forName("com.myorg.util.TargetClass",
    true, CurrentClass.class.getClassLoader());
```

Find All Resources with a Given Name

If you know the name and path of a resource, the best way to load it directly is to use the standard Java development kit `Class` or `ClassLoader` API.

Load a Single Resource

To load a single resource located in the same directory as your class or another class in your deployment, you can use the **Class.getResourceAsStream()** method.

Example 3.8. Load a single resource in your deployment.

```
InputStream inputStream =
    CurrentClass.class.getResourceAsStream("targetResourceName");
```

Load All Instances of a Single Resource

To load all instances of a single resource that are visible to your deployment's class loader, use the **Class.getClassLoader().getResources(String resourceName)** method, where **resourceName** is the fully qualified path of the resource. This method returns an `Enumeration` of all **URL** objects for resources accessible by the class loader with the given name. You can then iterate through the array of URLs to open each stream using the **openStream()** method.

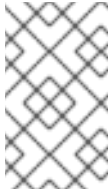
Example 3.9. Load all instances of a resource and iterate through the result.

```
Enumeration<URL> urls =
    CurrentClass.class.getClassLoader().getResources("full/path/to/resource");
while (urls.hasMoreElements()) {
    URL url = urls.nextElement();
    InputStream inputStream = null;
    try {
        inputStream = url.openStream();
        // Process the inputStream
        ...
    } catch (IOException ioException) {
        // Handle the error
    } finally {
```

```

        if (inputStream != null) {
            try {
                inputStream.close();
            } catch (Exception e) {
                // ignore
            }
        }
    }
}

```



NOTE

Because the URL instances are loaded from local storage, it is not necessary to use the `openConnection()` or other related methods. Streams are much simpler to use and minimize the complexity of the code.

Load a Class File From the Class Loader

If a class has already been loaded, you can load the class file that corresponds to that class using the following syntax:

Example 3.10. Load a class file for a class that has been loaded.

```

InputStream inputStream =
    CurrentClass.class.getResourceAsStream(TargetClass.class.getSimpleName() + ".class");

```

If the class is not yet loaded, you must use the class loader and translate the path:

Example 3.11. Load a class file for a class that has not been loaded.

```

String className = "com.myorg.util.TargetClass"
InputStream inputStream =
    CurrentClass.class.getClassLoader().getResourceAsStream(className.replace('.', '/') + ".class");

```

[Report a bug](#)

3.6.2. Programmatically Iterate Resources in a Deployment

The JBoss Modules library provides several APIs for iterating all deployment resources. The JavaDoc for the JBoss Modules API is located here: <http://docs.jboss.org/jbossmodules/1.3.0.Final/api/>. To use these APIs, you must add the following dependency to the `MANIFEST.MF`:

```
Dependencies: org.jboss.modules
```

It is important to note that while these APIs provide increased flexibility, they will also run much more slowly than a direct path lookup.

This topic describes some of the ways you can programmatically iterate through resources in your application code.

List Resources Within a Deployment and Within All Imports

There are times when it is not possible to look up resources by the exact path. For example, the exact path may not be known or you may need to examine more than one file in a given path. In this case, the JBoss Modules library provides several APIs for iterating all deployment resources. You can iterate through resources in a deployment by utilizing one of two methods.

Iterate All Resources Found in a Single Module

The `ModuleClassLoader.iterateResources()` method iterates all the resources within this module class loader. This method takes two arguments: the starting directory name to search and a boolean that specifies whether it should recurse into subdirectories.

The following example demonstrates how to obtain the `ModuleClassLoader` and obtain the iterator for resources in the `bin/` directory, recursing into subdirectories.

Example 3.12. Find resources in the "bin" directory, recursing into subdirectories.

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Iterator<Resource> mclResources =
moduleClassLoader.iterateResources("bin", true);
```

The resultant iterator may be used to examine each matching resource and query its name and size (if available), open a readable stream, or acquire a URL for the resource.

Iterate All Resources Found in a Single Module and Imported Resources

The `Module.iterateResources()` method iterates all the resources within this module class loader, including the resources that are imported into the module. This method returns a much larger set than the previous method. This method requires an argument, which is a filter that narrows the result to a specific pattern. Alternatively, `PathFilters.acceptAll()` can be supplied to return the entire set.

Example 3.13. Find the entire set of resources in this module, including imports.

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Module module = moduleClassLoader.getModule();
Iterator<Resource> moduleResources =
module.iterateResources(PathFilters.acceptAll());
```

Find All Resources That Match a Pattern

If you need to find only specific resources within your deployment or within your deployment's full import set, you need to filter the resource iteration. The JBoss Modules filtering APIs give you several tools to accomplish this.

Examine the Full Set of Dependencies

If you need to examine the full set of dependencies, you can use the `Module.iterateResources()` method's `PathFilter` parameter to check the name of each resource for a match.

Examine Deployment Dependencies

If you need to look only within the deployment, use the **ModuleClassLoader.iterateResources()** method. However, you must use additional methods to filter the resultant iterator. The **PathFilters.filtered()** method can provide a filtered view of a resource iterator in this case. The **PathFilters** class includes many static methods to create and compose filters that perform various functions, including finding child paths or exact matches, or matching an Ant-style "glob" pattern.

Additional Code Examples For Filtering Resources

The following examples demonstrate how to filter resources based on different criteria.

Example 3.14. Find all files named "messages.properties" in your deployment.

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Iterator<Resource> mclResources =
PathFilters.filtered(PathFilters.match("**/messages.properties"),
moduleClassLoader.iterateResources("", true));
```

Example 3.15. Find all files named "messages.properties" in your deployment and imports.

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Module module = moduleClassLoader.getModule();
Iterator<Resource> moduleResources =
module.iterateResources(PathFilters.match("**/message.properties));
```

Example 3.16. Find all files inside any directory named "my-resources" in your deployment.

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Iterator<Resource> mclResources =
PathFilters.filtered(PathFilters.match("**/my-resources/**"),
moduleClassLoader.iterateResources("", true));
```

Example 3.17. Find all files named "messages" or "errors" in your deployment and imports.

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Module module = moduleClassLoader.getModule();
Iterator<Resource> moduleResources =
module.iterateResources(PathFilters.any(PathFilters.match("**/messages
"), PathFilters.match("**/errors"));
```

Example 3.18. Find all files in a specific package in your deployment.

-

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Iterator<Resource> mclResources =
moduleClassLoader.iterateResources("path/form/of/packageName",
false);
```

[Report a bug](#)

3.7. CLASS LOADING AND SUBDEPLOYMENTS

3.7.1. Modules and Class Loading in Enterprise Archives

Enterprise Archives (EAR) are not loaded as a single module like JAR or WAR deployments. They are loaded as multiple unique modules.

The following rules determine what modules exist in an EAR.

- The contents of the **lib/** directory in the root of the EAR archive is a module. This is called the parent module.
- Each WAR and EJB JAR subdeployment is a module. These modules have the same behavior as any other module as well as implicit dependencies on the parent module.
- Subdeployments have implicit dependencies on the parent module and any other non-WAR subdeployments.

The implicit dependencies on non-WAR subdeployments occur because JBoss EAP 6 has subdeployment class loader isolation disabled by default. Dependencies on the parent module persist, regardless of subdeployment class loader isolation.



IMPORTANT

No subdeployment ever gains an implicit dependency on a WAR subdeployment. Any subdeployment can be configured with explicit dependencies on another subdeployment as would be done for any other module.

Subdeployment class loader isolation can be enabled if strict compatibility is required. This can be enabled for a single EAR deployment or for all EAR deployments. The Java EE 6 specification recommends that portable applications should not rely on subdeployments being able to access each other unless dependencies are explicitly declared as **Class-Path** entries in the **MANIFEST.MF** file of each subdeployment.

[Report a bug](#)

3.7.2. Subdeployment Class Loader Isolation

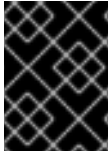
Each subdeployment in an Enterprise Archive (EAR) is a dynamic module with its own class loader. By default a subdeployment can access the resources of other subdeployments.

If a subdeployment is not to be allowed to access the resources of other subdeployments, strict subdeployment isolation can be enabled.

[Report a bug](#)

3.7.3. Enable Subdeployment Class Loader Isolation Within a EAR

This task shows you how to enable subdeployment class loader isolation in an EAR deployment by using a special deployment descriptor in the EAR. This does not require any changes to be made to the application server and does not affect any other deployments.



IMPORTANT

Even when subdeployment class loader isolation is disabled it is not possible to add a WAR deployment as a dependency.

1. Add the deployment descriptor file

Add the `jboss-deployment-structure.xml` deployment descriptor file to the **META-INF** directory of the EAR if it doesn't already exist and add the following content:

```
<jboss-deployment-structure>

</jboss-deployment-structure>
```

2. Add the `<ear-subdeployments-isolated>` element

Add the `<ear-subdeployments-isolated>` element to the `jboss-deployment-structure.xml` file if it doesn't already exist with the content of `true`.

```
<ear-subdeployments-isolated>true</ear-subdeployments-isolated>
```

Result:

Subdeployment class loader isolation will now be enabled for this EAR deployment. This means that the subdeployments of the EAR will not have automatic dependencies on each of the non-WAR subdeployments.

[Report a bug](#)

3.8. DEPLOY TAG LIBRARY DESCRIPTORS (TLDs) IN A CUSTOM MODULE

Summary

If you have multiple applications that use common Tag Library Descriptors (TLDs), it may be useful to separate the TLDs from the applications so that they are located in one central and unique location. This enables easier additions and updates to TLDs without necessarily having to update each individual application that uses them.

This can be done by creating a custom JBoss EAP 6 module that contains the TLD JARs, and declaring a dependency on that module in the applications.

Prerequisites

- At least one JAR containing TLDs. Ensure that the TLDs are packed in **META-INF**.

Procedure 3.7. Deploy TLDs in a Custom Module

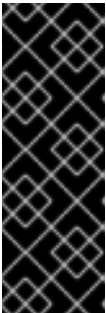
1. Using the Management CLI, connect to your JBoss EAP 6 instance and execute the following command to create the custom module containing the TLD JAR:

```
module add --name=MyTagLibs --resources=/path/to/TLDarchive.jar
```

If the TLDs are packaged with classes that require dependencies, use the `--dependencies=DEPENDENCY` option to ensure that you specify those dependencies when creating the custom module.

When creating the module, you can specify multiple JAR resources by separating each one with `:`. For example, `--resources=/path/to/one.jar:/path/to/two.jar`

2. In your applications, declare a dependency on the new *MyTagLibs* custom module using one of the methods described in [Section 3.2, “Add an Explicit Module Dependency to a Deployment”](#).



IMPORTANT

Ensure that you also import **META-INF** when declaring the dependency. For example, for **MANIFEST.MF**:

```
Dependencies: com.MyTagLibs meta-inf
```

Or, for **jboss-deployment-structure.xml**, use the **meta-inf** attribute.

Result

In your applications you can use TLDs that are contained in the new custom module.

[Report a bug](#)

3.9. REFERENCE

3.9.1. Implicit Module Dependencies

The following table lists the modules that are automatically added to deployments as dependencies and the conditions that trigger the dependency.

Table 3.1. Implicit Module Dependencies

Subsystem Responsible for Adding the Dependency	Dependencies That Are Always Added	Dependencies That Are Conditionally Added	Conditions That Trigger the Addition of the Dependency

Subsystem Responsible for Adding the Dependency	Dependencies That Are Always Added	Dependencies That Are Conditionally Added	Conditions That Trigger the Addition of the Dependency
Core Server	<ul style="list-style-type: none"> • javax.api • ibm.jdk • sun.jdk • org.jboss.vfs 		
EE subsystem	<ul style="list-style-type: none"> • javaee.api • org.hibernate.validator • org.jboss.invocation • org.jboss.as.ee 		
EJB 3 subsystem	<ul style="list-style-type: none"> • javax.ejb.api • org.jboss.ejb-client • org.jboss.iiop-client • org.jboss.as.ejb3 	<ul style="list-style-type: none"> • org.jboss.as.jacorb 	<p>The presence of an ejb-jar.xml file within a valid location in the deployment, as described in the Java EE 6 specification.</p> <p>The presence of annotation-based EJBs, for example: @Stateless, @Stateful, @MessageDriven</p>

Subsystem Responsible for Adding the Dependency	Dependencies That Are Always Added	Dependencies That Are Conditionally Added	Conditions That Trigger the Addition of the Dependency
JAX-RS (RESTEasy) subsystem	<ul style="list-style-type: none"> javax.xml.bind.api javax.ws.rs.api 	<ul style="list-style-type: none"> org.jboss.resteasy.resteasy-atom-provider org.jboss.resteasy.resteasy-hibernatevalidator-provider org.jboss.resteasy.resteasy-jaxrs org.jboss.resteasy.resteasy-jaxb-provider org.jboss.resteasy.resteasy-jackson-provider org.jboss.resteasy.resteasy-jettison-provider org.jboss.resteasy.resteasy-jsapi org.jboss.resteasy.resteasy-multipart-provider org.jboss.resteasy.resteasy-yaml-provider org.codehaus.jackson-jackson-core-asl 	The presence of JAX-RS annotations in the deployment.
JCA subsystem	<ul style="list-style-type: none"> javax.resource.api 	<ul style="list-style-type: none"> javax.jms.api javax.validation.api org.jboss.ironjacamar.api org.jboss.ironjacamar.impl org.hibernate.validator 	The deployment of a resource adapter (RAR) archive.

Subsystem Responsible for Adding the Dependency	Dependencies That Are Always Added	Dependencies That Are Conditionally Added	Conditions That Trigger the Addition of the Dependency
JPA (Hibernate) subsystem	<ul style="list-style-type: none"> • javax.persistence.api 	<ul style="list-style-type: none"> • javax.persistence • org.jboss.as.jpa • org.jboss.as.jpa.spi • org.javassist • org.jboss.as.jpa.hibernate3 / org.jboss.as.jpa.hibernate3.HibernatePersistenceProviderAdaptor • org.hibernate.envers • org.jboss.as.naming • org.jboss.jandex 	<p>The presence of an <code>@PersistenceUnit</code> or <code>@PersistenceContext</code> annotation, or a <code><persistence-unit-ref></code> or <code><persistence-context-ref></code> element in a deployment descriptor.</p> <p>JBoss EAP 6 maps persistence provider names to module names. If you name a specific provider in the <code>persistence.xml</code> file, a dependency is added for the appropriate module. If this not the desired behavior, you can exclude it using a <code>jboss-deployment-structure.xml</code> file.</p>
Logging subsystem	<ul style="list-style-type: none"> • org.jboss.logging • org.apache.log4j • org.apache.commons.logging • org.slf4j • org.jboss.logging.jul-to-slf4j-stub 		<p>These dependencies are always added unless the <code>add-logging-api-dependencies</code> attribute is set to false.</p>
SAR subsystem		<ul style="list-style-type: none"> • org.jboss.modules • org.jboss.as.system-jmx • org.jboss.common-beans 	<p>The deployment of a SAR archive.</p>

Subsystem Responsible for Adding the Dependency	Dependencies That Are Always Added	Dependencies That Are Conditionally Added	Conditions That Trigger the Addition of the Dependency
Security subsystem	<ul style="list-style-type: none"> org.picketbox org.jboss.as.security javax.security.jacc.api javax.security.auth.message.api 		
Web subsystem		<ul style="list-style-type: none"> javax.servlet.api javax.servlet.jsp.api javax.websocket.api javax.servlet.jstl.api org.jboss.as.web 	The deployment of a WAR archive. JavaServer Faces (JSF) is added only if it is used.
Web Services subsystem	<ul style="list-style-type: none"> javax.jws.api javax.xml.soap.api javax.xml.ws.api 	<ul style="list-style-type: none"> org.jboss.ws.api org.jboss.ws.spi 	If it is not application client type, then it will add the conditional dependencies
Weld (CDI) Subsystem	<ul style="list-style-type: none"> javax.enterprise.api javax.inject.api 	<ul style="list-style-type: none"> javax.persistence.api javaee.api org.javassist org.jboss.as.weld org.jboss.weld.core org.jboss.weld.api org.jboss.weld.spi 	The presence of a beans.xml file in the deployment.

Subsystem Responsible for Adding the Dependency	Dependencies That Are Always Added	Dependencies That Are Conditionally Added	Conditions That Trigger the Addition of the Dependency
Container Managed Persistence (CMP) Subsystem		<ul style="list-style-type: none"> org.jboss.as.cmp 	

[Report a bug](#)

3.9.2. Included Modules

A table listing the JBoss EAP 6 included modules and whether they are supported can be found on the Customer Portal at <https://access.redhat.com/articles/1122333>.

[Report a bug](#)

3.9.3. JBoss Deployment Structure Deployment Descriptor Reference

The key tasks that can be performed using this deployment descriptor are:

- Defining explicit module dependencies.
- Preventing specific implicit dependencies from loading.
- Defining additional modules from the resources of that deployment.
- Changing the subdeployment isolation behavior in that EAR deployment.
- Adding additional resource roots to a module in an EAR.

[Report a bug](#)

CHAPTER 4. VALVES

4.1. ABOUT VALVES

A Valve is a Java class that gets inserted into the request processing pipeline for an application. It is inserted in the pipeline before servlet filters. Valves can make changes to the request before passing it on or perform other processing such as authentication or even canceling the request.

Valves can be configured at the server level or at the application level. The only difference is in how they are configured and packaged.

- Global Valves are configured at the server level and apply to all applications deployed to the server. Instructions to configure Global Valves are located in the *Administration and Configuration Guide* for JBoss EAP.
- Valves configured at the application level are packaged with the application deployment and only affect the specific application. Instructions to configure Valves at the application level are located in the *Development Guide* for JBoss EAP.

Version 6.1.0 and later supports global valves.

[Report a bug](#)

4.2. ABOUT GLOBAL VALVES

A Global Valve is a valve that is inserted into the request processing pipeline of all deployed applications. A valve is made global by being packaged and installed as a static module in JBoss EAP 6. Global valves are configured in the web subsystem.

Only version 6.1.0 and later supports global valves.

For instructions on how to configure Global Valves, see the chapter entitled *Global Valves* in the *Administration and Configuration Guide for JBoss EAP*.

[Report a bug](#)

4.3. ABOUT AUTHENTICATOR VALVES

An authenticator valve is a valve that authenticates the credentials of a request. Such valve is a subclass of `org.apache.catalina.authenticator.AuthenticatorBase` and overrides the `authenticate(Request request, Response response, LoginConfig config)` method.

This can be used to implement additional authentication schemes.

[Report a bug](#)

4.4. CONFIGURE A WEB APPLICATION TO USE A VALVE

Valves that are not installed as global valves must be included with your application and configured in the `jboss-web.xml` deployment descriptor.



IMPORTANT

Valves that are installed as global valves are automatically applied to all deployed applications. For instructions on how to configure Global Valves, see *Global Valves* in the *JBoss EAP Administration and Configuration Guide*.

Prerequisites

- The valve must be created and included in your application's classpath. This can be done by either including it in the application's WAR file or any module that is added as a dependency. Examples of such modules include a static module installed on the server or a JAR file in the `lib/` directory of an EAR archive if the WAR is deployed in an EAR.
- The application must include a `jboss-web.xml` deployment descriptor.

Procedure 4.1. Configure an application for a local valve

1. Configure a Valve

Create a **valve** element containing the **class-name** child element in the application's `jboss-web.xml` file. The **class-name** is the name of the valve class.

```
<valve>
  <class-name>VALVE_CLASS_NAME</class-name>
</valve>
```

Example 4.1. Valve element configured in the `jboss-web.xml` file

```
<valve>
  <class-
name>org.jboss.security.negotiation.NegotiationAuthenticator</clas
s-name>
</valve>
```

2. Configure a Custom Valve

If the valve has configurable parameters, add a **param** child element to the **valve** element for each parameter, specifying the **param-name** and **param-value** for each.

Example 4.2. Custom valve element configured in the `jboss-web.xml` file

```
<valve>
  <class-
name>org.jboss.web.tomcat.security.GenericHeaderAuthenticator</cla
ss-name>
  <param>
    <param-name>httpHeaderForSSOAuth</param-name>
    <param-value>sm_ssoid,ct-remote-user,HTTP_OBLIX_UID</param-
value>
  </param>
  <param>
    <param-name>sessionCookieForSSOAuth</param-name>
```

```

        <param-value>SMSESSION,CTSESSION,ObSSOCookie</param-value>
      </param>
    </valve>

```

When the application is deployed, the valve will be enabled for the application with the specified configuration.

Example 4.3. jboss-web.xml valve configuration

```

<valve>
  <class-name>org.jboss.samplevalves.RestrictedUserAgentsValve</class-
name>
  <param>
    <param-name>restrictedUserAgents</param-name>
    <param-value>^.*MS Web Services Client Protocol.*$</param-value>
  </param>
</valve>

```

[Report a bug](#)

4.5. CONFIGURE A WEB APPLICATION TO USE AN AUTHENTICATOR VALVE

Configuring an application to use an authenticator valve requires the valve to be installed and configured (either local to the application or as a global valve) and the **web.xml** deployment descriptor of the application to be configured. In the simplest case, the **web.xml** configuration is the same as using **BASIC** authentication except the **auth-method** child element of **login-config** is set to the name of the valve performing the configuration.

Prerequisites

- Authentication valve must already be created.
- If the authentication valve is a global valve then it must already be installed and configured, and you must know the name that it was configured as.
- You need to know the realm name of the security realm that the application will use.

If you do not know the valve or security realm name to use, ask your server administrator for this information.

Procedure 4.2. Configure an Application to use an Authenticator Valve

1. Configure the valve

When using a local valve, it must be configured in the application's **jboss-web.xml** deployment descriptor. See [Section 4.4, "Configure a Web Application to use a Valve"](#).

When using a global valve, this is not necessary.

2. Add security configuration to web.xml

Add the security configuration to the **web.xml** file for your application, using the standard

elements such as **security-constraint**, **login-config**, and **security-role**. In the **login-config** element, set the value of **auth-method** to the name of the authenticator valve. The **realm-name** element must also be set to the name of the JBoss security realm being used by the application.

```
<login-config>
  <auth-method>VALVE_NAME</auth-method>
  <realm-name>REALM_NAME</realm-name>
</login-config>
```

When the application is deployed, the authentication of requests is handled by the configured authentication valve.

[Report a bug](#)

4.6. CREATE A CUSTOM VALVE

A Valve is a Java class that gets inserted into the request processing pipeline for an application before the application's servlet filters. This can be used to modify the request or perform any other behavior. This task demonstrates the basic steps required for implementing a valve.

Procedure 4.3. Create a Custom Valve

1. Configure the Maven dependencies.

Add the following dependency configuration to the project **pom.xml** file.

```
<dependency>
  <groupId>org.jboss.web</groupId>
  <artifactId>jbossweb</artifactId>
  <version>7.5.7.Final-redhat-1</version>
  <scope>provided</scope>
</dependency>
```



NOTE

The **jbossweb-VERSION.jar** file should not be included in the application. It is available to the JBoss EAP server runtime classpath as a JBoss module at this location:

EAP_HOME/modules/system/layers/base/org/jboss/as/web/main/jbossweb-7.5.7.Final-redhat-1.jar.

2. Create the Valve class

Create a subclass of **org.apache.catalina.valves.ValveBase**.

```
package org.jboss.samplevalves;

import org.apache.catalina.valves.ValveBase;
import org.apache.catalina.connector.Request;
import org.apache.catalina.connector.Response;

public class RestrictedUserAgentsValve extends ValveBase {

}
```

3. Implement the invoke method.

The **invoke()** method is called when this valve is executed in the pipeline. The request and response objects are passed as parameters. Perform any processing and modification of the request and response here.

```
public void invoke(Request request, Response response)
{
}
```

4. Invoke the next pipeline step.

The last thing the invoke method must do is invoke the next step of the pipeline and pass the modified request and response objects along. This is done using the **getNext().invoke()** method

```
getNext().invoke(request, response);
```

5. Optional: Specify parameters.

If the valve must be configurable, enable this by adding a parameter. Do this by adding an instance variable and a setter method for each parameter.

```
private String restrictedUserAgents = null;

public void setRestricteduserAgents(String mystring)
{
    this.restrictedUserAgents = mystring;
}
```

6. Review the completed code example.

The class should now look like the following example.

Example 4.4. Sample Custom Valve

```
package org.jboss.samplevalves;

import java.io.IOException;
import java.util.regex.Pattern;

import javax.servlet.ServletException;
import org.apache.catalina.valves.ValveBase;
import org.apache.catalina.connector.Request;
import org.apache.catalina.connector.Response;

public class RestrictedUserAgentsValve extends ValveBase
{
    private String restrictedUserAgents = null;

    public void setRestrictedUserAgents(String mystring)
    {
        this.restrictedUserAgents = mystring;
    }
}
```

```
public void invoke(Request request, Response response) throws
IOException, ServletException
{
    String agent = request.getHeader("User-Agent");
    System.out.println("user-agent: " + agent + " : " +
restrictedUserAgents);
    if (Pattern.matches(restrictedUserAgents, agent))
    {
        System.out.println("user-agent: " + agent + " matches: "
+ restrictedUserAgents);
        response.addHeader("Connection", "close");
    }
    getNext().invoke(request, response);
}
```

[Report a bug](#)

CHAPTER 5. LOGGING FOR DEVELOPERS

5.1. INTRODUCTION

5.1.1. About Logging

Logging is the practice of recording a series of messages from an application that provide a record (or log) of the application's activities.

Log messages provide important information for developers when debugging an application and for system administrators maintaining applications in production.

Most modern logging frameworks in Java also include other details such as the exact time and the origin of the message.

[Report a bug](#)

5.1.2. Application Logging Frameworks Supported By JBoss LogManager

JBoss LogManager supports the following logging frameworks:

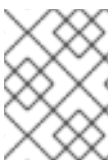
- JBoss Logging - included with JBoss EAP 6
- Apache Commons Logging - <http://commons.apache.org/logging/>
- Simple Logging Facade for Java (SLF4J) - <http://www.slf4j.org/>
- Apache log4j - <http://logging.apache.org/log4j/1.2/>
- Java SE Logging (java.util.logging) - <http://download.oracle.com/javase/6/docs/api/java/util/logging/package-summary.html>

JBoss LogManager supports the following APIs:

- java.util.logging
- JBoss Logging
- Log4j
- SLF4J
- commons-logging

JBoss LogManager also supports the following SPIs:

- java.util.logging Handler
- Log4j Appender



NOTE

If you are using the **Log4j API** and a **Log4J Appender**, then Objects will be converted to **string** before being passed.

[Report a bug](#)

5.1.3. About Log Levels

Log levels are an ordered set of enumerated values that indicate the nature and severity of a log message. The level of a given log message is specified by the developer using the appropriate methods of their chosen logging framework to send the message.

JBoss EAP 6 supports all the log levels used by the supported application logging frameworks. The most commonly used six log levels are (in order of lowest to highest): **TRACE**, **DEBUG**, **INFO**, **WARN**, **ERROR** and **FATAL**.

Log levels are used by log categories and handlers to limit the messages they are responsible for. Each log level has an assigned numeric value which indicates its order relative to other log levels. Log categories and handlers are assigned a log level and they only process log messages of that level or higher. For example a log handler with the level of **WARN** will only record messages of the levels **WARN**, **ERROR** and **FATAL**.

[Report a bug](#)

5.1.4. Supported Log Levels

Table 5.1. Supported Log Levels

Log Level	Value	Description
FINEST	300	-
FINER	400	-
TRACE	400	Use for messages that provide detailed information about the running state of an application. Log messages of TRACE are usually only captured when debugging an application.
DEBUG	500	Use for messages that indicate the progress individual requests or activities of an application. Log messages of DEBUG are usually only captured when debugging an application.
FINE	500	-
CONFIG	700	-
INFO	800	Use for messages that indicate the overall progress of the application. Often used for application startup, shutdown and other major lifecycle events.
WARN	900	Use to indicate a situation that is not in error but is not considered ideal. May indicate circumstances that may lead to errors in the future.
WARNING	900	-
ERROR	1000	Use to indicate an error that has occurred that could prevent the current activity or request from completing but will not prevent the application from running.

Log Level	Value	Description
SEVERE	1000	-
FATAL	1100	Use to indicate events that could cause critical service failure and application shutdown and possibly cause JBoss EAP 6 to shutdown.

[Report a bug](#)

5.1.5. Default Log File Locations

These are the log files that get created for the default logging configurations. The default configuration writes the server log files using periodic log handlers

Table 5.2. Default Log File for a standalone server

Log File	Description
<i>EAP_HOME/standalone/log/server.log</i>	Server Log. Contains all server log messages, including server startup messages.
<i>EAP_HOME/standalone/log/gc.log</i>	Garbage collection log. Contains details of all garbage collection.

Table 5.3. Default Log Files for a managed domain

Log File	Description
<i>EAP_HOME/domain/log/host-controller.log</i>	Host Controller boot log. Contains log messages related to the startup of the host controller.
<i>EAP_HOME/domain/log/process-controller.log</i>	Process controller boot log. Contains log messages related to the startup of the process controller.
<i>EAP_HOME/domain/servers/SERVERNAME/log/server.log</i>	The server log for the named server. Contains all log messages for that server, including server startup messages.

[Report a bug](#)

5.2. LOGGING WITH THE JBOSS LOGGING FRAMEWORK

5.2.1. About JBoss Logging

JBoss Logging is the application logging framework that is included in JBoss EAP 6.

JBoss Logging provide an easy way to add logging to an application. You add code to your application that uses the framework to send log messages in a defined format. When the application is deployed to an application server, these messages can be captured by the server and displayed and/or written to file

according to the server's configuration.

[Report a bug](#)

5.2.2. Features of JBoss Logging

- Provides an innovative, easy to use "typed" logger.
- Full support for internationalization and localization. Translators work with message bundles in properties files while developers can work with interfaces and annotations.
- Build-time tooling to generate typed loggers for production, and runtime generation of typed loggers for development.

[Report a bug](#)

5.2.3. Add Logging to an Application with JBoss Logging

To log messages from your application you create a Logger object (`org.jboss.logging.Logger`) and call the appropriate methods of that object. This task describes the steps required to add support for this to your application.

Prerequisites

- If you are using Maven as your build system, the project must be configured to include the JBoss Maven Repository. Refer to [Section 2.3.2, "Configure the JBoss EAP 6 Maven Repository Using the Maven Settings"](#)
- The JBoss Logging JAR files must be in the build path for your application. How you do this depends on whether you build your application using Red Hat JBoss Developer Studio or with Maven.
 - When building using Red Hat JBoss Developer Studio select **Properties** from the **Project** menu, then select **Targeted Runtimes** and ensure the runtime for JBoss EAP 6 is checked.
 - When building using Maven add the following dependency configuration to your project's `pom.xml` file.

```
<dependency>
  <groupId>org.jboss.logging</groupId>
  <artifactId>jboss-logging</artifactId>
  <version>3.1.2.GA-redhat-1</version>
  <scope>provided</scope>
</dependency>
```

You do not need to include the JARs in your built application because JBoss EAP 6 provides them to deployed applications.

Procedure 5.1. Add Logging to an Application

Complete the following procedure for each class to which you want to add logging:

1. Add imports

Add the **import** statements for the JBoss Logging class namespaces that you will be using. At a minimum you will need to import **import org.jboss.logging.Logger**.

```
import org.jboss.logging.Logger;
```

2. Create a Logger object

Create an instance of **org.jboss.logging.Logger** and initialize it by calling the static method **Logger.getLogger(Class)**. Red Hat recommends creating this as a single instance variable for each class.

```
private static final Logger LOGGER =
    Logger.getLogger>HelloWorld.class);
```

3. Add logging messages

Add calls to the methods of the **Logger** object to your code where you want it to send log messages. The **Logger** object has many different methods with different parameters for different types of messages. The easiest to use are:

```
debug(Object message)
```

```
info(Object message)
```

```
error(Object message)
```

```
trace(Object message)
```

```
fatal(Object message)
```

These methods send a log message with the corresponding log level and the **message** parameter as a string.

```
LOGGER.error("Configuration file not found.");
```

For the complete list of JBoss Logging methods refer to the **org.jboss.logging** package in the JBoss EAP 6 API Documentation.

Example 5.1. Using JBoss Logging when opening a properties file

This example shows an extract of code from a class that loads customized configuration for an application from a properties file. If the specified file is not found, an ERROR level log message is recorded.

```
import org.jboss.logging.Logger;
public class LocalSystemConfig
{
    private static final Logger LOGGER =
        Logger.getLogger(LocalSystemConfig.class);

    public Properties openCustomProperties(String configname) throws
        CustomConfigFileNotFoundException
    {
        Properties props = new Properties();
        try
        {
```

```

        LOGGER.info("Loading custom configuration from "+configname);
        props.load(new FileInputStream(configname));
    }
    catch(IOException e) //catch exception in case properties file
does not exist
    {
        LOGGER.error("Custom configuration file (" +configname+" ) not
found. Using defaults.");
        throw new CustomConfigFileNotFoundException(configname);
    }

    return props;
}

```

[Report a bug](#)

5.3. PER-DEPLOYMENT LOGGING

5.3.1. About Per-deployment Logging

Per-deployment logging allows a developer to configure in advance the logging configuration for their application. When the application is deployed, logging begins according to the defined configuration. The log files created through this configuration contain information only about the behavior of the application.

This approach has advantages and disadvantages over using system-wide logging. An advantage is that the administrator of the JBoss EAP instance does not need to configure logging. A disadvantage is that the per-deployment logging configuration is read only on startup and so cannot be changed at runtime.

[Report a bug](#)

5.3.2. Add Per-deployment Logging to an Application

To configure per-deployment logging, add the logging configuration file **logging.properties** into the deployment. This configuration file is recommended because it can be used with any logging facade as the JBoss Log Manager is the underlying log manager used.

If you are using **Simple Logging Facade for Java (SLF4J)** or **Apache log4j**, the **logging.properties** configuration file is suitable. If you are using Apache log4j appenders then the configuration file **log4j.properties** is required. The configuration file **jboss-logging.properties** is supported only for legacy deployments.

Procedure 5.2. Add Configuration File to the Application

- **The directory into which the configuration file is added depends on the deployment method: EAR, WAR or JAR.**
 - **EAR deployment**
Copy the logging configuration file to the **META-INF** directory.
 - **WAR or JAR deployment**
Copy the logging configuration file to either the **META-INF** or **WEB-INF/classes** directory.

[Report a bug](#)

5.3.3. Example logging.properties File

```
# Additional loggers to configure (the root logger is always configured)
loggers=
# Root logger configuration
logger.level=INFO
logger.handlers=FILE

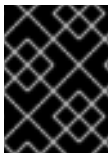
# A handler configuration
handler.FILE=org.jboss.logmanager.handlers.FileHandler
handler.FILE.level=ALL
handler.FILE.formatter=PATTERN
handler.FILE.properties=append,autoFlush,enabled,suffix,fileName
handler.FILE.constructorProperties=fileName,append
handler.FILE.append=true
handler.FILE.autoFlush=true
handler.FILE.enabled=true
handler.FILE.fileName=${jboss.server.log.dir}/app.log

# The formatter to use
formatter.PATTERN=org.jboss.logmanager.formatters.PatternFormatter
formatter.PATTERN.properties=pattern
formatter.PATTERN.constructorProperties=pattern
formatter.PATTERN.pattern=%d %-5p %c: %m%n
```

[Report a bug](#)

5.4. LOGGING PROFILES

5.4.1. About Logging Profiles



IMPORTANT

Logging profiles are only available in version 6.1.0 and later. They cannot be configured using the management console.

Logging profiles are independent sets of logging configuration that can be assigned to deployed applications. As with the regular logging subsystem, a logging profile can define handlers, categories and a root logger but cannot refer to configuration in other profiles or the main logging subsystem. The design of logging profiles mimics the logging subsystem for ease of configuration.

The use of logging profiles allows administrators to create logging configuration that are specific to one or more applications without affecting any other logging configuration. Because each profile is defined in the server configuration, the logging configuration can be changed without requiring that the affected applications be redeployed.

Each logging profile can have the following configuration:

- A unique name. This is required.
- Any number of log handlers.
- Any number of log categories.

- Up to one root logger.

An application can specify a logging profile to use in its **MANIFEST.MF** file, using the ***logging-profile*** attribute.

[Report a bug](#)

5.4.2. Specify a Logging Profile in an Application

An application specifies the logging profile to use in its **MANIFEST.MF** file.

Prerequisites:

1. You must know the name of the logging profile that has been setup on the server for this application to use. Ask your server administrator for the name of the profile to use.

Procedure 5.3. Add Logging Profile configuration to an Application

- **Edit MANIFEST.MF**

If your application does not have a **MANIFEST.MF** file: create one with the following content, replacing *NAME* with the required profile name.

```
Manifest-Version: 1.0
Logging-Profile: NAME
```

If your application already has a **MANIFEST.MF** file: add the following line to it, replacing *NAME* with the required profile name.

```
Logging-Profile: NAME
```

NOTE

If you are using Maven and the **maven-war-plugin**, you can put your **MANIFEST.MF** file in **src/main/resources/META-INF/** and add the following configuration to your **pom.xml** file.

```
<plugin>
  <artifactId>maven-war-plugin</artifactId>
  <configuration>
    <archive>
      <manifestFile>src/main/resources/META-
INF/MANIFEST.MF</manifestFile>
    </archive>
  </configuration>
</plugin>
```

When the application is deployed it will use the configuration in the specified logging profile for its log messages.

[Report a bug](#)

CHAPTER 6. INTERNATIONALIZATION AND LOCALIZATION

6.1. INTRODUCTION

6.1.1. About Internationalization

Internationalization is the process of designing software so that it can be adapted to different languages and regions without engineering changes.

[Report a bug](#)

6.1.2. About Localization

Localization is the process of adapting internationalized software for a specific region or language by adding locale-specific components and translations of text.

[Report a bug](#)

6.2. JBOSS LOGGING TOOLS

6.2.1. Overview

6.2.1.1. JBoss Logging Tools Internationalization and Localization

JBoss Logging Tools is a Java API that provides support for the internationalization and localization of log messages, exception messages, and generic strings. In addition to providing a mechanism for translation, JBoss Logging tools also provides support for unique identifiers for each log message.

Internationalized messages and exceptions are created as method definitions inside of interfaces annotated using **`org.jboss.logging`** annotations. It is not necessary to implement the interfaces, JBoss Logging Tools does this at compile time. Once defined you can use these methods to log messages or obtain exception objects in your code.

Internationalized logging and exception interfaces created with JBoss Logging Tools can be localized by creating a properties file for each bundle containing the translations for a specific language and region. JBoss Logging Tools can generate template property files for each bundle that can then be edited by a translator.

JBoss Logging Tools creates an implementation of each bundle for each corresponding translations property file in your project. All you have to do is use the methods defined in the bundles and JBoss Logging Tools ensures that the correct implementation is invoked for your current regional settings.

Message ids and project codes are unique identifiers that are prepended to each log message. These unique identifiers can be used in documentation to make it easy to find information about log messages. With adequate documentation, the meaning of a log message can be determined from the identifiers regardless of the language that the message was written in.

[Report a bug](#)

6.2.1.2. JBoss Logging Tools Quickstart

The JBoss Logging Tools quickstart, **logging-tools**, contains a simple Maven project that demonstrates the features of JBoss Logging Tools. It has been used extensively in this documentation for code samples.

Refer to this quickstart for a complete working demonstration of all the features described in this documentation.

[Report a bug](#)

6.2.1.3. Message Logger

A Message Logger is an interface that is used to define internationalized log messages. A Message Logger interface is annotated with **@org.jboss.logging.MessageLogger**.

[Report a bug](#)

6.2.1.4. Message Bundle

A message bundle is an interface that can be used to define generic translatable messages and Exception objects with internationalized messages . A message bundle is not used for creating log messages.

A message bundle interface is annotated with **@org.jboss.logging.MessageBundle**.

[Report a bug](#)

6.2.1.5. Internationalized Log Messages

Internationalized Log Messages are log messages created by defining a method in a Message Logger. The method must be annotated with the **@LogMessage** and **@Message** annotations and specify the log message using the value attribute of **@Message**. Internationalized log messages are localized by providing translations in a properties file.

JBoss Logging Tools generates the required logging classes for each translation at compile time and invokes the correct methods for the current locale at runtime.

[Report a bug](#)

6.2.1.6. Internationalized Exceptions

An internationalized exception is an exception object returned from a method defined in a message bundle. Message bundle methods that return Java Exception objects can be annotated to define a default exception message. The default message is replaced with a translation if one is found in a matching properties file for the current locale. Internationalized exceptions can also have project codes and message ids assigned to them.

[Report a bug](#)

6.2.1.7. Internationalized Messages

An internationalized message is a string returned from a method defined in a message bundle. Message bundle methods that return Java String objects can be annotated to define the default content of that String, known as the message. The default message is replaced with a translation if one is found in a matching properties file for the current locale.

[Report a bug](#)

6.2.1.8. Translation Properties Files

Translation properties files are Java properties files that contain the translations of messages from one interface for one locale, country, and variant. Translation properties files are used by the JBoss Logging Tools to generate the classes that return the messages.

[Report a bug](#)

6.2.1.9. JBoss Logging Tools Project Codes

Project codes are strings of characters that identify groups of messages. They are displayed at the beginning of each log message, prepended to the message id. Project codes are defined with the **projectCode** attribute of the **@MessageLogger** annotation.

[Report a bug](#)

6.2.1.10. JBoss Logging Tools Message IDs

Message IDs are numbers, that when combined with a project code, uniquely identify a log message. Message IDs are displayed at the beginning of each log message, appended to the project code for the message. Message IDs are defined with the **id** attribute of the **@Message** annotation.

[Report a bug](#)

6.2.2. Creating Internationalized Loggers, Messages and Exceptions

6.2.2.1. Create Internationalized Log Messages

This task shows you how to use JBoss Logging Tools to create internationalized log messages by creating **MessageLogger** interfaces. It does not cover all optional features or the localization of those log messages.

Refer to the **logging-tools** quick start for a complete example.

Prerequisites:

1. You must already have a working Maven project. Refer to [Section 6.2.6.1, “JBoss Logging Tools Maven Configuration”](#).
2. The project must have the required Maven configuration for JBoss Logging Tools.

Procedure 6.1. Create an Internationalized Log Message Bundle

1. Create an Message Logger interface

Add a Java interface to your project to contain the log message definitions. Name the interface descriptively for the log messages that will be defined in it.

The log message interface has the following requirements:

- It must be annotated with **@org.jboss.logging.MessageLogger**.
- It must extend **org.jboss.logging.BasicLogger**.

- The interface must define a field of that is a typed logger that implements this interface. Do this with the `getMessageLogger()` method of `org.jboss.logging.Logger`.

```
package com.company.accounts.loggers;

import org.jboss.logging.BasicLogger;
import org.jboss.logging.Logger;
import org.jboss.logging.MessageLogger;

@MessageLogger(projectCode="")
interface AccountsLogger extends BasicLogger
{
    AccountsLogger LOGGER = Logger.getMessageLogger(
        AccountsLogger.class,
        AccountsLogger.class.getPackage().getName() );
}
```

2. Add method definitions

Add a method definition to the interface for each log message. Name each method descriptively for the log message that it represents.

Each method has the following requirements:

- The method must return **void**.
- It must be annotated with the `@org.jboss.logging.LogMessage` annotation.
- It must be annotated with the `@org.jboss.logging.Message` annotation.
- The value attribute of `@org.jboss.logging.Message` contains the default log message. This is the message that is used if no translation is available.

```
@LogMessage
@Message(value = "Customer query failed, Database not available.")
void customerQueryFailDBClosed();
```

The default log level is **INFO**.

3. Invoke the methods

Add the calls to the interface methods in your code where the messages must be logged from. It is not necessary to create implementations of the interfaces, the annotation processor does this for you when the project is compiled.

```
AccountsLogger.LOGGER.customerQueryFailDBClosed();
```

The custom loggers are sub-classed from `BasicLogger` so the logging methods of **BasicLogger** (`debug()`, `error()` etc) can also be used. It is not necessary to create other loggers to log non-internationalized messages.

```
AccountsLogger.LOGGER.error("Invalid query syntax.");
```

Result

The project now supports one or more internationalized loggers that can be localized.

[Report a bug](#)

6.2.2.2. Create and Use Internationalized Messages

This task shows you how to create internationalized messages and how to use them. This task does not cover all optional features or the process of localizing those messages.

Refer to the **logging-tools** quickstart for a complete example.

Prerequisites

1. You have a working Maven project using the JBoss EAP 6 repository. Refer to [Section 2.3.2, “Configure the JBoss EAP 6 Maven Repository Using the Maven Settings”](#).
2. The required Maven configuration for JBoss Logging Tools has been added. Refer to [Section 6.2.6.1, “JBoss Logging Tools Maven Configuration”](#).

Procedure 6.2. Create and Use Internationalized Messages

1. Create an interface for the exceptions

JBoss Logging Tools defines internationalized messages in interfaces. Name each interface descriptively for the messages that will be defined in it.

The interface has the following requirements:

- It must be declared as public
- It must be annotated with `@org.jboss.logging.MessageBundle`.
- The interface must define a field that is a message bundle of the same type as the interface.

```
@MessageBundle(projectCode="")
public interface GreetingMessageBundle
{
    GreetingMessageBundle MESSAGES =
        Messages.getBundle(GreetingMessageBundle.class);
}
```

2. Add method definitions

Add a method definition to the interface for each message. Name each method descriptively for the message that it represents.

Each method has the following requirements:

- It must return an object of type **String**.
- It must be annotated with the `@org.jboss.logging.Message` annotation.
- The value attribute of `@org.jboss.logging.Message` must be set to the default message. This is the message that is used if no translation is available.

```
@Message(value = "Hello world.")
String helloworldString();
```

3. Invoke methods

Invoke the interface methods in your application where you need to obtain the message.

```
System.console.out.println(helloworldString());
```

RESULT: the project now supports internationalized message strings that can be localized.

[Report a bug](#)

6.2.2.3. Create Internationalized Exceptions

This task shows you how to create internationalized exceptions and how to use them. This task does not cover all optional features or the process of localization of those exceptions.

Refer to the **logging-tools** quick start for a complete example.

For this task it is assumed that you already have a software project, that is being built in either Red Hat JBoss Developer Studio or Maven, to which you want to add internationalized exceptions.

Procedure 6.3. Create and use Internationalized Exceptions

1. **Add JBoss Logging Tools configuration**

Add the required project configuration to support JBoss Logging Tools. Refer to [Section 6.2.6.1, “JBoss Logging Tools Maven Configuration”](#)

2. **Create an interface for the exceptions**

JBoss Logging Tools defines internationalized exceptions in interfaces. Name each interface descriptively for the exceptions that will be defined in it.

The interface has the following requirements:

- It must be declared as **public**.
- It must be annotated with **@org.jboss.logging.MessageBundle**.
- The interface must define a field that is a message bundle of the same type as the interface.

```
@MessageBundle(projectCode="")
public interface ExceptionBundle
{
    ExceptionBundle EXCEPTIONS =
        Messages.getBundle(ExceptionBundle.class);
}
```

3. **Add method definitions**

Add a method definition to the interface for each exception. Name each method descriptively for the exception that it represents.

Each method has the following requirements:

- It must return an object of type **Exception** or a sub-type of **Exception**.
- It must be annotated with the **@org.jboss.logging.Message** annotation.
- The value attribute of **@org.jboss.logging.Message** must be set to the default exception message. This is the message that is used if no translation is available.

- If the exception being returned has a constructor that requires parameters in addition to a message string, then those parameters must be supplied in the method definition using the `@Param` annotation. The parameters must be the same type and order as the constructor.

```
@Message(value = "The config file could not be opened.")
IOException configFileAccessError();

@Message(id = 13230, value = "Date string '%s' was invalid.")
ParseException dateWasInvalid(String dateString, @Param int
errorOffset);
```

4. Invoke methods

Invoke the interface methods in your code where you need to obtain one of the exceptions. The methods do not throw the exceptions, they return the exception object which you can then throw.

```
try
{
    propsInFile=new File(configname);
    props.load(new FileInputStream(propsInFile));
}
catch(IOException ioex) //in case props file does not exist
{
    throw ExceptionBundle.EXCEPTIONS.configFileAccessError();
}
```

RESULT: the project now supports internationalized exceptions that can be localized.

[Report a bug](#)

6.2.3. Localizing Internationalized Loggers, Messages and Exceptions

6.2.3.1. Generate New Translation Properties Files with Maven

Projects that are being built with Maven can generate empty translation property files for each Message Logger and Message Bundle it contains. These files can then be used as new translation property files.

The following procedure shows how to configure a Maven project to generate new translation property files.

Refer to the **logging-tools** quick start for a complete example.

Prerequisites:

1. You must already have a working Maven project.
2. The project must already be configured for JBoss Logging Tools.
3. The project must contain one or more interfaces that define internationalized log messages or exceptions.

Procedure 6.4. Generate New Translation Properties Files with Maven

1. **Add Maven configuration**

Add the **-AgeneratedTranslationFilePath** compiler argument to the Maven compiler plug-in configuration and assign it the path where the new files will be created.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>2.3.2</version>
  <configuration>
    <source>1.6</source>
    <target>1.6</target>
    <compilerArgument>
      -
      AgeneratedTranslationFilesPath=${project.basedir}/target/generated-
      translation-files
    </compilerArgument>
    <showDeprecation>true</showDeprecation>
  </configuration>
</plugin>
```

The above configuration will create the new files in the **target/generated-translation-files** directory of your Maven project.

2. Build the project

Build the project using Maven.

```
[Localhost]$ mvn compile
```

One properties file is created per interface annotated with **@MessageBundle** or **@MessageLogger**. The new files are created in a subdirectory corresponding to the Java package that each interface is declared in.

Each new file is named using the following syntax where **InterfaceName** is the name of the interface that this file was generated for: **InterfaceName.i18n_locale_COUNTRY_VARIANT.properties**.

These files can now be copied into your project as the basis for new translations.

[Report a bug](#)

6.2.3.2. Translate an Internationalized Logger, Exception or Message

Logging and Exception messages defined in interfaces using JBoss Logging Tools can have translations provided in properties files.

The following procedure shows how to create and use a translation properties file. It is assumed that you already have a project with one or more interfaces defined for internationalized exceptions or log messages.

Refer to the **logging-tools** quick start for a complete example.

Prerequisites

1. You must already have a working Maven project.
2. The project must already be configured for JBoss Logging Tools.

3. The project must contain one or interfaces that define internationalized log messages or exceptions.
4. The project must be configured to generate template translation property files.

Procedure 6.5. Translate an internationalized logger, exception or message

1. **Generate the template properties files**

Run the **mvn compile** command to create the template translation properties files.

2. **Add the template file to your project**

Copy the template for the interfaces that you want to translate from the directory where they were created into the **src/main/resources** directory of your project. The properties files must be in the same package as the interfaces they are translating.

3. **Rename the copied template file**

Rename the copy of the template file according to the translation it will contain. E.g. **GreeterLogger.i18n_fr_FR.properties**.

4. **Translate the contents of the template.**

Edit the new translation properties file to contain the appropriate translation.

```
# Level: Logger.Level.INFO
# Message: Hello message sent.
logHelloMessageSent=Bonjour message envoyé.
```

Repeat steps two, three, and four for each translation of each bundle being performed.

RESULT: The project now contains translations for one or more message or logger bundles. Building the project will generate the appropriate classes to log messages with the supplied translations. It is not necessary to explicitly invoke methods or supply parameters for specific languages, JBoss Logging Tools automatically uses the correct class for the current locale of the application server.

The source code of the generated classes can be viewed under **target/generated-sources/annotations/**.

[Report a bug](#)

6.2.4. Customizing Internationalized Log Messages

6.2.4.1. Add Message IDs and Project Codes to Log Messages

This task shows how to add message IDs and project codes to internationalized log messages created using JBoss Logging Tools. A log message must have both a project code and message ID for them to be displayed in the log. If a message does not have both a project code and a message ID, then neither is displayed.

Refer to the **logging-tools** quick start for a complete example.

Prerequisites

1. You must already have a project with internationalized log messages. Refer to [Section 6.2.2.1, “Create Internationalized Log Messages”](#).

2. You need to know the project code you will be using. You can use a single project code, or define different ones for each interface.

Procedure 6.6. Add message IDs and Project Codes to Log Messages

1. **Specify the project code for the interface.**

Specify the project code using the `projectCode` attribute of the `@MessageLogger` annotation attached to a custom logger interface. All messages that are defined in the interface will use that project code.

```
@MessageLogger(projectCode="ACCNTS")
interface AccountsLogger extends BasicLogger
{

}
```

2. **Specify Message IDs**

Specify a message ID for each message using the `id` attribute of the `@Message` annotation attached to the method that defines the message.

```
@LogMessage
@Message(id=43, value = "Customer query failed, Database not
available.") void customerQueryFailDBClosed();
```

The log messages that have both a message ID and project code associated with them will prepend these to the logged message.

```
10:55:50,638 INFO [com.company.accounts.ejb] (MSC service thread 1-4)
ACCNTS000043: Customer query failed, Database not available.
```

[Report a bug](#)

6.2.4.2. Specify the Log Level for a Message

The default log level of a message defined by an interface by JBoss Logging Tools is **INFO**. A different log level can be specified with the `level` attribute of the `@LogMessage` annotation attached to the logging method.

Procedure 6.7. Specify the log level for a message

1. **Specify level attribute**

Add the `level` attribute to the `@LogMessage` annotation of the log message method definition.

2. **Assign log level**

Assign the `level` attribute the value of the log level for this message. The valid values for `level` are the six enumerated constants defined in `org.jboss.logging.Logger.Level`: **DEBUG**, **ERROR**, **FATAL**, **INFO**, **TRACE**, and **WARN**.

```
Import org.jboss.logging.Logger.Level;

@LogMessage(level=Level.ERROR)
@Message(value = "Customer query failed, Database not available.")
void customerQueryFailDBClosed();
```

Invoking the logging method in the above sample will produce a log message at the level of **ERROR**.

```
10:55:50,638 ERROR [com.company.app.Main] (MSC service thread 1-4)
  Customer query failed, Database not available.
```

[Report a bug](#)

6.2.4.3. Customize Log Messages with Parameters

Custom logging methods can define parameters. These parameters are used to pass additional information to be displayed in the log message. Where the parameters appear in the log message is specified in the message itself using either explicit or ordinary indexing.

Procedure 6.8. Customize log messages with parameters

1. Add parameters to method definition

Parameters of any type can be added to the method definition. Regardless of type, the String representation of the parameter is what is displayed in the message.

2. Add parameter references to the log message

References can use explicit or ordinary indexes.

- To use ordinary indexes, insert the characters `%s` in the message string where you want each parameter to appear. The first instance of `%s` will insert the first parameter, the second instance will insert the second parameter, and so on.
- To use explicit indexes, insert the characters `%{#}$s` in the message, where `#` indicates the number of the parameter you wish to appear.



IMPORTANT

Using explicit indexes allows the parameter references in the message to be in a different order than they are defined in the method. This is important for translated messages which may require different ordering of parameters.

The number of parameters must match the number of references to the parameters in the specified message or the code will not compile. A parameter marked with the **@Cause** annotation is not included in the number of parameters.

Example 6.1. Message parameters using ordinary indexes

```
@LogMessage(level=Logger.Level.DEBUG)
@Message(id=2, value="Customer query failed, customerid:%s, user:%s")
void customerLookupFailed(Long customerid, String username);
```

Example 6.2. Message parameters using explicit indexes

```
@LogMessage(level=Logger.Level.DEBUG)
@Message(id=2, value="Customer query failed, user:%2$s,
customerid:%1$s")
void customerLookupFailed(Long customerid, String username);
```


[Report a bug](#)

6.2.4.4. Specify an Exception as the Cause of a Log Message

JBoss Logging Tools allows one parameter of a custom logging method to be defined as the cause of the message. This parameter must be of the type **Throwable** or any of its sub-classes and is marked with the **@Cause** annotation. This parameter cannot be referenced in the log message like other parameters and is displayed after the log message.

The following procedure shows how to update a logging method using the **@Cause** parameter to indicate the "causing" exception. It is assumed that you have already created internationalized logging messages to which you want to add this functionality.

Procedure 6.9. Specify an exception as the cause of a log message

1. Add the parameter

Add a parameter of the type **Throwable** or a sub-class to the method.

```
@LogMessage
@Message(id=404, value="Loading configuration failed. Config
file:%s")
void loadConfigFailed(Exception ex, File file);
```

2. Add the annotation

Add the **@Cause** annotation to the parameter.

```
import org.jboss.logging.Cause

@LogMessage
@Message(value = "Loading configuration failed. Config file: %s")
void loadConfigFailed(@Cause Exception ex, File file);
```

3. Invoke the method

When the method is invoked in your code, an object of the correct type must be passed and will be displayed after the log message.

```
try
{
    confFile=new File(filename);
    props.load(new FileInputStream(confFile));
}
catch(Exception ex) //in case properties file cannot be read
{
    ConfigLogger.LOGGER.loadConfigFailed(ex, filename);
}
```

Below is the output of the above code samples if the code threw an exception of type **FileNotFoundException**.

```
10:50:14,675 INFO [com.company.app.Main] (MSC service thread 1-3)
Loading configuration failed. Config file: customised.properties
java.io.FileNotFoundException: customised.properties (No such file
```

```

    or directory)
    at java.io.FileInputStream.open(Native Method)
    at java.io.FileInputStream.<init>(FileInputStream.java:120)
    at com.company.app.demo.Main.openCustomProperties(Main.java:70)
    at com.company.app.Main.go(Main.java:53)
    at com.company.app.Main.main(Main.java:43)

```

[Report a bug](#)

6.2.5. Customizing Internationalized Exceptions

6.2.5.1. Add Message IDs and Project Codes to Exception Messages

The following procedure shows the steps required to add message IDs and project codes to internationalized Exception messages created using JBoss Logging Tools.

Message IDs and project codes are unique identifiers that are prepended to each message displayed by internationalized exceptions. These identifying codes make it possible to create a reference of all the exception messages for an application so that someone can lookup the meaning of an exception message written in language that they do not understand.

Prerequisites

1. You must already have a project with internationalized exceptions. Refer to [Section 6.2.2.3, “Create Internationalized Exceptions”](#).
2. You need to know the project code you will be using. You can use a single project code, or define different ones for each interface.

Procedure 6.10. Add Message IDs and Project Codes to Exception Messages

1. Specify a project code

Specify the project code using the **projectCode** attribute of the **@MessageBundle** annotation attached to a exception bundle interface. All messages that are defined in the interface will use that project code.

```

@MessageBundle(projectCode="ACCTS")
interface ExceptionBundle
{
    ExceptionBundle EXCEPTIONS =
    Messages.getBundle(ExceptionBundle.class);
}

```

2. Specify message IDs

Specify a message ID for each exception using the **id** attribute of the **@Message** annotation attached to the method that defines the exception.

```

@Message(id=143, value = "The config file could not be opened.")
IOException configFileAccessError();

```



IMPORTANT

A message that has both a project code and message ID displays them prepended to the message. If a message does not have both a project code and a message ID, neither is displayed.

Example 6.3. Creating internationalized exceptions

This exception bundle interface has the project code of ACCTS, with a single exception method with the ID of 143.

```
@MessageBundle(projectCode="ACCTS")
interface ExceptionBundle
{
    ExceptionBundle EXCEPTIONS =
        Messages.getBundle(ExceptionBundle.class);

    @Message(id=143, value = "The config file could not be opened.")
    IOException configFileAccessError();
}
```

The exception object can be obtained and thrown using the following code.

```
throw ExceptionBundle.EXCEPTIONS.configFileAccessError();
```

This would display an exception message like the following:

```
Exception in thread "main" java.io.IOException: ACCTS000143: The config
file could not be opened.
at com.company.accounts.Main.openCustomProperties(Main.java:78)
at com.company.accounts.Main.go(Main.java:53)
at com.company.accounts.Main.main(Main.java:43)
```

[Report a bug](#)

6.2.5.2. Customize Exception Messages with Parameters

Exception bundle methods that define exceptions can specify parameters to pass additional information to be displayed in the exception message. Where the parameters appear in the exception message is specified in the message itself using either explicit or ordinary indexing.

The following procedure shows the steps required to use method parameters to customize method exceptions.

Procedure 6.11. Customize an exception message with parameters

1. **Add parameters to method definition**

Parameters of any type can be added to the method definition. Regardless of type, the **String** representation of the parameter is what is displayed in the message.

2. **Add parameter references to the exception message**

References can use explicit or ordinary indexes.

- To use ordinary indexes, insert the characters `%s` in the message string where you want each parameter to appear. The first instance of `%s` will insert the first parameter, the second instance will insert the second parameter, and so on.
- To use explicit indexes, insert the characters `%{#}$s` in the message where `#` indicates the number of the parameter which you wish to appear.

Using explicit indexes allows the parameter references in the message to be in a different order than they are defined in the method. This is important for translated messages which may require different ordering of parameters.



IMPORTANT

The number of parameters must match the number of references to the parameters in the specified message or the code will not compile. A parameter marked with the `@Cause` annotation is not included in the number of parameters.

Example 6.4. Using ordinary indexes

```
@Message(id=2, value="Customer query failed, customerid:%s, user:%s")
void customerLookupFailed(Long customerid, String username);
```

Example 6.5. Using explicit indexes

```
@Message(id=2, value="Customer query failed, user:%2$s,
customerid:%1$s")
void customerLookupFailed(Long customerid, String username);
```

[Report a bug](#)

6.2.5.3. Specify One Exception as the Cause of Another Exception

Exceptions returned by exception bundle methods can have another exception specified as the underlying cause. This is done by adding a parameter to the method and annotating the parameter with `@Cause`. This parameter is used to pass the causing exception. This parameter cannot be referenced in the exception message.

The following procedure shows how to update a method from an exception bundle using the `@Cause` parameter to indicate the causing exception. It is assumed that you have already created an exception bundle to which you want to add this functionality.

Procedure 6.12. Specify one exception as the cause of another exception

1. Add the parameter

Add the a parameter of the type `Throwable` or a sub-class to the method.

```
@Message(id=328, value = "Error calculating: %s.")
ArithmeticException calculationError(Throwable cause, String msg);
```

2. Add the annotation

Add the **@Cause** annotation to the parameter.

```
import org.jboss.logging.Cause

@Message(id=328, value = "Error calculating: %s.")
ArithmeticException calculationError(@Cause Throwable cause, String
msg);
```

3. Invoke the method

Invoke the interface method to obtain an exception object. The most common use case is to throw a new exception from a catch block using the caught exception as the cause.

```
try
{
    ...
}
catch(Exception ex)
{
    throw ExceptionBundle.EXCEPTIONS.calculationError(
        ex, "calculating payment due
per day");
}
```

Example 6.6. Specify one exception as the cause of another exception

This exception bundle defines a single method that returns an exception of type `ArithmeticException`.

```
@MessageBundle(projectCode = "TPS")
interface CalcExceptionBundle
{
    CalcExceptionBundle EXCEPTIONS =
    Messages.getBundle(CalcExceptionBundle.class);

    @Message(id=328, value = "Error calculating: %s.")
    ArithmeticException calcError(@Cause Throwable cause, String value);
}
```

This code snippet performs an operation that throws an exception because it attempts to divide an integer by zero. The exception is caught and a new exception is created using the first one as the cause.

```
int totalDue = 5;
int daysToPay = 0;
int amountPerDay;

try
{
    amountPerDay = totalDue/daysToPay;
}
catch (Exception ex)
{
}
```

```
        throw CalcExceptionBundle.EXCEPTIONS.calcError(ex, "payments per
day");
    }
```

This is what the exception message looks like:

```
Exception in thread "main" java.lang.ArithmeticException: TPS000328:
Error calculating: payments per day.
    at com.company.accounts.Main.go(Main.java:58)
    at com.company.accounts.Main.main(Main.java:43)
Caused by: java.lang.ArithmeticException: / by zero
    at com.company.accounts.Main.go(Main.java:54)
    ... 1 more
```

[Report a bug](#)

6.2.6. Reference

6.2.6.1. JBoss Logging Tools Maven Configuration

To build a Maven project that uses JBoss Logging Tools for internationalization you must make the following changes to the project's configuration in the **pom.xml** file.

Refer to the **logging-tools** quick start for an example of a complete working **pom.xml** file.

1. JBoss Maven Repository must be enabled for the project. Refer to [Section 2.3.2, "Configure the JBoss EAP 6 Maven Repository Using the Maven Settings"](#).
2. The Maven dependencies for **jboss-logging** and **jboss-logging-processor** must be added. Both of dependencies are available in JBoss EAP 6 so the scope element of each can be set to **provided** as shown.

```
<dependency>
  <groupId>org.jboss.logging</groupId>
  <artifactId>jboss-logging-processor</artifactId>
  <version>1.0.0.Final</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>org.jboss.logging</groupId>
  <artifactId>jboss-logging</artifactId>
  <version>3.1.0.GA</version>
  <scope>provided</scope>
</dependency>
```

3. The **maven-compiler-plugin** must be at least version **2.2** and be configured for target and generated sources of **1.6**.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
```

```

    <version>2.3.2</version>
    <configuration>
      <source>1.6</source>
      <target>1.6</target>
    </configuration>
  </plugin>

```

[Report a bug](#)

6.2.6.2. Translation Property File Format

The property files used for translations of messages in JBoss Logging Tools are standard Java property files. The format of the file is the simple line-oriented, **key=value** pair format described in the documentation for the `java.util.Properties` class, <http://docs.oracle.com/javase/6/docs/api/java/util/Properties.html>.

The file name format has the following format:

```
InterfaceName.i18n_locale_COUNTRY_VARIANT.properties
```

- **InterfaceName** is the name of the interface that the translations apply to.
- **locale**, **COUNTRY**, and **VARIANT** identify the regional settings that the translation applies to.
- **locale** and **COUNTRY** specify the language and country using the ISO-639 and ISO-3166 Language and Country codes respectively. **COUNTRY** is optional.
- **VARIANT** is an optional identifier that can be used to identify translations that only apply to a specific operating system or browser.

The properties contained in the translation file are the names of the methods from the interface being translated. The assigned value of the property is the translation. If a method is overloaded then this is indicated by appending a dot and then the number of parameters to the name. Methods for translation can only be overloaded by supplying a different number of parameters.

Example 6.7. Sample Translation Properties File

File name: **GreeterService.i18n_fr_FR_POSIX.properties**.

```

# Level: Logger.Level.INFO
# Message: Hello message sent.
logHelloMessageSent=Bonjour message envoyé.

```

[Report a bug](#)

6.2.6.3. JBoss Logging Tools Annotations Reference

The following annotations are defined in JBoss Logging for use with internationalization and localization of log messages, strings, and exceptions.

Table 6.1. JBoss Logging Tools Annotations

Annotation	Target	Description	Attributes
@MessageBundle	Interface	Defines the interface as a Message Bundle.	projectCode
@MessageLogger	Interface	Defines the interface as a Message Logger.	projectCode
@Message	Method	Can be used in Message Bundles and Message Loggers. In a Message Logger it defines a method as being a localized logger. In a Message Bundle it defines the method as being one that returns a localized String or Exception object.	value, id
@LogMessage	Method	Defines a method in a Message Logger as being a logging method.	level1 (default INFO)
@Cause	Parameter	Defines a parameter as being one that passes an Exception as the cause of either a Log message or another Exception.	-
@Param	Parameter	Defines a parameter as being one that is passed to the constructor of the Exception.	-

[Report a bug](#)

CHAPTER 7. REMOTE JNDI LOOKUP

7.1. REGISTERING OBJECTS TO JNDI

The Java Naming and Directory Interface (JNDI) is a Java API for a directory service that allows Java software clients to discover and look up objects using a name. To look up an object, you must first register that object to JNDI using the **java:jboss/exported** context.

The following is an example of how to register a JMS queue to JNDI in the **messaging** subsystem so that it can be looked up by remote JNDI clients.

```
java:jboss/exported/jms/queue/myTestQueue
```

Remote JNDI clients can then look up the object using the above name; however, it is not necessary to specify the **java:jboss/exported/** prefix when looking up a remote client. The remote JNDI clients can look up the remote object up using the following name.

```
jms/queue/myTestQueue
```

Example 7.1. Example of Standalone Server JMS Queue Configuration

```
<subsystem xmlns="urn:jboss:domain:messaging:1.4">
  <hornetq-server>
    ...
    <jms-destinations>
      <jms-queue name="myTestQueue">
        <entry name="java:jboss/exported/jms/queue/myTestQueue"/>
      </jms-queue>
    </jms-destinations>
  </hornetq-server>
</subsystem>
```

[Report a bug](#)

7.2. CONFIGURING A REMOTE JNDI CLIENT

Remote JNDI clients can look up and connect to objects by name using JNDI. The client must have **jboss-client.jar** on its class path.

The following example shows how to look up the **myTestQueue** JMS queue from a remote JNDI client:

Example 7.2. Example Remote JNDI Lookup

```
Properties properties = new Properties();
properties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
properties.put(Context.PROVIDER_URL, "remote://<hostname>:4447");
context = new InitialContext(properties);
Queue myTestQueue = (Queue) context.lookup("jms/queue/myTestQueue");
```

[Report a bug](#)

CHAPTER 8. ENTERPRISE JAVABEANS

8.1. INTRODUCTION

8.1.1. Overview of Enterprise JavaBeans

Enterprise JavaBeans (EJB) 3.1 is an API for developing distributed, transactional, secure and portable Java EE applications through the use of server-side components called Enterprise Beans. Enterprise Beans implement the business logic of an application in a decoupled manner that encourages reuse. Enterprise JavaBeans 3.1 is documented as the Java EE specification JSR-318.

JBoss EAP 6 has full support for applications built using the Enterprise JavaBeans 3.1 specification.

[Report a bug](#)

8.1.2. EJB 3.1 Feature Set

The following features are supported in EJB 3.1

- Session Beans
- Message Driven Beans
- No-interface views
- local interfaces
- remote interfaces
- JAX-WS web services
- JAX-RS web services
- Timer Service
- Asynchronous Calls
- Interceptors
- RMI/IIOP interoperability
- Transaction support
- Security
- Embeddable API

The following features are supported in EJB 3.1 but are proposed for "pruning". This means that these features may become optional in Java EE 7.

- Entity Beans (container and bean-managed persistence)
- EJB 2.1 Entity Bean client views
- EJB Query Language (EJB QL)

- JAX-RPC based Web Services (endpoints and client views)

[Report a bug](#)

8.1.3. EJB 3.1 Lite

EJB Lite is a sub-set of the EJB 3.1 specification. It provides a simpler version of the full EJB 3.1 specification as part of the Java EE 6 web profile.

EJB Lite simplifies the implementation of business logic in web applications with enterprise beans by:

1. Only supporting the features that make sense for web-applications, and
2. allowing EJBs to be deployed in the same WAR file as a web-application.

[Report a bug](#)

8.1.4. EJB 3.1 Lite Features

EJB Lite includes the following features:

- Stateless, stateful, and singleton session beans
- Local business interfaces and "no interface" beans
- Interceptors
- Container-managed and bean-managed transactions
- Declarative and programmatic security
- Embeddable API

The following features of EJB 3.1 are specifically not included:

- Remote interfaces
- RMI-IIOP Interoperability
- JAX-WS Web Service Endpoints
- EJB Timer Service
- Asynchronous session bean invocations
- Message-driven beans

[Report a bug](#)

8.1.5. Enterprise Beans

Enterprise beans are server-side application components as defined in the Enterprise JavaBeans (EJB) 3.1 specification, JSR-318. Enterprise beans are designed for the implementation of application business logic in a decoupled manner to encourage reuse.

Enterprise beans are written as Java classes and annotated with the appropriate EJB annotations. They can be deployed to the application server in their own archive (a JAR file) or be deployed as part of a

Java EE application. The application server manages the lifecycle of each enterprise bean and provides services to them such as security, transactions, and concurrency management.

An enterprise bean can also define any number of business interfaces. Business interfaces provide greater control over which of the bean's methods are available to clients and can also allow access to clients running in remote JVMs.

There are three types of Enterprise Bean: Session beans, Message-driven beans and Entity beans.



IMPORTANT

Entity beans are now deprecated in EJB 3.1 and Red Hat recommends the use of JPA entities instead. Red Hat only recommends the use of Entity beans for backwards compatibility with legacy systems.

[Report a bug](#)

8.1.6. Overview of Writing Enterprise Beans

Enterprise beans are server-side components designed to encapsulate business logic in a manner decoupled from any one specific application client. By implementing your business logic within enterprise beans you will be able to reuse those beans in multiple applications.

Enterprise beans are written as annotated Java classes and do not have to implement any specific EJB interfaces or be sub-classed from any EJB super classes to be considered an enterprise bean.

EJB 3.1 enterprise beans are packaged and deployed in Java archive (JAR) files. An enterprise bean JAR file can be deployed to your application server, or included in an enterprise archive (EAR) file and deployed with that application. It is also possible to deploy enterprise beans in a WAR file along side a web application.

[Report a bug](#)

8.1.7. Session Bean Business Interfaces

8.1.7.1. Enterprise Bean Business Interfaces

An EJB business interface is a Java interface written by the bean developer which provides declarations of the public methods of a session bean that are available for clients. Session beans can implement any number of interfaces including none (a "no-interface" bean).

Business interfaces can be declared as local or remote interfaces but not both.

[Report a bug](#)

8.1.7.2. EJB Local Business Interfaces

An EJB local business interface declares the methods which are available when the bean and the client are in the same JVM. When a session bean implements a local business interface only the methods declared in that interface will be available to clients.

[Report a bug](#)

8.1.7.3. EJB Remote Business Interfaces

An EJB remote business interface declares the methods which are available to remote clients. Remote access to a session bean that implements a remote interface is automatically provided by the EJB container.

A remote client is any client running in a different JVM and can include desktop applications as well as web applications, services and enterprise beans deployed to a different application server.

Local clients can access the methods exposed by a remote business interface.

[Report a bug](#)

8.1.7.4. EJB No-interface Beans

A session bean that does not implement any business interfaces is called a no-interface bean. All of the public methods of no-interface beans are accessible to local clients.

A session bean that implements a business interface can also be written to expose a "no-interface" view.

[Report a bug](#)

8.2. CREATING ENTERPRISE BEAN PROJECTS

8.2.1. Create an EJB Archive Project Using Red Hat JBoss Developer Studio

This task describes how to create an Enterprise JavaBeans (EJB) project in Red Hat JBoss Developer Studio.

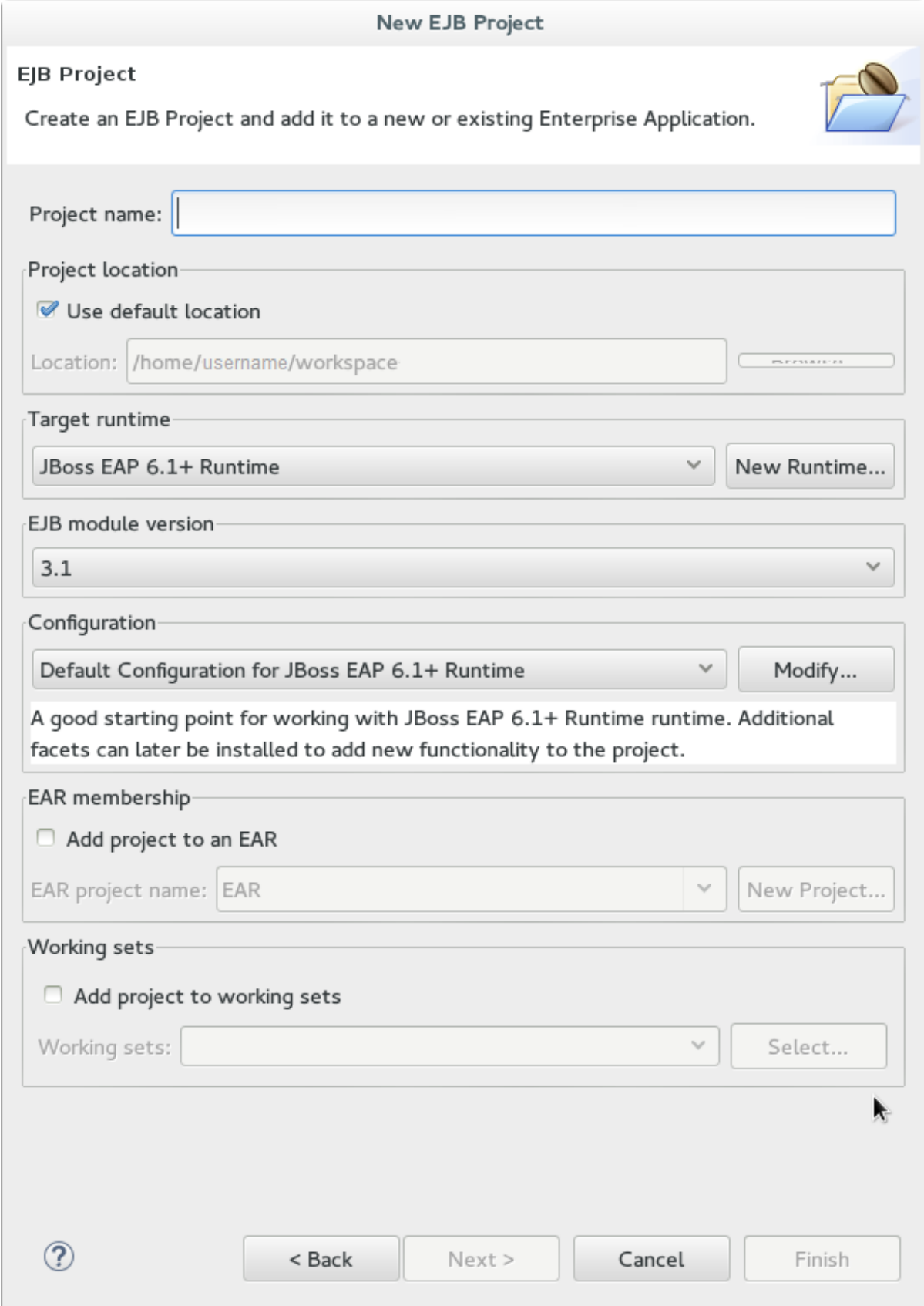
Prerequisites

- A server and server runtime for JBoss EAP 6 has been set up. See [Section 1.3.1.5, “Add the JBoss EAP Server Using Define New Server”](#).

Procedure 8.1. Create an EJB Project in Red Hat JBoss Developer Studio

1. **Create new project**

To open the New EJB Project wizard, navigate to the **File** menu, select **New**, and then **EJB Project**.



New EJB Project

EJB Project
Create an EJB Project and add it to a new or existing Enterprise Application.

Project name:

Project location

☒ Use default location

Location:

Target runtime

EJB module version

Configuration

A good starting point for working with JBoss EAP 6.1+ Runtime runtime. Additional facets can later be installed to add new functionality to the project.

EAR membership

☐ Add project to an EAR

EAR project name:

Working sets

☐ Add project to working sets

Working sets:

Figure 8.1. New EJB Project wizard

2. Specify Details

Supply the following details:

- Project name.

As well as the being the name of the project that appears in Red Hat JBoss Developer Studio this is also the default filename for the deployed JAR file.

- Project location.

The directory where the project's files will be saved. The default is a directory in the current workspace.

- Target Runtime.

This is the server runtime used for the project. This will need to be set to the same JBoss EAP 6 runtime used by the server that you will be deploying to.

- EJB module version. This is the version of the EJB specification that your enterprise beans will comply with. Red Hat recommends using **3.1**.
- Configuration. This allows you to adjust the supported features in your project. Use the default configuration for your selected runtime.

Click **Next** to continue.

3. Java Build Configuration

This screen allows you to customize the directories will contain Java source files and the directory where the built output is placed.

Leave this configuration unchanged and click **Next**.

4. EJB Module settings

Check the **Generate ejb-jar.xml deployment descriptor** checkbox if a deployment descriptor is required. The deployment descriptor is optional in EJB 3.1 and can be added later if required.

Click **Finish** and the project is created and will be displayed in the Project Explorer.

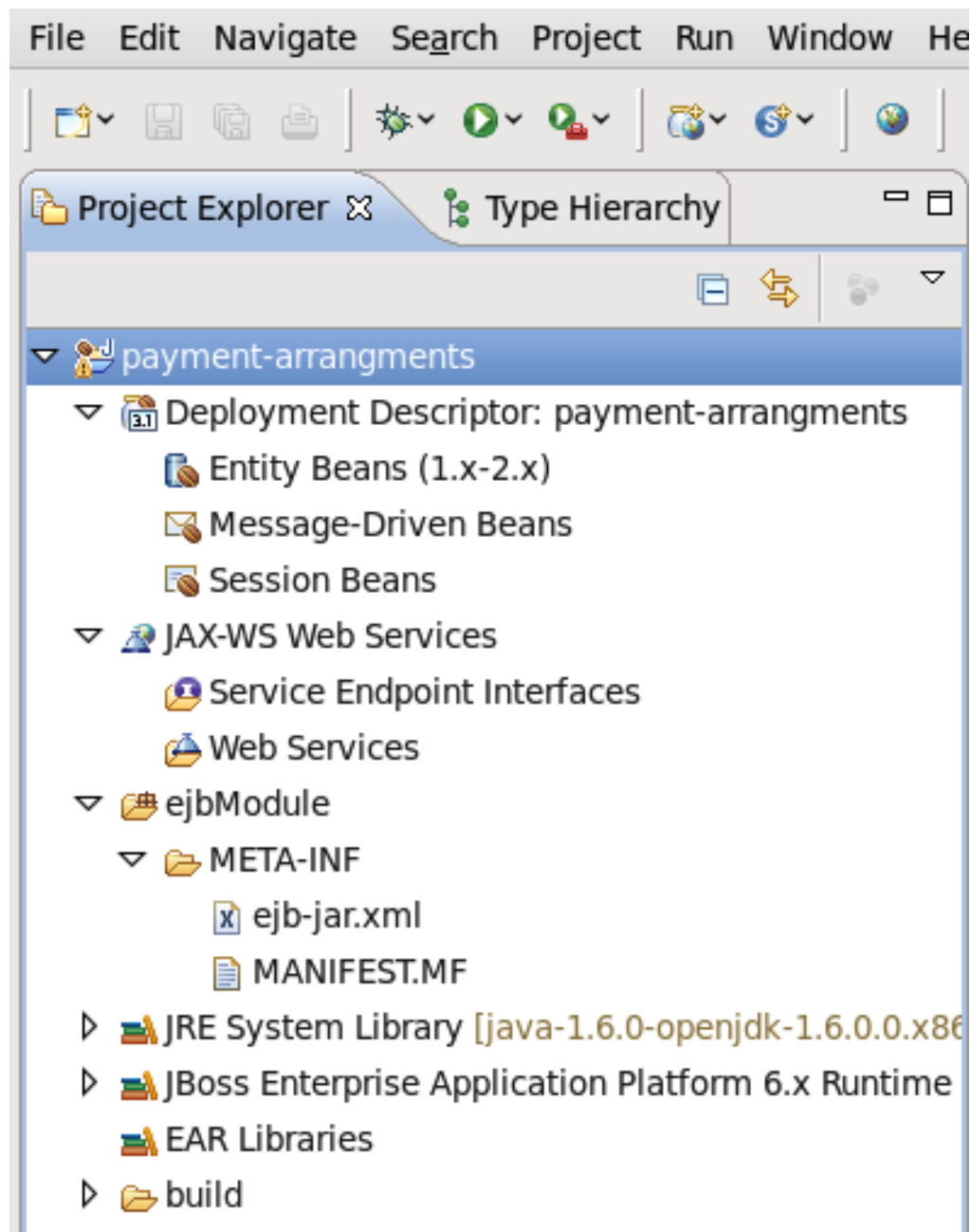


Figure 8.2. Newly created EJB Project in the Project Explorer

5. Add Build Artifact to Server for Deployment

Open the **Add and Remove** dialog by right-clicking on the server you want to deploy the built artifact to in the server tab, and select "Add and Remove".

Select the resource to deploy from the **Available** column and click the **Add** button. The resource will be moved to the **Configured** column. Click **Finish** to close the dialog.

Add and Remove

Modify the resources that are configured on the server



Move resources to the right to configure them on the server

Available:		Configured:
payment-arrangments	Add >	
	< Remove	
	Add All >>	
	<< Remove All	

☒ If server is started, publish changes immediately

< Back **Next >** **Cancel** **Finish**

Figure 8.3. Add and Remove dialog

Result

You now have an EJB Project in Red Hat JBoss Developer Studio that can build and deploy to the specified server.

If no enterprise beans are added to the project then Red Hat JBoss Developer Studio will display the warning "An EJB module must contain one or more enterprise beans." This warning will disappear once one or more enterprise beans have been added to the project.

[Report a bug](#)

8.2.2. Create an EJB Archive Project in Maven

This task demonstrates how to create a project using Maven that contains one or more enterprise beans packaged in a JAR file.

Prerequisites:

- Maven is already installed.
- You understand the basic usage of Maven.

Procedure 8.2. Create an EJB Archive project in Maven

1. Create the Maven project

An EJB project can be created using Maven's archetype system and the **ejb-javaee6** archetype. To do this run the **mvn** command with parameters as shown:

```
mvn archetype:generate -
DarchetypeGroupId=org.codehaus.mojo.archetypes -
DarchetypeArtifactId=ejb-javaee6
```

Maven will prompt you for the **groupId**, **artifactId**, **version** and **package** for your project.

```
[localhost]$ mvn archetype:generate -
DarchetypeGroupId=org.codehaus.mojo.archetypes -
DarchetypeArtifactId=ejb-javaee6
[INFO] Scanning for projects...
[INFO]
[INFO] -----
-----
[INFO] Building Maven Stub Project (No POM) 1
[INFO] -----
-----
[INFO]
[INFO] >>> maven-archetype-plugin:2.0:generate (default-cli) @
standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.0:generate (default-cli) @
standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.0:generate (default-cli) @
standalone-pom ---
[INFO] Generating project in Interactive mode
[INFO] Archetype [org.codehaus.mojo.archetypes:ejb-javaee6:1.5]
found in catalog remote
Define value for property 'groupId': : com.shinysparkly
Define value for property 'artifactId': : payment-arrangments
Define value for property 'version': 1.0-SNAPSHOT: :
Define value for property 'package': com.shinysparkly: :
Confirm properties configuration:
groupId: com.company
artifactId: payment-arrangments
version: 1.0-SNAPSHOT
package: com.company.collections
Y: :
[INFO] -----
-----
[INFO] BUILD SUCCESS
[INFO] -----
-----
[INFO] Total time: 32.440s
[INFO] Finished at: Mon Oct 31 10:11:12 EST 2011
[INFO] Final Memory: 7M/81M
[INFO] -----
-----
[localhost]$
```

2. Add your enterprise beans

Write your enterprise beans and add them to the project under the **src/main/java** directory in the appropriate sub-directory for the bean's package.

3. Build the project

To build the project, run the **mvn package** command in the same directory as the **pom.xml** file. This will compile the Java classes and package the JAR file. The built JAR file is named **artifactId-version.jar** and is placed in the **target/** directory.

RESULT: You now have a Maven project that builds and packages a JAR file. This project can contain enterprise beans and the JAR file can be deployed to an application server.

[Report a bug](#)

8.2.3. Create an EAR Project containing an EJB Project

This task describes how to create a new Enterprise Archive (EAR) project in Red Hat JBoss Developer Studio that contains an EJB Project.

Prerequisites

- A server and server runtime for JBoss EAP 6 has been set up. See [Section 1.3.1.5, “Add the JBoss EAP Server Using Define New Server”](#).

Procedure 8.3. Create an EAR Project containing an EJB Project

1. Open the New EAR Application Project Wizard

Navigate to the **File** menu, select **New**, then **Project** and the **New Project** wizard appears. Select **Java EE/Enterprise Application Project** and click **Next**.

New EAR Application Project

EAR Application Project
Create a EAR application.

Project name:

/home/username/workspace

Project location
☒ Use default location
 Location:

Target runtime

EAR version

Configuration

 A good starting point for working with JBoss EAP 6.1+ Runtime runtime. Additional facets can later be installed to add new functionality to the project.

Working sets
☐ Add project to working sets
 Working sets:

? < Back Next > Cancel Finish

Figure 8.4. New EAR Application Project Wizard

2. Supply details

Supply the following details:

- Project name.

As well as the being the name of the project that appears in Red Hat JBoss Developer Studio this is also the default filename for the deployed EAR file.

- Project location.

The directory where the project's files will be saved. The default is a directory in the current workspace.

- Target Runtime.

This is the server runtime used for the project. This will need to be set to the same JBoss EAP 6 runtime used by the server that you will be deploying to.

- EAR version.

This is the version of the Java Enterprise Edition specification that your project will comply with. Red Hat recommends using **6**.

- Configuration. This allows you to adjust the supported features in your project. Use the default configuration for your selected runtime.

Click **Next** to continue.

3. Add a new EJB Module

New Modules can be added from the **Enterprise Application** page of the wizard. To add a new EJB Project as a module follow the steps below:

a. Add new EJB Module

Click **New Module**, uncheck **Create Default Modules** checkbox, select the **Enterprise Java Bean** and click **Next**. The **New EJB Project** wizard appears.

b. Create EJB Project

New EJB Project wizard is the same as the wizard used to create new standalone EJB Projects and is described in [Section 8.2.1, “Create an EJB Archive Project Using Red Hat JBoss Developer Studio”](#).

The minimal details required to create the project are:

- Project name
- Target Runtime
- EJB Module version
- Configuration

All the other steps of the wizard are optional. Click **Finish** to complete creating the EJB Project.

The newly created EJB project is listed in the Java EE module dependencies and the checkbox is checked.

4. Optional: add an application.xml deployment descriptor

Check the **Generate application.xml deployment descriptor** checkbox if one is required.

5. Click Finish

Two new project will appear, the EJB project and the EAR project

6. Add Build Artifact to Server for Deployment

Open the **Add and Remove** dialog by right-clicking in the **Servers** tab on the server you want to deploy the built artifact to in the server tab, and select **Add and Remove**.

Select the EAR resource to deploy from the **Available** column and click the **Add** button. The resource will be moved to the **Configured** column. Click **Finish** to close the dialog.

Add and Remove

Modify the resources that are configured on the server



Move resources to the right to configure them on the server

Available:		Configured:
	Add >	<div> <div>CollectionsApp</div> <div> <div>CollectionsAppEJB</div> </div> </div>
	< Remove	
	Add All >>	
	<< Remove All	

☒ If server is started, publish changes immediately

?
< Back
Next >
Cancel
Finish

Figure 8.5. Add and Remove dialog

Result

You now have an Enterprise Application Project with a member EJB Project. This will build and deploy to the specified server as a single EAR deployment containing an EJB subdeployment.

[Report a bug](#)

8.2.4. Add a Deployment Descriptor to an EJB Project

An EJB deployment descriptor can be added to an EJB project that was created without one. To do this, follow the procedure below.

Perquisites:

- You have a EJB Project in Red Hat JBoss Developer Studio to which you want to add an EJB deployment descriptor.

Procedure 8.4. Add an Deployment Descriptor to an EJB Project

1. Open the Project

Open the project in Red Hat JBoss Developer Studio.

2. Add Deployment Descriptor

Right-click on the Deployment Descriptor folder in the project view and select **Generate Deployment Descriptor Stub**.

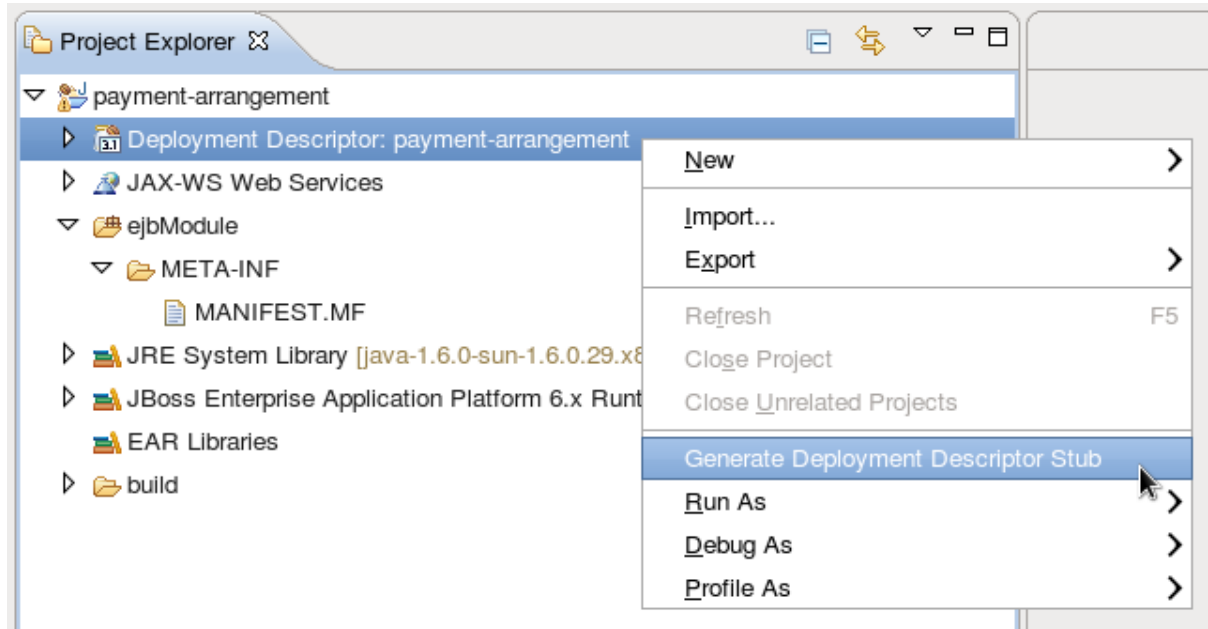


Figure 8.6. Adding a Deployment Descriptor

The new file, `ejb-jar.xml`, is created in `ejbModule/META-INF/`. Double-clicking on the Deployment Descriptor folder in the project view will also open this file.

[Report a bug](#)

8.3. SESSION BEANS

8.3.1. Session Beans

Session Beans are Enterprise Beans that encapsulate a set of related business processes or tasks and are injected into the classes that request them. There are three types of session bean: stateless, stateful, and singleton.

[Report a bug](#)

8.3.2. Stateless Session Beans

Stateless session beans are the simplest yet most widely used type of session bean. They provide business methods to client applications but do not maintain any state between method calls. Each method is a complete task that does not rely on any shared state within that session bean. Because there is no state, the application server is not required to ensure that each method call is performed on the same instance. This makes stateless session beans very efficient and scalable.

[Report a bug](#)

8.3.3. Stateful Session Beans

Stateful session beans are Enterprise Beans that provide business methods to client applications and maintain conversational state with the client. They should be used for tasks that must be done in several steps (method calls), each of which relies on the state of the previous step being maintained. The application server ensures that each client receives the same instance of a stateful session bean for each method call.

[Report a bug](#)

8.3.4. Singleton Session Beans

Singleton session beans are session beans that are instantiated once per application and every client request for a singleton bean goes to the same instance. Singleton beans are an implementation of the Singleton Design Pattern as described in the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides; published by Addison-Wesley in 1994.

Singleton beans provide the smallest memory footprint of all the session bean types but must be designed as thread-safe. EJB 3.1 provides container-managed concurrency (CMC) to allow developers to implement thread safe singleton beans easily. However singleton beans can also be written using traditional multi-threaded code (bean-managed concurrency or BMC) if CMC does not provide enough flexibility.

[Report a bug](#)

8.3.5. Add Session Beans to a Project in Red Hat JBoss Developer Studio

Red Hat JBoss Developer Studio has several wizards that can be used to quickly create enterprise bean classes. The following procedure shows how to use the Red Hat JBoss Developer Studio wizards to add a session bean to a project.

Prerequisites:

- You have a EJB or Dynamic Web Project in Red Hat JBoss Developer Studio to which you want to add one or more session beans.

Procedure 8.5. Add Session Beans to a Project in Red Hat JBoss Developer Studio

1. **Open the Project**
Open the project in Red Hat JBoss Developer Studio.
2. **Open the "Create EJB 3.x Session Bean" wizard**
To open the **Create EJB 3.x Session Bean** wizard, navigate to the **File** menu, select **New**, and then **Session Bean (EJB 3.x)**.

Create EJB 3.x Session Bean

Specify class file destination.



Project:

Source folder:

Java package:

Class name:

Superclass:

State type:

Create business interface

☐ Remote

☐ Local

☒ No-interface View

Figure 8.7. Create EJB 3.x Session Bean wizard

3. Specify class information

Supply the following details:

- Project

Verify the correct project is selected.

- Source folder

This is the folder that the Java source files will be created in. This should not usually need to be changed.

- Package

Specify the package that the class belongs to.

- Class name

Specify the name of the class that will be the session bean.

- Superclass

The session bean class can inherit from a super class. Specify that here if your session has a super class.

- State type

Specify the state type of the session bean: stateless, stateful, or singleton.

- Business Interfaces

By default the No-interface box is checked so no interfaces will be created. Check the boxes for the interfaces you wish to define and adjust the names if necessary.

Remember that enterprise beans in a web archive (WAR) only support EJB 3.1 Lite and this does not include remote business interfaces.

Click **Next**.

4. Session Bean Specific Information

You can enter in additional information here to further customize the session bean. It is not required to change any of the information here.

Items that you can change are:

- Bean name.
- Mapped name.
- Transaction type (Container managed or Bean managed).
- Additional interfaces can be supplied that the bean must implement.
- You can also specify EJB 2.x Home and Component interfaces if required.

5. Finish

Click **Finish** and the new session bean will be created and added to the project. The files for any new business interfaces will also be created if they were specified.

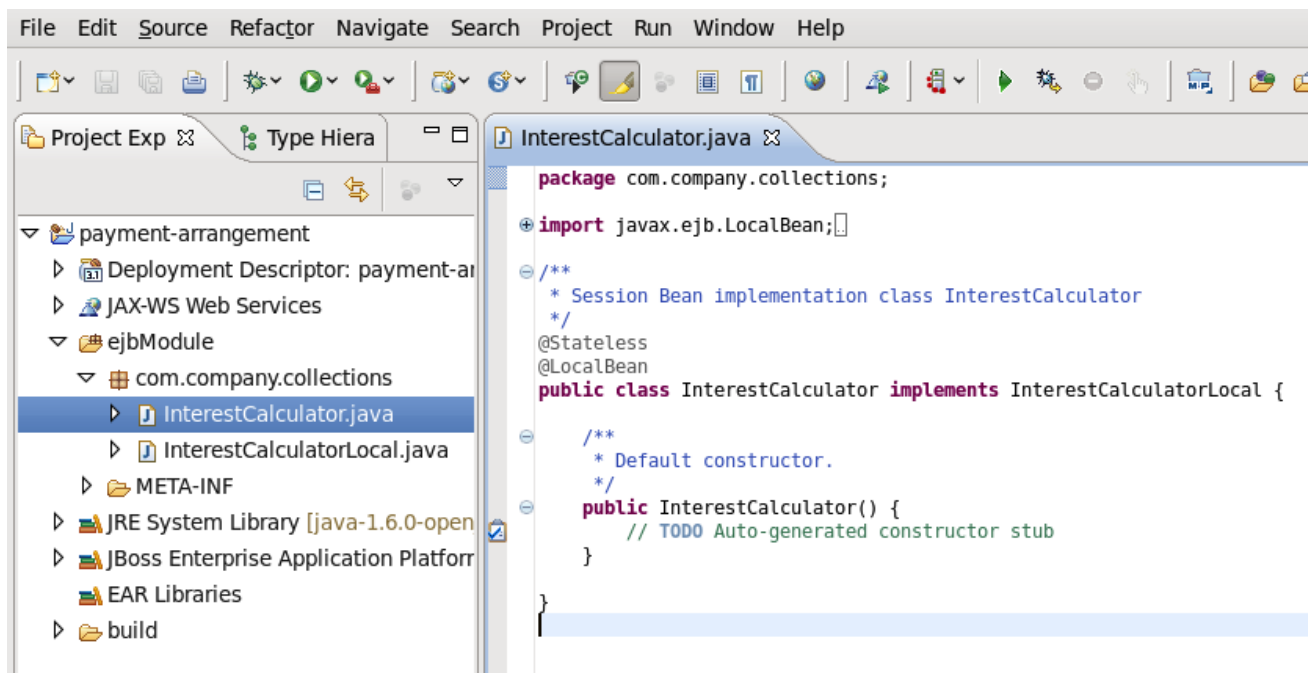


Figure 8.8. New Session Bean in Red Hat JBoss Developer Studio

[Report a bug](#)

8.4. MESSAGE-DRIVEN BEANS

8.4.1. Message-Driven Beans

Message-driven Beans (MDBs) provide an event driven model for application development. The methods of MDBs are not injected into or invoked from client code but are triggered by the receipt of messages from a messaging service such as a Java Messaging Service (JMS) server. The Java EE 6 specification requires that JMS is supported but other messaging systems can be supported as well.

[Report a bug](#)

8.4.2. Resource Adapters

A resource adapter is a deployable Java EE component that provides communication between a Java EE application and an Enterprise Information System (EIS) using the Java Connector Architecture (JCA) specification. A resource adapter is often provided by EIS vendors to allow easy integration of their products with Java EE applications.

An Enterprise Information System can be any other software system within an organization. Examples include Enterprise Resource Planning (ERP) systems, database systems, e-mail servers and proprietary messaging systems.

A resource adapter is packaged in a Resource Adapter Archive (RAR) file which can be deployed to JBoss EAP 6. A RAR file may also be included in an Enterprise Archive (EAR) deployment.

[Report a bug](#)

8.4.3. Create a JMS-based Message-Driven Bean in Red Hat JBoss Developer Studio

This procedure shows how to add a JMS-based Message-Driven Bean to a project in Red Hat JBoss Developer Studio. This procedure creates an EJB 3.x Message-Driven Bean that uses annotations.

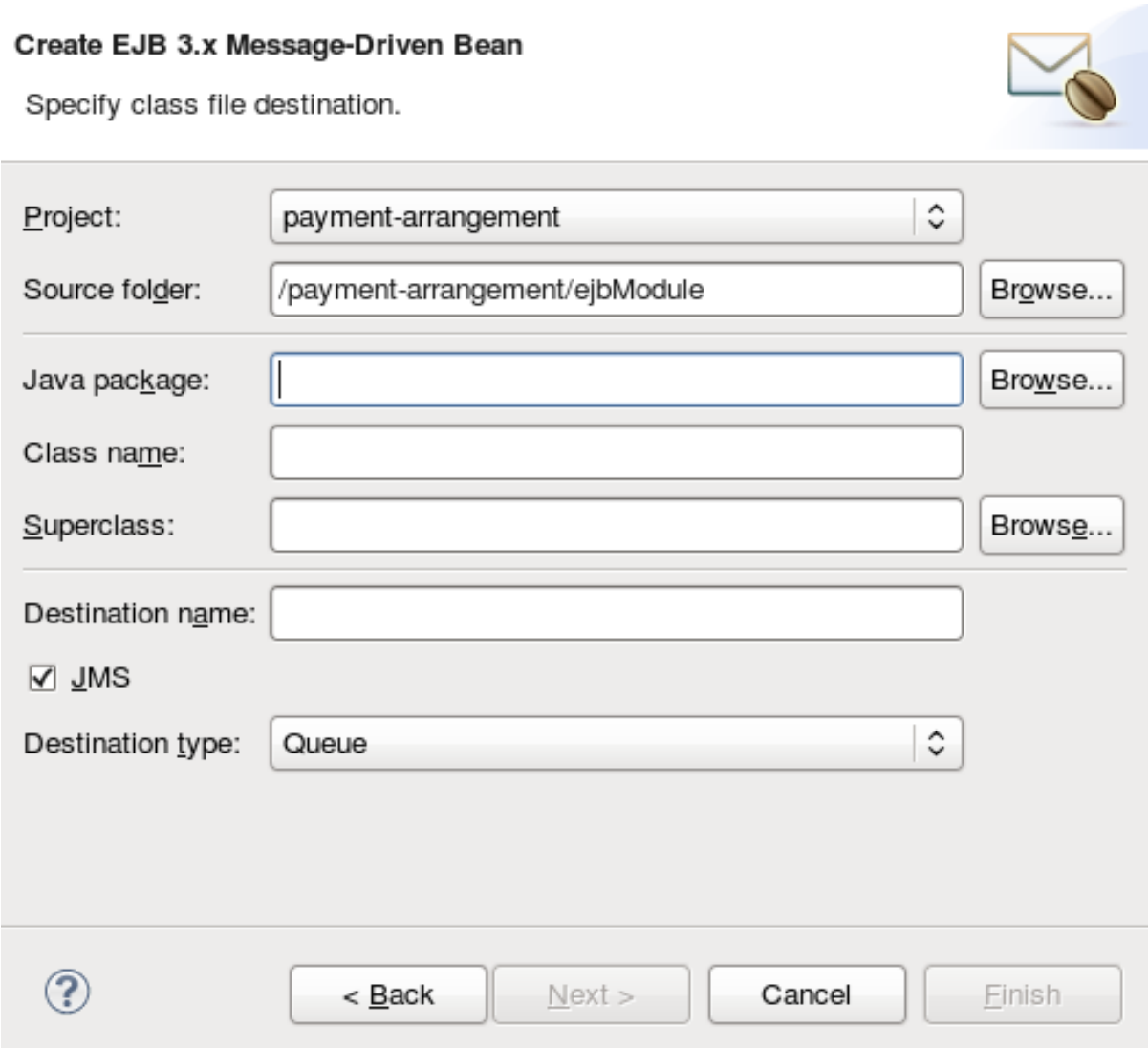
Prerequisites:

1. You must have an existing project open in Red Hat JBoss Developer Studio.
2. You must know the name and type of the JMS destination that the bean will be listening to.
3. Support for Java Messaging Service (JMS) must be enabled in the JBoss EAP 6 configuration to which this bean will be deployed.

Procedure 8.6. Add a JMS-based Message-Driven Bean in Red Hat JBoss Developer Studio

1. Open the Create EJB 3.x Message-Driven Bean Wizard

Go to **File** → **New** → **Other**. Select **EJB/Message-Driven Bean (EJB 3.x)** and click the **Next** button.



Create EJB 3.x Message-Driven Bean

Specify class file destination.

Project: payment-arrangement

Source folder: /payment-arrangement/ejbModule **Browse...**

Java package: **Browse...**

Class name:

Superclass: **Browse...**

Destination name:

☒ **JMS**

Destination type: Queue

< Back **Next >** **Cancel** **Finish**

Figure 8.9. Create EJB 3.x Message-Driven Bean Wizard

2. Specify class file destination details

There are three sets of details to specify for the bean class here: Project, Java class, and message destination.

Project

- If multiple projects exist in the **Workspace**, ensure that the correct one is selected in the **Project** menu.
- The folder where the source file for the new bean will be created is **ejbModule** under the selected project's directory. Only change this if you have a specific requirement.

Java class

- The required fields are: **Java package** and **class name**.
- It is not necessary to supply a **Superclass** unless the business logic of your application requires it.

Message Destination

These are the details you must supply for a JMS-based Message-Driven Bean:

- **Destination name**. This is the queue or topic name that contains the messages that the bean will respond to.
- By default the **JMS** checkbox is selected. Do not change this.
- Set **Destination type** to **Queue** or **Topic** as required.

Click the **Next** button.

3. Enter Message-Driven Bean specific information

The default values here are suitable for a JMS-based Message-Driven bean using Container-managed transactions.

- Change the Transaction type to Bean if the Bean will use Bean-managed transactions.
- Change the Bean name if a different bean name than the class name is required.
- The JMS Message Listener interface will already be listed. You do not need to add or remove any interfaces unless they are specific to your applications business logic.
- Leave the checkboxes for creating method stubs selected.

Click the **Finish** button.

Result: The Message-Driven Bean is created with stub methods for the default constructor and the **onMessage()** method. A Red Hat JBoss Developer Studio editor window opened with the corresponding file.

[Report a bug](#)

8.4.4. Specifying a Resource Adapter in `jboss-ejb3.xml` for an MDB

In the `jboss-ejb3.xml` deployment descriptor you can specify a resource adapter for an MDB to use. Alternatively, to configure a JBoss EAP 6 server-wide default resource adapter for MDBs, see *Configuring Message-Driven Beans* in the *Administration and Configuration Guide*.

To specify a resource adapter in `jboss-ejb3.xml` for an MDB, use the following example.

Example 8.1. `jboss-ejb3.xml` Configuration for an MDB Resource Adapter

```
<jboss xmlns="http://www.jboss.com/xml/ns/javaee"
  xmlns:jee="http://java.sun.com/xml/ns/javaee"
  xmlns:mdb="urn:resource-adapter-binding">
  <jee:assembly-descriptor>
    <mdb:resource-adapter-binding>
      <jee:ejb-name>MyMDB</jee:ejb-name>
      <mdb:resource-adapter-name>MyResourceAdapter.rar</mdb:resource-
adapter-name>
    </mdb:resource-adapter-binding>
  </jee:assembly-descriptor>
</jboss>
```

For a resource adapter located in an EAR, you must use the following syntax for `<mdb:resource-adapter-name>`:

- For a resource adapter that is in another EAR:

```
<mdb:resource-adapter-
name>OtherDeployment.ear#MyResourceAdapter.rar</mdb:resource-
adapter-name>
```

- For a resource adapter that is in the same EAR as the MDB, you can omit the EAR name:

```
<mdb:resource-adapter-name>#MyResourceAdapter.rar</mdb:resource-
adapter-name>
```

[Report a bug](#)

8.4.5. Enable EJB and MDB Property Substitution in an Application

A new feature in Red Hat JBoss Enterprise Application Platform allows you to enable property substitution in EJBs and MDBs using the `@ActivationConfigProperty` and `@Resource` annotations. Property substitution requires the following configuration and code changes.

- You must enable property substitution in the JBoss EAP server configuration file.
- You must define the system properties in the server configuration file or pass them as arguments when you start the JBoss EAP server.
- You must modify the code to use the substitution variables.

Procedure 8.7. Implement Property Substitution in an MDB Application

The following code examples are based on the `helloworld-mdb` quickstart that ships with JBoss EAP 6.3 or later. This topic shows you how to modify that quickstart to enable property substitution.

1. **Configure the JBoss EAP server to enable property substitution.**

The JBoss EAP server must be configured to enable property substitution. To do this, set the **<annotation-property-replacement>** attribute in the **ee** subsystem of the server configuration file to **true**.

- a. Back up the server configuration file. The **helloworld-mdb** quickstart example requires the full profile for a standalone server, so this is the **standalone/configuration/standalone-full.xml** file. If you are running your server in a managed domain, this is the **domain/configuration/domain.xml** file.

- b. Start the JBoss EAP server with the full profile.

For Linux:

```
EAP_HOME/bin/standalone.sh -c standalone-full.xml
```

For Windows:

```
EAP_HOME\bin\standalone.bat -c standalone-full.xml
```

- c. Launch the Management CLI using the command for your operating system.

For Linux:

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

For Windows:

```
EAP_HOME\bin\jboss-cli.bat --connect
```

- d. Type the following command to enable annotation property substitution.

```
/subsystem=ee:write-attribute(name=annotation-property-replacement,value=true)
```

- e. You should see the following result:

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

- f. Review the changes to the JBoss EAP server configuration file. The **ee** subsystem should now contain the following XML.

```
<subsystem xmlns="urn:jboss:domain:ee:1.2">
  <spec-descriptor-property-replacement>false</spec-descriptor-
  property-replacement>
  <jboss-descriptor-property-replacement>true</jboss-
  descriptor-property-replacement>
  <annotation-property-replacement>true</annotation-property-
  replacement>
</subsystem>
```

2. Define the system properties.

You can specify the system properties in the server configuration file or you can pass them as command line arguments when you start the JBoss EAP server. System properties defined in

the server configuration file take precedence over those passed on the command line when you start the server.

- o Define the system properties in the server configuration file.
 - a. Start the JBoss EAP server and Management API as described in the previous step.
 - b. Use the following command syntax to configure a system property in the JBoss EAP server:

```
/system-property=PROPERTY_NAME:add(value=PROPERTY_VALUE)
```

For the **helloworld-mdb** quickstart, we configure the following system properties:

```
/system-  
property=property.helloworldmdb.queue:add(value=java:/queue/HEL  
LOWORLDMDBPropQueue)  
/system-  
property=property.helloworldmdb.topic:add(value=java:/topic/HE  
LOWORLDMDBPropTopic)  
/system-  
property=property.connection.factory:add(value=java:/Connectio  
nFactory)
```

- c. Review the changes to the JBoss EAP server configuration file. The following system properties should now appear in the after the **<extensions>**.

```
<system-properties>  
  <property name="property.helloworldmdb.queue"  
value="java:/queue/HELLOWORLDMDBPropQueue"/>  
  <property name="property.helloworldmdb.topic"  
value="java:/topic/HELLOWORLDMDBPropTopic"/>  
  <property name="property.connection.factory"  
value="java:/ConnectionFactory"/>  
</system-properties>
```

- o Pass the system properties as arguments on the command line when you start the JBoss EAP server in the form of **-DPROPERTY_NAME=PROPERTY_VALUE**. The following is an example of how to pass the arguments for the system properties defined in the previous step.

```
EAP_HOME/bin/standalone.sh -c standalone-full.xml -  
Dproperty.helloworldmdb.queue=java:/queue/HELLOWORLDMDBPropQueue  
-Dproperty.helloworldmdb.topic=java:/topic/HELLOWORLDMDBPropTopic  
-Dproperty.connection.factory=java:/ConnectionFactory
```

3. Modify the code to use the system property substitutions.

Replace hard-coded **@ActivationConfigProperty** and **@Resource** annotation values with substitutions for the newly defined system properties. The following are examples of how to change the **helloworld-mdb** quickstart to use the newly defined system property substitutions within the annotations in the source code.

- a. Change the **@ActivationConfigProperty destination** property value in the **HelloWorldQueueMDB** class to use the substitution for the system property. The **@MessageDriven** annotation should now look like this:

```
@MessageDriven(name = "HelloWorldQueueMDB", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType",
        propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "destination",
        propertyValue = "${property.helloworldmdb.queue}"),
    @ActivationConfigProperty(propertyName = "acknowledgeMode",
        propertyValue = "Auto-acknowledge") })
```

- b. Change the **@ActivationConfigProperty destination** property value in the **HelloWorldTopicMDB** class to use the substitution for the system property. The **@MessageDriven** annotation should now look like this:

```
@MessageDriven(name = "HelloWorldQTopicMDB", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType",
        propertyValue = "javax.jms.Topic"),
    @ActivationConfigProperty(propertyName = "destination",
        propertyValue = "${property.helloworldmdb.topic}"),
    @ActivationConfigProperty(propertyName = "acknowledgeMode",
        propertyValue = "Auto-acknowledge") })
```

- c. Change the **@Resource** annotations in the **HelloWorldMDBServletClient** class to use the system property substitutions. The code should now look like this:

```
@Resource(mappedName = "${property.connection.factory}")
private ConnectionFactory connectionFactory;

@Resource(mappedName = "${property.helloworldmdb.queue}")
private Queue queue;

@Resource(mappedName = "${property.helloworldmdb.topic}")
private Topic topic;
```

- d. Modify the **hornetq-jms.xml** file to use the system property substitution values.

```
<?xml version="1.0" encoding="UTF-8"?>
<messaging-deployment xmlns="urn:jboss:messaging-deployment:1.0">
    <hornetq-server>
        <jms-destinations>
            <jms-queue name="HELLOWORLDMDBQueue">
                <entry name="${property.helloworldmdb.queue}"/>
            </jms-queue>
            <jms-topic name="HELLOWORLDMDBTopic">
                <entry name="${property.helloworldmdb.topic}"/>
            </jms-topic>
        </jms-destinations>
    </hornetq-server>
</messaging-deployment>
```

4. Deploy the application. The application will now use the values specified by the system properties for the **@Resource** and **@ActivationConfigProperty** property values.

[Report a bug](#)

8.5. INVOKING SESSION BEANS

8.5.1. Invoke a Session Bean Remotely using JNDI

This task describes how to add support to a remote client for the invocation of session beans using JNDI. The task assumes that the project is being built using Maven.

The **ejb-remote** quickstart contains working Maven projects that demonstrate this functionality. The quickstart contains projects for both the session beans to deploy and the remote client. The code samples below are taken from the remote client project.

This task assumes that the session beans do not require authentication.



WARNING

Red Hat recommends that you explicitly disable SSL in favor of TLSv1.1 or TLSv1.2 in all affected packages.

Prerequisites

The following prerequisites must be satisfied before beginning:

- You must already have a Maven project created ready to use.
- Configuration for the JBoss EAP 6 Maven repository has already been added.
- The session beans that you want to invoke are already deployed.
- The deployed session beans implement remote business interfaces.
- The remote business interfaces of the session beans are available as a Maven dependency. If the remote business interfaces are only available as a JAR file then it is recommended to add the JAR to your Maven repository as an artifact. Refer to the Maven documentation for the **install:install-file** goal for directions, <http://maven.apache.org/plugins/maven-install-plugin/usage.html>
- You need to know the hostname and JNDI port of the server hosting the session beans.

To invoke a session bean from a remote client you must first configure the project correctly.

Procedure 8.8. Add Maven Project Configuration for Remote Invocation of Session Beans

1. Add the required project dependencies

The **pom.xml** for the project must be updated to include the necessary dependencies.

2. Add the **jboss-ejb-client.properties** file

The JBoss EJB client API expects to find a file in the root of the project named **jboss-ejb-client.properties** that contains the connection information for the JNDI service. Add this file to the **src/main/resources/** directory of your project with the following content.

```
# In the following line, set SSL_ENABLED to true for SSL
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
remote.connections=default
# Uncomment the following line to set SSL_STARTTLS to true for SSL
#
remote.connection.default.connect.options.org.xnio.Options.SSL_STARTTLS=true
remote.connection.default.host=localhost
remote.connection.default.port = 4447
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
# Add any of the following SASL options if required
#
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
#
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOPLAINTEXT=false
#
remote.connection.default.connect.options.org.xnio.Options.SASL_DISALLOWED_MECHANISMS=JBOSS-LOCAL-USER
```

Change the host name and port to match your server. **4447** is the default port number. For a secure connection, set the **SSL_ENABLED** line to **true** and uncomment the **SSL_STARTTLS** line. The Remoting interface in the container supports secured and unsecured connections using the same port.

3. Add dependencies for the remote business interfaces

Add the Maven dependencies to the **pom.xml** for the remote business interfaces of the session beans.

```
<dependency>
  <groupId>org.jboss.as.quickstarts</groupId>
  <artifactId>jboss-ejb-remote-server-side</artifactId>
  <type>ejb-client</type>
  <version>${project.version}</version>
</dependency>
```

Now that the project has been configured correctly, you can add the code to access and invoke the session beans.

Procedure 8.9. Obtain a Bean Proxy using JNDI and Invoke Methods of the Bean

1. Handle checked exceptions

Two of the methods used in the following code (**InitialContext()** and **lookup()**) have a checked exception of type **javax.naming.NamingException**. These method calls must either be enclosed in a try/catch block that catches **NamingException** or in a method that is declared to throw **NamingException**. The **ejb-remote** quickstart uses the second technique.

2. Create a JNDI Context

A JNDI Context object provides the mechanism for requesting resources from the server. Create a JNDI context using the following code:

```
final Hashtable jndiProperties = new Hashtable();
jndiProperties.put(Context.URL_PKG_PREFIXES,
    "org.jboss.ejb.client.naming");
final Context context = new InitialContext(jndiProperties);
```

The connection properties for the JNDI service are read from the **jboss-ejb-client.properties** file.

3. Use the JNDI Context's lookup() method to obtain a bean proxy

Invoke the **lookup()** method of the bean proxy and pass it the JNDI name of the session bean you require. This will return an object that must be cast to the type of the remote business interface that contains the methods you want to invoke.

```
final RemoteCalculator statelessRemoteCalculator =
    (RemoteCalculator) context.lookup(
        "ejb://jboss-ejb-remote-server-side//CalculatorBean!" +
        RemoteCalculator.class.getName());
```

Session bean JNDI names are defined using a special syntax. For more information, see [Section 8.8.1, “EJB JNDI Naming Reference”](#).

4. Invoke methods

Now that you have a proxy bean object you can invoke any of the methods contained in the remote business interface.

```
int a = 204;
int b = 340;
System.out.println("Adding " + a + " and " + b + " via the remote
    stateless calculator deployed on the server");
int sum = statelessRemoteCalculator.add(a, b);
System.out.println("Remote calculator returned sum = " + sum);
```

The proxy bean passes the method invocation request to the session bean on the server, where it is executed. The result is returned to the proxy bean which then returns it to the caller. The communication between the proxy bean and the remote session bean is transparent to the caller.

You should now be able to configure a Maven project to support invoking session beans on a remote server and write the code invoke the session beans methods using a proxy bean retrieved from the server using JNDI.

[Report a bug](#)

8.5.2. About EJB Client Contexts

JBoss EAP 6 introduced the EJB client API for managing remote EJB invocations. The JBoss EJB client API uses the `EJBClientContext`, which may be associated with and be used by one or more threads concurrently. This means an `EJBClientContext` can potentially contain any number of EJB receivers. An

EJB receiver is a component that knows how to communicate with a server that is capable of handling the EJB invocation. Typically, EJB remote applications can be classified into the following:

- A remote client, which runs as a standalone Java application.
- A remote client, which runs within another JBoss EAP 6 instance.

Depending on the type of remote client, from an EJB client API point of view, there can potentially be more than one `EJBClientContext` within a JVM.

While standalone applications typically have a single `EJBClientContext` that may be backed by any number of EJB receivers, this isn't mandatory. If a standalone application has more than one `EJBClientContext`, an EJB client context selector is responsible for returning the appropriate context.

In case of remote clients that run within another JBoss EAP 6 instance, each deployed application will have a corresponding EJB client context. Whenever that application invokes another EJB, the corresponding EJB client context is used to find the correct EJB receiver, which then handles the invocation.

[Report a bug](#)

8.5.3. Considerations When Using a Single EJB Context

Summary

You must consider your application requirements when using a single EJB client context with standalone remote clients. For more information about the different types of remote clients, refer to: [Section 8.5.2, “About EJB Client Contexts”](#).

Typical Process for a Remote Standalone Client with a Single EJB Client Context

A remote standalone client typically has just one EJB client context backed by any number of EJB receivers. The following is an example of a standalone remote client application:

```
public class MyApplication {
    public static void main(String args[]) {
        final javax.naming.Context ctxOne = new
javax.naming.InitialContext();
        final MyBeanInterface beanOne =
ctxOne.lookup("ejb:app/module/distinct/bean!interface");
        beanOne.doSomething();
        ...
    }
}
```

Remote client JNDI lookups are usually backed by a `jboss-ejb-client.properties` file, which is used to set up the EJB client context and the EJB receivers. This configuration also includes the security credentials, which are then used to create the EJB receiver that connects to the JBoss EAP 6 server. When the above code is invoked, the EJB client API looks for the EJB client context, which is then used to select the EJB receiver that will receive and process the EJB invocation request. In this case, there is just the single EJB client context, so that context is used by the above code to invoke the bean. The procedure to invoke a session bean remotely using JNDI is described in greater detail here: [Section 8.5.1, “Invoke a Session Bean Remotely using JNDI”](#).

Remote Standalone Client Requiring Different Credentials

A user application may want to invoke a bean more than once, but connect to the JBoss EAP 6 server using different security credentials. The following is an example of a standalone remote client application that invokes the same bean twice:

```
public class MyApplication {
    public static void main(String args[]) {
        // Use the "foo" security credential connect to the server and
        // invoke this bean instance
        final javax.naming.Context ctxOne = new
        javax.naming.InitialContext();
        final MyBeanInterface beanOne =
        ctxOne.lookup("ejb:app/module/distinct/bean!interface");
        beanOne.doSomething();
        ...

        // Use the "bar" security credential to connect to the server and
        // invoke this bean instance
        final javax.naming.Context ctxTwo = new
        javax.naming.InitialContext();
        final MyBeanInterface beanTwo =
        ctxTwo.lookup("ejb:app/module/distinct/bean!interface");
        beanTwo.doSomething();
        ...
    }
}
```

In this case, the application wants to connect to the same server instance to invoke the EJB hosted on that server, but wants to use two different credentials while connecting to the server. Because the client application has a single EJB client context, which can have only one EJB receiver for each server instance, this means the above code uses just one credential to connect to the server and the code does not execute as the application expects it to.

Solution

Scoped EJB client contexts offer a solution to this issue. They provide a way to have more control over the EJB client contexts and their associated JNDI contexts, which are typically used for EJB invocations. For more information about scoped EJB client contexts, refer to [Section 8.5.4, “Using Scoped EJB Client Contexts”](#) and [Section 8.5.5, “Configure EJBs Using a Scoped EJB Client Context”](#).

[Report a bug](#)

8.5.4. Using Scoped EJB Client Contexts

Summary

To invoke an EJB In earlier versions of JBoss EAP 6, you would typically create a JNDI context and pass it the PROVIDER_URL, which would point to the target server. Any invocations done on EJB proxies that were looked up using that JNDI context, would end up on that server. With scoped EJB client contexts, user applications have control over which EJB receiver is used for a specific invocation.

Use Scoped EJB Client Context in a Remote Standalone Client

Prior to the introduction of scoped EJB client contexts, the context was typically scoped to the client application. Scoped client contexts now allow the EJB client contexts to be scoped with the JNDI contexts. The following is an example of a standalone remote client application that invokes the same bean twice using a scoped EJB client context:

```

public class MyApplication {
    public static void main(String args[]) {

        // Use the "foo" security credential connect to the server and
        // invoke this bean instance
        final Properties ejbClientContextPropsOne =
        getPropsForEJBClientContextOne():
        final javax.naming.Context ctxOne = new
        javax.naming.InitialContext(ejbClientContextPropsOne);
        final MyBeanInterface beanOne =
        ctxOne.lookup("ejb:app/module/distinct/bean!interface");
        beanOne.doSomething();
        ...
        ctxOne.close();

        // Use the "bar" security credential to connect to the server and
        // invoke this bean instance
        final Properties ejbClientContextPropsTwo =
        getPropsForEJBClientContextTwo():
        final javax.naming.Context ctxTwo = new
        javax.naming.InitialContext(ejbClientContextPropsTwo);
        final MyBeanInterface beanTwo =
        ctxTwo.lookup("ejb:app/module/distinct/bean!interface");
        beanTwo.doSomething();
        ...
        ctxTwo.close();
    }
}

```

To use the scoped EJB client context, you configure EJB client properties programmatically and pass the properties on context creation. The properties are the same set of properties that are used in the standard **jboss-ejb-client.properties** file. To scope the EJB client context to the JNDI context, you must also specify the **org.jboss.ejb.client.scoped.context** property and set its value to **true**. This property notifies the EJB client API that it must create an EJB client context, which is backed by EJB receivers, and that the created context is then scoped or visible only to the JNDI context that created it. Any EJB proxies looked up or invoked using this JNDI context will only know of the EJB client context associated with this JNDI context. Other JNDI contexts used by the application to lookup and invoke EJBs will not know about the other scoped EJB client contexts.

JNDI contexts that do not pass the **org.jboss.ejb.client.scoped.context** property and aren't scoped to an EJB client context will use the default behavior, which is to use the existing EJB client context that is typically tied to the entire application.

Scoped EJB client contexts provide user applications with the flexibility that was associated with the JNP based JNDI invocations in previous versions of JBoss EAP. It provides user applications with more control over which JNDI context communicates to which server and how it connects to that server.



NOTE

With the scoped context, the underlying resources are no longer handled by the container or the API, so you must close the **InitialContext** when it is no longer needed. When the **InitialContext** is closed, the resources are released immediately. The proxies that are bound to it are no longer valid and any invocation will throw an Exception. Failure to close the **InitialContext** may result in resource and performance issues.

[Report a bug](#)

8.5.5. Configure EJBs Using a Scoped EJB Client Context

Summary

EJBs can be configured using a map-based scoped context. This is achieved by programmatically populating a **Properties** map using the standard properties found in the **jboss-ejb-client.properties**, specifying **true** for the **org.jboss.ejb.client.scoped.context** property, and passing the properties on the **InitialContext** creation.

The benefit of using a scoped context is that it allows you to configure access without directly referencing the EJB or importing JBoss classes. It also provides a way to configure and load balance a host at runtime in a multithreaded environment.

Procedure 8.10. Configure an EJB Using a Map-Based Scoped Context

1. Set the Properties

Configure the EJB client properties programmatically, specifying the same set of properties that are used in the standard **jboss-ejb-client.properties** file. To enable the scoped context, you must specify the **org.jboss.ejb.client.scoped.context** property and set its value to **true**. The following is an example that configures the properties programmatically.

```
// Configure EJB Client properties for the InitialContext
Properties ejbClientContextProps = new Properties();
ejbClientContextProps.put("remote.connections", "name1");
ejbClientContextProps.put("remote.connection.name1.host", "localhost"
);
ejbClientContextProps.put("remote.connection.name1.port", "4447");
// Property to enable scoped EJB client context which will be tied
to the JNDI context
ejbClientContextProps.put("org.jboss.ejb.client.scoped.context",
"true");
```

2. Pass the Properties on the Context Creation

```
// Create the context using the configured properties
InitialContext ic = new InitialContext(ejbClientContextProps);
MySLSB bean = ic.lookup("ejb:myapp/ejb//MySLSBBean!" +
MySLSB.class.getName());
```

Additional Information

- Contexts generated by lookup EJB proxies are bound by this scoped context and use only the relevant connection parameters. This makes it possible to create different contexts to access data within a client application or to independently access servers using different logins.
- In the client, both the scoped **InitialContext** and the scoped proxy are passed to threads, allowing each thread to work with the given context. It is also possible to pass the proxy to multiple threads that can use it concurrently.
- The scoped context EJB proxy is serialized on the remote call and then deserialized on the server. When it is deserialized, the scoped context information is removed and it returns to its default state. If the deserialized proxy is used on the remote server, because it no longer has the

scoped context that was used when it was created, this can result in an **EJBCLIENT000025** error or possibly call an unwanted target by using the EJB name.

[Report a bug](#)

8.5.6. EJB Client Properties


Summary

The following tables list properties that can be configured programmatically or in the **jboss-ejb-client.properties** file.

EJB Client Global Properties

The following table lists properties that are valid for the whole library within the same scope.

Table 8.1. Global Properties

Property Name	Description
endpoint.name	<p>Name of the client endpoint. If not set, the default value is client-endpoint</p> <p>This can be helpful to distinguish different endpoint settings because the thread name contains this property.</p>
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED	<p>Boolean value that specifies whether the SSL protocol is enabled for all connections.</p> <div>  <p>WARNING</p> <p>Red Hat recommends that you explicitly disable SSL in favor of TLSv1.1 or TLSv1.2 in all affected packages.</p> </div>
deployment.node.selector	<p>The fully qualified name of the implementation of org.jboss.ejb.client.DeploymentNodesElector.</p> <p>This is used to load balance the invocation for the EJBs.</p>
invocation.timeout	<p>The timeout for the EJB handshake or method invocation request/response cycle. The value is in milliseconds.</p> <p>The invocation of any method throws a java.util.concurrent.TimeoutException if the execution takes longer than the timeout period. The execution completes and the server is not interrupted.</p>

Property Name	Description
<code>reconnect.tasks.timeout</code>	<p>The timeout for the background reconnect tasks. The value is in milliseconds.</p> <p>If a number of connections are down, the next client EJB invocation will use an algorithm to decide if a reconnect is necessary to find the right node.</p>
<code>org.jboss.ejb.client.scoped.context</code>	<p>Boolean value that specifies whether to enable the scoped EJB client context. The default value is false.</p> <p>If set to true, the EJB Client will use the scoped context that is tied to the JNDI context. Otherwise the EJB client context will use the global selector in the JVM to determine the properties used to call the remote EJB and host.</p>

EJB Client Connection Properties

The connection properties start with the prefix `remote.connection.CONNECTION_NAME` where the `CONNECTION_NAME` is a local identifier only used to uniquely identify the connection.

Table 8.2. Connection Properties

Property Name	Description
<code>remote.connections</code>	A comma-separated list of active connection-names . Each connection is configured by using this name.
<code>remote.connection.CONNECTION_NAME.host</code>	The host name or IP for the connection.
<code>remote.connection.CONNECTION_NAME.port</code>	The port for the connection. The default value is 4447.
<code>remote.connection.CONNECTION_NAME.username</code>	The user name used to authenticate connection security.
<code>remote.connection.CONNECTION_NAME.password</code>	The password used to authenticate the user.
<code>remote.connection.CONNECTION_NAME.connect.timeout</code>	The timeout period for the initial connection. After that, the reconnect task will periodically check whether the connection can be established. The value is in milliseconds.
<code>remote.connection.CONNECTION_NAME.callback.handler.class</code>	Fully qualified name of the CallbackHandler class. It will be used to establish the connection and can not be changed as long as the connection is open.

Property Name	Description
<code>remote.connection.CONNECTION_NAME.channel.options.org.jboss.remoting3.RemotingOptions.MAX_OUTBOUND_MESSAGES</code>	<p>Integer value specifying the maximum number of outbound requests. The default is 80.</p> <p>There is only one connection from the client (JVM) to the server to handle all invocations.</p>
<code>remote.connection.CONNECTION_NAME.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS</code>	<p>Boolean value that determines whether credentials must be provided by the client to connect successfully. The default value is true.</p> <p>If set to true, the client must provide credentials. If set to false, invocation is allowed as long as the remoting connector does not request a security realm.</p>
<code>remote.connection.CONNECTION_NAME.connect.options.org.xnio.Options.SASL_DISALLOWED_MECHANISMS</code>	<p>Disables certain SASL mechanisms used for authenticating during connection creation.</p> <p>JBoss-LOCAL-USER means the silent authentication mechanism, used when the client and server are on the same machine, is disabled.</p>
<code>remote.connection.CONNECTION_NAME.connect.options.org.xnio.Options.SASL_POLICY_NOPLAINTEXT</code>	<p>Boolean value that enables or disables the use of plain text messages during the authentication. If using JAAS, it must be set to false to allow a plain text password.</p>
<code>remote.connection.CONNECTION_NAME.connect.options.org.xnio.Options.SSL_ENABLED</code>	<p>Boolean value that specifies whether the SSL protocol is enabled for this connection.</p> <div>  <p>WARNING</p> <p>Red Hat recommends that you explicitly disable SSL in favor of TLSv1.1 or TLSv1.2 in all affected packages.</p> </div>
<code>remote.connection.CONNECTION_NAME.connect.options.org.jboss.remoting3.RemotingOptions.HEARTBEAT_INTERVAL</code>	<p>Interval to send a heartbeat between client and server to prevent automatic close, for example, in the case of a firewall. The value is in milliseconds.</p>

EJB Client Cluster Properties

If the initial connection connects to a clustered environment, the topology of the cluster is received automatically and asynchronously. These properties are used to connect to each received member. Each property starts with the prefix `remote.cluster.CLUSTER_NAME` where the `CLUSTER_NAME` refers to the related to the servers Infinispan subsystem configuration.

Table 8.3. Cluster Properties

Property Name	Description
<code>remote.cluster.CLUSTER_NAME.clusterNode.selector</code>	The fully qualified name of the implementation of <code>org.jboss.ejb.client.ClusterNodeSelector</code> . This class, rather than <code>org.jboss.ejb.client.DeploymentNodesSelector</code> , is used to load balance EJB invocations in a clustered environment. If the cluster is completely down, the invocation will fail with No ejb receiver available .
<code>remote.cluster.CLUSTER_NAME.channel.options.org.jboss.remoting3.RemotingOptions.MAX_OUTBOUND_MESSAGES</code>	Integer value specifying the maximum number of outbound requests that can be made to the entire cluster.
<code>remote.cluster.CLUSTER_NAME.node.NO_DE_NAME.channel.options.org.jboss.remoting3.RemotingOptions.MAX_OUTBOUND_MESSAGES</code>	Integer value specifying the maximum number of outbound requests that can be made to this specific cluster-node.

[Report a bug](#)

8.5.7. Remote EJB Data Compression

Previous versions of JBoss EAP included a feature where the message stream that contained the EJB protocol message could be compressed. This feature has been included in JBoss EAP 6.3 and later.



NOTE

Compression currently can only be specified by annotations on the EJB interface which should be on the client and server side. There is not currently an XML equivalent to specify compression hints.

Data compression hints can be specified via the JBoss annotation `org.jboss.ejb.client.annotation.CompressionHint`. The hint values specify whether to compress the request, response or request and response. Adding `@CompressionHint` defaults to `compressResponse=true` and `compressRequest=true`.

The annotation can be specified at the interface level to apply to all methods in the EJB's interface such as:

```
import org.jboss.ejb.client.annotation.CompressionHint;

@CompressionHint(compressResponse = false)
public interface ClassLevelRequestCompressionRemoteView {
    String echo(String msg);
}
```

Or the annotation can be applied to specific methods in the EJB's interface such as:

-

```
import org.jboss.ejb.client.annotation.CompressionHint;

public interface CompressableDataRemoteView {

    @CompressionHint(compressResponse = false, compressionLevel =
Deflater.BEST_COMPRESSION)
    String echoWithRequestCompress(String msg);

    @CompressionHint(compressRequest = false)
    String echoWithResponseCompress(String msg);

    @CompressionHint
    String echoWithRequestAndResponseCompress(String msg);

    String echoWithNoCompress(String msg);
}
```

The **compressionLevel** setting shown above can have the following values:

- BEST_COMPRESSION
- BEST_SPEED
- DEFAULT_COMPRESSION
- NO_COMPRESSION

The **compressionLevel** setting defaults to **Deflater.DEFAULT_COMPRESSION**.

Class level annotation with method level overrides:

```
@CompressionHint
public interface MethodOverrideDataCompressionRemoteView {

    @CompressionHint(compressRequest = false)
    String echoWithResponseCompress(final String msg);

    @CompressionHint(compressResponse = false)
    String echoWithRequestCompress(final String msg);

    String echoWithNoExplicitDataCompressionHintOnMethod(String msg);
}
```

On the client side ensure the ***org.jboss.ejb.client.view.annotation.scan.enabled*** system property is set to **true**. This property tells JBoss EJB Client to scan for annotations.

[Report a bug](#)

8.6. CONTAINER INTERCEPTORS

8.6.1. About Container Interceptors

Standard Java EE interceptors, as defined by the [JSR 318](#), [Enterprise JavaBeans 3.1](#) specification, are expected to run after the container has completed security context propagation, transaction

management, and other container provided invocation processing. This is a problem if the application must intercept a call before a specific container interceptor is run.

Releases prior to JBoss EAP 6.0 provided a way to plug server side interceptors into the invocation flow so you could run specific application logic before the container completed the invocation processing. This feature was implemented in JBoss EAP 6.1. This implementation allows standard Java EE interceptors to be used as container interceptors, meaning they use the same XSD elements that are allowed in `ejb-jar.xml` file for the 3.1 version of the `ejb-jar` deployment descriptor.

Positioning of the Container Interceptor in the Interceptor Chain

The container interceptors configured for an EJB are guaranteed to be run before the JBoss EAP provided security interceptors, transaction management interceptors, and other server provided interceptors. This allows specific application container interceptors to process or configure relevant context data before the invocation proceeds.

Differences Between the Container Interceptor and the Java EE Interceptor API

Although container interceptors are modeled to be similar to Java EE interceptors, there are some differences in the semantics of the API. For example, it is illegal for container interceptors to invoke the `javax.interceptor.InvocationContext.getTarget()` method because these interceptors are invoked long before the EJB components are setup or instantiated.

[Report a bug](#)

8.6.2. Create a Container Interceptor Class

Summary

Container interceptor classes are simple Plain Old Java Objects (POJOs). They use the `@javax.annotation.AroundInvoke` to mark the method that is invoked during the invocation on the bean.

The following is an example of a container interceptor class that marks the `iAmAround` method for invocation:

Example 8.2. Container Interceptor Class Example

```
public class ClassLevelContainerInterceptor {
    @AroundInvoke
    private Object iAmAround(final InvocationContext invocationContext)
    throws Exception {
        return this.getClass().getName() + " " +
        invocationContext.proceed();
    }
}
```

For an example of a container interceptor descriptor file configured to use this class, see the `jboss-ejb3.xml` file described here: [Section 8.6.3, “Configure a Container Interceptor”](#).

[Report a bug](#)

8.6.3. Configure a Container Interceptor

Summary

Container interceptors use the standard Java EE interceptor libraries, meaning they use the same XSD elements that are allowed in `ejb-jar.xml` file for the 3.1 version of the `ejb-jar` deployment descriptor. Because they are based on the standard Java EE interceptor libraries, container interceptors may only be configured using deployment descriptors. This was done by design so applications would not require any JBoss specific annotation or other library dependencies. For more information about container interceptors, refer to: [Section 8.6.1, “About Container Interceptors”](#).

Procedure 8.11. Create the Descriptor File to Configure the Container Interceptor

1. Create a `jboss-ejb3.xml` file in the **META-INF** directory of the EJB deployment.
2. Configure the container interceptor elements in the descriptor file.
 - a. Use the `urn:container-interceptors:1.0` namespace to specify configuration of container interceptor elements.
 - b. Use the `<container-interceptors>` element to specify the container interceptors.
 - c. Use the `<interceptor-binding>` elements to bind the container interceptor to the EJBs. The interceptors can be bound in either of the following ways:
 - Bind the interceptor to all the EJBs in the deployment using the `*` wildcard.
 - Bind the interceptor at the individual bean level using the specific EJB name.
 - Bind the interceptor at the specific method level for the EJBs.



NOTE

These elements are configured using the EJB 3.1 XSD in the same way it is done for Java EE interceptors.

3. Review the following descriptor file for examples of the above elements.

Example 8.3. `jboss-ejb3.xml`

```
<jboss xmlns="http://www.jboss.com/xml/ns/javaee"
      xmlns:jee="http://java.sun.com/xml/ns/javaee"
      xmlns:ci="urn:container-interceptors:1.0">

  <jee:assembly-descriptor>
    <ci:container-interceptors>
      <!-- Default interceptor -->
      <jee:interceptor-binding>
        <ejb-name>*</ejb-name>
        <interceptor-
class>org.jboss.as.test.integration.ejb.container.interceptor.Cont
ainerInterceptorOne</interceptor-class>
      </jee:interceptor-binding>
      <!-- Class level container-interceptor -->
      <jee:interceptor-binding>
        <ejb-name>AnotherFlowTrackingBean</ejb-name>
        <interceptor-
```



```

class>org.jboss.as.test.integration.ejb.container.interceptor.ClassLevelContainerInterceptor</interceptor-class>
  </jee:interceptor-binding>
  <!-- Method specific container-interceptor -->
  <jee:interceptor-binding>
    <ejb-name>AnotherFlowTrackingBean</ejb-name>
    <interceptor-
class>org.jboss.as.test.integration.ejb.container.interceptor.MethodSpecificContainerInterceptor</interceptor-class>
    <method>
      <method-
name>echoWithMethodSpecificContainerInterceptor</method-name>
    </method>
  </jee:interceptor-binding>
  <!-- container interceptors in a specific order -->
  <jee:interceptor-binding>
    <ejb-name>AnotherFlowTrackingBean</ejb-name>
    <interceptor-order>
      <interceptor-
class>org.jboss.as.test.integration.ejb.container.interceptor.ClassLevelContainerInterceptor</interceptor-class>
      <interceptor-
class>org.jboss.as.test.integration.ejb.container.interceptor.MethodSpecificContainerInterceptor</interceptor-class>
      <interceptor-
class>org.jboss.as.test.integration.ejb.container.interceptor.ContainerInterceptorOne</interceptor-class>
    </interceptor-order>
    <method>
      <method-
name>echoInSpecificOrderOfContainerInterceptors</method-name>
    </method>
  </jee:interceptor-binding>
</ci:container-interceptors>
</jee:assembly-descriptor>
</jboss>

```

The XSD for the **urn:container-interceptors:1.0** namespace is available at **EAP_HOME/docs/schema/jboss-ejb-container-interceptors_1_0.xsd**.

[Report a bug](#)

8.6.4. Change the Security Context Identity

Summary

By default, when you make a remote call to an EJB deployed to the application server, the connection to the server is authenticated and any request received over this connection is executed as the identity that authenticated the connection. This is true for both client-to-server and server-to-server calls. If you need to use different identities from the same client, you normally need to open multiple connections to the server so that each one is authenticated as a different identity. Rather than open multiple client connections, you can give permission to the authenticated user to execute a request as a different user.

This topic describes how to switch identities on the existing client connection. The code examples are abridged versions of the code in the quickstart. Refer to the **ejb-security-interceptors** quickstart for a complete working example.

Procedure 8.12. Change the Identity of the Security Context

To change the identity of a secured connection, you must create the following 3 components.

1. Create the client side interceptor

The client side interceptor must implement the **org.jboss.ejb.client.EJBClientInterceptor** interface. The interceptor must pass the requested identity through the context data map, which can be obtained via a call to **EJBClientInvocationContext.getContextData()**. The following is an example of client side interceptor code:

```
public class ClientSecurityInterceptor implements
EJBClientInterceptor {

    public void handleInvocation(EJBClientInvocationContext context)
throws Exception {
        Principal currentPrincipal =
SecurityActions.securityContextGetPrincipal();

        if (currentPrincipal != null) {
            Map<String, Object> contextData =
context.getContextData();

            contextData.put(ServerSecurityInterceptor.DELEGATED_USER_KEY,
currentPrincipal.getName());
        }
        context.sendRequest();
    }

    public Object handleInvocationResult(EJBClientInvocationContext
context) throws Exception {
        return context.getResult();
    }
}
```

User applications can insert the interceptor into the interceptor chain in the **EJBClientContext** in one of the following ways:

o Programmatically

With this approach, you call the

org.jboss.ejb.client.EJBClientContext.registerInterceptor(int order, EJBClientInterceptor interceptor) method and pass the **order** and the **interceptor** instance. The **order** determines where this client interceptor is placed in the interceptor chain.

o ServiceLoader Mechanism

With this approach, you create a **META-INF/services/org.jboss.ejb.client.EJBClientInterceptor** file and place or

package it in the classpath of the client application. The rules for the file are dictated by the [Java ServiceLoader Mechanism](#). This file is expected to contain a separate line for each fully qualified class name of the EJB client interceptor implementation. The EJB client

interceptor classes must be available in the classpath. EJB client interceptors added using the **ServiceLoader** mechanism are added to the end of the client interceptor chain, in the order they are found in the classpath. The **ejb-security-interceptors** quickstart uses this approach.

2. Create and configure the server side container interceptor

Container interceptor classes are simple Plain Old Java Objects (POJOs). They use the **@javax.annotation.AroundInvoke** to mark the method that will be invoked during the invocation on the bean. For more information about container interceptors, refer to:

[Section 8.6.1, “About Container Interceptors”](#).

a. Create the container interceptor

This interceptor receives the **InvocationContext** with the identity and requests the switch to that new identity. The following is an abridged version of the actual code example:

```
public class ServerSecurityInterceptor {

    private static final Logger logger =
        Logger.getLogger(ServerSecurityInterceptor.class);

    static final String DELEGATED_USER_KEY =
        ServerSecurityInterceptor.class.getName() + ".DelegationUser";

    @AroundInvoke
    public Object aroundInvoke(final InvocationContext
        invocationContext) throws Exception {
        Principal desiredUser = null;
        UserPrincipal connectionUser = null;

        Map<String, Object> contextData =
            invocationContext.getContextData();
        if (contextData.containsKey(DELEGATED_USER_KEY)) {
            desiredUser = new SimplePrincipal((String)
                contextData.get(DELEGATED_USER_KEY));

            Collection<Principal> connectionPrincipals =
                SecurityActions.getConnectionPrincipals();

            if (connectionPrincipals != null) {
                for (Principal current : connectionPrincipals) {
                    if (current instanceof UserPrincipal) {
                        connectionUser = (UserPrincipal)
                            current;
                        break;
                    }
                }
            } else {
                throw new IllegalStateException("Delegation user
                    requested but no user on connection found.");
            }
        }

        ContextStateCache stateCache = null;
        try {
```

```

        if (desiredUser != null && connectionUser != null
            &&
            (desiredUser.getName().equals(connectionUser.getName()) ==
false)) {
            // The final part of this check is to verify
            that the change does actually indicate a change in user.
            try {
                // We have been requested to use an
authentication token
                // so now we attempt the switch.
                stateCache =
SecurityActions.pushIdentity(desiredUser, new
OuterUserCredential(connectionUser));
            } catch (Exception e) {
                logger.error("Failed to switch security
context for user", e);
                // Don't propagate the exception stacktrace
back to the client for security reasons
                throw new EJBAccessException("Unable to
attempt switching of user.");
            }
        }

        return invocationContext.proceed();
    } finally {
        // switch back to original context
        if (stateCache != null) {
            SecurityActions.popIdentity(stateCache);
        }
    }
}

```

b. Configure the container interceptor

For information on how to configure server side container interceptors, refer to:
[Section 8.6.3, “Configure a Container Interceptor”](#).

3. Create the JAAS LoginModule

This component is responsible for verifying that user is allowed to execute requests as the requested identity. The following abridged code examples show the methods that perform the login and validation:

```

@SuppressWarnings("unchecked")
@Override
public boolean login() throws LoginException {
    if (super.login() == true) {
        log.debug("super.login()==true");
        return true;
    }

    // Time to see if this is a delegation request.
    NameCallback ncb = new NameCallback("Username:");
    ObjectCallback ocb = new ObjectCallback("Password:");

    try {
        callbackHandler.handle(new Callback[] { ncb, ocb });
    } catch (Exception e) {

```

```

        if (e instanceof RuntimeException) {
            throw (RuntimeException) e;
        }
        return false; // If the CallbackHandler can not handle
the required callbacks then no chance.
    }

    String name = ncb.getName();
    Object credential = ocb.getCredential();

    if (credential instanceof OuterUserCredential) {
        // This credential type will only be seen for a
delegation request, if not seen then the request is not for us.

        if (delegationAcceptable(name, (OuterUserCredential)
credential)) {

            identity = new SimplePrincipal(name);
            if (getUseFirstPass()) {
                String userName = identity.getName();
                if (log.isDebugEnabled())
                    log.debug("Storing username '" + userName +
"'" and empty password");
                // Add the username and an empty password to
the shared state map
                sharedState.put("javax.security.auth.login.name", identity);
                sharedState.put("javax.security.auth.login.password", "");
            }
            loginOk = true;
            return true;
        }
    }

    return false; // Attempted login but not successful.
}

protected boolean delegationAcceptable(String requestedUser,
OuterUserCredential connectionUser) {
    if (delegationMappings == null) {
        return false;
    }

    String[] allowedMappings =
loadPropertyValue(connectionUser.getName(),
connectionUser.getRealm());
    if (allowedMappings.length == 1 &&
"".equals(allowedMappings[1])) {
        // A wild card mapping was found.
        return true;
    }
    for (String current : allowedMappings) {
        if (requestedUser.equals(current)) {
            return true;
        }
    }
}

```

```
    }  
    return false;  
}
```

See the **ejb-security-interceptors** quickstart **README.html** file for complete instructions and more detailed information about the code.

[Report a bug](#)

8.6.5. Use a Client Side Interceptor in an Application

You can plug a client-side interceptor into an application programmatically or using a `ServiceLoader` mechanism. The following procedure describes the two methods.

Plug the Interceptor into an Application Programmatically

With this approach, you call the `org.jboss.ejb.client.EJBClientContext.registerInterceptor(int order, EJBClientInterceptor interceptor)` API and pass the `order` and the `interceptor` instance. The `order` is used to determine where exactly in the client interceptor chain this `interceptor` is placed.

Plug the Interceptor into an Application via the ServiceLoader Mechanism

With this approach, you create a `META-INF/services/org.jboss.ejb.client.EJBClientInterceptor` file and place or package it in the classpath of the client application. The rules for the file are dictated by the [Java ServiceLoader Mechanism](#). This file is expected to contain a separate line for each fully qualified class name of the EJB client interceptor implementation. The EJB client interceptor classes must be available in the classpath. EJB client interceptors added using the **ServiceLoader** mechanism are added to the end of the client interceptor chain, in the order they are found in the classpath. The **ejb-security-interceptors** quickstart uses this approach.

[Report a bug](#)

8.7. CLUSTERED ENTERPRISE JAVABEANS

8.7.1. About Clustered Enterprise JavaBeans (EJBs)

EJB components can be clustered for high-availability scenarios. They use different protocols than HTTP components, so they are clustered in different ways. EJB 2 and 3 stateful and stateless beans can be clustered.

For information on singletons, refer here: [Section 10.3, “Implement an HA Singleton”](#).



NOTE

EJB 2 entity beans cannot be clustered in EAP 6 and henceforth. This is a migration issue.

[Report a bug](#)

8.7.2. Standalone and In-server Client Configuration

To connect an EJB client to a clustered EJB application, you need to expand the existing configuration in standalone EJB client or in-server EJB client to include cluster connection configuration. The **jboss-ejb-client.properties** for standalone EJB client, or even **jboss-ejb-client.xml** file for a server-side application must be expanded to include a cluster configuration.



NOTE

An EJB client is any program that uses an EJB on a remote server. A client is **in-server** when the JVM doing the calling to the remote server is itself running inside of a server. In other words, an EAP instance calling out to another EAP instance would be considered an in-server client.

Example 8.4. Standalone client with jboss-ejb-client.properties configuration

This example shows the additional cluster configuration required for a standalone EJB client.

```
remote.clusters=ejb
remote.cluster.ejb.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
remote.cluster.ejb.connect.options.org.xnio.Options.SSL_ENABLED=false
remote.cluster.ejb.username=test
remote.cluster.ejb.password=password
```

If an application uses the remote-outbound-connection, you need to configure **jboss-ejb-client.xml** file and add cluster configuration as shown in the following example:

Example 8.5. Client application which is deployed in another EAP 6 instance (Configuring jboss-ejb-client.xml file)

```
<jboss-ejb-client xmlns:xsi="urn:jboss:ejb-client:1.2"
xsi:noNamespaceSchemaLocation="jboss-ejb-client_1_2.xsd">
  <client-context>
    <ejb-receivers>
      <!-- this is the connection to access the app-one -->
      <remoting-ejb-receiver outbound-connection-ref="remote-ejb-connection-1" />
      <!-- this is the connection to access the app-two -->
      <remoting-ejb-receiver outbound-connection-ref="remote-ejb-connection-2" />
    </ejb-receivers>

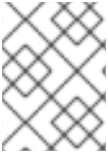
    <!-- if an outbound connection connects to a cluster; a list of members
    is provided after successful connection.
    To connect to this node this cluster element must be defined. -->

    <clusters>
      <!-- cluster of remote-ejb-connection-1 -->
      <cluster name="ejb" security-realm="ejb-security-realm-1"
username="quickuser1">
        <connection-creation-options>
          <property name="org.xnio.Options.SSL_ENABLED" value="false"
/>
          <property name="org.xnio.Options.SASL_POLICY_NOANONYMOUS"
```

```

value="false" />
    </connection-creation-options>
  </cluster>
</clusters>
</client-context>
</jboss-ejb-client>

```



NOTE

For a secure connection you need to add the credentials to cluster configuration in order to avoid an authentication exception.

[Report a bug](#)

8.7.3. Implementing a Custom Load Balancing Policy for EJB Calls

It is possible to implement a custom/alternate load balancing policy so that servers for the application do not handle the same amount of EJB calls in general or for a specific time period.

You can implement **AllClusterNodeSelector** for EJB calls. The node selection behavior of **AllClusterNodeSelector** is similar to default selector except that **AllClusterNodeSelector** uses all available cluster nodes even in case of a large cluster (number of nodes > 20). If an unconnected cluster node is returned it is opened automatically. The following example shows **AllClusterNodeSelector** implementation:

```

package org.jboss.as.quickstarts.ejb.clients.selector;

import java.util.Arrays;
import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;

import org.jboss.ejb.client.ClusterNodeSelector;
public class AllClusterNodeSelector implements ClusterNodeSelector {
    private static final Logger LOG =
        Logger.getLogger(AllClusterNodeSelector.class.getName());

    @Override
    public String selectNode(final String clusterName, final String[]
connectedNodes, final String[] availableNodes) {
        if (LOG.isLoggable(Level.FINER)) {
            LOG.finer("INSTANCE "+this+" : cluster:"+clusterName+"
connected:"+Arrays.deepToString(connectedNodes)+"
available:"+Arrays.deepToString(availableNodes));
        }

        if (availableNodes.length == 1) {
            return availableNodes[0];
        }
        final Random random = new Random();
        final int randomSelection = random.nextInt(availableNodes.length);
        return availableNodes[randomSelection];
    }
}

```



```

    }
}

```

You can also implement the **SimpleLoadFactorNodeSelector** for EJB calls. Load balancing in **SimpleLoadFactorNodeSelector** happens based on a load factor. The load factor (2/3/4) is calculated based on the names of nodes (A/B/C) irrespective of the load on each node. The following example shows **SimpleLoadFactorNodeSelector** implementation:

```

package org.jboss.as.quickstarts.ejb.clients.selector;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

import org.jboss.ejb.client.DeploymentNodeSelector;
public class SimpleLoadFactorNodeSelector implements
DeploymentNodeSelector {
    private static final Logger LOGGER =
Logger.getLogger(SimpleLoadFactorNodeSelector.class.getName());
    private final Map<String, List<String>[]> nodes = new HashMap<String,
List<String>[]>();
    private final Map<String, Integer> cursor = new HashMap<String, Integer>
();

    private ArrayList<String> calculateNodes(Collection<String>
eligibleNodes) {
        ArrayList<String> nodeList = new ArrayList<String>();

        for (String string : eligibleNodes) {
            if(string.contains("A") || string.contains("2")) {
                nodeList.add(string);
                nodeList.add(string);
            } else if(string.contains("B") || string.contains("3")) {
                nodeList.add(string);
                nodeList.add(string);
                nodeList.add(string);
            } else if(string.contains("C") || string.contains("4")) {
                nodeList.add(string);
                nodeList.add(string);
                nodeList.add(string);
                nodeList.add(string);
            }
        }
        return nodeList;
    }

    @SuppressWarnings("unchecked")
    private void checkNodeNames(String[] eligibleNodes, String key) {
        if(!nodes.containsKey(key) || nodes.get(key)[0].size() !=
eligibleNodes.length || !nodes.get(key)

```

```

[0].containsAll(Arrays.asList(eligibleNodes))) {
    // must be synchronized as the client might call it concurrent
    synchronized (nodes) {
        if(!nodes.containsKey(key) || nodes.get(key)[0].size() !=
eligibleNodes.length || !nodes.get(key)
[0].containsAll(Arrays.asList(eligibleNodes))) {
            ArrayList<String> nodeList = new ArrayList<String>();
            nodeList.addAll(Arrays.asList(eligibleNodes));

            nodes.put(key, new List[] { nodeList, calculateNodes(nodeList)
});
        }
    }
}

private synchronized String nextNode(String key) {
    Integer c = cursor.get(key);
    List<String> nodeList = nodes.get(key)[1];

    if(c == null || c >= nodeList.size()) {
        c = Integer.valueOf(0);
    }

    String node = nodeList.get(c);
    cursor.put(key, Integer.valueOf(c + 1));

    return node;
}

@Override
public String selectNode(String[] eligibleNodes, String appName, String
moduleName, String distinctName) {
    if (LOGGER.isLoggable(Level.FINER)) {
        LOGGER.finer("INSTANCE " + this + " : nodes:" +
Arrays.deepToString(eligibleNodes) + " appName:" + appName + "
moduleName:" + moduleName
        + " distinctName:" + distinctName);
    }

    // if there is only one there is no sense to choice
    if (eligibleNodes.length == 1) {
        return eligibleNodes[0];
    }
    final String key = appName + "|" + moduleName + "|" + distinctName;

    checkNodeNames(eligibleNodes, key);
    return nextNode(key);
}
}

```

Configuration with `jboss-ejb-client.properties`

You need to add the property `remote.cluster.ejb.clusternode.selector` with the name of your implementation class (**AllClusterNodeSelector** or **SimpleLoadFactorNodeSelector**). The selector will see all configured servers which are available at the invocation time. The following example uses **AllClusterNodeSelector** as the deployment node selector:

```

remote.clusters=ejb
remote.cluster.ejb.clusternode.selector=org.jboss.as.quickstarts.ejb.clients.selector.AllClusterNodeSelector
remote.cluster.ejb.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
remote.cluster.ejb.connect.options.org.xnio.Options.SSL_ENABLED=false
remote.cluster.ejb.username=test
remote.cluster.ejb.password=password

remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
remote.connections=one,two
remote.connection.one.host=localhost
remote.connection.one.port = 4447
remote.connection.one.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
remote.connection.one.username=user
remote.connection.one.password=user123
remote.connection.two.host=localhost
remote.connection.two.port = 4547
remote.connection.two.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false

```

Using JBoss ejb-client API

You need to add the property **remote.cluster.ejb.clusternode.selector** to the list for the **PropertiesBasedEJBClientConfiguration** constructor. The following example uses **AllClusterNodeSelector** as the deployment node selector:

```

Properties p = new Properties();
p.put("remote.clusters", "ejb");
p.put("remote.cluster.ejb.clusternode.selector",
"org.jboss.as.quickstarts.ejb.clients.selector.AllClusterNodeSelector");
p.put("remote.cluster.ejb.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS", "false");
p.put("remote.cluster.ejb.connect.options.org.xnio.Options.SSL_ENABLED", "false");
p.put("remote.cluster.ejb.username", "test");
p.put("remote.cluster.ejb.password", "password");

p.put("remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED", "false");
p.put("remote.connections", "one,two");
p.put("remote.connection.one.port", "4447");
p.put("remote.connection.one.host", "localhost");
p.put("remote.connection.two.port", "4547");
p.put("remote.connection.two.host", "localhost");

EJBClientConfiguration cc = new PropertiesBasedEJBClientConfiguration(p);
ContextSelector<EJBClientContext> selector = new
ConfigBasedEJBClientContextSelector(cc);
EJBClientContext.setSelector(selector);

p = new Properties();
p.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
InitialContext context = new InitialContext(p);

```

Server application side configuration with `jboss-ejb-client.xml`

To use the load balancing policy for server to server communication; package the class together with the application and configure it within the `jboss-ejb-client.xml` settings (located in **META-INF** folder). The following example uses **AllClusterNodeSelector** as the deployment node selector:

```
<jboss-ejb-client xmlns:xsi="urn:jboss:ejb-client:1.2"
xsi:noNamespaceSchemaLocation="jboss-ejb-client_1_2.xsd">
  <client-context deployment-node-
selector="org.jboss.ejb.client.DeploymentNodeSelector">
    <ejb-receivers>
      <!-- this is the connection to access the app -->
      <remoting-ejb-receiver outbound-connection-ref="remote-ejb-
connection-1" />
    </ejb-receivers>

    <!-- if an outbound connection connect to a cluster a list of members
is provided after successful connection.
To connect to this node this cluster element must be defined.
-->
    <clusters>
      <!-- cluster of remote-ejb-connection-1 -->
      <cluster name="ejb" security-realm="ejb-security-realm-1"
username="test" cluster-node-
selector="org.jboss.as.quickstarts.ejb.clients.selector.AllClusterNodeSele
ctor">
        <connection-creation-options>
          <property name="org.xnio.Options.SSL_ENABLED" value="false" />
          <property name="org.xnio.Options.SASL_POLICY_NOANONYMOUS"
value="false" />
        </connection-creation-options>
      </cluster>
    </clusters>
  </client-context>
</jboss-ejb-client>
```

To use the above configuration with security, you will need to add **ejb-security-realm-1** to client-server configuration. The following example shows the CLI commands for adding security realm (**ejb-security-realm-1**) the value is the base64 encoded password for the user "test":

```
core-service=management/security-realm=ejb-security-realm-1:add()
core-service=management/security-realm=ejb-security-realm-1/server-
identity=secret:add(value=cXVpY2sxMjMr)
```



NOTE

If you are using standalone mode use the start option **-Djboss.node.name=** or the server configuration file **standalone.xml** to configure the server name (server name=""). Ensure that the server name is unique. In domain mode, the controller automatically validates that the names are unique.

[Report a bug](#)

8.7.4. Transaction Behavior of EJB Invocations

Server to Server Invocations

Transaction attributes for distributed JBoss EAP applications need to be handled in a way as if the application is called on the same server. To discontinue a transaction, the destination method must be marked **REQUIRES_NEW** using different interfaces.



NOTE

JBoss EAP 6 does not require Java Transaction Services (JTS) for transaction propagation on server-to-server EJB invocations if both servers are JBoss EAP 6. JBoss EJB client API library handles it itself.

Client Side Invocations

To invoke EJB session beans with a JBoss EAP 6 standalone client, the client must have a reference to the **InitialContext** object while the EJB proxies or **UserTransaction** are used. It is also important to keep the **InitialContext** object open while EJB proxies or **UserTransaction** are being used. Control of the connections will be inside the classes created by the **InitialContext** with the properties.

The following example shows EJB client API which holds a reference to the **InitialContext** object.

Example 8.6. EJB client API referencing InitialContext object

```
package org.jboss.as.quickstarts.ejb.multi.server;

import java.util.Date;
import java.util.Properties;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.naming.Context;
import javax.naming.InitialContext;

import org.jboss.as.quickstarts.ejb.multi.server.app.MainApp;
import org.jboss.ejb.client.ContextSelector;
import org.jboss.ejb.client.EJBClientConfiguration;
import org.jboss.ejb.client.EJBClientContext;
import org.jboss.ejb.client.PropertiesBasedEJBClientConfiguration;
import org.jboss.ejb.client.remoting.ConfigBasedEJBClientContextSelector;

public class Client {

    /**
     * @param args no args needed
     * @throws Exception
     */
    public static void main(String[] args) throws Exception {
        // suppress output of client messages
        Logger.getLogger("org.jboss").setLevel(Level.OFF);
        Logger.getLogger("org.xnio").setLevel(Level.OFF);

        Properties p = new Properties();
```

```

p.put("remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED", "false");
p.put("remote.connections", "one");
p.put("remote.connection.one.port", "4447");
p.put("remote.connection.one.host", "localhost");
p.put("remote.connection.one.username", "quickuser");
p.put("remote.connection.one.password", "quick-123");

EJBClientConfiguration cc = new
PropertiesBasedEJBClientConfiguration(p);
ContextSelector<EJBClientContext> selector = new
ConfigBasedEJBClientContextSelector(cc);
EJBClientContext.setSelector(selector);

Properties props = new Properties();
props.put(Context.URL_PKG_PREFIXES,
"org.jboss.ejb.client.naming");
InitialContext context = new InitialContext(props);

final String rcal = "ejb:jboss-ejb-multi-server-app-main/ejb//"
+ ("MainAppBean") + "!" + MainApp.class.getName();
final MainApp remote = (MainApp) context.lookup(rcal);
final String result = remote.invokeAll("Client call at " + new
Date());

System.out.println("InvokeAll succeed: " + result);
}
}

```

NOTE

Obtaining a **UserTransaction** reference on the client is unsupported for scenarios with a scoped EJB client context and for invocations which use the **remote-naming** protocol. This is because in these scenarios, **InitialContext** encapsulates its own EJB client context instance; which cannot be accessed using the static methods of the **EJBClient** class. When **EJBClient.getUserTransaction()** is called, it returns a transaction from default (global) EJB client context (which might not be initialized) and not from the desired one.

UserTransaction reference on the Client Side

The following example shows how to get **UserTransaction** reference on a standalone client.

Example 8.7. Standalone client referencing UserTransaction object

```

import org.jboss.ejb.client.EJBClient;
import javax.transaction.UserTransaction;
.
.
Context context=null;

```

```

UserTransaction tx=null;
try {
    Properties props = new Properties();
    // REMEMBER: there must be a jboss-ejb-client.properties with the
    connection parameter
    //           in the clients classpath
    props.put(Context.URL_PKG_PREFIXES,
"org.jboss.ejb.client.naming");
    context = new InitialContext(props);
    System.out.println("\n\tGot initial Context: "+context);
    tx=EJBClient.getUserTransaction("yourServerName");
    System.out.println("UserTransaction = "+tx.getStatus());
    tx.begin();
    // do some work
    ...
}catch (Exception e) {
    e.printStackTrace();
    tx.rollback();
}finally{
    if(context != null) {
        context.close();
    }
}

```

NOTE

To get **UserTransaction** reference on the client side; start your server with the following system property **-Djboss.node.name=yourServerName** and then use it on client side as following:

```
tx=EJBClient.getUserTransaction("yourServerName");
```

Replace "yourServerName" with the name of your server. If a user transaction is started on a node all invocations are sticky on the node and the node must have all the needed EJBs. It is not possible to use **UserTransaction** with remote-naming protocol and scoped-context.

[Report a bug](#)

8.8. REFERENCE

8.8.1. EJB JNDI Naming Reference

The JNDI lookup name for a session bean has the syntax of:

```
ejb:<appName>/<moduleName>/<distinctName>/<beanName>!<viewClassName>?
stateful
```

<appName>

If the session bean's JAR file has been deployed within an enterprise archive (EAR) then this is the name of that EAR. By default, the name of an EAR is its filename without the **.ear** suffix. The

application name can also be overridden in its **application.xml** file. If the session bean is not deployed in an EAR then leave this blank.

<moduleName>

The module name is the name of the JAR file that the session bean is deployed in. By the default, the name of the JAR file is its filename without the **.jar** suffix. The module name can also be overridden in the JAR's **ejb-jar.xml** file.

<distinctName>

JBoss EAP 6 allows each deployment to specify an optional distinct name. If the deployment does not have a distinct name then leave this blank.

<beanName>

The bean name is the classname of the session bean to be invoked.

<viewClassName>

The view class name is the fully qualified classname of the remote interface. This includes the package name of the interface.

?stateful

The **?stateful** suffix is required when the JNDI name refers to a stateful session bean. It is not included for other bean types.

[Report a bug](#)

8.8.2. EJB Reference Resolution

This section covers how JBoss implements **@EJB** and **@Resource**. Please note that XML always overrides annotations but the same rules apply.

Rules for the @EJB annotation

- The **@EJB** annotation also has a **mappedName ()** attribute. The specification leaves this as vendor specific metadata, but JBoss recognizes **mappedName ()** as the global JNDI name of the EJB you are referencing. If you have specified a **mappedName ()**, then all other attributes are ignored and this global JNDI name is used for binding.
- If you specify **@EJB** with no attributes defined:

```
@EJB
ProcessPayment myEjbref;
```

Then the following rules apply:

- The EJB jar of the referencing bean is searched for an EJB with the interface used in the **@EJB** injection. If there are more than one EJB that publishes same business interface, then an exception is thrown. If there is only one bean with that interface then that one is used.
- Search the EAR for EJBs that publish that interface. If there are duplicates, then an exception is thrown. Otherwise the matching bean is returned.

- Search globally in JBoss runtime for an EJB of that interface. Again, if duplicates are found, an exception is thrown.
- **@EJB.beanName()** corresponds to **<ejb-link>**. If the **beanName()** is defined, then use the same algorithm as **@EJB** with no attributes defined except use the **beanName()** as a key in the search. An exception to this rule is if you use the **ejb-link '#'** syntax. The '#' syntax allows you to put a relative path to a jar in the EAR where the EJB you are referencing is located. Refer to the EJB 3.1 specification for more details.

[Report a bug](#)

8.8.3. Project dependencies for Remote EJB Clients

Maven projects that include the invocation of session beans from remote clients require the following dependencies from the JBoss EAP 6 Maven repository.

Table 8.4. Maven dependencies for Remote EJB Clients

GroupID	ArtifactID
org.jboss.spec	jboss-javaee-6.0
org.jboss.as	jboss-as-ejb-client-bom
org.jboss.spec.javax.transaction	jboss-transaction-api_1.1_spec
org.jboss.spec.javax.ejb	jboss-ejb-api_3.1_spec
org.jboss	jboss-ejb-client
org.jboss.xnio	xnio-api
org.jboss.xnio	xnio-nio
org.jboss.remoting3	jboss-remoting
org.jboss.sasl	jboss-sasl
org.jboss.marshalling	jboss-marshalling-river

With the exception of **jboss-javaee-6.0** and **jboss-as-ejb-client-bom**, these dependencies must be added to the **<dependencies>** section of the **pom.xml** file.

The **jboss-javaee-6.0** and **jboss-as-ejb-client-bom** dependencies should be added to the **<dependencyManagement>** section of your **pom.xml** with the scope of **import**.



NOTE

The **artifactID**'s versions are subject to change. Refer to the Maven repository for the relevant version.

```

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.spec</groupId>
      <artifactId>jboss-javaee-6.0</artifactId>
      <version>3.0.0.Final-redhat-1</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>

    <dependency>
      <groupId>org.jboss.as</groupId>
      <artifactId>jboss-as-ejb-client-bom</artifactId>
      <version>7.1.1.Final-redhat-1</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

```

Refer to **ejb-remote/client/pom.xml** in the quickstart files for a complete example of dependency configuration for remote session bean invocation.

[Report a bug](#)

8.8.4. jboss-ejb3.xml Deployment Descriptor Reference

jboss-ejb3.xml is a custom deployment descriptor that can be used in either EJB JAR or WAR archives. In an EJB JAR archive it must be located in the **META-INF/** directory. In a WAR archive it must be located in the **WEB-INF/** directory.

The format is similar to **ejb-jar.xml**, using some of the same namespaces and providing some other additional namespaces. The contents of **jboss-ejb3.xml** are merged with the contents of **ejb-jar.xml**, with the **jboss-ejb3.xml** items taking precedence.

This document only covers the additional non-standard namespaces used by **jboss-ejb3.xml**. Refer to <http://java.sun.com/xml/ns/javaee/> for documentation on the standard namespaces.

The root namespace is **http://www.jboss.com/xml/ns/javaee**.

Assembly descriptor namespaces

The following namespaces can all be used in the **<assembly-descriptor>** element. They can be used to apply their configuration to a single bean, or to all beans in the deployment by using ***** as the **ejb-name**.

The clustering namespace: urn:clustering:1.0

```
xmlns:c="urn:clustering:1.0"
```

This allows you to mark EJB's as clustered. It is the deployment descriptor equivalent to **@org.jboss.ejb3.annotation.Clustered**.

```
<c:clustering>
```

```

    <ejb-name>DDBasedClusteredSFSB</ejb-name>
    <c:clustered>true</c:clustered>
  </c:clustering>

```

The security namespace (urn:security)

```
xmlns:s="urn:security"
```

This allows you to set the **security-domain** and the **run-as-principal** for an EJB.

```

<s:security>
  <ejb-name>*</ejb-name>
  <s:security-domain>myDomain</s:security-domain>
  <s:run-as-principal>myPrincipal</s:run-as-principal>
</s:security>

```

The resource adapter namespace: urn:resource-adapter-binding

```
xmlns:r="urn:resource-adapter-binding"
```

This allows you to set the resource adapter for a Message-Driven Bean.

```

<r:resource-adapter-binding>
  <ejb-name>*</ejb-name>
  <r:resource-adapter-name>myResourceAdapter</r:resource-adapter-name>
</r:resource-adapter-binding>

```

The IIOP namespace: urn:iio

```
xmlns:u="urn:iio"
```

The IIOP namespace is where IIOP settings are configured.

The pool namespace: urn:ejb-pool:1.0

```
xmlns:p="urn:ejb-pool:1.0"
```

This allows you to select the pool that is used by the included stateless session beans or Message-Driven Beans. Pools are defined in the server configuration.

```

<p:pool>
  <ejb-name>*</ejb-name>
  <p:bean-instance-pool-ref>my-pool</p:bean-instance-pool-ref>
</p:pool>

```

The cache namespace: urn:ejb-cache:1.0

```
xmlns:c="urn:ejb-cache:1.0"
```

This allows you to select the cache that is used by the included stateful session beans. Caches are defined in the server configuration.

```

<c:cache>
  <ejb-name>*</ejb-name>
  <c:cache-ref>my-cache</c:cache-ref>
</c:cache>

```

Example 8.8. jboss-ejb3.xml file

```

<?xml version="1.1" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:c="urn:clustering:1.0"
  xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd"
  version="3.1" impl-version="2.0">
  <enterprise-beans>
    <message-driven>
      <ejb-name>ReplyingMDB</ejb-name>
      <ejb-
class>org.jboss.as.test.integration.ejb.mdb.messagedestination.ReplyingM
DB</ejb-class>
      <activation-config>
        <activation-config-property>
          <activation-config-property-name>destination</activation-
config-property-name>
          <activation-config-property-
value>java:jboss/mdbtest/messageDestinationQueue
          </activation-config-property-value>
        </activation-config-property>
      </activation-config>
    </message-driven>
  </enterprise-beans>
  <assembly-descriptor>
    <c:clustering>
      <ejb-name>DDBasedClusteredSFSB</ejb-name>
      <c:clustered>true</c:clustered>
    </c:clustering>
  </assembly-descriptor>
</jboss:ejb-jar>

```



NOTE

There are known issues with the **jboss-ejb3-spec-2_0.xsd** that may result in schema validation errors. You can ignore these errors. For more information, see https://bugzilla.redhat.com/show_bug.cgi?id=1192591.

[Report a bug](#)

CHAPTER 9. JBOSS MBEAN SERVICES

9.1. WRITING JBOSS MBEAN SERVICES

Writing a custom MBean service that relies on a JBoss service requires the service interface method pattern. JBoss MBean service interface method pattern consists of a set of life cycle operations which inform an MBean service when it can **create**, **start**, **stop**, and **destroy** itself.

You can manage the dependency state using any of the following approaches:

- If you want specific methods to be called on your MBean, declare those methods in your MBean interface. This approach allows your MBean implementation to avoid dependencies on JBoss specific classes
- If you are not bothered about dependencies on JBoss specific classes then you may have your MBean interface extend the **ServiceMBean** interface and **ServiceMBeanSupport** class. The **ServiceMBeanSupport** class provides implementations of the service lifecycle methods like **create**, **start** and **stop**. To handle a specific event like the **start()** event, you need to override **startService()** method provided by the **ServiceMBeanSupport** class.

[Report a bug](#)

9.2. A STANDARD MBEAN EXAMPLE

This section develops two sample MBean services packaged together in a service archive (**.sar**).

ConfigServiceMBean interface declares specific methods like the **start**, **getTimeout** and **stop** methods to **start**, **hold** and **stop** the MBean correctly without using any JBoss specific classes. **ConfigService** class implements **ConfigServiceMBean** interface and consequently implements the methods used within that interface.

PlainThread class extends **ServiceMBeanSupport** class and implements **PlainThreadMBean** interface. **PlainThread** starts a thread and uses **ConfigServiceMBean.getTimeout()** to determine how long the thread should sleep.

Example 9.1. Sample MBean services

```
package org.jboss.example.mbean.support;

public interface ConfigServiceMBean {

    int getTimeout();

    void start();

    void stop();

}

package org.jboss.example.mbean.support;

public class ConfigService implements ConfigServiceMBean {
    int timeout;
```

```

        @Override
        public int getTimeout() {
            return timeout;
        }

        @Override
        public void start() {
            //Create a random number between 3000 and 6000 milliseconds
            timeout = (int)Math.round(Math.random() * 3000) + 3000;
            System.out.println("Random timeout set to " + timeout + "
seconds");
        }

        @Override
        public void stop() {
            timeout = 0;
        }
    }

package org.jboss.example.mbean.support;

import org.jboss.system.ServiceMBean;

public interface PlainThreadMBean extends ServiceMBean {
    void setConfigService(ConfigServiceMBean configServiceMBean);
}

package org.jboss.example.mbean.support;

import org.jboss.system.ServiceMBeanSupport;

public class PlainThread extends ServiceMBeanSupport implements
PlainThreadMBean {

    private ConfigServiceMBean configService;
    private Thread thread;
    private volatile boolean done;

    @Override
    public void setConfigService(ConfigServiceMBean configService) {
        this.configService = configService;
    }

    @Override
    protected void startService() throws Exception {
        System.out.println("Starting Plain Thread MBean");
        done = false;
        thread = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    while (!done) {
                        System.out.println("Sleeping...");
                        Thread.sleep(configService.getTimeout());
                    }
                }
            }
        });
    }
}

```

```

        System.out.println("Slept!");
    }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
    }
});
thread.start();
}

@Override
protected void stopService() throws Exception {
    System.out.println("Stopping Plain Thread MBean");
    done = true;
}
}
}

```

The **jboss-service.xml** descriptor shows how **ConfigService** class is injected into **PlainThread** class using **inject** tag. The **inject** tag establishes a dependency between **PlainThreadMBean** and **ConfigServiceMBean** and thus allows **PlainThreadMBean** use **ConfigServiceMBean** easily.

Example 9.2. JBoss-service.xml Service Descriptor

```

<server xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:jboss:service:7.0 jboss-
service_7_0.xsd"
        xmlns="urn:jboss:service:7.0">
    <mbean code="org.jboss.example.mbean.support.ConfigService"
name="jboss.support:name=ConfigBean"/>
    <mbean code="org.jboss.example.mbean.support.PlainThread"
name="jboss.support:name=ThreadBean">
        <attribute name="configService">
            <inject bean="jboss.support:name=ConfigBean"/>
        </attribute>
    </mbean>
</server>

```

After writing the sample MBeans you can package the classes and the **jboss-service.xml** descriptor in the **META-INF** folder of a service archive (**.sar**).

[Report a bug](#)

9.3. DEPLOYING JBOSS MBEAN SERVICES

To build and deploy the sample MBeans (**ServiceMBeanTest.sar**) in **Domain** mode use the following commands:

```
[domain@localhost:9999 /] deploy ~/Desktop/ServiceMBeanTest.sar
```

```
[domain@localhost:9999 /] deploy ~/Desktop/ServiceMBeanTest.sar --all-server-groups
```

To build and deploy the sample MBeans (**ServiceMBeanTest.sar**) in **Standalone** mode use the following command:

```
[standalone@localhost:9999 /] deploy ~/Desktop/ServiceMBeanTest.sar
```

To undeploy the sample MBeans use the following command:

```
[standalone@localhost:9999 /] undeploy ServiceMBeanTest.sar
```

[Report a bug](#)

CHAPTER 10. CLUSTERING IN WEB APPLICATIONS

10.1. SESSION REPLICATION

10.1.1. About HTTP Session Replication

Session replication ensures that client sessions of distributable applications are not disrupted by failovers of nodes in a cluster. Each node in the cluster shares information about ongoing sessions, and can take them over if the originally-involved node disappears.

Session replication is the mechanism by which `mod_cluster`, `mod_jk`, `mod_proxy`, ISAPI, and NSAPI clusters provide high availability.

[Report a bug](#)

10.1.2. About the Web Session Cache

The web session cache can be configured when you use any of the HA profiles, including the **standalone-ha.xml** profile, or the managed domain profiles **ha** or **full-ha**. The most commonly configured elements are the cache mode and the number of cache owners for a distributed cache. The **owners** parameter works only in the **DIST** mode.

Cache Mode

The cache mode can either be **REPL** (the default) or **DIST**.

REPL

The **REPL** mode replicates the entire cache to every other node in the cluster. This is the safest option, but introduces more overhead.

DIST

The **DIST** mode is similar to the *buddy mode* provided in previous implementations. It reduces overhead by distributing the cache to the number of nodes specified in the **owners** parameter. This number of owners defaults to **2**.

Owners

The **owners** parameter controls how many cluster nodes hold replicated copies of the session. The default is **2**.

[Report a bug](#)

10.1.3. Configure the Web Session Cache

The web session cache defaults to **REPL**. If you wish to use **DIST** mode, run the following two commands in the Management CLI. If you use a different profile, change the profile name in the commands. If you use a standalone server, remove the **/profile=ha** portion of the commands.

Procedure 10.1. Configure the Web Session Cache

1. **Change the default cache mode to DIST.**

```
/profile=ha/subsystem=infinispan/cache-container=web/:write-attribute(name=default-cache,value=dist)
```

2. Set the number of owners for a distributed cache.

The following command sets **5** owners. The default is **2**.

```
/profile=ha/subsystem=infinispan/cache-container=web/distributed-cache=dist/:write-attribute(name=owners,value=5)
```

3. Change the default cache mode back to REPL.

```
/profile=ha/subsystem=infinispan/cache-container=web/:write-attribute(name=default-cache,value=repl)
```

4. Restart the Server

After changing the web cache mode, you must restart the server.

Result

Your server is configured for session replication. To use session replication in your own applications, refer to the following topic: [Section 10.1.4, “Enable Session Replication in Your Application”](#).

[Report a bug](#)

10.1.4. Enable Session Replication in Your Application

Summary

To take advantage of JBoss EAP 6 High Availability (HA) features, you must configure your application to be distributable. This procedure shows how to do that, and then explains some of the advanced configuration options you can use.

Procedure 10.2. Make your Application Distributable

1. Required: Indicate that your application is distributable.

If your application is not marked as distributable, its sessions will never be distributed. Add the `<distributable/>` element inside the `<web-app>` tag of your application's `web.xml` descriptor file. Here is an example.

Example 10.1. Minimum Configuration for a Distributable Application

```
<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">

  <distributable/>

</web-app>
```

2. Modify the default replication behavior if desired.

If you want to change any of the values affecting session replication, you can override them inside a `<replication-config>` element which is a child element of the `<jboss-web>` element of your application's `jboss-web.xml` file. For a given element, only include it if you want to override the defaults. The following example lists all of the default settings, and is followed by a table which explains the most commonly changed options.

Example 10.2. Example `<replication-config>` Values

```
<!DOCTYPE jboss-web PUBLIC
    "-//JBoss//DTD Web Application 5.0//EN"
    "http://www.jboss.org/j2ee/dtd/jboss-web_5_0.dtd">

<jboss-web>

    <replication-config>
        <replication-
trigger>SET_AND_NON_PRIMITIVE_GET</replication-trigger>
        <replication-granularity>SESSION</replication-granularity>
        <use-jk>>false</use-jk>
        <max-unreplicated-interval>30</max-unreplicated-interval>
        <snapshot-mode>INSTANT</snapshot-mode>
        <snapshot-interval>1000</snapshot-interval>
        <session-notification-
policy>com.example.CustomSessionNotificationPolicy</session-
notification-policy>
    </replication-config>

</jboss-web>
```

Table 10.1. Common Options for Session Replication

Option	Description
--------	-------------

Option	Description
<replication-trigger>	<p>Controls which conditions should trigger session data replication across the cluster. This option is necessary because after a mutable object (stored as a session attribute) is accessed from the session, the container has no clear way to know if the object has been modified and needs to be replicated, unless method setAttribute() is called directly.</p> <p>Valid Values for <replication-trigger></p> <p>SET_AND_GET</p> <p>This is the safest but worst-performing option. Session data is always replicated, even if its content has only been accessed, and not modified. This setting is preserved for legacy purposes only. To get the same behavior with better performance, you may, instead of using this setting, set <max-unreplicated-interval> to 0.</p> <p>SET_AND_NON_PRIMITIVE_GET</p> <p>The default value. Session data is only replicated if an object of a non-primitive type is accessed. This means that the object is not of a well-known Java type such as Integer, Long, or String.</p> <p>SET</p> <p>This option assumes that the application will explicitly call setAttribute on the session when the data needs to be replicated. It prevents unnecessary replication and can benefit overall performance, but is inherently unsafe.</p> <p>Regardless of the setting, you can always trigger session replication by calling setAttribute().</p>
<replication-granularity>	<p>Determines the granularity of data that is replicated. It defaults to SESSION, but can be set to ATTRIBUTE instead, to increase performance on sessions where most attributes remain unchanged.</p> <p>Valid values for <replication-granularity></p> <p>ATTRIBUTE</p> <p>This is only for dirty attributes in the session and for some session data like the last-accessed timestamp.</p> <p>SESSION</p> <p>The default value. The entire session object is replicated if any attribute is dirty. The shared object references are maintained on remote nodes since the entire session is serialized in one unit.</p> <div data-bbox="593 1912 699 2024"> </div> <p>NOTE</p> <p>FIELD is not supported in JBoss EAP 6.</p>

The following options rarely need to be changed.

Table 10.2. Less Commonly Changed Options for Session Replication

Option	Description
<use-jk>	Whether to assume that a load balancer such as mod_cluster , mod_jk , or mod_proxy is in use. The default is false . If set to true , the container examines the session ID associated with each request and replaces the jvmRoute portion of the session ID if there is a failover.
<max-unreplicated-interval>	<p>The maximum interval (in seconds) to wait after a session was accessed before triggering a replication of a session's timestamp, even if it is considered to be unchanged. This ensures that cluster nodes are aware of each session's timestamp and that an unreplicated session will not expire incorrectly during a failover. It also ensures that you can rely on a correct value for calls to method HttpSession.getLastAccessedTime() during a failover.</p> <p>By default, no value is specified. A value of 0 causes the timestamp to be replicated whenever the session is accessed. A value of -1 causes the timestamp to be replicated only if other activity during the request triggers a replication. A positive value greater than HttpSession.getMaxInactiveInterval() is treated as a misconfiguration and converted to 0.</p>
<snapshot-mode>	<p>Specifies when sessions are replicated to other nodes. The default is INSTANT and the other possible value is INTERVAL.</p> <p>In INSTANT mode, changes are replicated at the end of a request, by means of the request processing thread. The <snapshot-interval> option is ignored.</p> <p>In INTERVAL mode, a background task runs at the interval specified by <snapshot-interval>, and replicates modified sessions.</p>
<snapshot-interval>	The interval, in milliseconds, at which modified sessions should be replicated when using INTERVAL for the value of <snapshot-mode> .
<session-notification-policy>	The fully-qualified class name of the implementation of interface ClusteredSessionNotificationPolicy which governs whether servlet specification notifications are emitted to any registered HttpSessionListener , HttpSessionAttributeListener , or HttpSessionBindingListener .

[Report a bug](#)

10.2. HTTPSESSION PASSIVATION AND ACTIVATION

10.2.1. About HTTP Session Passivation and Activation

Passivation is the process of controlling memory usage by removing relatively unused sessions from memory while storing them in persistent storage.

Activation is when passivated data is retrieved from persisted storage and put back into memory.

Passivation occurs at three different times in a HTTP session's lifetime:

- When the container requests the creation of a new session, if the number of currently active session exceeds a configurable limit, the server attempts to passivate some sessions to make room for the new one.
- Periodically, at a configured interval, a background task checks to see if sessions should be passivated.
- When a web application is deployed and a backup copy of sessions active on other servers is acquired by the newly deploying web application's session manager, sessions may be passivated.

A session is passivated if it meets the following conditions:

- The session has not been in use for longer than a configurable maximum idle time.
- The number of active sessions exceeds a configurable maximum and the session has not been in use for longer than a configurable minimum idle time.

Sessions are always passivated using a Least Recently Used (LRU) algorithm.

[Report a bug](#)

10.2.2. Configure HttpSession Passivation in Your Application

Overview

HttpSession passivation is configured in your application's **WEB_INF/jboss-web.xml** or **META_INF/jboss-web.xml** file.

Example 10.3. jboss-web.xml File

```
<!DOCTYPE jboss-web PUBLIC
    "-//JBoss//DTD Web Application 5.0//EN"
    "http://www.jboss.org/j2ee/dtd/jboss-web_5_0.dtd">

<jboss-web version="6.0"
    xmlns="http://www.jboss.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-web_6_0.xsd">

    <max-active-sessions>20</max-active-sessions>
    <passivation-config>
        <use-session-passivation>true</use-session-passivation>
        <passivation-min-idle-time>60</passivation-min-idle-time>
        <passivation-max-idle-time>600</passivation-max-idle-time>
    </passivation-config>

</jboss-web>
```

Passivation Configuration Elements

`<max-active-sessions>`

The maximum number of active sessions allowed. If the number of sessions managed by the session manager exceeds this value and passivation is enabled, the excess will be passivated based on the configured `<passivation-min-idle-time>`. Then, if the number of active sessions still exceeds this limit, attempts to create new sessions will fail. The default value of `-1` sets no limit on the maximum number of active sessions.

`<passivation-config>`

This element holds the rest of the passivation configuration parameters, as child elements.

`<passivation-config>` Child Elements

`<use-session-passivation>`

Whether or not to use session passivation. The default value is `false`.

`<passivation-min-idle-time>`

The minimum time, in seconds, that a session must be inactive before the container will consider passivating it in order to reduce the active session count to conform to value defined by `max-active-sessions`. The default value of `-1` disables passivating sessions before `<passivation-max-idle-time>` has elapsed. Neither a value of `-1` nor a high value are recommended if `<max-active-sessions>` is set.

`<passivation-max-idle-time>`

The maximum time, in seconds, that a session can be inactive before the container attempts to passivate it to save memory. Passivation of such sessions takes place regardless of whether the active session count exceeds `<max-active-sessions>`. This value should be less than the `<session-timeout>` setting in the `web.xml`. The default value of `-1` disables passivation based on maximum inactivity.

NOTE

The total number of sessions in memory includes sessions replicated from other cluster nodes that are not being accessed on this node. Take this into account when setting `<max-active-sessions>`. The number of sessions replicated from other nodes also depends on whether **REPL** or **DIST** cache mode is enabled. In **REPL** cache mode, each session is replicated to each node. In **DIST** cache mode, each session is replicated only to the number of nodes specified by the **owners** parameter. See [Section 10.1.2, “About the Web Session Cache”](#) and [Section 10.1.3, “Configure the Web Session Cache”](#) for information on configuring session cache modes.

For example, consider an eight node cluster, where each node handles requests from 100 users. With **REPL** cache mode, each node would store 800 sessions in memory. With **DIST** cache mode enabled, and the default **owners** setting of `2`, each node stores 200 sessions in memory.

[Report a bug](#)

10.3. IMPLEMENT AN HA SINGLETON

Summary

The following procedure demonstrates how to deploy a service that is wrapped with the `SingletonService` decorator and used as a cluster-wide singleton service. The service activates a scheduled timer, which is started only once in the cluster.

Procedure 10.3. Implement an HA Singleton Service

1. Write the HA singleton service application.

The following is a simple example of a **Service** that is wrapped with the **SingletonService** decorator to be deployed as a singleton service. A complete example can be found in the **cluster-ha-singleton** quickstart that ships with Red Hat JBoss Enterprise Application Platform 6. This quickstart contains all the instructions to build and deploy the application.

- a. Create a service.

The following listing is an example of a service:

```
package org.jboss.as.quickstarts.cluster.hsingleton.service.ejb;

import java.util.Date;
import java.util.concurrent.atomic.AtomicBoolean;

import javax.naming.InitialContext;
import javax.naming.NamingException;

import org.jboss.logging.Logger;
import org.jboss.msc.service.Service;
import org.jboss.msc.service.ServiceName;
import org.jboss.msc.service.StartContext;
import org.jboss.msc.service.StartException;
import org.jboss.msc.service.StopContext;

/**
 * @author <a href="mailto:wfink@redhat.com">Wolf-Dieter Fink</a>
 */
public class HATimerService implements Service<String> {
    private static final Logger LOG =
        Logger.getLogger(HATimerService.class);
    private static final ServiceName SINGLETON_SERVICE_NAME =
        ServiceName.JBOSS.append("quickstart", "ha", "singleton",
            "timer");

    /**
     * A flag whether the service is started.
     */
    private final AtomicBoolean started = new
        AtomicBoolean(false);

    /**
     * @return the name of the server node
     */
}
```



```

        public String getValue() throws IllegalStateException,
        IllegalArgumentException {
            LOGGER.infof("%s is %s at %s",
            HATimerService.class.getSimpleName(), (started.get() ? "started"
            : "not started"), System.getProperty("jboss.node.name"));
            return "";
        }

        public void start(StartContext arg0) throws StartException {
            if (!started.compareAndSet(false, true)) {
                throw new StartException("The service is still
started!");
            }
            LOGGER.info("Start HASingleton timer service '" +
this.getClass().getName() + "'");

            final String node =
System.getProperty("jboss.node.name");
            try {
                InitialContext ic = new InitialContext();
                ((Scheduler) ic.lookup("global/jboss-cluster-ha-
singleton-
service/SchedulerBean!org.jboss.as.quickstarts.cluster.hasingleto
n.service.ejb.Scheduler")).initialize("HASingleton timer @" +
node + " " + new Date());
            } catch (NamingException e) {
                throw new StartException("Could not initialize
timer", e);
            }
        }

        public void stop(StopContext arg0) {
            if (!started.compareAndSet(true, false)) {
                LOGGER.warn("The service '" +
this.getClass().getName() + "' is not active!");
            } else {
                LOGGER.info("Stop HASingleton timer service '" +
this.getClass().getName() + "'");
                try {
                    InitialContext ic = new InitialContext();
                    ((Scheduler) ic.lookup("global/jboss-cluster-ha-
singleton-
service/SchedulerBean!org.jboss.as.quickstarts.cluster.hasingleto
n.service.ejb.Scheduler")).stop();
                } catch (NamingException e) {
                    LOGGER.error("Could not stop timer", e);
                }
            }
        }
    }
}

```

- b. Create an activator that installs the **Service** as a clustered singleton.

The following listing is an example of a Service activator that installs the **HATimerService** as a clustered singleton service:

```

package org.jboss.as.quickstarts.cluster.hasingleton.service.ejb;

import org.jboss.as.clustering.singleton.SingletonService;
import org.jboss.logging.Logger;
import org.jboss.msc.service.DelegatingServiceContainer;
import org.jboss.msc.service.ServiceActivator;
import org.jboss.msc.service.ServiceActivatorContext;
import org.jboss.msc.service.ServiceController;

/**
 * Service activator that installs the HATimerService as a
 * clustered singleton service
 * during deployment.
 *
 * @author Paul Ferraro
 */
public class HATimerServiceActivator implements ServiceActivator
{
    private final Logger log = Logger.getLogger(this.getClass());

    @Override
    public void activate(ServiceActivatorContext context) {
        log.info("HATimerService will be installed!");

        HATimerService service = new HATimerService();
        SingletonService<String> singleton = new
SingletonService<String>(service,
HATimerService.SINGLETON_SERVICE_NAME);
        /*
         * To pass a chain of election policies to the singleton,
         for example,
         * to tell JGroups to prefer running the singleton on a
         node with a
         * particular name, uncomment the following line:
         */
        // singleton.setElectionPolicy(new
PreferredSingletonElectionPolicy(new
SimpleSingletonElectionPolicy(), new
NamePreference("node1/singleton"));

        singleton.build(new
DelegatingServiceContainer(context.getServiceTarget(),
context.getServiceRegistry()))
            .setInitialMode(ServiceController.Mode.ACTIVE)
            .install()
        ;
    }
}

```



NOTE

The above code example uses a class, **org.jboss.as.clustering.singleton.SingletonService**, that is part of the JBoss EAP private API. A public API will become available in the JBoss EAP 7 release and the private class will be deprecated, but these classes will be maintained and available for the duration of the JBoss EAP 6.x release cycle.

c. Create a ServiceActivator File

Create a file named **org.jboss.msc.service.ServiceActivator** in the application's **resources/META-INF/services/** directory. Add a line containing the fully qualified name of the ServiceActivator class created in the previous step.

```
org.jboss.as.quickstarts.cluster.hasingleton.service.ejb.HATimerServiceActivator
```

d. Create a Singleton bean that implements a timer to be used as a cluster-wide singleton timer.

This Singleton bean must not have a remote interface and you must not reference its local interface from another EJB in any application. This prevents a lookup by a client or other component and ensures the SingletonService has total control of the Singleton.

i. Create the Scheduler interface

```
package
org.jboss.as.quickstarts.cluster.hasingleton.service.ejb;

/**
 * @author <a href="mailto:wfink@redhat.com">Wolf-Dieter
 Fink</a>
 */
public interface Scheduler {

    void initialize(String info);

    void stop();

}
```

ii. Create the Singleton bean that implements the cluster-wide singleton timer.

```
package
org.jboss.as.quickstarts.cluster.hasingleton.service.ejb;

import javax.annotation.Resource;
import javax.ejb.ScheduleExpression;
import javax.ejb.Singleton;
import javax.ejb.Timeout;
import javax.ejb.Timer;
import javax.ejb.TimerConfig;
import javax.ejb.TimerService;
```

```

import org.jboss.logging.Logger;

/**
 * A simple example to demonstrate a implementation of a
 * cluster-wide singleton timer.
 *
 * @author <a href="mailto:wfink@redhat.com">Wolf-Dieter
 * Fink</a>
 */
@Singleton
public class SchedulerBean implements Scheduler {
    private static Logger LOGGER =
    Logger.getLogger(SchedulerBean.class);
    @Resource
    private TimerService timerService;

    @Timeout
    public void scheduler(Timer timer) {
        LOGGER.info("HASingletonTimer: Info=" +
timer.getInfo());
    }

    @Override
    public void initialize(String info) {
        ScheduleExpression sexpr = new ScheduleExpression();
        // set schedule to every 10 seconds for demonstration
        sexpr.hour("*").minute("*").second("0/10");
        // persistent must be false because the timer is
        started by the HASingleton service
        timerService.createCalendarTimer(sexpr, new
TimerConfig(info, false));
    }

    @Override
    public void stop() {
        LOGGER.info("Stop all existing HASingleton timers");
        for (Timer timer : timerService.getTimers()) {
            LOGGER.trace("Stop HASingleton timer: " +
timer.getInfo());
            timer.cancel();
        }
    }
}

```

2. Start each JBoss EAP 6 instance with clustering enabled.

To enable clustering for standalone servers, you must start each server with the **HA** profile, using a unique node name and port offset for each instance.

- For Linux, use the following command syntax to start the servers:

```

EAP_HOME/bin/standalone.sh --server-config=standalone-ha.xml -
Djboss.node.name=UNIQUE_NODE_NAME -Djboss.socket.binding.port-
offset=PORT_OFFSET

```

Example 10.4. Start multiple standalone servers on Linux

```
$ EAP_HOME/bin/standalone.sh --server-config=standalone-ha.xml
-Djboss.node.name=node1
$ EAP_HOME/bin/standalone.sh --server-config=standalone-ha.xml
-Djboss.node.name=node2 -Djboss.socket.binding.port-offset=100
```

- o For Microsoft Windows, use the following command syntax to start the servers:

```
EAP_HOME\bin\standalone.bat --server-config=standalone-ha.xml -
Djboss.node.name=UNIQUE_NODE_NAME -Djboss.socket.binding.port-
offset=PORT_OFFSET
```

Example 10.5. Start multiple standalone servers on Microsoft Windows

```
C:> EAP_HOME\bin\standalone.bat --server-config=standalone-
ha.xml -Djboss.node.name=node1
C:> EAP_HOME\bin\standalone.bat --server-config=standalone-
ha.xml -Djboss.node.name=node2 -Djboss.socket.binding.port-
offset=100
```

**NOTE**

If you prefer not to use command line arguments, you can configure the **standalone-ha.xml** file for each server instance to bind on a separate interface.

3. Deploy the application to the servers

The following Maven command deploys the application to a standalone server running on the default ports.

```
mvn clean install jboss-as:deploy
```

To deploy to additional servers, pass the server name. If it is on a different host, pass the host name and port number on the command line:

```
mvn clean package jboss-as:deploy -Djboss-as.hostname=localhost -
Djboss-as.port=10099
```

See the **cluster-ha-singleton** quickstart that ships with JBoss EAP 6 for Maven configuration and deployment details.

[Report a bug](#)

10.4. APACHE MOD_CLUSTER-MANAGER APPLICATION

10.4.1. About mod_cluster-manager Application

The mod_cluster-manager application is an administration web page which is available on Apache HTTP Server. It is used for monitoring the connected worker nodes and performing various administration tasks like enabling/disabling contexts and configuring the load-balancing properties of worker nodes in a cluster.

[Report a bug](#)

10.4.2. Exploring mod_cluster-manager Application

The mod_cluster-manager application can be used for performing various administration tasks on worker nodes.

The figure shown below represents the mod_cluster-manager application web page with annotations to highlight important components and administration options on the page.

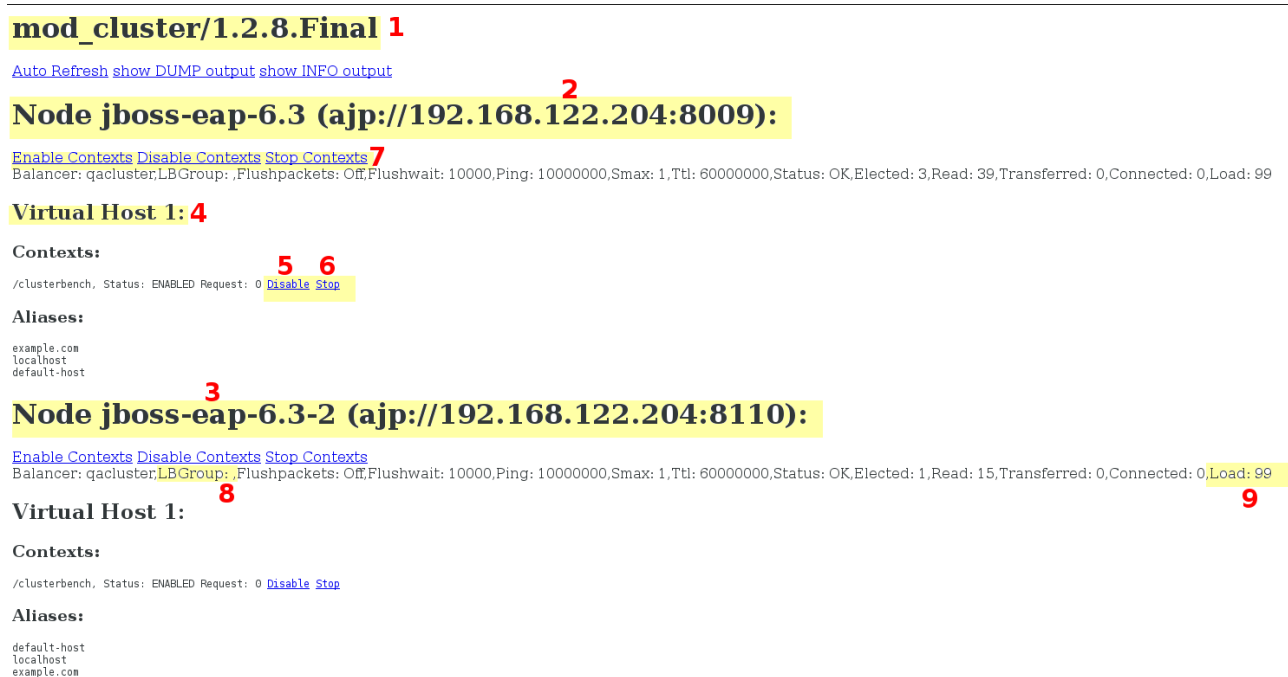


Figure 10.1. mod_cluster Administration Web Page

The annotations are explained below:

- **[1] mod_cluster/1.2.8.Final:** This denotes the version of the mod_cluster native library
- **[2] ajp://192.168.122.204:8099:** This denotes the protocol used (either one of AJP, HTTP, HTTPS), hostname or IP address of the worker node and the port
- **[3] jboss-eap-6.3-2:** This denotes the worker node's JVMRoute.
- **[4] Virtual Host 1:** This denotes the virtual host(s) configured on the worker node
- **[5] Disable :** This is an administration option which can be used to disable the creation of new sessions on the particular context. However the ongoing sessions do not get disabled and remain intact

- **[6] Stop** : This is an administration option which can be used to stop the routing of session requests to the context. The remaining sessions will failover to another node unless the property **sticky-session-force** is set to "true"
- **[7] Enable Contexts Disable Contexts Stop Contexts**: These denote operations which can be performed on the whole node. Selecting one of these options affects all the contexts of a node in all its virtual hosts.
- **[8] Load balancing group (LBGroup)**: The **load-balancing-group** property is set in the `mod_cluster` subsystem in EAP configuration to group all worker nodes into custom load balancing groups. Load balancing group (LBGroup) is an informational field which gives information about all set load balancing groups. If this field is not set, then all worker nodes are grouped into a single default load balancing group



NOTE

This is only an informational field and thus cannot be used to set **load-balancing-group** property. The property has to be set in `mod_cluster` subsystem in EAP configuration.

- **[9] Load (value)**: This indicates the load factor on the worker node. The load factor(s) are evaluated as below:

-load > 0 : A load factor with value 1 indicates that the worker node is overloaded. A load factor of 100 denotes a free and not-loaded node.

-load = 0 : A load factor of value 0 indicates that the worker node is in a standby mode. This means that no session requests will be routed to this node until and unless the other worker nodes are unavailable

-load = -1 : A load factor of value -1 indicates that the worker node is in an error state.

-load = -2 : A load factor of value -2 indicates that the worker node is undergoing CPing/CPong and is in a transition state

[Report a bug](#)

CHAPTER 11. CDI

11.1. OVERVIEW OF CDI

11.1.1. Overview of CDI

- [Section 11.1.2, “About Contexts and Dependency Injection \(CDI\)”](#)
- [Section 11.1.5, “Relationship Between Weld, Seam 2, and JavaServer Faces”](#)
- [Section 11.1.3, “Benefits of CDI”](#)

[Report a bug](#)

11.1.2. About Contexts and Dependency Injection (CDI)

Contexts and Dependency Injection (CDI) is a specification designed to enable EJB 3.0 components "to be used as Java Server Faces (JSF) managed beans, unifying the two component models and enabling a considerable simplification to the programming model for web-based applications in Java." The preceding quote is taken from the JSR-299 specification, which can be found at <http://www.jcp.org/en/jsr/detail?id=299>.

JBoss EAP 6 includes Weld, which is the reference implementation of JSR-299. For more information, about type-safe dependency injection, see [Section 11.1.4, “About Type-safe Dependency Injection”](#).

[Report a bug](#)

11.1.3. Benefits of CDI

Following are the benefits of CDI:

- It simplifies and shrinks your code base by replacing big chunks of code with annotations.
- It is flexible, allowing you to disable and enable injections and events, use alternative beans, and inject non-CDI objects easily.
- It is easy to use your old code with CDI. You only need to include a **beans.xml** in your **META-INF/** or **WEB-INF/** directory. The file can be empty.
- It simplifies packaging and deployments and reduces the amount of XML you need to add to your deployments.
- It provides lifecycle management via contexts. You can tie injections to requests, sessions, conversations, or custom contexts.
- It also provides type-safe dependency injection, which is safer and easier to debug than string-based injection.
- It decouples interceptors from beans.
- It provides complex event notification.

[Report a bug](#)

11.1.4. About Type-safe Dependency Injection

Before JSR-299 and CDI, the only way to inject dependencies in Java was to use strings. This was prone to errors. CDI introduces the ability to inject dependencies in a type-safe way.

For more information about CDI, refer to [Section 11.1.2, “About Contexts and Dependency Injection \(CDI\)”](#).

[Report a bug](#)

11.1.5. Relationship Between Weld, Seam 2, and JavaServer Faces

The goal of *Seam 2* was to unify Enterprise Java Beans (EJBs) and JavaServer Faces (JSF) managed beans.

JavaServer Faces (JSF) implements JSR-314. It is an API for building server-side user interfaces. *JBoss Web Framework Kit* includes *RichFaces*, which is an implementation of JavaServer Faces and AJAX.

Weld is the reference implementation of *Contexts and Dependency Injection (CDI)*, which is defined in JSR-299. Weld was inspired by Seam 2 and other dependency injection frameworks. Weld is included in JBoss EAP 6.

[Report a bug](#)

11.2. USE CDI

11.2.1. First Steps

11.2.1.1. Enable CDI

Summary

Contexts and Dependency Injection (CDI) is one of the core technologies in JBoss EAP 6, and is enabled by default. If for some reason it is disabled and you need to enable it, follow this procedure.

Procedure 11.1. Enable CDI in JBoss EAP 6

1. **Check to see if the CDI subsystem details are commented out of the configuration file.**

A subsystem can be disabled by commenting out the relevant section of the **domain.xml** or **standalone.xml** configuration files, or by removing the relevant section altogether.

To find the CDI subsystem in **EAP_HOME/domain/configuration/domain.xml** or **EAP_HOME/standalone/configuration/standalone.xml**, search them for the following string. If it exists, it is located inside the <extensions> section.

```
<extension module="org.jboss.as.weld"/>
```

The following line must also be present in the profile you are using. Profiles are in individual <profile> elements within the <profiles> section.

```
<subsystem xmlns="urn:jboss:domain:weld:1.0"/>
```

2. **Before editing any files, stop JBoss EAP 6.**

JBoss EAP 6 modifies the configuration files during the time it is running, so you must stop the server before you edit the configuration files directly.

3. Edit the configuration file to restore the CDI subsystem.

If the CDI subsystem was commented out, remove the comments.

If it was removed entirely, restore it by adding this line to the file in a new line directly above the `</extensions>` tag:

```
<extension module="org.jboss.as.weld"/>
```

4. You also need to add the following line to the relevant profile in the `<profiles>` section.

```
<subsystem xmlns="urn:jboss:domain:weld:1.0"/>
```

5. Restart JBoss EAP 6.

Start JBoss EAP 6 with your updated configuration.

Result

JBoss EAP 6 starts with the CDI subsystem enabled.

[Report a bug](#)

11.2.2. Use CDI to Develop an Application

11.2.2.1. Use CDI to Develop an Application

Introduction

Contexts and Dependency Injection (CDI) gives you tremendous flexibility in developing applications, reusing code, adapting your code at deployment or run-time, and unit testing. JBoss EAP 6 includes Weld, the reference implementation of CDI. These tasks show you how to use CDI in your enterprise applications.

- [Section 11.2.1.1, “Enable CDI”](#)
- [Section 11.2.2.2, “Use CDI with Existing Code”](#)
- [Section 11.2.2.3, “Exclude Beans From the Scanning Process”](#)
- [Section 11.2.2.4, “Use an Injection to Extend an Implementation”](#)
- [Section 11.2.3.3, “Use a Qualifier to Resolve an Ambiguous Injection”](#)
- [Section 11.2.7.4, “Override an Injection with an Alternative”](#)
- [Section 11.2.7.2, “Use Named Beans”](#)
- [Section 11.2.6.1, “Manage the Lifecycle of a Bean”](#)
- [Section 11.2.6.2, “Use a Producer Method”](#)
- [Section 11.2.10.2, “Use Interceptors with CDI”](#)
- [Section 11.2.8.2, “Use Stereotypes”](#)

- [Section 11.2.9.3, “Fire and Observe Events”](#)

[Report a bug](#)

11.2.2.2. Use CDI with Existing Code

Almost every concrete Java class that has a constructor with no parameters, or a constructor designated with the annotation `@Inject`, is a bean. The only thing you need to do before you can start injecting beans is create a file called **beans.xml** in the **META-INF/** or **WEB-INF/** directory of your archive. The file can be empty.

Procedure 11.2. Use legacy beans in CDI applications

1. **Package your beans into an archive.**

Package your beans into a JAR or WAR archive.

2. **Include a beans.xml file in your archive.**

Place a **beans.xml** file into your JAR archive's **META-INF/** or your WAR archive's **WEB-INF/** directory. The file can be empty.

Result:

You can use these beans with CDI. The container can create and destroy instances of your beans and associate them with a designated context, inject them into other beans, use them in EL expressions, specialize them with qualifier annotations, and add interceptors and decorators to them, without any modifications to your existing code. In some circumstances, you may need to add some annotations.

[Report a bug](#)

11.2.2.3. Exclude Beans From the Scanning Process

Summary

One of the features of Weld, the JBoss EAP 6 implementation of CDI, is the ability to exclude classes in your archive from scanning, having container lifecycle events fired, and being deployed as beans. This is not part of the JSR-299 specification.

Example 11.1. Exclude packages from your bean

The following example has several `<weld:exclude>` tags.

1. The first one excludes all Swing classes.
2. The second excludes Google Web Toolkit classes if Google Web Toolkit is not installed.
3. The third excludes classes which end in the string **Blether** (using a regular expression), if the system property `verbosity` is set to **low**.
4. The fourth excludes Java Server Faces (JSF) classes if Wicket classes are present and the `viewlayer` system property is not set.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:weld="http://jboss.org/schema/weld/beans"
       xsi:schemaLocation="
```

```

        http://java.sun.com/xml/ns/javaee
        http://docs.jboss.org/cdi/beans_1_0.xsd
        http://jboss.org/schema/weld/beans
        http://jboss.org/schema/weld/beans_1_1.xsd">

    <weld:scan>

        <!-- Don't deploy the classes for the swing app! -->
        <weld:exclude name="com.acme.swing.**" />

        <!-- Don't include GWT support if GWT is not installed -->
        <weld:exclude name="com.acme.gwt.**">
            <weld:if-class-available name="!com.google.GWT"/>
        </weld:exclude>

        <!--
            Exclude classes which end in Blether if the system property
            verbosity is set to low
            i.e.
            java ... -Dverbosity=low
        -->
        <weld:exclude pattern="^(.*)Blether$">
            <weld:if-system-property name="verbosity" value="low"/>
        </weld:exclude>

        <!--
            Don't include JSF support if Wicket classes are present,
            and the viewlayer system
            property is not set
        -->
        <weld:exclude name="com.acme.jsf.**">
            <weld:if-class-available name="org.apache.wicket.Wicket"/>
            <weld:if-system-property name="!viewlayer"/>
        </weld:exclude>
    </weld:scan>
</beans>

```

The formal specification of Weld-specific configuration options can be found at http://jboss.org/schema/weld/beans_1_1.xsd.

[Report a bug](#)

11.2.2.4. Use an Injection to Extend an Implementation

Summary

You can use an injection to add or change a feature of your existing code. This example shows you how to add a translation ability to an existing class. The translation is a hypothetical feature and the way it is implemented in the example is pseudo-code, and only provided for illustration.

The example assumes you already have a `Welcome` class, which has a method `buildPhrase`. The `buildPhrase` method takes as an argument the name of a city, and outputs a phrase like "Welcome to Boston." Your goal is to create a version of the `Welcome` class which can translate the greeting into a different language.

■

Example 11.2. Inject a Translator Bean Into the Welcome Class

The following pseudo-code injects a hypothetical **Translator** object into the **Welcome** class. The **Translator** object may be an EJB stateless bean or another type of bean, which can translate sentences from one language to another. In this instance, the **Translator** is used to translate the entire greeting, without actually modifying the original **Welcome** class at all. The **Translator** is injected before the **buildPhrase** method is implemented.

The code sample below is an example Translating Welcome class.

```
public class TranslatingWelcome extends Welcome {

    @Inject Translator translator;

    public String buildPhrase(String city) {
        return translator.translate("Welcome to " + city + "!");
    }
    ...
}
```

[Report a bug](#)

11.2.3. Ambiguous or Unsatisfied Dependencies

11.2.3.1. About Ambiguous or Unsatisfied Dependencies

Ambiguous dependencies exist when the container is unable to resolve an injection to exactly one bean.

Unsatisfied dependencies exist when the container is unable to resolve an injection to any bean at all.

The container takes the following steps to try to resolve dependencies:

1. It resolves the qualifier annotations on all beans that implement the bean type of an injection point.
2. It filters out disabled beans. Disabled beans are `@Alternative` beans which are not explicitly enabled.

In the event of an ambiguous or unsatisfied dependency, the container aborts deployment and throws an exception.

To fix an ambiguous dependency, see [Section 11.2.3.3, “Use a Qualifier to Resolve an Ambiguous Injection”](#).

[Report a bug](#)

11.2.3.2. About Qualifiers

A qualifier is an annotation which ties a bean to a bean type. It allows you to specify exactly which bean you mean to inject. Qualifiers have a retention and a target, which are defined as in the example below.

Example 11.3. Define the `@Synchronous` and `@Asynchronous` Qualifiers

```

@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Synchronous {}

```

```

@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Asynchronous {}

```

Example 11.4. Use the @Synchronous and @Asynchronous Qualifiers

```

@Synchronous
public class SynchronousPaymentProcessor implements PaymentProcessor {

    public void process(Payment payment) { ... }

}

@Asynchronous
public class AsynchronousPaymentProcessor implements PaymentProcessor {

    public void process(Payment payment) { ... }

}

```

[Report a bug](#)

11.2.3.3. Use a Qualifier to Resolve an Ambiguous Injection

Summary

This task shows an ambiguous injection and removes the ambiguity with a qualifier. Read more about ambiguous injections at [Section 11.2.3.1, “About Ambiguous or Unsatisfied Dependencies”](#).

Example 11.5. Ambiguous injection

You have two implementations of **Welcome**, one which translates and one which does not. In that situation, the injection below is ambiguous and needs to be specified to use the translating **Welcome**.

```

public class Greeter {
    private Welcome welcome;

    @Inject
    void init(Welcome welcome) {
        this.welcome = welcome;
    }
    ...
}

```

Procedure 11.3. Resolve an Ambiguous Injection with a Qualifier

1. Create a qualifier annotation called `@Translating`.

```
@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETERS})
public @interface Translating{}
```

2. Annotate your translating `Welcome` with the `@Translating` annotation.

```
@Translating
public class TranslatingWelcome extends Welcome {
    @Inject Translator translator;
    public String buildPhrase(String city) {
        return translator.translate("Welcome to " + city + "!");
    }
    ...
}
```

3. Request the translating `Welcome` in your injection.

You must request a qualified implementation explicitly, similar to the factory method pattern. The ambiguity is resolved at the injection point.

```
public class Greeter {
    private Welcome welcome;
    @Inject
    void init(@Translating Welcome welcome) {
        this.welcome = welcome;
    }
    public void welcomeVisitors() {
        System.out.println(welcome.buildPhrase("San Francisco"));
    }
}
```

Result

The `TranslatingWelcome` is used, and there is no ambiguity.

[Report a bug](#)

11.2.4. Managed Beans

11.2.4.1. About Managed Beans

Prior to Java EE 6, there was no clear definition of the term *bean* in the Java EE platform. There were several concepts referred to as beans in the Java EE specifications, including EJB beans and JSF managed beans. Third-party frameworks such as Spring and Seam introduced their own ideas of what defined a **bean**.

Java EE 6 established a common definition in the Managed Beans specification. Managed Beans are defined as container-managed objects with minimal programming restrictions, otherwise known by the acronym POJO (Plain Old Java Object). They support a small set of basic services, such as resource

injection, lifecycle callbacks and interceptors. Companion specifications, such as EJB and CDI, build on this basic model.

With very few exceptions, almost every concrete Java class that has a constructor with no parameters (or a constructor designated with the annotation **@Inject**) is a bean. This includes every JavaBean and every EJB session bean. The only requirement to enable the mentioned services in beans is that they reside in an archive (a JAR, or a Java EE module such as a WAR or EJB JAR) that contains a special marker file: **META-INF/beans.xml**.

[Report a bug](#)

11.2.4.2. Types of Classes That are Beans

A managed bean is a Java class. The basic lifecycle and semantics of a managed bean are defined by the Managed Beans specification. You can explicitly declare a managed bean by annotating the bean class **@ManagedBean**, but in CDI you do not need to. According to the specification, the CDI container treats any class that satisfies the following conditions as a managed bean:

- It is not a non-static inner class.
- It is a concrete class, or is annotated **@Decorator**.
- It is not annotated with an EJB component-defining annotation or declared as an EJB bean class in **ejb-jar.xml**.
- It does not implement interface **javax.enterprise.inject.spi.Extension**.
- It has either a constructor with no parameters, or a constructor annotated with **@Inject**.

The unrestricted set of bean types for a managed bean contains the bean class, every superclass and all interfaces it implements directly or indirectly.

If a managed bean has a public field, it must have the default scope **@Dependent**.

[Report a bug](#)

11.2.4.3. Use CDI to Inject an Object Into a Bean

When your deployment archive includes a **META-INF/beans.xml** or **WEB-INF/beans.xml** file, each object in your deployment can be injected using CDI.

This procedure introduces the main ways to inject objects into other objects.

1. **Inject an object into any part of a bean with the **@Inject** annotation.**

To obtain an instance of a class, within your bean, annotate the field with **@Inject**.

Example 11.6. Injecting a **TextTranslator** instance into a **TranslateController**

```
public class TranslateController {  
  
    @Inject TextTranslator textTranslator;  
    ...  
}
```

2. **Use your injected object's methods**

You can use your injected object's methods directly. Assume that **TextTranslator** has a method **translate**.

Example 11.7. Use your injected object's methods

```
// in TranslateController class

public void translate() {

    translation = textTranslator.translate(inputText);

}
```

3. Use injection in the constructor of a bean

You can inject objects into the constructor of a bean, as an alternative to using a factory or service locator to create them.

Example 11.8. Using injection in the constructor of a bean

```
public class TextTranslator {

    private SentenceParser sentenceParser;

    private Translator sentenceTranslator;

    @Inject

    TextTranslator(SentenceParser sentenceParser, Translator
sentenceTranslator) {

        this.sentenceParser = sentenceParser;

        this.sentenceTranslator = sentenceTranslator;

    }

    // Methods of the TextTranslator class
    ...

}
```

4. Use the **Instance(<T>)** interface to get instances programmatically.

The **Instance** interface can return an instance of **TextTranslator** when parameterized with the bean type.

Example 11.9. Obtaining an instance programmatically

```
@Inject Instance<TextTranslator> textTranslatorInstance;

...
```

```
public void translate() {  
    textTranslatorInstance.get().translate(inputText);  
}
```

Result:

When you inject an object into a bean all of the object's methods and properties are available to your bean. If you inject into your bean's constructor, instances of the injected objects are created when your bean's constructor is called, unless the injection refers to an instance which already exists. For instance, a new instance would not be created if you inject a session-scoped bean during the lifetime of the session.

[Report a bug](#)

11.2.5. Contexts, Scopes, and Dependencies

11.2.5.1. Contexts and Scopes

A context, in terms of CDI, is a storage area which holds instances of beans associated with a specific scope.

A scope is the link between a bean and a context. A scope/context combination may have a specific lifecycle. Several pre-defined scopes exist, and you can create your own scopes. Examples of pre-defined scopes are **@RequestScoped**, **@SessionScoped**, and **@ConversationScope**.

[Report a bug](#)

11.2.5.2. Available Contexts

Table 11.1. Available contexts

Context	Description
@Dependent	The bean is bound to the lifecycle of the bean holding the reference.
@ApplicationScoped	Bound to the lifecycle of the application.
@RequestScoped	Bound to the lifecycle of the request.
@SessionScoped	Bound to the lifecycle of the session.
@ConversationScoped	Bound to the lifecycle of the conversation. The conversation scope is between the lengths of the request and the session, and is controlled by the application.

Context	Description
Custom scopes	If the above contexts do not meet your needs, you can define custom scopes.

[Report a bug](#)

11.2.6. Bean Lifecycle

11.2.6.1. Manage the Lifecycle of a Bean

Summary

This task shows you how to save a bean for the life of a request. Several other scopes exist, and you can define your own scopes.

The default scope for an injected bean is **@Dependent**. This means that the bean's lifecycle is dependent upon the lifecycle of the bean which holds the reference. For more information, see [Section 11.2.5.1, “Contexts and Scopes”](#).

Procedure 11.4. Manage Bean Lifecycles

1. **Annotate the bean with the scope corresponding to your desired scope.**

```
@RequestScoped
@Named("greeter")
public class GreeterBean {
    private Welcome welcome;
    private String city; // getter & setter not shown
    @Inject void init(Welcome welcome) {
        this.welcome = welcome;
    }
    public void welcomeVisitors() {
        System.out.println(welcome.buildPhrase(city));
    }
}
```

2. **When your bean is used in the JSF view, it holds state.**

```
<h:form>
    <h:inputText value="#{greeter.city}"/>
    <h:commandButton value="Welcome visitors" action="#"
        {greeter.welcomeVisitors}"/>
</h:form>
```

Result:

Your bean is saved in the context relating to the scope that you specify, and lasts as long as the scope applies.

- [Section 11.2.13.1, “About Bean Proxies”](#)
- [Section 11.2.13.2, “Use a Proxy in an Injection”](#)

[Report a bug](#)

11.2.6.2. Use a Producer Method

Summary

This task shows how to use producer methods to produce a variety of different objects which are not beans for injection.

Example 11.10. Use a producer method instead of an alternative, to allow polymorphism after deployment

The **@Preferred** annotation in the example is a qualifier annotation. For more information about qualifiers, refer to: [Section 11.2.3.2, “About Qualifiers”](#).

```
@SessionScoped
public class Preferences implements Serializable {
    private PaymentStrategyType paymentStrategy;
    ...
    @Produces @Preferred
    public PaymentStrategy getPaymentStrategy() {
        switch (paymentStrategy) {
            case CREDIT_CARD: return new CreditCardPaymentStrategy();
            case CHECK: return new CheckPaymentStrategy();
            default: return null;
        }
    }
}
```

The following injection point has the same type and qualifier annotations as the producer method, so it resolves to the producer method using the usual CDI injection rules. The producer method is called by the container to obtain an instance to service this injection point.

```
@Inject @Preferred PaymentStrategy paymentStrategy;
```

Example 11.11. Assign a scope to a producer method

The default scope of a producer method is **@Dependent**. If you assign a scope to a bean, it is bound to the appropriate context. The producer method in this example is only called once per session.

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy() {
    ...
}
```

Example 11.12. Use an injection inside a producer method

Objects instantiated directly by an application cannot take advantage of dependency injection and do not have interceptors. However, you can use dependency injection into the producer method to obtain bean instances.

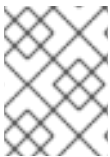
```
@Produces @Preferred @SessionScoped
```

```

public PaymentStrategy getPaymentStrategy(CreditCardPaymentStrategy
ccps,
                                           CheckPaymentStrategy cps )
{
    switch (paymentStrategy) {
        case CREDIT_CARD: return ccps;
        case CHEQUE: return cps;
        default: return null;
    }
}

```

If you inject a request-scoped bean into a session-scoped producer, the producer method promotes the current request-scoped instance into session scope. This is almost certainly not the desired behavior, so use caution when you use a producer method in this way.



NOTE

The scope of the producer method is not inherited from the bean that declares the producer method.

Result

Producer methods allow you to inject non-bean objects and change your code dynamically.

[Report a bug](#)

11.2.7. Named Beans and Alternative Beans

11.2.7.1. About Named Beans

A bean is named by using the **@Named** annotation. Naming a bean allows you to use it directly in Java Server Faces (JSF).

The **@Named** annotation takes an optional parameter, which is the bean name. If this parameter is omitted, the lower-cased bean name is used as the name.

[Report a bug](#)

11.2.7.2. Use Named Beans

1. Use the **@Named** annotation to assign a name to a bean.

```

@Named("greeter")
public class GreeterBean {
    private Welcome welcome;

    @Inject
    void init (Welcome welcome) {
        this.welcome = welcome;
    }

    public void welcomeVisitors() {

```

```

        System.out.println(welcome.buildPhrase("San Francisco"));
    }
}

```

The bean name itself is optional. If it is omitted, the bean is named after the class name, with the first letter decapitalized. In the example above, the default name would be **greeterBean**.

2. Use the named bean in a JSF view.

```

<h:form>
    <h:commandButton value="Welcome visitors" action="#"
        {greeter.welcomeVisitors}"/>
</h:form>

```

Result:

Your named bean is assigned as an action to the control in your JSF view, with a minimum of coding.

[Report a bug](#)

11.2.7.3. About Alternative Beans

Alternatives are beans whose implementation is specific to a particular client module or deployment scenario.

Example 11.13. Defining Alternatives

This alternative defines a mock implementation of both `@Synchronous PaymentProcessor` and `@Asynchronous PaymentProcessor`, all in one:

```

@Alternative @Synchronous @Asynchronous

public class MockPaymentProcessor implements PaymentProcessor {

    public void process(Payment payment) { ... }

}

```

By default, `@Alternative` beans are disabled. They are enabled for a specific bean archive by editing its `beans.xml` file.

[Report a bug](#)

11.2.7.4. Override an Injection with an Alternative

Summary

Alternative beans let you override existing beans. They can be thought of as a way to plug in a class which fills the same role, but functions differently. They are disabled by default. This task shows you how to specify and enable an alternative.

Procedure 11.5. Override an Injection

This task assumes that you already have a **TranslatingWelcome** class in your project, but you want to override it with a "mock" **TranslatingWelcome** class. This would be the case for a test deployment, where the true **Translator** bean cannot be used.

1. Define the alternative.

```
@Alternative
@Translating
public class MockTranslatingWelcome extends Welcome {
    public String buildPhrase(string city) {
        return "Bienvenue Ã " + city + "!";
    }
}
```

2. Substitute the alternative.

To activate the substitute implementation, add the fully-qualified class name to your **META-INF/beans.xml** or **WEB-INF/beans.xml** file.

```
<beans>
  <alternatives>
    <class>com.acme.MockTranslatingWelcome</class>
  </alternatives>
</beans>
```

Result

The alternative implementation is now used instead of the original one.

[Report a bug](#)

11.2.8. Stereotypes

11.2.8.1. About Stereotypes

In many systems, use of architectural patterns produces a set of recurring bean roles. A stereotype allows you to identify such a role and declare some common metadata for beans with that role in a central place.

A stereotype encapsulates any combination of:

- default scope
- a set of interceptor bindings

A stereotype may also specify either of these two scenarios:

- all beans with the stereotype have defaulted bean EL names
- all beans with the stereotype are alternatives

A bean may declare zero, one or multiple stereotypes. Stereotype annotations may be applied to a bean class or producer method or field.

A stereotype is an annotation, annotated **@Stereotype**, that packages several other annotations.

A class that inherits a scope from a stereotype may override that stereotype and specify a scope directly on the bean.

In addition, if a stereotype has a **@Named** annotation, any bean it is placed on has a default bean name. The bean may override this name if the **@Named** annotation is specified directly on the bean. For more information about named beans, see [Section 11.2.7.1, “About Named Beans”](#).

[Report a bug](#)

11.2.8.2. Use Stereotypes

Summary

Without stereotypes, annotations can become cluttered. This task shows you how to use stereotypes to reduce the clutter and streamline your code. For more information about what stereotypes are, see [Section 11.2.8.1, “About Stereotypes”](#).

Example 11.14. Annotation clutter

```
@Secure
@Transactional
@RequestScoped
@Named
public class AccountManager {
    public boolean transfer(Account a, Account b) {
        ...
    }
}
```

Procedure 11.6. Define and Use Stereotypes

1. Define the stereotype,

```
@Secure
@Transactional
@RequestScoped
@Named
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
public @interface BusinessComponent {
    ...
}
```

2. Use the stereotype.

```
@BusinessComponent
public class AccountManager {
    public boolean transfer(Account a, Account b) {
        ...
    }
}
```


Result:

Stereotypes streamline and simplify your code.

[Report a bug](#)

11.2.9. Observer Methods

11.2.9.1. About Observer Methods

Observer methods receive notifications when events occur.

CDI also provides *transactional observer methods*, which receive event notifications during the *before completion* or *after completion* phase of the transaction in which the event was fired.

[Report a bug](#)

11.2.9.2. Transactional Observers

Transactional observers receive the event notifications before or after the completion phase of the transaction in which the event was raised. For example, the following observer method refreshes a query result set cached in the application context, but only when transactions that update the Category tree are successful:

```
public void refreshCategoryTree(@Observes(during = AFTER_SUCCESS)
CategoryUpdateEvent event) { ... }
```

There are five kinds of transactional observers:

- **IN_PROGRESS**: By default, observers are invoked immediately.
- **AFTER_SUCCESS**: Observers are invoked after the completion phase of the transaction, but only if the transaction completes successfully.
- **AFTER_FAILURE**: Observers are invoked after the completion phase of the transaction only if the transaction fails to complete successfully.
- **AFTER_COMPLETION**: Observers are invoked after the completion phase of the transaction.
- **BEFORE_COMPLETION**: Observers are invoked before the completion phase of the transaction.

Transactional observers are important in a stateful object model because state is often held for longer than a single atomic transaction.

Assume we have cached a JPA query result set in the application scope:

```
import javax.ejb.Singleton;
import javax.enterprise.inject.Produces;

@ApplicationScoped @Singleton

public class Catalog {
    @PersistenceContext EntityManager em;
    List<Product> products;
```

```

    @Produces @Catalog
    List<Product> getCatalog() {
        if (products==null) {
            products = em.createQuery("select p from Product p where
p.deleted = false")
                .getResultList();
        }
        return products;
    }
}

```

Occasionally a Product is created or deleted. When this occurs, we need to refresh the Product catalog. But we have to wait for the transaction to complete successfully before performing this refresh.

The bean that creates and deletes Products triggers events, for example:

```

import javax.enterprise.event.Event;

@Stateless

public class ProductManager {
    @PersistenceContext EntityManager em;
    @Inject @Any Event<Product> productEvent;
    public void delete(Product product) {
        em.delete(product);
        productEvent.select(new AnnotationLiteral<Deleted>()
{}).fire(product);
    }

    public void persist(Product product) {
        em.persist(product);
        productEvent.select(new AnnotationLiteral<Created>()
{}).fire(product);
    }
    ...
}

```

The Catalog can now observe the events after successful completion of the transaction:

```

import javax.ejb.Singleton;

@ApplicationScoped @Singleton
public class Catalog {
    ...
    void addProduct(@Observes(during = AFTER_SUCCESS) @Created Product
product) {
        products.add(product);
    }

    void removeProduct(@Observes(during = AFTER_SUCCESS) @Deleted Product
product) {
        products.remove(product);
    }
}

```

}

[Report a bug](#)

11.2.9.3. Fire and Observe Events

Example 11.15. Fire an event

This code shows an event being injected and used in a method.

```

public class AccountManager {
    @Inject Event<Withdrawal> event;

    public boolean transfer(Account a, Account b) {
        ...
        event.fire(new Withdrawal(a));
    }
}

```

Example 11.16. Fire an event with a qualifier

You can annotate your event injection with a qualifier, to make it more specific. For more information about qualifiers, see [Section 11.2.3.2, “About Qualifiers”](#).

```

public class AccountManager {
    @Inject @Suspicious Event <Withdrawal> event;

    public boolean transfer(Account a, Account b) {
        ...
        event.fire(new Withdrawal(a));
    }
}

```

Example 11.17. Observe an event

To observe an event, use the **@Observes** annotation.

```

public class AccountObserver {
    void checkTran(@Observes Withdrawal w) {
        ...
    }
}

```

Example 11.18. Observe a qualified event

You can use qualifiers to observe only specific types of events. For more information about qualifiers, see [Section 11.2.3.2, “About Qualifiers”](#).

```

public class AccountObserver {

```

```

    void checkTran(@Observes @Suspicious Withdrawal w) {
        ...
    }
}

```

[Report a bug](#)

11.2.10. Interceptors

11.2.10.1. About Interceptors

Interceptors are defined as part of the Enterprise JavaBeans specification, which can be found at <http://jcp.org/aboutJava/communityprocess/final/jsr318/>. Interceptors allow you to add functionality to the business methods of a bean without modifying the bean's method directly. The interceptor is executed before any of the business methods of the bean.

CDI enhances this functionality by allowing you to use annotations to bind interceptors to beans.

Interception points

business method interception

A business method interceptor applies to invocations of methods of the bean by clients of the bean.

lifecycle callback interception

A lifecycle callback interceptor applies to invocations of lifecycle callbacks by the container.

timeout method interception

A timeout method interceptor applies to invocations of the EJB timeout methods by the container.

[Report a bug](#)

11.2.10.2. Use Interceptors with CDI

Example 11.19. Interceptors without CDI

Without CDI, interceptors have two problems.

- The bean must specify the interceptor implementation directly.
- Every bean in the application must specify the full set of interceptors in the correct order. This makes adding or removing interceptors on an application-wide basis time-consuming and error-prone.

```

@Interceptors({
    SecurityInterceptor.class,
    TransactionInterceptor.class,
    LoggingInterceptor.class
})
@Stateful public class BusinessComponent {

```

```
...
}
```

Procedure 11.7. Use interceptors with CDI

1. Define the interceptor binding type.

```
@InterceptorBinding
@Retention(RUNTIME)
@Target({TYPE, METHOD})
public @interface Secure {}
```

2. Mark the interceptor implementation.

```
@Secure
@Interceptor
public class SecurityInterceptor {
    @AroundInvoke
    public Object aroundInvoke(InvocationContext ctx) throws Exception
    {
        // enforce security ...
        return ctx.proceed();
    }
}
```

3. Use the interceptor in your business code.

```
@Secure
public class AccountManager {
    public boolean transfer(Account a, Account b) {
        ...
    }
}
```

4. Enable the interceptor in your deployment, by adding it to META-INF/beans.xml or WEB-INF/beans.xml.

```
<beans>
  <interceptors>
    <class>com.acme.SecurityInterceptor</class>
    <class>com.acme.TransactionInterceptor</class>
  </interceptors>
</beans>
```

The interceptors are applied in the order listed.

Result:

CDI simplifies your interceptor code and makes it easier to apply to your business code.

[Report a bug](#)

11.2.11. About Decorators

A decorator intercepts invocations from a specific Java interface, and is aware of all the semantics attached to that interface. Decorators are useful for modeling some kinds of business concerns, but do not have the generality of interceptors. They are a bean, or even an abstract class, that implements the type it decorates, and are annotated with **@Decorator**. To invoke a decorator in a CDI application, it must be specified in the **beans.xml** file.

Example 11.20. Example Decorator

```
@Decorator

public abstract class LargeTransactionDecorator
    implements Account {

    @Inject @Delegate @Any Account account;

    @PersistenceContext EntityManager em;

    public void withdraw(BigDecimal amount) {

        ...

    }

    public void deposit(BigDecimal amount);

    ...

}
}
```

A decorator must have exactly one **@Delegate** injection point to obtain a reference to the decorated object.

[Report a bug](#)

11.2.12. About Portable Extensions

CDI is intended to be a foundation for frameworks, extensions and integration with other technologies. Therefore, CDI exposes a set of SPIs for the use of developers of portable extensions to CDI. Extensions can provide the following types of functionality:

- integration with Business Process Management engines
- integration with third-party frameworks such as Spring, Seam, GWT or Wicket
- new technology based upon the CDI programming model

According to the JSR-299 specification, a portable extension may integrate with the container in the following ways:

- Providing its own beans, interceptors and decorators to the container
- Injecting dependencies into its own objects using the dependency injection service
- Providing a context implementation for a custom scope
- Augmenting or overriding the annotation-based metadata with metadata from some other source

[Report a bug](#)

11.2.13. Bean Proxies

11.2.13.1. About Bean Proxies

Clients of an injected bean do not usually hold a direct reference to a bean instance. Unless the bean is a dependent object (scope `@Dependent`), the container must redirect all injected references to the bean using a proxy object.

This bean proxy referred to as client proxy is responsible for ensuring the bean instance that receives a method invocation is the instance associated with the current context. The client proxy also allows beans bound to contexts such as the session context to be serialized to disk without recursively serializing other injected beans.

Due to Java limitations, some Java types cannot be proxied by the container. If an injection point declared with one of these types resolves to a bean with any scope other than `@Dependent`, the container aborts the deployment.

Java types that cannot be proxied by the container

- Classes which do not have a non-private constructor with no parameters
- Classes which are declared **final** or have a **final** method
- Arrays and primitive types

[Report a bug](#)

11.2.13.2. Use a Proxy in an Injection

Overview

A proxy is used for injection when the lifecycles of the beans are different from each other. The proxy is a subclass of the bean that is created at run-time, and overrides all the non-private methods of the bean class. The proxy forwards the invocation onto the actual bean instance.

In this example, the **PaymentProcessor** instance is not injected directly into **Shop**. Instead, a proxy is injected, and when the **processPayment()** method is called, the proxy looks up the current **PaymentProcessor** bean instance and calls the **processPayment()** method on it.

Example 11.21. Proxy Injection

```
@ConversationScoped
class PaymentProcessor
{
    public void processPayment(int amount)
```

```
    {  
        System.out.println("I'm taking $" + amount);  
    }  
}  
  
@ApplicationScoped  
public class Shop  
{  
  
    @Inject  
    PaymentProcessor paymentProcessor;  
  
    public void buyStuff()  
    {  
        paymentProcessor.processPayment(100);  
    }  
}
```

For more information about proxies, including which types of classes can be proxied, refer to [Section 11.2.13.1, “About Bean Proxies”](#).

[Report a bug](#)

CHAPTER 12. JAVA TRANSACTION API (JTA)

12.1. OVERVIEW

12.1.1. Overview of Java Transactions API (JTA)

Introduction

These topics provide a foundational understanding of the Java Transactions API (JTA).

- [Section 12.2.5, “About Java Transactions API \(JTA\)”](#)
- [Section 12.5.2, “Lifecycle of a JTA Transaction”](#)
- [Section 12.9.2, “JTA Transaction Example”](#)

[Report a bug](#)

12.2. TRANSACTION CONCEPTS

12.2.1. About Transactions

A transaction consists of two or more actions which must either all succeed or all fail. A successful outcome is a commit, and a failed outcome is a roll-back. In a roll-back, each member's state is reverted to its state before the transaction attempted to commit.

The typical standard for a well-designed transaction is that it is *Atomic, Consistent, Isolated, and Durable (ACID)*.

[Report a bug](#)

12.2.2. About ACID Properties for Transactions

ACID is an acronym which stands for **A**tomicity, **C**onsistency, **I**solation, and **D**urability. This terminology is usually used in the context of databases or transactional operations.

ACID Definitions

Atomicity

For a transaction to be atomic, all transaction members must make the same decision. Either they all commit, or they all roll back. If atomicity is broken, what results is termed a *heuristic outcome*.

Consistency

Consistency means that data written to the database is guaranteed to be valid data, in terms of the database schema. The database or other data source must always be in a consistent state. One example of an inconsistent state would be a field in which half of the data is written before an operation aborts. A consistent state would be if all the data were written, or the write were rolled back when it could not be completed.

Isolation

Isolation means that data being operated on by a transaction must be locked before modification, to prevent processes outside the scope of the transaction from modifying the data.

Durability

Durability means that in the event of an external failure after transaction members have been instructed to commit, all members will be able to continue committing the transaction when the failure is resolved. This failure may be related to hardware, software, network, or any other involved system.

[Report a bug](#)

12.2.3. About the Transaction Coordinator or Transaction Manager

The terms *Transaction Coordinator* and *Transaction Manager* are mostly interchangeable in terms of transactions with JBoss EAP 6. The term Transaction Coordinator is usually used in the context of distributed transactions.

In JTA transactions, The *Transaction Manager* runs within JBoss EAP 6 and communicates with transaction participants during the *two-phase commit* protocol.

The Transaction Manager tells transaction participants whether to commit or roll back their data, depending on the outcome of other transaction participants. In this way, it ensures that transactions adhere to the ACID standard.

In JTS transactions, the Transaction Coordinator manages interactions between transaction managers on different servers.

- [Section 12.2.4, “About Transaction Participants”](#)
- [Section 12.2.2, “About ACID Properties for Transactions”](#)
- [Section 12.2.9, “About the 2-Phase Commit Protocol”](#)

[Report a bug](#)

12.2.4. About Transaction Participants

A transaction participant is any process within a transaction, which has the ability to commit or roll back state. This may be a database or other application. Each participant of a transaction independently decides whether it is able to commit or roll back its state, and only if all participants can commit, does the transaction as a whole succeed. Otherwise, each participant rolls back its state, and the transaction as a whole fails. The Transaction Manager coordinates the commit or rollback operations and determines the outcome of the transaction.

- [Section 12.2.1, “About Transactions”](#)
- [Section 12.2.3, “About the Transaction Coordinator or Transaction Manager”](#)

[Report a bug](#)

12.2.5. About Java Transactions API (JTA)

Java Transactions API (JTA) is part of Java Enterprise Edition specification. It is defined in JSR-907.

Implementation of JTA is done using Transaction manager, which is covered by project Narayana for JBoss EAP application server. Transaction manager allows application to assign various resources, for example, database or JMS brokers, through a single global transaction. The global transaction is referred as XA transaction. Only resources with XA capabilities can be included in a transaction.

In this document, JTA refers to Java Transaction API, this term is used to indicate how the transaction manager processes the transactions. Transaction manager works in JTA transactions mode, the data is shared via memory and transaction context is transferred by remote EJB calls. In JTS mode, the data is shared by sending Common Object Request Broker Architecture (CORBA) messages and transaction context is transferred by IIOP calls. Both modes support distribution of transaction over multiple EAP servers.

Annotations is a method for creating and controlling transactions within your code.

- [Section 12.2.7, “About XA Datasources and XA Transactions”](#)
- [Section 12.2.11, “About Distributed Transactions”](#)
- [Section 12.8.2, “Configure the ORB for JTS Transactions”](#)

[Report a bug](#)

12.2.6. About Java Transaction Service (JTS)

Java Transaction Service (JTS) is a mapping of the Object Transaction Service (OTS) to Java. Java applications use the JTA API to manage transactions. JTA then interacts with a JTS transaction implementation when the transaction manager is switched to JTS mode. To use special JTS capabilities, for example, nested transactions, you need to manually use the JTS API.

JTS works over the IIOP protocol. Transaction managers that use JTS, communicate with each other using a process called an *Object Request Broker (ORB)*, using a communication standard called *Common Object Request Broker Architecture (CORBA)*.

Using JTA API from an application standpoint, a JTS transaction behaves in the same way as a JTA transaction.



NOTE

The implementation of JTS included in JBoss EAP 6 supports distributed transactions. The difference from fully-compliant JTS transactions is interoperability with external third-party ORBs. This feature is unsupported with JBoss EAP 6. Supported configurations distribute transactions across multiple JBoss EAP 6 containers only.

- [Section 12.2.3, “About the Transaction Coordinator or Transaction Manager”](#)

[Report a bug](#)

12.2.7. About XA Datasources and XA Transactions

An XA datasource is a datasource which can participate in an XA global transaction.

An XA transaction is a transaction which can span multiple resources. It involves a coordinating transaction manager, with one or more databases or other transactional resources, all involved in a single global transaction.

[Report a bug](#)

12.2.8. About XA Recovery

The Java Transaction API (JTA) allows distributed transactions across multiple *X/Open XA resources*.

XA stands for *Extended Architecture* which was developed by the X/Open Group to define a transaction which uses more than one back-end data store. The XA standard describes the interface between a global *Transaction Manager (TM)* and a local resource manager. XA allows multiple resources, such as application servers, databases, caches, and message queues, to participate in the same transaction, while preserving atomicity of the transaction. Atomicity means that if one of the participants fails to commit its changes, the other participants abort the transaction, and restore their state to the same status as before the transaction occurred.

XA Recovery is the process of ensuring that all resources affected by a transaction are updated or rolled back, even if any of the resources are transaction participants crash or become unavailable. Within the scope of JBoss EAP 6, the Transaction subsystem provides the mechanisms for XA Recovery to any XA resources or subsystems which use them, such as XA datasources, JMS message queues, and JCA resource adapters.

XA Recovery happens without user intervention. In the event of an XA Recovery failure, errors are recorded in the log output. Contact Red Hat Global Support Services if you need assistance.

[Report a bug](#)

12.2.9. About the 2-Phase Commit Protocol

The Two-phase commit protocol (2PC) refers to an algorithm to determine the outcome of a transaction.

Phase 1

In the first phase, the transaction participants notify the transaction coordinator whether they are able to commit the transaction or must roll back.

Phase 2

In the second phase, the transaction coordinator makes the decision about whether the overall transaction should commit or roll back. If any one of the participants cannot commit, the transaction must roll back. Otherwise, the transaction can commit. The coordinator directs the transactions about what to do, and they notify the coordinator when they have done it. At that point, the transaction is finished.

[Report a bug](#)

12.2.10. About Transaction Timeouts

In order to preserve atomicity and adhere to the ACID standard for transactions, some parts of a transaction can be long-running. Transaction participants need to lock parts of datasources when they commit, and the transaction manager needs to wait to hear back from each transaction participant before it can direct them all whether to commit or roll back. Hardware or network failures can cause resources to be locked indefinitely.

Transaction timeouts can be associated with transactions in order to control their lifecycle. If a timeout threshold passes before the transaction commits or rolls back, the timeout causes the transaction to be rolled back automatically.

You can configure default timeout values for the entire transaction subsystem, or you disable default timeout values, and specify timeouts on a per-transaction basis.

[Report a bug](#)

12.2.11. About Distributed Transactions

A *distributed transaction*, is a transaction with participants on multiple JBoss EAP 6 servers. *Java*

Transaction Service (JTS) specification mandates that JTS transactions be able to be distributed across application servers from different vendors (transaction distribution among servers from different vendors is not a supported feature). Java Transaction API (JTA) does not define that but JBoss EAP 6 supports distributed JTA transactions among JBoss EAP6 servers.



NOTE

In other app server vendor documentation, you can find that term distributed transaction means XA transaction. In context of JBoss EAP 6 documentation, the distributed transaction refers transactions distributed among several application servers. Transaction which consists from different resources (for example, database resource and jms resource) are referred as XA transactions in this document. For more information, refer [Section 12.2.6, “About Java Transaction Service \(JTS\)”](#) and [Section 12.2.7, “About XA Datasources and XA Transactions”](#).

[Report a bug](#)

12.2.12. About the ORB Portability API

The Object Request Broker (ORB) is a process which sends and receives messages to transaction participants, coordinators, resources, and other services distributed across multiple application servers. An ORB uses a standardized Interface Description Language (IDL) to communicate and interpret messages. *Common Object Request Broker Architecture (CORBA)* is the IDL used by the ORB in JBoss EAP 6.

The main type of service which uses an ORB is a system of distributed Java Transactions, using the Java Transaction Service (JTS) protocol. Other systems, especially legacy systems, may choose to use an ORB for communication, rather than other mechanisms such as remote Enterprise JavaBeans or JAX-WS or JAX-RS Web Services.

The ORB Portability API provides mechanisms to interact with an ORB. This API provides methods for obtaining a reference to the ORB, as well as placing an application into a mode where it listens for incoming connections from an ORB. Some of the methods in the API are not supported by all ORBs. In those cases, an exception is thrown.

The API consists of two different classes:

ORB Portability API Classes

- `com.arjuna.orbportability.orb`
- `com.arjuna.orbportability.oa`

Refer to the JBoss EAP 6 Javadocs bundle from the [Red Hat Customer Portal](#) for specific details about the methods and properties included in the ORB Portability API.

[Report a bug](#)

12.2.13. About Nested Transactions

Nested transactions are transactions where some participants are also transactions.

Benefits of Nested Transactions

Fault Isolation

If a subtransaction rolls back, perhaps because an object it is using fails, the enclosing transaction does not need to roll back.

Modularity

If a transaction is already associated with a call when a new transaction begins, the new transaction is nested within it. Therefore, if you know that an object requires transactions, you can create them within the object. If the object's methods are invoked without a client transaction, then the object's transactions are top-level. Otherwise, they are nested within the scope of the client's transactions. Likewise, a client does not need to know whether an object is transactional. It can begin its own transaction.

Nested Transactions are only supported as part of the Java Transaction Service (JTS) API, and not part of the Java Transaction API (JTA). Attempting to nest (non-distributed) JTA transactions results in an exception.

Modifying JBoss EAP 6 configuration of transaction subsystem to use JTS does not indicate that nested transaction will be used or activated. If you need to use them, you have to directly use ORB API as JTA API does not provide any method to start the nested transaction.

[Report a bug](#)

12.2.14. About XML Transaction Service

The XML Transaction Service (XTS) component supports the coordination of private and public Web Services in a business transaction. Using XTS, you can coordinate complex business transactions in a controlled and reliable manner. The XTS API supports a transactional coordination model based on the WS-Coordination, WS-Atomic Transaction, and WS-Business Activity protocols.

- [Section 12.2.14.1, “Overview of Protocols Used by XTS”](#)
- [Section 12.2.14.2, “Web Services-Atomic Transaction Process”](#)
- [Section 12.2.14.3, “Web Services-Business Activity Process”](#)
- [Section 12.2.14.4, “Transaction Bridging Overview”](#)

[Report a bug](#)

12.2.14.1. Overview of Protocols Used by XTS

This topic describes the fundamental concepts associated with the WS-Coordination (WS-C), WS-Atomic Transaction (WS-AT) and WS-Business Activity (WS-BA) protocols, as defined in the specifications of each protocol.

The WS-C specification defines a framework that allows different coordination protocols to be plugged in to coordinate work between clients, services, and participants.

WS-T protocol comprises the pair of transaction coordination protocols, WS-Atomic Transaction (WS-AT) and WS-Business Activity (WS-BA) , which utilize the coordination framework provided by WS-C. WS-T is developed to unify existing traditional transaction processing systems, allowing them to communicate reliably with one another.

- [Section 12.2.14.2, “Web Services-Atomic Transaction Process”](#)

- [Section 12.2.14.3, “Web Services-Business Activity Process”](#)

[Report a bug](#)

12.2.14.2. Web Services-Atomic Transaction Process

An atomic transaction (AT) is designed to support short duration interactions where ACID semantics are appropriate. Within the scope of an AT, Web Services typically employ bridging to access XA resources, such as databases and message queues, under the control of the WS-T. When the transaction terminates, the participant propagates the outcome decision of the AT to the XA resources, and the appropriate commit or rollback actions are taken by each participant.

Atomic Transaction Process

1. To initiate an AT, the client application first locates a WS-C Activation Coordinator Web Service that supports WS-T.
2. The client sends a WS-C **CreateCoordinationContext** message to the service, specifying <http://schemas.xmlsoap.org/ws/2004/10/wsat> as its coordination type.
3. The client receives an appropriate WS-T context from the activation service.
4. The response to the **CreateCoordinationContext** message, the transaction context, has its **CoordinationType** element set to the WS-AT namespace, <http://schemas.xmlsoap.org/ws/2004/10/wsat>. It also contains a reference to the atomic transaction coordinator endpoint, the WS-C Registration Service, where participants can be enlisted.
5. The client normally proceeds to invoke Web Services and complete the transaction, either committing all the changes made by the Web Services, or rolling them back. In order to be able to drive this completion, the client must register itself as a participant for the Completion protocol, by sending a register message to the Registration Service whose endpoint was returned in the Coordination Context.
6. Once registered for completion, the client application then interacts with Web Services to accomplish its business-level work. With each invocation of a business Web Service, the client inserts the transaction context into a SOAP header block, such that each invocation is implicitly scoped by the transaction. The toolkits that support WS-AT aware Web Services provide facilities to correlate contexts found in SOAP header blocks with back-end operations. This ensures that modifications made by the Web Service are done within the scope of the same transaction as the client and subject to commit or rollback by the Transaction Coordinator.
7. Once all the necessary application work is complete, the client can terminate the transaction, with the intent of making any changes to the service state permanent. The completion participant instructs the coordinator to try to commit or roll back the transaction. When the commit or rollback operation completes, a status is returned to the participant to indicate the outcome of the transaction.

For more details, see [Web Services-Transaction Documentation](#).

[Report a bug](#)

12.2.14.3. Web Services-Business Activity Process

Web Services-Business Activity (WS-BA) defines a protocol for Web Services based applications to enable existing business processing and workflow systems to wrap their proprietary mechanisms and interoperate across implementations and business boundaries.

Unlike the WS-AT protocol model, where participants inform the transaction coordinator of their state only when asked, a child activity within a BA can specify its outcome to the coordinator directly, without waiting for a request. A participant may choose to exit the activity or notify the coordinator of a failure at any point. This feature is useful when tasks fail because the notification can be used to modify the goals and drive processing forward, without waiting until the end of the transaction to identify failures.

WS-BA Process

1. Services are requested to do work.
2. Wherever these services have the ability to undo any work, they inform the BA, in case the BA later decides to cancel the work. If the BA suffers a failure, it can instruct the service to execute its undo behavior.

The BA protocols employ a compensation-based transaction model. When a participant in a business activity completes its work, it may choose to exit the activity. This choice does not allow any subsequent rollback. Alternatively, the participant can complete its activity, signaling to the coordinator that the work it has done can be compensated if, at some later point, another participant notifies a failure to the coordinator. In this latter case, the coordinator asks each non-exited participant to compensate for the failure, giving them the opportunity to execute whatever compensating action they consider appropriate. If all participants exit or complete without failure, the coordinator notifies each completed participant that the activity has been closed.

For more details, see [Web Services-Transaction Documentation](#).

[Report a bug](#)

12.2.14.4. Transaction Bridging Overview

Transaction Bridging describes the process of linking the Java EE and WS-T domains. The transaction bridge component **txbridge** provides bi-directional linkage, such that either type of transaction may encompass business logic designed for use with the other type. The technique used by the bridge is a combination of interposition and protocol mapping.

In the transaction bridge, an interposed coordinator is registered into the existing transaction and performs the additional task of protocol mapping; that is, it appears to its parent coordinator to be a resource of its native transaction type, whilst appearing to its children to be a coordinator of their native transaction type, even though these transaction types differ.

The transaction bridge resides in the package **org.jboss.jbossts.txbridge** and its sub-packages. It consists of two distinct sets of classes, one for bridging in each direction.

For more details, see [Transaction Bridge Documentation](#).

[Report a bug](#)

12.3. TRANSACTION OPTIMIZATIONS

12.3.1. Overview of Transaction Optimizations

Introduction

The Transactions subsystem of JBoss EAP 6 includes several optimizations which you can take advantage of in your applications.

- [Section 12.3.2, “About the LRCO Optimization for Single-phase Commit \(1PC\)”](#)
- [Section 12.3.3, “About the Presumed-Abort Optimization”](#)
- [Section 12.3.4, “About the Read-Only Optimization”](#)

[Report a bug](#)

12.3.2. About the LRCO Optimization for Single-phase Commit (1PC)

Although the 2-phase commit protocol (2PC) is more commonly encountered with transactions, some situations do not require, or cannot accommodate, both phases. In these cases, you can use the *single phase commit (1PC)* protocol. One situation where this might happen is when a non-XA-aware datasource needs to participate in the transaction.

In these situations, an optimization known as the *Last Resource Commit Optimization (LRCO)* is employed. The single-phase resource is processed last in the prepare phase of the transaction, and an attempt is made to commit it. If the commit succeeds, the transaction log is written and the remaining resources go through the 2PC. If the last resource fails to commit, the transaction is rolled back.

While this protocol allows for most transactions to complete normally, certain types of error can cause an inconsistent transaction outcome. Therefore, use this approach only as a last resort.

Where a single local TX datasource is used in a transaction, the LRCO is automatically applied to it.

- [Section 12.2.9, “About the 2-Phase Commit Protocol”](#)

[Report a bug](#)

12.3.2.1. Commit Markable Resource

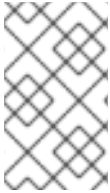
Summary

Configuring access to a resource manager via the Commit Markable Resource (CMR) interface ensures that a 1PC resource manager can be reliably enlisted in a 2PC transaction. It is an implementation of the LRCO algorithm, which makes non-XA resource fully recoverable.

Previously, adding 1PC resources to a 2PC transaction was achieved via the LRCO method, however there is a window of failure in LRCO. Following the procedure below for adding 1PC resources to a 2PC transaction via the LRCO method:

1. Prepare 2PC
2. Commit LRCO
3. Write tx log
4. Commit 2PC

If the procedure crashes between steps 2 and step 3, you cannot commit the 2PC. CMR eliminates this restriction and allows 1PC to be reliably enlisted in a 2PC transaction.



NOTE

Use the **exception-sorter** parameter in the datasource configuration. You can follow the datasource configuration examples mentioned in the JBoss EAP *Administration and Configuration Guide*.

Restrictions

A transaction may contain only one CMR resource.

Prerequisites

You must have a table created for which the following SQL would work:

```
SELECT xid,actionuid FROM _tableName_ WHERE transactionManagerID IN
(String[])
DELETE FROM _tableName_ WHERE xid IN (byte[])
INSERT INTO _tableName_ (xid, transactionManagerID, actionuid) VALUES
(byte[],String,byte[])
```

Example 12.1. Some examples of the SQL query

Sybase:

```
CREATE TABLE xids (xid varbinary(144), transactionManagerID varchar(64),
actionuid varbinary(28))
```

Oracle:

```
CREATE TABLE xids (xid RAW(144), transactionManagerID varchar(64),
actionuid RAW(28))
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

IBM:

```
CREATE TABLE xids (xid VARCHAR(255) for bit data not null,
transactionManagerID
varchar(64), actionuid VARCHAR(255) for bit data not null)
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

SQL Server:

```
CREATE TABLE xids (xid varbinary(144), transactionManagerID varchar(64),
actionuid varbinary(28))
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

Postgres:

```
CREATE TABLE xids (xid bytea, transactionManagerID varchar(64),
actionuid bytea)
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

Enabling a resource manager as CMR

By default, the CMR feature is disabled for datasources. To enable it, you must create or modify the datasource configuration and ensure that the connectable attribute is set to true. An example configuration entry in the datasources section of a server xml configuration file could be as follows:

```
<datasource enabled="true" jndi-
name="java:jboss/datasources/ConnectableDS" pool-name="ConnectableDS"
jta="true" use-java-context="true" spy="false" use-ccm="true"
connectable="true"/>
```



NOTE

This feature is not applicable to XA datasources.

You can also enable a resource manager as CMR using CLI as follows:

```
/subsystem=datasources/data-source=ConnectableDS:add(enabled="true", jndi-
name="java:jboss/datasources/ConnectableDS", jta="true", use-java-
context="true", spy="false", use-ccm="true", connectable="true",
connection-url="validConnectionURL", exception-
sorter="org.jboss.jca.adapters.jdbc.extensions.mssql.MSQLExceptionSorter"
, driver-name="h2")
```

Updating an existing resource to use the new CMR feature

If you only need to update an existing resource to use the new CMR feature, then simply modify the connectable attribute:

```
/subsystem=datasources/data-source=ConnectableDS:write-
attribute(name=connectable,value=true)
```

Identifying CMR capable datasources

The transaction subsystem identifies the datasources that are CMR capable through an entry to the transaction subsystem config section as shown below:

```
<subsystem xmlns="urn:jboss:domain:transactions:3.0">
  ...
  <commit-markable-resources>
    <commit-markable-resource jndi-
name="java:jboss/datasources/ConnectableDS">
      <xid-location name="xids" batch-size="100" immediate-
cleanup="false"/>
    </commit-markable-resource>
    ...
  </commit-markable-resources>
</subsystem>
```



NOTE

You must restart the server after adding the CMR.

[Report a bug](#)

12.3.3. About the Presumed-Abort Optimization

If a transaction is going to roll back, it can record this information locally and notify all enlisted participants. This notification is only a courtesy, and has no effect on the transaction outcome. After all participants have been contacted, the information about the transaction can be removed.

If a subsequent request for the status of the transaction occurs there will be no information available. In this case, the requester assumes that the transaction has aborted and rolled back. This *presumed-abort* optimization means that no information about participants needs to be made persistent until the transaction has decided to commit, since any failure prior to this point will be assumed to be an abort of the transaction.

[Report a bug](#)

12.3.4. About the Read-Only Optimization

When a participant is asked to prepare, it can indicate to the coordinator that it has not modified any data during the transaction. Such a participant does not need to be informed about the outcome of the transaction, since the fate of the participant has no effect on the transaction. This *read-only* participant can be omitted from the second phase of the commit protocol.

[Report a bug](#)

12.4. TRANSACTION OUTCOMES

12.4.1. About Transaction Outcomes

There are three possible outcomes for a transaction.

Roll-back

If any transaction participant cannot commit, or the transaction coordinator cannot direct participants to commit, the transaction is rolled back. See [Section 12.4.3, “About Transaction Roll-Back”](#) for more information.

Commit

If every transaction participant can commit, the transaction coordinator directs them to do so. See [Section 12.4.2, “About Transaction Commit”](#) for more information.

Heuristic outcome

If some transaction participants commit and others roll back, it is termed a heuristic outcome. Heuristic outcomes require human intervention. See [Section 12.4.4, “About Heuristic Outcomes”](#) for more information.

[Report a bug](#)

12.4.2. About Transaction Commit

When a transaction participant commits, it makes its new state durable. The new state is created by the participant doing the work involved in the transaction. The most common example is when a transaction member writes records to a database.

After commit, information about the transaction is removed from the transaction coordinator, and the newly-written state is now the durable state.

[Report a bug](#)

12.4.3. About Transaction Roll-Back

A transaction participant rolls back by restoring its state to reflect the state before the transaction began. After a roll-back, the state is the same as if the transaction had never been started.

[Report a bug](#)

12.4.4. About Heuristic Outcomes

A heuristic outcome, or non-atomic outcome, is a transaction anomaly. It refers to a situation where some transaction participants committed their state, and others rolled back. A heuristic outcome causes state to be inconsistent.

Heuristic outcomes typically happen during the second phase of the 2-phase commit (2PC) protocol. They are often caused by failures to the underlying hardware or communications subsystems of the underlying servers.

There are four different types of heuristic outcome.

Heuristic rollback

The commit operation failed because some or all of the participants unilaterally rolled back the transaction.

Heuristic commit

An attempted rollback operation failed because all of the participants unilaterally committed. This may happen if, for example, the coordinator is able to successfully prepare the transaction but then decides to roll it back because of a failure on its side, such as a failure to update its log. In the interim, the participants may decide to commit.

Heuristic mixed

Some participants committed and others rolled back.

Heuristic hazard

The outcome of some of the updates is unknown. For the ones that are known, they have either all committed or all rolled back.

Heuristic outcomes can cause loss of integrity to the system, and usually require human intervention to resolve. Do not write code which relies on them.

- [Section 12.2.9, “About the 2-Phase Commit Protocol”](#)

[Report a bug](#)

12.4.5. JBoss Transactions Errors and Exceptions

For details about exceptions thrown by methods of the **UserTransaction** class, see the *UserTransaction API* specification at <http://docs.oracle.com/javaee/6/api/javax/transaction/UserTransaction.html>.

[Report a bug](#)

12.5. OVERVIEW OF JTA TRANSACTIONS

12.5.1. About Java Transactions API (JTA)

Java Transactions API (JTA) is part of Java Enterprise Edition specification. It is defined in JSR-907.

Implementation of JTA is done using Transaction manager, which is covered by project Narayana for JBoss EAP application server. Transaction manager allows application to assign various resources, for example, database or JMS brokers, through a single global transaction. The global transaction is referred as XA transaction. Only resources with XA capabilities can be included in a transaction.

In this document, JTA refers to Java Transaction API, this term is used to indicate how the transaction manager processes the transactions. Transaction manager works in JTA transactions mode, the data is shared via memory and transaction context is transferred by remote EJB calls. In JTS mode, the data is shared by sending Common Object Request Broker Architecture (CORBA) messages and transaction context is transferred by IIOP calls. Both modes support distribution of transaction over multiple EAP servers.

Annotations is a method for creating and controlling transactions within your code.

- [Section 12.2.7, “About XA Datasources and XA Transactions”](#)
- [Section 12.2.11, “About Distributed Transactions”](#)
- [Section 12.8.2, “Configure the ORB for JTS Transactions”](#)

[Report a bug](#)

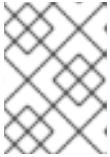
12.5.2. Lifecycle of a JTA Transaction

When a resource asks to participate in a transaction, a chain of events is set in motion. The *Transaction Manager* is a process that lives within the application server and manages transactions. *Transaction participants* are objects which participate in a transaction. *Resources* are datasources, JMS connection factories, or other JCA connections.

1. Your application starts a new transaction

To begin a transaction, your application obtains an instance of class **UserTransaction** from JNDI or, if it is an EJB, from an annotation. The **UserTransaction** interface includes methods for beginning, committing, and rolling back top-level transactions. Newly-created transactions are automatically associated with their invoking thread. Nested transactions are not supported in JTA, so all transactions are top-level transactions.

Calling **UserTransaction.begin()** using annotations starts a transaction when an EJB method is called (driven by TransactionAttribute rules). Any resource that is used after that point is associated with the transaction. If more than one resource is enlisted, your transaction becomes an XA transaction, and participates in the two-phase commit protocol at commit time.

**NOTE**

The **UserTransaction** object is used only for BMT transactions. In CMT, the UserTransaction object is not permitted.

2. **Your application modifies its state.**

In the next step, your application performs its work and makes changes to its state.

3. **Your application decides to commit or roll back**

When your application has finished changing its state, it decides whether to commit or roll back. It calls the appropriate method, either **UserTransaction.commit()** or **UserTransaction.rollback()**.

4. **The transaction manager removes the transaction from its records.**

After the commit or rollback completes, the transaction manager cleans up its records and removes information about your transaction from the transaction log.

Failure recovery

Failure recovery happens automatically. If a resource, transaction participant, or the application server become unavailable, the Transaction Manager handles recovery when the underlying failure is resolved and the resource is available again.

- [Section 12.2.1, “About Transactions”](#)
- [Section 12.2.3, “About the Transaction Coordinator or Transaction Manager”](#)
- [Section 12.2.4, “About Transaction Participants”](#)
- [Section 12.2.9, “About the 2-Phase Commit Protocol”](#)
- [Section 12.2.7, “About XA Datasources and XA Transactions”](#)

[Report a bug](#)

12.6. TRANSACTION SUBSYSTEM CONFIGURATION

12.6.1. Transactions Configuration Overview

Introduction

The following procedures show you how to configure the transactions subsystem of JBoss EAP 6.

- [Section 12.6.2.3, “Configure Your Datasource to Use JTA Transaction API”](#)
- [Section 12.6.2.1, “Configure an XA Datasource”](#)
- [Section 12.7.8.2, “Configure the Transaction Manager”](#)
- [Section 12.6.3.2, “Configure Logging for the Transaction Subsystem”](#)

[Report a bug](#)

12.6.2. Transactional Datasource Configuration

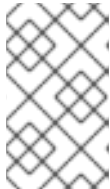
12.6.2.1. Configure an XA Datasource

Prerequisites

Log into the Management Console.

1. **Add a new datasource.**

Add a new datasource to JBoss EAP 6. Click the **XA Datasource** tab at the top.



NOTE

Refer to *Create an XA Datasource with the Management Interfaces* section of the *Administration and Configuration Guide* on the Red Hat Customer Portal for information on how to add a new datasource to JBoss EAP 6.

2. **Configure additional properties as appropriate.**

All datasource parameters are listed in [Section 12.6.2.5, “Datasource Parameters”](#).

Result

Your XA Datasource is configured and ready to use.

[Report a bug](#)

12.6.2.2. Create a Non-XA Datasource with the Management Interfaces

Summary

This topic covers the steps required to create a non-XA datasource, using either the Management Console or the Management CLI.

Prerequisites

- The JBoss EAP 6 server must be running.



NOTE

Prior to version 10.2 of the Oracle datasource, the `<no-tx-separate-pools/>` parameter was required, as mixing non-transactional and transactional connections would result in an error. This parameter may no longer be required for certain applications.



NOTE

To prevent issues such as duplication of driver listing, selected driver not available in a profile, or driver not displayed if a server for the profile is not running, in JBoss EAP 6.4 onwards, only JDBC drivers that are installed as modules and correctly referenced from profiles are detectable while creating a datasource using the Management Console in domain mode.

Procedure 12.1. Create a Datasource using either the Management CLI or the Management Console

- - **Management CLI**
 - a. Launch the CLI tool and connect to your server.

- b. Run the following Management CLI command to create a non-XA datasource, configuring the variables as appropriate:



NOTE

The value for *DRIVER_NAME* depends on the number of classes listed in the */META-INF/services/java.sql.Driver* file located in the JDBC driver JAR. If there is only one class, the value is the name of the JAR. If there are multiple classes, the value is the name of the JAR + *driverClassName* + "_" + *majorVersion* + "_" + *minorVersion*. Failure to do so will result in the following error being logged:

```
JBAS014775:      New missing/unsatisfied dependencies
```

For example, the *DRIVER_NAME* value required for the MySQL 5.1.31 driver, is **mysql-connector-java-5.1.31-bin.jarcom.mysql.jdbc.Driver_5_1**.

```
data-source add --name=DATASOURCE_NAME --jndi-name=JNDI_NAME -
-driver-name=DRIVER_NAME --connection-url=CONNECTION_URL
```

- c. Enable the datasource:

```
data-source enable --name=DATASOURCE_NAME
```

o Management Console

- a. Login to the Management Console.
- b. **Navigate to the Datasources panel in the Management Console**
 - i. Select the **Configuration** tab from the top of the console.
 - ii. For Domain mode only, select a profile from the drop-down box in the top left.
 - iii. Expand the **Subsystems** menu on the left of the console, then expand the **Connector** menu.
 - iv. Select **Datasources** from the menu on the left of the console.
- c. **Create a new datasource**
 - i. Click **Add** at the top of the **Datasources** panel.
 - ii. Enter the new datasource attributes in the **Create Datasource** wizard and proceed with the **Next** button.
 - iii. Enter the JDBC driver details in the **Create Datasource** wizard and click **Next** to continue.
 - iv. Enter the connection settings in the **Create Datasource** wizard.
 - v. Click the **Test Connection** button to test the connection to the datasource and verify the settings are correct.

- vi. Click **Done** to finish

Result

The non-XA datasource has been added to the server. It is now visible in either the **standalone.xml** or **domain.xml** file, as well as the management interfaces.

[Report a bug](#)

12.6.2.3. Configure Your Datasource to Use JTA Transaction API

Summary

This task shows you how to enable Java Transaction API (JTA) on your datasource.

Prerequisites

You must meet the following conditions before continuing with this task:

- Your database or other resource must support Java Transaction API. If in doubt, consult the documentation for your database or other resource.
- Create a datasource. Refer to [Section 12.6.2.2, “Create a Non-XA Datasource with the Management Interfaces”](#).
- Stop JBoss EAP 6.
- Have access to edit the configuration files directly, in a text editor.

Procedure 12.2. Configure the Datasource to use Java Transaction API

1. **Open the configuration file in a text editor.**

Depending on whether you run JBoss EAP 6 in a managed domain or standalone server, your configuration file will be in a different location.

- **Managed domain**

The default configuration file for a managed domain is in ***EAP_HOME/domain/configuration/domain.xml*** for Red Hat Enterprise Linux, and ***EAP_HOME\domain\configuration\domain.xml*** for Microsoft Windows Server.

- **Standalone server**

The default configuration file for a standalone server is in ***EAP_HOME/standalone/configuration/standalone.xml*** for Red Hat Enterprise Linux, and ***EAP_HOME\standalone\configuration\standalone.xml*** for Microsoft Windows Server.

2. **Locate the `<datasource>` tag that corresponds to your datasource.**

The datasource will have the **`jndi-name`** attribute set to the one you specified when you created it. For example, the ExampleDS datasource looks like this:

```
<datasource jndi-name="java:jboss/datasources/ExampleDS" pool-
name="H2DS" enabled="true" jta="true" use-java-context="true" use-
ccm="true">
```

3. **Set the `jta` attribute to `true`.**

Add the following to the contents of your **<datasource>** tag, as they appear in the previous step: **jta="true"**

Unless you have a specific use case (such as defining a read only datasource) Red Hat discourages overriding the default value of **jta=true**. This setting indicates that the datasource will honor the Java Transaction API and allows better tracking of connections by the JCA implementation.

4. Save the configuration file.

Save the configuration file and exit the text editor.

5. Start JBoss EAP 6.

Relaunch the JBoss EAP 6 server.

Result:

JBoss EAP 6 starts, and your datasource is configured to use Java Transaction API.

[Report a bug](#)

12.6.2.4. Configure Database Connection Validation Settings

Overview

Database maintenance, network problems, or other outage events may cause JBoss EAP 6 to lose the connection to the database. You enable database connection validation using the **<validation>** element within the **<datasource>** section of the server configuration file. Follow the steps below to configure the datasource settings to enable database connection validation in JBoss EAP 6.

Procedure 12.3. Configure Database Connection Validation Settings

1. Choose a Validation Method

Select one of the following validation methods.

- o **<validate-on-match>true</validate-on-match>**

When the **<validate-on-match>** option is set to **true**, the database connection is validated every time it is checked out from the connection pool using the validation mechanism specified in the next step.

If a connection is not valid, a warning is written to the log and it retrieves the next connection in the pool. This process continues until a valid connection is found. If you prefer not to cycle through every connection in the pool, you can use the **<use-fast-fail>** option. If a valid connection is not found in the pool, a new connection is created. If the connection creation fails, an exception is returned to the requesting application.

This setting results in the quickest recovery but creates the highest load on the database. However, this is the safest selection if the minimal performance hit is not a concern.

- o **<background-validation>true</background-validation>**

When the **<background-validation>** option is set to **true**, it is used in combination with the **<background-validation-millis>** value to determine how often background validation runs. The default value for the **<background-validation-millis>** parameter is 0 milliseconds, meaning it is disabled by default. This value should not be set to the same value as your **<idle-timeout-minutes>** setting.

It is a balancing act to determine the optimum **<background-validation-millis>**

value for a particular system. The lower the value, the more frequently the pool is validated and the sooner invalid connections are removed from the pool. However, lower values take more database resources. Higher values result in less frequent connection validation checks and use less database resources, but dead connections are undetected for longer periods of time.



NOTE

If the **<validate-on-match>** option is set to **true**, the **<background-validation>** option should be set to **false**. The reverse is also true. If the **<background-validation>** option is set to **true**, the **<validate-on-match>** option should be set to **false**.

2. Choose a Validation Mechanism

Select one of the following validation mechanisms.

o Specify a **<valid-connection-checker>** Class Name

This is the preferred mechanism as it optimized for the particular RDBMS in use. JBoss EAP 6 provides the following connection checkers:

- `org.jboss.jca.adapters.jdbc.extensions.db2.DB2ValidConnectionChecker`
- `org.jboss.jca.adapters.jdbc.extensions.mssql.MSSQLValidConnectionChecker`
- `org.jboss.jca.adapters.jdbc.extensions.mysql.MySQLReplicationValidConnectionChecker`
- `org.jboss.jca.adapters.jdbc.extensions.mysql.MySQLValidConnectionChecker`
- `org.jboss.jca.adapters.jdbc.extensions.novendor.JDBC4ValidConnectionChecker`
- `org.jboss.jca.adapters.jdbc.extensions.novendor.NullValidConnectionChecker`
- `org.jboss.jca.adapters.jdbc.extensions.oracle.OracleValidConnectionChecker`
- `org.jboss.jca.adapters.jdbc.extensions.postgres.PostgreSQLValidConnectionChecker`
- `org.jboss.jca.adapters.jdbc.extensions.sybase.SybaseValidConnectionChecker`

o Specify SQL for **<check-valid-connection-sql>**

You provide the SQL statement used to validate the connection.

The following is an example of how you might specify a SQL statement to validate a connection for Oracle:

```
<check-valid-connection-sql>select 1 from dual</check-valid-connection-sql>
```

For MySQL or PostgreSQL, you might specify the following SQL statement:

```
<check-valid-connection-sql>select 1</check-valid-connection-sql>
```

3. Set the **<exception-sorter>** Class Name

When an exception is marked as fatal, the connection is closed immediately, even if the connection is participating in a transaction. Use the exception sorter class option to properly detect and clean up after fatal connection exceptions. JBoss EAP 6 provides the following

exception sorters:

- `org.jboss.jca.adapters.jdbc.extensions.db2.DB2ExceptionSorter`
- `org.jboss.jca.adapters.jdbc.extensions.informix.InformixExceptionSorter`
- `org.jboss.jca.adapters.jdbc.extensions.mysql.MySQLExceptionSorter`
- `org.jboss.jca.adapters.jdbc.extensions.novendor.NullExceptionSorter`
- `org.jboss.jca.adapters.jdbc.extensions.oracle.OracleExceptionSorter`
- `org.jboss.jca.adapters.jdbc.extensions.postgres.PostgreSQLExceptionSorter`
- `org.jboss.jca.adapters.jdbc.extensions.sybase.SybaseExceptionSorter`
- `org.jboss.jca.adapters.jdbc.extensions.mssql.MSQLExceptionSorter`

[Report a bug](#)

12.6.2.5. Datasource Parameters

Table 12.1. Datasource parameters common to non-XA and XA datasources

Parameter	Description
<code>jndi-name</code>	The unique JNDI name for the datasource.
<code>pool-name</code>	The name of the management pool for the datasource.
<code>enabled</code>	Whether or not the datasource is enabled.
<code>use-java-context</code>	Whether to bind the datasource to global JNDI.
<code>spy</code>	Enable spy functionality on the JDBC layer. This logs all JDBC traffic to the datasource. Note that the logging category jboss.jdbc.spy must also be set to the log level DEBUG in the logging subsystem.
<code>use-ccm</code>	Enable the cached connection manager.
<code>new-connection-sql</code>	A SQL statement which executes when the connection is added to the connection pool.

Parameter	Description
transaction-isolation	One of the following: <ul style="list-style-type: none"> TRANSACTION_READ_UNCOMMITTED TRANSACTION_READ_COMMITTED TRANSACTION_REPEATABLE_READ TRANSACTION_SERIALIZABLE TRANSACTION_NONE
url-selector-strategy-class-name	A class that implements interface org.jboss.jca.adapters.jdbc.URLSelectorStrategy .
security	Contains child elements which are security settings. See Table 12.6, “Security parameters” .
validation	Contains child elements which are validation settings. See Table 12.7, “Validation parameters” .
timeout	Contains child elements which are timeout settings. See Table 12.8, “Timeout parameters” .
statement	Contains child elements which are statement settings. See Table 12.9, “Statement parameters” .

Table 12.2. Non-XA datasource parameters

Parameter	Description
jta	Enable JTA integration for non-XA datasources. Does not apply to XA datasources.
connection-url	The JDBC driver connection URL.
driver-class	The fully-qualified name of the JDBC driver class.
connection-property	Arbitrary connection properties passed to the method Driver.connect(url, props) . Each connection-property specifies a string name/value pair. The property name comes from the name, and the value comes from the element content.
pool	Contains child elements which are pooling settings. See Table 12.4, “Pool parameters common to non-XA and XA datasources” .
url-delimiter	The delimiter for URLs in a connection-url for High Availability (HA) clustered databases.

Table 12.3. XA datasource parameters

Parameter	Description
xa-datasource-property	A property to assign to implementation class XADataSource . Specified by <i>name=value</i> . If a setter method exists, in the format setName , the property is set by calling a setter method in the format of setName(value) .
xa-datasource-class	The fully-qualified name of the implementation class javax.sql.XADataSource .
driver	A unique reference to the class loader module which contains the JDBC driver. The accepted format is <i>driverName#majorVersion.minorVersion</i> .
xa-pool	Contains child elements which are pooling settings. See Table 12.4, “Pool parameters common to non-XA and XA datasources” and Table 12.5, “XA pool parameters”.
recovery	Contains child elements which are recovery settings. See Table 12.10, “Recovery parameters”.

Table 12.4. Pool parameters common to non-XA and XA datasources

Parameter	Description
min-pool-size	The minimum number of connections a pool holds.
max-pool-size	The maximum number of connections a pool can hold.
prefill	Whether to try to prefill the connection pool. The default is false .
use-strict-min	Whether the idle connection scan should strictly stop marking for closure of any further connections, once the min-pool-size has been reached. The default value is false .
flush-strategy	<p>Whether the pool is flushed in the case of an error. Valid values are:</p> <ul style="list-style-type: none"> FailingConnectionOnly IdleConnections EntirePool <p>The default is FailingConnectionOnly.</p>

Parameter	Description
allow-multiple-users	Specifies if multiple users will access the datasource through the <code>getConnection(user, password)</code> method, and whether the internal pool type accounts for this behavior.

Table 12.5. XA pool parameters

Parameter	Description
is-same-rm-override	Whether the <code>javax.transaction.xa.XAResource.isSameRM(XAResource)</code> class returns <code>true</code> or <code>false</code> .
interleaving	Whether to enable interleaving for XA connection factories.
no-tx-separate-pools	Whether to create separate sub-pools for each context. This is required for Oracle datasources, which do not allow XA connections to be used both inside and outside of a JTA transaction. Using this option will cause your total pool size to be twice max-pool-size , because two actual pools will be created.
pad-xid	Whether to pad the Xid.
wrap-xa-resource	Whether to wrap the XAResource in an <code>org.jboss.tm.XAResourceWrapper</code> instance.

Table 12.6. Security parameters

Parameter	Description
user-name	The username to use to create a new connection.
password	The password to use to create a new connection.
security-domain	Contains the name of a JAAS security-manager which handles authentication. This name correlates to the application-policy/name attribute of the JAAS login configuration.
reauth-plugin	Defines a reauthentication plug-in to use to reauthenticate physical connections.

Table 12.7. Validation parameters

Parameter	Description
valid-connection-checker	An implementation of interface org.jboss.jca.adapters.jdbc.ValidConnectionChecker which provides a SQLException.isValidConnection(Connection e) method to validate a connection. An exception means the connection is destroyed. This overrides the parameter check-valid-connection-sql if it is present.
check-valid-connection-sql	An SQL statement to check validity of a pool connection. This may be called when a managed connection is taken from a pool for use.
validate-on-match	Indicates whether connection level validation is performed when a connection factory attempts to match a managed connection for a given set. Specifying "true" for validate-on-match is typically not done in conjunction with specifying "true" for background-validation . Validate-on-match is needed when a client must have a connection validated prior to use. This parameter is false by default.
background-validation	Specifies that connections are validated on a background thread. Background validation is a performance optimization when not used with validate-on-match . If validate-on-match is true, using background-validation could result in redundant checks. Background validation does leave open the opportunity for a bad connection to be given to the client for use (a connection goes bad between the time of the validation scan and prior to being handed to the client), so the client application must account for this possibility.
background-validation-millis	The amount of time, in milliseconds, that background validation runs.
use-fast-fail	If true, fail a connection allocation on the first attempt, if the connection is invalid. Defaults to false .
stale-connection-checker	An instance of org.jboss.jca.adapters.jdbc.StaleConnectionChecker which provides a Boolean isStaleConnection(SQLException e) method. If this method returns true , the exception is wrapped in an org.jboss.jca.adapters.jdbc.StaleConnectionException , which is a subclass of SQLException .

Parameter	Description
exception-sorter	An instance of org.jboss.jca.adapters.jdbc.ExceptionSorter which provides a Boolean isExceptionFatal(SQLException e) method. This method validates whether an exception is broadcast to all instances of javax.resource.spi.ConnectionEventListener as a connectionErrorOccurred message.

Table 12.8. Timeout parameters

Parameter	Description
use-try-lock	Uses tryLock() instead of lock() . This attempts to obtain the lock for the configured number of seconds, before timing out, rather than failing immediately if the lock is unavailable. Defaults to 60 seconds. As an example, to set a timeout of 5 minutes, set <use-try-lock>300</use-try-lock> .
blocking-timeout-millis	The maximum time, in milliseconds, to block while waiting for a connection. After this time is exceeded, an exception is thrown. This blocks only while waiting for a permit for a connection, and does not throw an exception if creating a new connection takes a long time. Defaults to 30000, which is 30 seconds.
idle-timeout-minutes	The maximum time, in minutes, before an idle connection is closed. If not specified, the default is 30 minutes. The actual maximum time depends upon the idleRemover scan time, which is half of the smallest idle-timeout-minutes of any pool.
set-tx-query-timeout	Whether to set the query timeout based on the time remaining until transaction timeout. Any configured query timeout is used if no transaction exists. Defaults to false .
query-timeout	Timeout for queries, in seconds. The default is no timeout.
allocation-retry	The number of times to retry allocating a connection before throwing an exception. The default is 0 , so an exception is thrown upon the first failure.
allocation-retry-wait-millis	How long, in milliseconds, to wait before retrying to allocate a connection. The default is 5000, which is 5 seconds.

Parameter	Description
xa-resource-timeout	If non-zero, this value is passed to method XAResource.setTimeout .

Table 12.9. Statement parameters

Parameter	Description
track-statements	<p>Whether to check for unclosed statements when a connection is returned to a pool and a statement is returned to the prepared statement cache. If false, statements are not tracked.</p> <p>Valid values</p> <ul style="list-style-type: none"> • true: statements and result sets are tracked, and a warning is issued if they are not closed. • false: neither statements or result sets are tracked. • nowarn: statements are tracked but no warning is issued. This is the default.
prepared-statement-cache-size	The number of prepared statements per connection, in a Least Recently Used (LRU) cache.
share-prepared-statements	Whether JBoss EAP should cache, instead of close or terminate, the underlying physical statement when the wrapper supplied to the application is closed by application code. The default is false .

Table 12.10. Recovery parameters

Parameter	Description
recover-credential	A username/password pair or security domain to use for recovery.
recover-plugin	An implementation of the org.jboss.jca.core.spi.recoveryRecoveryPlugin class, to be used for recovery.

[Report a bug](#)

12.6.3. Transaction Logging

12.6.3.1. About Transaction Log Messages

To track transaction status while keeping the log files readable, use the **DEBUG** log level for the transaction logger. For detailed debugging, use the **TRACE** log level. Refer to [Section 12.6.3.2, “Configure Logging for the Transaction Subsystem”](#) for information on configuring the transaction logger.

The transaction manager can generate a lot of logging information when configured to log in the **TRACE** log level. Following are some of the most commonly-seen messages. This list is not comprehensive, so you may see other messages than these.

Table 12.11. Transaction State Change

Transaction Begin	<p>When a transaction begins, the following code is executed:</p> <pre>com.arjuna.ats.arjuna.coordinator .BasicAction::Begin:1342 tsLogger.logger.trace("BasicActio n::Begin() for action-id "+ get_uid());</pre>
Transaction Commit	<p>When a transaction commits, the following code is executed:</p> <pre>com.arjuna.ats.arjuna.coordinator .BasicAction::End:1342 tsLogger.logger.trace("BasicActio n::End() for action-id "+ get_uid());</pre>
Transaction Rollback	<p>When a transaction rolls back, the following code is executed:</p> <pre>com.arjuna.ats.arjuna.coordinator .BasicAction::Abort:1575 tsLogger.logger.trace("BasicActio n::Abort() for action-id "+ get_uid());</pre>

Transaction Timeout	<p>When a transaction times out, the following code is executed:</p> <pre>com.arjuna.ats.arjuna.coordinator .TransactionReaper::doCancellatio ns:349 tsLogger.logger.trace("Reaper Worker " + Thread.currentThread() + " attempting to cancel " + e._control.get_uid());</pre> <p>You will then see the same thread rolling back the transaction as shown above.</p>
---------------------	---

[Report a bug](#)

12.6.3.2. Configure Logging for the Transaction Subsystem

Summary

Use this procedure to control the amount of information logged about transactions, independent of other logging settings in JBoss EAP 6. The main procedure shows how to do this in the web-based Management Console. The Management CLI command is given afterward.

Procedure 12.4. Configure the Transaction Logger Using the Management Console

1. **Navigate to the Logging configuration area.**

In the Management Console, click the **Configuration** tab. If you use a managed domain, choose the server profile you wish to configure, from the **Profile** selection box at the top left.

Expand the **Core** menu, and select **Logging**.

2. **Edit the `com.arjuna` attributes.**

Select the **Log Categories** tab. Select **`com.arjuna`** and click **Edit** in the **Details** section. This is where you can add class-specific logging information. The **`com.arjuna`** class is already present. You can change the log level and whether to use parent handlers.

Log Level

The log level is **WARN** by default. Because transactions can produce a large quantity of logging output, the meaning of the standard logging levels is slightly different for the transaction logger. In general, messages tagged with levels at a lower severity than the chosen level are discarded.

Transaction Logging Levels, from Most to Least Verbose

- TRACE
- DEBUG
- INFO
- WARN

- ERROR
- FAILURE

Use Parent Handlers

Whether the logger should send its output to its parent logger. The default behavior is **true**.

3. Changes take effect immediately.

[Report a bug](#)

12.6.3.3. Browse and Manage Transactions

The Management CLI supports the ability to browse and manipulate transaction records. This functionality is provided by the interaction between the Transaction Manager and the management API of JBoss EAP 6.

The Transaction Manager stores information about each pending transaction and the participants involved the transaction, in a persistent storage called the *object store*. The management API exposes the object store as a resource called the **log-store**. An API operation called **probe** reads the transaction logs and creates a node for each log. You can call the **probe** command manually, whenever you need to refresh the **log-store**. It is normal for transaction logs to appear and disappear quickly.

Example 12.2. Refresh the Log Store

This command refreshes the log store for server groups which use the profile **default** in a managed domain. For a standalone server, remove the **profile=default** from the command.

```
/profile=default/subsystem=transactions/log-store=log-store/:probe
```

Example 12.3. View All Prepared Transactions

To view all prepared transactions, first refresh the log store (see [Example 12.2, “Refresh the Log Store”](#)), then run the following command, which functions similarly to a filesystem **ls** command.

```
ls /profile=default/subsystem=transactions/log-store=log-store/transactions
```

Each transaction is shown, along with its unique identifier. Individual operations can be run against an individual transaction (see [Manage a Transaction](#)).

Manage a Transaction

View a transaction's attributes.

To view information about a transaction, such as its JNDI name, EIS product name and version, or its status, use the **:read-resource** CLI command.

```
/profile=default/subsystem=transactions/log-store=log-store/transactions=0\:ffff7f000001\:-b66efc2\:4f9e6f8f\:9\:read-resource
```

View the participants of a transaction.

Each transaction log contains a child element called **participants**. Use the **read-resource** CLI command on this element to see the participants of the transaction. Participants are identified by their JNDI names.

```
/profile=default/subsystem=transactions/log-store=log-
store/transactions=0\:ffff7f000001\:-
b66efc2\:4f9e6f8f\:9/participants=java\:/JmsXA:read-resource
```

The result may look similar to this:

```
{
  "outcome" => "success",
  "result" => {
    "eis-product-name" => "HornetQ",
    "eis-product-version" => "2.0",
    "jndi-name" => "java:/JmsXA",
    "status" => "HEURISTIC",
    "type" => "/StateManager/AbstractRecord/XAResourceRecord"
  }
}
```

The outcome status shown here is in a **HEURISTIC** state and is eligible for recovery. See [Recover a transaction](#) for more details.

In special cases it is possible to create orphan records in the object store, that is XAResourceRecords, which do not have any corresponding transaction record in the log. For example, XA resource prepared but crashed before the TM recorded and is inaccessible for the domain management API. To access such records you need to set management option **expose-all-logs** to **true**. This option is not saved in management model and is restored to **false** when the server is restarted.

```
/profile=default/subsystem=transactions/log-store=log-store:write-
attribute(name=expose-all-logs, value=true)
```

Delete a transaction.

Each transaction log supports a **:delete** operation, to delete the transaction log representing the transaction.

```
/profile=default/subsystem=transactions/log-store=log-
store/transactions=0\:ffff7f000001\:-b66efc2\:4f9e6f8f\:9:delete
```

Recover a transaction.

Each transaction participant supports recovery via the **:recover** CLI command.

```
/profile=default/subsystem=transactions/log-store=log-
store/transactions=0\:ffff7f000001\:-
b66efc2\:4f9e6f8f\:9/participants=2:recover
```

Recovery of heuristic transactions and participants

- If the transaction's status is **HEURISTIC**, the recovery operation changes the state to **PREPARE** and triggers a recovery.
- If one of the transaction's participants is heuristic, the recovery operation tries to replay the **commit** operation. If successful, the participant is removed from the transaction log. You can verify this by re-running the **:probe** operation on the **log-store** and checking that the participant is no longer listed. If this is the last participant, the transaction is also deleted.

Refresh the status of a transaction which needs recovery.

If a transaction needs recovery, you can use the **:refresh** CLI command to be sure it still requires recovery, before attempting the recovery.

```
/profile=default/subsystem=transactions/log-store=log-store/transactions=0\:ffff7f000001\:-b66efc2\:4f9e6f8f\:9/participants=2:refresh
```

View Transaction Statistics

If Transaction Manager statistics are enabled, you can view statistics about the Transaction Manager and transaction subsystem. See [Section 12.7.8.2, “Configure the Transaction Manager”](#) for information about how to enable Transaction Manager statistics.

You can view statistics either via the management console or the Management CLI. In the management console, transaction statistics are available via **Runtime** → **Status** → **Subsystems** → **Transactions**.

Transaction statistics are available for each server in a managed domain. To view the status of a different server, select **Change Server** in the left-hand menu and select the server from the list.

The following table shows each available statistic, its description, and the Management CLI command to view the statistic.

Table 12.12. Transaction Subsystem Statistics

Statistic	Description	CLI Command
Total	The total number of transactions processed by the Transaction Manager on this server.	<pre>/host=master/server=server-one/subsystem=transactions/:read-attribute(name=number-of-transactions,include-defaults=true)</pre>

Statistic	Description	CLI Command
Committed	The number of committed transactions processed by the Transaction Manager on this server.	<pre>/host=<i>master</i>/server=<i>server-one</i>/subsystem=transactions/:read-attribute(name=number-of-committed-transactions,include-defaults=true)</pre>
Aborted	The number of aborted transactions processed by the Transaction Manager on this server.	<pre>/host=<i>master</i>/server=<i>server-one</i>/subsystem=transactions/:read-attribute(name=number-of-aborted-transactions,include-defaults=true)</pre>
Timed Out	The number of timed out transactions processed by the Transaction Manager on this server.	<pre>/host=<i>master</i>/server=<i>server-one</i>/subsystem=transactions/:read-attribute(name=number-of-timed-out-transactions,include-defaults=true)</pre>
Heuristics	Not available in the Management Console. Number of transactions in a heuristic state.	<pre>/host=<i>master</i>/server=<i>server-one</i>/subsystem=transactions/:read-attribute(name=number-of-heuristics,include-defaults=true)</pre>

Statistic	Description	CLI Command
In-Flight Transactions	Not available in the Management Console. Number of transactions which have begun but not yet terminated.	<pre>/host=<i>master</i>/server=<i>server-one</i>/subsystem=transactions/:read-attribute(name=number-of-inflight-transactions,include-defaults=true)</pre>
Failure Origin - Applications	The number of failed transactions whose failure origin was an application.	<pre>/host=<i>master</i>/server=<i>server-one</i>/subsystem=transactions/:read-attribute(name=number-of-application-rollback,include-defaults=true)</pre>
Failure Origin - Resources	The number of failed transactions whose failure origin was a resource.	<pre>/host=<i>master</i>/server=<i>server-one</i>/subsystem=transactions/:read-attribute(name=number-of-resource-rollback,include-defaults=true)</pre>
Participant ID	The ID of the participant.	<pre>/host=<i>master</i>/server=<i>server-one</i>/subsystem=transactions/log-store=log-store/transactions=0\:<i>ffff7f000001</i>\:-<i>b66efc2</i>\:<i>4f9e6f8f</i>\:<i>9</i>:read-children-names(child-type=participants)</pre>

Statistic	Description	CLI Command
List of all transactions	The complete list of transactions.	<pre>/host=master/server=server-one/subsystem=transactions/log-store=log-store:read-children-names(child-type=transactions)</pre>

[Report a bug](#)

12.7. USE JTA TRANSACTIONS

12.7.1. Transactions JTA Task Overview

Introduction

The following procedures are useful when you need to use transactions in your application.

- [Section 12.7.2, “Control Transactions”](#)
- [Section 12.7.3, “Begin a Transaction”](#)
- [Section 12.7.5, “Commit a Transaction”](#)
- [Section 12.7.6, “Roll Back a Transaction”](#)
- [Section 12.7.7, “Handle a Heuristic Outcome in a Transaction”](#)
- [Section 12.7.8.2, “Configure the Transaction Manager”](#)
- [Section 12.7.9.1, “Handle Transaction Errors”](#)

[Report a bug](#)

12.7.2. Control Transactions

Introduction

This list of procedures outlines the different ways to control transactions in your applications which use JTA or JTS APIs.

- [Section 12.7.3, “Begin a Transaction”](#)
- [Section 12.7.5, “Commit a Transaction”](#)
- [Section 12.7.6, “Roll Back a Transaction”](#)
- [Section 12.7.7, “Handle a Heuristic Outcome in a Transaction”](#)

[Report a bug](#)

12.7.3. Begin a Transaction

This procedure shows how to begin a new transaction. The API is the same either you run Transaction Manager configured with JTA or JTS.

1. Get an instance of `UserTransaction`.

You can get the instance using JNDI, injection, or an EJB's context, if the EJB uses bean-managed transactions, by means of a

`@TransactionManagement(TransactionManagementType.BEAN)` annotation.

◦ JNDI

```
new InitialContext().lookup("java:comp/UserTransaction")
```

◦ Injection

```
@Resource UserTransaction userTransaction;
```

◦ Context

■ In a stateless/stateful bean:

```
@Resource SessionContext ctx;
ctx.getUserTransaction();
```

■ In a message-driven bean:

```
@Resource MessageDrivenContext ctx;
ctx.getUserTransaction();
```

2. Call `UserTransaction.begin()` after you connect to your datasource.

```
...
try {
    System.out.println("\nCreating connection to database: "+url);
    stmt = conn.createStatement(); // non-tx statement
    try {
        System.out.println("Starting top-level transaction.");
        userTransaction.begin();
        stmtx = conn.createStatement(); // will be a tx-statement
        ...
    }
}
```

Participate in an existing transaction using the JTS API.

One of the benefits of EJBs (either used with CMT or BMT) is that the container manages all the internals of the transactional processing, that is, you are free from taking care of transaction being part of XA transaction or transaction distribution amongst EAP containers.

Result:

The transaction begins. All uses of your datasource until you commit or roll back the transaction are transactional.



NOTE

For a full example, see [Section 12.9.2, “JTA Transaction Example”](#).

[Report a bug](#)

12.7.4. Nested Transactions

Nested transactions allow an application to create a transaction that is embedded in an existing transaction. In this model, multiple subtransactions can be embedded recursively in a transaction. Subtransactions can be committed or rolled back without committing or rolling back the parent transaction. However, the results of a commit operation are contingent upon the commitment of all the transaction's ancestors.

For implementation specific information, refer JBossTS JTS Development guide at https://docs.jboss.org/jbosstm/latest/guides/narayana-jts-development_guide.

Nested transactions are available only when used with the JTS API. Nested transaction are not a supported feature of EAP application server. In addition, many database vendors do not support nested transactions, so consult your database vendor before you add nested transactions to your application.

[Report a bug](#)

12.7.5. Commit a Transaction

This procedure shows how to commit a transaction using the Java Transaction API (JTA).

Prerequisites

You must begin a transaction before you can commit it. For information on how to begin a transaction, refer to [Section 12.7.3, “Begin a Transaction”](#).

1. Call the `commit()` method on the `UserTransaction`.

When you call the `commit()` method on the `UserTransaction`, the Transaction Manager attempts to commit the transaction.

```
@Inject
private UserTransaction userTransaction;

public void updateTable(String key, String value)
    EntityManager entityManager =
entityManagerFactory.createEntityManager();
    try {
        userTransaction.begin();
        <!-- Perform some data manipulation using entityManager -->
        ...
        // Commit the transaction
        userTransaction.commit();
    } catch (Exception ex) {
        <!-- Log message or notify Web page -->
        ...
    }
    try {
```

```

        userTransaction.rollback();
    } catch (SystemException se) {
        throw new RuntimeException(se);
    }
    throw new RuntimeException(e);
} finally {
    entityManager.close();
}
}

```

2. If you use Container Managed Transactions (CMT), you do not need to manually commit.

If you configure your bean to use Container Managed Transactions, the container will manage the transaction lifecycle for you based on annotations you configure in the code.

```

@PersistenceContext
private EntityManager em;

@Transactional(TransactionAttributeType.REQUIRED)
public void updateTable(String key, String value)
    <!-- Perform some data manipulation using entityManager -->
    ...
}

```

Result

Your datasource commits and your transaction ends, or an exception is thrown.



NOTE

For a full example, see [Section 12.9.2, “JTA Transaction Example”](#).

[Report a bug](#)

12.7.6. Roll Back a Transaction

This procedure shows how to roll back a transaction using the Java Transaction API (JTA).

Prerequisites

You must begin a transaction before you can roll it back. For information on how to begin a transaction, refer to [Section 12.7.3, “Begin a Transaction”](#).

1. Call the `rollback()` method on the `UserTransaction`.

When you call the `rollback()` method on the `UserTransaction`, the Transaction Manager attempts to roll back the transaction and return the data to its previous state.

```

@Inject
private UserTransaction userTransaction;

public void updateTable(String key, String value)
    EntityManager entityManager =
    entityManagerFactory.createEntityManager();
    try {
        userTransaction.begin():

```

```

        <!-- Perform some data manipulation using entityManager -->
        ...
        // Commit the transaction
        userTransaction.commit();
    } catch (Exception ex) {
        <!-- Log message or notify Web page -->
        ...
        try {
            userTransaction.rollback();
        } catch (SystemException se) {
            throw new RuntimeException(se);
        }
        throw new RuntimeException(e);
    } finally {
        entityManager.close();
    }
}

```

2. If you use Container Managed Transactions (CMT), you do not need to manually roll back the transaction.

If you configure your bean to use Container Managed Transactions, the container will manage the transaction lifecycle for you based on annotations you configure in the code.

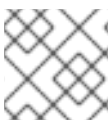


NOTE

Rollback for CMT occurs if `RuntimeException` is thrown. You can also explicitly call the `setRollbackOnly` method to gain the rollback. Or, use the `@ApplicationException(rollback=true)` for application exception to rollback.

Result

Your transaction is rolled back by the Transaction Manager.



NOTE

For a full example, see [Section 12.9.2, “JTA Transaction Example”](#).

[Report a bug](#)

12.7.7. Handle a Heuristic Outcome in a Transaction

This procedure shows how to handle a heuristic outcome of a transaction using the Java Transaction API (JTA).

Heuristic transaction outcomes are uncommon and usually have exceptional causes. The word *heuristic* means “by hand”, and that is the way that these outcomes usually have to be handled. Refer to [Section 12.4.4, “About Heuristic Outcomes”](#) for more information about heuristic transaction outcomes.

Procedure 12.5. Handle a heuristic outcome in a transaction

1. Determine the cause

The over-arching cause of a heuristic outcome in a transaction is that a resource manager promised it could commit or roll-back, and then failed to fulfill the promise. This could be due to a

problem with a third-party component, the integration layer between the third-party component and JBoss EAP 6, or JBoss EAP 6 itself.

By far, the most common two causes of heuristic errors are transient failures in the environment and coding errors in the code dealing with resource managers.

2. Fix transient failures in the environment

Typically, if there is a transient failure in your environment, you will know about it before you find out about the heuristic error. This could be a network outage, hardware failure, database failure, power outage, or a host of other things.

If you experienced the heuristic outcome in a test environment, during stress testing, it provides information about weaknesses in your environment.



WARNING

JBoss EAP 6 will automatically recover transactions that were in a non-heuristic state at the time of the failure, but it does not attempt to recover heuristic transactions.

3. Contact resource manager vendors

If you have no obvious failure in your environment, or the heuristic outcome is easily reproducible, it is probably a coding error. Contact third-party vendors to find out if a solution is available. If you suspect the problem is in the transaction manager of JBoss EAP 6 itself, contact Red Hat Global Support Services.

4. In a test environment, delete the logs and restart JBoss EAP 6.

In a test environment, or if you do not care about the integrity of the data, deleting the transaction logs and restarting JBoss EAP 6 gets rid of the heuristic outcome. The transaction logs are located in ***EAP_HOME/standalone/data/tx-object-store/*** for a standalone server, or ***EAP_HOME/domain/servers/SERVER_NAME/data/tx-object-store*** in a managed domain, by default. In the case of a managed domain, *SERVER_NAME* refers to the name of the individual server participating in a server group.



NOTE

The location of the transaction log also depends on the object store in use and the values set for the ***object-store-relative-to*** and ***object-store-path*** parameters. For file system logs (such as a standard shadow and HornetQ logs) the default direction location is used, but when using a JDBC object store, the transaction logs are stored in a database.

5. Resolve the outcome by hand

The process of resolving the transaction outcome by hand is very dependent on the exact circumstance of the failure. Typically, you need to take the following steps, applying them to your situation:

- a. Identify which resource managers were involved.
- b. Examine the state in the transaction manager and the resource managers.

- c. Manually force log cleanup and data reconciliation in one or more of the involved components.

The details of how to perform these steps are out of the scope of this documentation.

[Report a bug](#)

12.7.8. Transaction Timeouts

12.7.8.1. About Transaction Timeouts

In order to preserve atomicity and adhere to the ACID standard for transactions, some parts of a transaction can be long-running. Transaction participants need to lock parts of datasources when they commit, and the transaction manager needs to wait to hear back from each transaction participant before it can direct them all whether to commit or roll back. Hardware or network failures can cause resources to be locked indefinitely.

Transaction timeouts can be associated with transactions in order to control their lifecycle. If a timeout threshold passes before the transaction commits or rolls back, the timeout causes the transaction to be rolled back automatically.

You can configure default timeout values for the entire transaction subsystem, or you disable default timeout values, and specify timeouts on a per-transaction basis.

[Report a bug](#)

12.7.8.2. Configure the Transaction Manager

You can configure the Transaction Manager (TM) using the web-based Management Console or the command-line Management CLI. For each command or option given, the assumption is made that you are running JBoss EAP 6 as a Managed Domain. If you use a Standalone Server or you want to modify a different profile than **default**, you may need to modify the steps and commands in the following ways.

Notes about the Example Commands

- For the Management Console, the **default** profile is the one which is selected when you first log into the console. If you need to modify the Transaction Manager's configuration in a different profile, select your profile instead of **default**, in each instruction.

Similarly, substitute your profile for the **default** profile in the example CLI commands.

- If you use a Standalone Server, only one profile exists. Ignore any instructions to choose a specific profile. In CLI commands, remove the **/profile=default** portion of the sample commands.



NOTE

In order for the TM options to be visible in the Management Console or Management CLI, the **transactions** subsystem must be enabled. It is enabled by default, and required for many other subsystems to function properly, so it is very unlikely that it would be disabled.

Configure the TM Using the Management Console

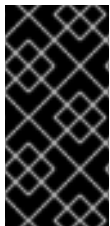
To configure the TM using the web-based Management Console, select the **Configuration** tab from the top of the screen. If you use a managed domain, choose the correct profile from the **Profile** selection box at the top left. Expand the **Container** menu and select **Transactions**.

Most options are shown in the Transaction Manager configuration page. The **Recovery** options are hidden by default. Click the **Recovery** tab to see the recovery options. Click **Edit** to edit any of the options. Changes take effect immediately.

Click the **Need Help?** label to display in-line help text.

Configure the TM using the Management CLI

In the Management CLI, you can configure the TM using a series of commands. The commands all begin with `/profile=default/subsystem=transactions/` for a managed domain with profile **default**, or `/subsystem=transactions` for a Standalone Server.



IMPORTANT

If transaction subsystem is configured to use hornetq journal as storage type for transaction logs, then two instances of JBoss EAP is not permitted to use the same directory for storing the journal. Application server instances can't share the same location and each has to configure unique location for it.

Table 12.13. TM Configuration Options

Option	Description	CLI Command
Enable Statistics	Whether to enable transaction statistics. These statistics can be viewed in the Management Console in the Subsystem Metrics section of the Runtime tab.	<code>/profile=default/subsystem=transactions/:write-attribute(name=enable-statistics,value=true)</code>
Enable TSM Status	Whether to enable the transaction status manager (TSM) service, which is used for out-of-process recovery. Running an out of process recovery manager to contact the ActionStatusService from different process is not supported (it is normally contacted in memory).	This configuration option is unsupported.
Default Timeout	The default transaction timeout. This defaults to 300 seconds. You can override this programmatically, on a per-transaction basis.	<code>/profile=default/subsystem=transactions/:write-attribute(name=default-timeout,value=300)</code>
Object Store Path	A relative or absolute filesystem path where the TM object store stores data. By default relative to the object-store-relative-to parameter's value.	<code>/profile=default/subsystem=transactions/:write-attribute(name=object-store-path,value=tx-object-store)</code>

Option	Description	CLI Command
Object Store Path Relative To	References a global path configuration in the domain model. The default value is the data directory for JBoss EAP 6, which is the value of the property jboss.server.data.dir , and defaults to EAP_HOME/domain/data/ for a Managed Domain, or EAP_HOME/standalone/data/ for a Standalone Server instance. The value of the object store object-store-path TM attribute is relative to this path.	/profile=default/subsystem=transactions/:write-attribute(name=object-store-relative-to,value=jboss.server.data.dir)
Socket Binding	Specifies the name of the socket binding used by the Transaction Manager for recovery and generating transaction identifiers, when the socket-based mechanism is used. Refer to process-id-socket-max-ports for more information on unique identifier generation. Socket bindings are specified per server group in the Server tab of the Management Console.	/profile=default/subsystem=transactions/:write-attribute(name=socket-binding,value=txn-recovery-environment)
Status Socket Binding	Specifies the socket binding to use for the Transaction Status manager.	This configuration option is unsupported.
Recovery Listener	Whether or not the Transaction Recovery process should listen on a network socket. Defaults to false .	/profile=default/subsystem=transactions/:write-attribute(name=recovery-listener,value=false)

The following options are for advanced use and can only be modified using the Management CLI. Be cautious when changing them from the default configuration. Contact [Red Hat Global Support Services](#) for more information.

Table 12.14. Advanced TM Configuration Options

Option	Description	CLI Command
jts	Whether to use Java Transaction Service (JTS) transactions. Defaults to false , which uses JTA transactions only.	/profile=default/subsystem=transactions/:write-attribute(name=jts,value=false)

Option	Description	CLI Command
node-identifier	<p>The node identifier for the Transaction Manager. This option is required in the following situations:</p> <ul style="list-style-type: none"> • For JTS to JTS communications • When two Transaction Managers access shared resource managers • When two Transaction Managers access shared object stores <p>The node-identifier must be unique for each Transaction Manager as it is required to enforce data integrity during recovery. The node-identifier must also be unique for JTA because multiple nodes may interact with the same resource manager or share a transaction object store.</p>	/profile=default/subsystem=transactions/:write-attribute(name=node-identifier,value=1)
process-id-socket-max-ports	<p>The Transaction Manager creates a unique identifier for each transaction log. Two different mechanisms are provided for generating unique identifiers: a socket-based mechanism and a mechanism based on the process identifier of the process.</p> <p>In the case of the socket-based identifier, a socket is opened and its port number is used for the identifier. If the port is already in use, the next port is probed, until a free one is found. The process-id-socket-max-ports represents the maximum number of sockets the TM will try before failing. The default value is 10.</p>	/profile=default/subsystem=transactions/:write-attribute(name=process-id-socket-max-ports,value=10)

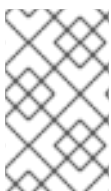
Option	Description	CLI Command
process-id-uuid	Set to true to use the process identifier to create a unique identifier for each transaction. Otherwise, the socket-based mechanism is used. Defaults to true . Refer to process-id-socket-max-ports for more information. To enable process-id-socket-binding , set process-id-uuid to false .	<code>/profile=default/subsystem=transactions/:write - attribute(name=process-id-uuid,value=true)</code>
process-id-socket-binding	The name of the socket binding configuration to use if the transaction manager should use a socket-based process id. Will be undefined if process-id-uuid is true ; otherwise must be set.	<code>/profile=default/subsystem=transactions/:write - attribute(name=process-id-socket-binding,value=true)</code>
use-hornetq-store	Use HornetQ's journaled storage mechanisms instead of file-based storage, for the transaction logs. This is disabled by default, but can improve I/O performance. It is not recommended for JTS transactions on separate Transaction Managers. When changing this option, the server has to be restarted using the shutdown command for the change to take effect.	<code>/profile=default/subsystem=transactions/:write - attribute(name=use-hornetq-store,value=false)</code>

[Report a bug](#)

12.7.9. JTA Transaction Error Handling

12.7.9.1. Handle Transaction Errors

Transaction errors are challenging to solve because they are often dependent on timing. Here are some common errors and ideas for troubleshooting them.



NOTE

These guidelines do not apply to heuristic errors. If you experience heuristic errors, refer to [Section 12.7.7, “Handle a Heuristic Outcome in a Transaction”](#) and contact Red Hat Global Support Services for assistance.

The transaction timed out but the business logic thread did not notice

This type of error often manifests itself when Hibernate is unable to obtain a database connection for lazy loading. If it happens frequently, you can lengthen the timeout value. Refer to [Section 12.7.8.2, “Configure the Transaction Manager”](#).

If that is not feasible, you may be able to tune your external environment to perform more quickly, or restructure your code to be more efficient. Contact Red Hat Global Support Services if you still have trouble with timeouts.

The transaction is already running on a thread, or you receive a `NotSupportedException` exception

The **`NotSupportedException`** exception usually indicates that you attempted to nest a JTA transaction, and this is not supported. If you were not attempting to nest a transaction, it is likely that another transaction was started in a thread pool task, but finished the task without suspending or ending the transaction.

Applications typically use **`UserTransaction`**, which handles this automatically. If so, there may be a problem with a framework.

If your code does use **`TransactionManager`** or **`Transaction`** methods directly, be aware of the following behavior when committing or rolling back a transaction. If your code uses **`TransactionManager`** methods to control your transactions, committing or rolling back a transaction disassociates the transaction from the current thread. However, if your code uses **`Transaction`** methods, the transaction may not be associated with the running thread, and you need to disassociate it from its threads manually, before returning it to the thread pool.

You are unable to enlist a second local resource

This error happens if you try to enlist a second non-XA resource into a transaction. If you need multiple resources in a transaction, they must be XA.

[Report a bug](#)

12.8. ORB CONFIGURATION

12.8.1. About Common Object Request Broker Architecture (CORBA)

Common Object Request Broker Architecture (CORBA) is a standard that enables applications and services to work together even when they are written in multiple, otherwise-incompatible, languages or hosted on separate platforms. CORBA requests are brokered by a server-side component called an *Object Request Broker (ORB)*. JBoss EAP 6 provides an ORB instance, by means of the JacORB component.

The ORB is used internally for *Java Transaction Service (JTS)* transactions, and is also available for use by your own applications.

[Report a bug](#)

12.8.2. Configure the ORB for JTS Transactions

In a default installation of JBoss EAP 6, the ORB is disabled. You can enable the ORB using the command-line Management CLI.

Procedure 12.6. Configure the ORB using the Management Console

1. **View the profile settings.**

Select **Configuration** from the top of the management console. If you use a managed domain, select either the **full** or **full-ha** profile from the selection box at the top left.

2. Modify the Initializers Settings

Expand the **Subsystems** menu. Expand the **Container** menu and select **JacORB**.

In the form that appears in the main screen, select the **Initializers** tab and click the **Edit** button.

Enable the security interceptors by setting the value of **Security** to **on**.

To enable the ORB for JTS, set the **Transaction Interceptors** value to **on**, rather than the default **spec**.

Refer to the **Need Help?** link in the form for detailed explanations about these values. Click **Save** when you have finished editing the values.

3. Advanced ORB Configuration

Refer to the other sections of the form for advanced configuration options. Each section includes a **Need Help?** link with detailed information about the parameters.

Configure the ORB using the Management CLI

You can configure each aspect of the ORB using the Management CLI. The following commands configure the initializers to the same values as the procedure above, for the Management Console. This is the minimum configuration for the ORB to be used with JTS.

These commands are configured for a managed domain using the **full** profile. If necessary, change the profile to suit the one you need to configure. If you use a standalone server, omit the **/profile=full** portion of the commands.

Example 12.4. Enable the Security Interceptors

```
/profile=full/subsystem=jacorb/:write-attribute(name=security,value=on)
```

Example 12.5. Enable Transactions in the JacORB Subsystem

```
/profile=full/subsystem=jacorb/:write-attribute(name=transactions,value=on)
```

Example 12.6. Enable JTS in the Transaction Subsystem

```
/profile=full/subsystem=transactions:write-attribute(name=jts,value=true)
```



NOTE

For JTS activation, the server must be restarted as reload is not enough.

[Report a bug](#)

12.9. TRANSACTION REFERENCES

12.9.1. JBoss Transactions Errors and Exceptions

For details about exceptions thrown by methods of the **UserTransaction** class, see the *UserTransaction API* specification at <http://docs.oracle.com/javaee/6/api/javax/transaction/UserTransaction.html>.

[Report a bug](#)

12.9.2. JTA Transaction Example

This example illustrates how to begin, commit, and roll back a JTA transaction. You need to adjust the connection and datasource parameters to suit your environment, and set up two test tables in your database.

Example 12.7. JTA Transaction example

```
public class JDBCExample {
    public static void main (String[] args) {
        Context ctx = new InitialContext();
        // Change these two lines to suit your environment.
        DataSource ds = (DataSource)ctx.lookup("jdbc/ExampleDS");
        Connection conn = ds.getConnection("testuser", "testpwd");
        Statement stmt = null; // Non-transactional statement
        Statement stmtx = null; // Transactional statement
        Properties dbProperties = new Properties();

        // Get a UserTransaction
        UserTransaction txn = new
InitialContext().lookup("java:comp/UserTransaction");

        try {
            stmt = conn.createStatement(); // non-tx statement

            // Check the database connection.
            try {
                stmt.executeUpdate("DROP TABLE test_table");
                stmt.executeUpdate("DROP TABLE test_table2");
            }
            catch (Exception e) {
                // assume not in database.
            }

            try {
                stmt.executeUpdate("CREATE TABLE test_table (a
INTEGER,b INTEGER)");
                stmt.executeUpdate("CREATE TABLE test_table2 (a
INTEGER,b INTEGER)");
            }
            catch (Exception e) {
            }
        }
    }
}
```



```

try {
    System.out.println("Starting top-level transaction.");

    txn.begin();

    stmtx = conn.createStatement(); // will be a tx-
statement

    // First, we try to roll back changes

    System.out.println("\nAdding entries to table 1.");

    stmtx.executeUpdate("INSERT INTO test_table (a, b)
VALUES (1,2)");

    ResultSet res1 = null;

    System.out.println("\nInspecting table 1.");

    res1 = stmtx.executeQuery("SELECT * FROM test_table");

    while (res1.next()) {
        System.out.println("Column 1: "+res1.getInt(1));
        System.out.println("Column 2: "+res1.getInt(2));
    }
    System.out.println("\nAdding entries to table 2.");

    stmtx.executeUpdate("INSERT INTO test_table2 (a, b)
VALUES (3,4)");
    res1 = stmtx.executeQuery("SELECT * FROM test_table2");

    System.out.println("\nInspecting table 2.");

    while (res1.next()) {
        System.out.println("Column 1: "+res1.getInt(1));
        System.out.println("Column 2: "+res1.getInt(2));
    }

    System.out.print("\nNow attempting to rollback
changes.");

    txn.rollback();

    // Next, we try to commit changes
    txn.begin();
    stmtx = conn.createStatement();
    ResultSet res2 = null;

    System.out.println("\nNow checking state of table 1.");

    res2 = stmtx.executeQuery("SELECT * FROM test_table");

    while (res2.next()) {
        System.out.println("Column 1: "+res2.getInt(1));
        System.out.println("Column 2: "+res2.getInt(2));
    }

```

```

    }

    System.out.println("\nNow checking state of table 2.");

    stmtx = conn.createStatement();

    res2 = stmtx.executeQuery("SELECT * FROM test_table2");

    while (res2.next()) {
        System.out.println("Column 1: "+res2.getInt(1));
        System.out.println("Column 2: "+res2.getInt(2));
    }

    txn.commit();
}
catch (Exception ex) {
    ex.printStackTrace();
    System.exit(0);
}
}
catch (Exception sysEx) {
    sysEx.printStackTrace();
    System.exit(0);
}
}
}
}

```

[Report a bug](#)

12.9.3. API Documentation for JBoss Transactions JTA

The API documentation for the Transaction subsystem of JBoss EAP 6 is available at the following location:

- UserTransaction - <http://docs.oracle.com/javaee/6/api/javax/transaction/UserTransaction.html>

If you use Red Hat JBoss Developer Studio to develop your applications, the API documentation is included in the **Help** menu.

[Report a bug](#)

12.9.4. Limitations of the XA Recovery Process

XA recovery has the following limitations.

The transaction log may not be cleared from a successfully committed transaction.

If the JBoss EAP server crashes after an **XAResource** commit method successfully completes and commits the transaction, but before the coordinator can update the log, you may see the following warning message in the log when you restart the server:

```
ARJUNA016037: Could not find new XAResource to use for recovering non-serializable XAResource XAResourceRecord
```

This is because upon recovery, the JBoss Transaction Manager sees the transaction participants in the log and attempts to retry the commit. Eventually the JBoss Transaction Manager assumes the resources are committed and no longer retries the commit. In this situation, can safely ignore this warning as the transaction is committed and there is no loss of data.

To prevent the warning, set the `com.arjuna.ats.jta.xaAssumeRecoveryComplete` property value to **true**. This property is checked whenever a new **XAResource** instance cannot be located from any registered **XAResourceRecovery** instance. When set to **true**, the recovery assumes that a previous commit attempt succeeded and the instance can be removed from the log with no further recovery attempts. This property must be used with care because it is global and when used incorrectly could result in **XAResource** instances remaining in an uncommitted state.

Rollback is not called for JTS transaction when a server crashes at the end of XAResource.prepare().

If the JBoss EAP server crashes after the completion of an **XAResource prepare()** method call, all of the participating XAResources are locked in the prepared state and remain that way upon server restart. The transaction is not rolled back and the resources remain locked until the transaction times out or a DBA manually rolls back the resources and clears the transaction log.

Periodic recovery can occur on committed transactions.

When the server is under excessive load, the server log may contain the following warning message, followed by a stacktrace:

```
ARJUNA016027: Local XAResourceRecoveryModule.xaRecovery got XA exception
XAException.XAER_NOTA: javax.transaction.xa.XAException
```

Under heavy load, the processing time taken by a transaction can overlap with the timing of the periodic recovery process's activity. The periodic recovery process detects the transaction still in progress and attempts to initiate a rollback but in fact the transaction continues to completion. At the time the periodic recovery attempts but fails the rollback, it records the rollback failure in the server log. The underlying cause of this issue will be addressed in a future release, but in the meantime a workaround is available.

Increase the interval between the two phases of the recovery process by setting the `com.arjuna.ats.jta.orphanSafetyInterval` property to a value higher than the default value of 10000 milliseconds. A value of 40000 milliseconds is recommended. Please note that this does not solve the issue, instead it decreases the probability that it will occur and that the warning message will be shown in the log.

[Report a bug](#)

CHAPTER 13. HIBERNATE

13.1. ABOUT HIBERNATE CORE

Hibernate Core is an object/relational mapping library. It provides the framework for mapping Java classes to database tables, allowing applications to avoid direct interaction with the database.

For more information, refer to [Section 13.2.2, “Hibernate EntityManager”](#) and the [Section 13.2.1, “About JPA”](#).

[Report a bug](#)

13.2. JAVA PERSISTENCE API (JPA)

13.2.1. About JPA

The Java Persistence API (JPA) is the standard for using persistence in Java projects. Java EE 6 applications use the Java Persistence 2.0 specification, documented here: <http://www.jcp.org/en/jsr/detail?id=317>.

Hibernate EntityManager implements the programming interfaces and life-cycle rules defined by the specification. It provides JBoss EAP 6 with a complete Java Persistence solution.

JBoss EAP 6 is 100% compliant with the Java Persistence 2.0 specification. Hibernate also provides additional features to the specification.

To get started with JPA and JBoss EAP 6, refer to the **bean-validation**, **greeter**, and **kitchensink** quickstarts: [Section 1.4.1.1, “Access the Quickstarts”](#).

[Report a bug](#)

13.2.2. Hibernate EntityManager

Hibernate EntityManager implements the programming interfaces and life-cycle rules defined by the [JPA 2.0 specification](#). It provides JBoss EAP 6 with a complete Java Persistence solution.

For more information about Java Persistence or Hibernate, refer to the [Section 13.2.1, “About JPA”](#) and [Section 13.1, “About Hibernate Core”](#).

[Report a bug](#)

13.2.3. Getting Started

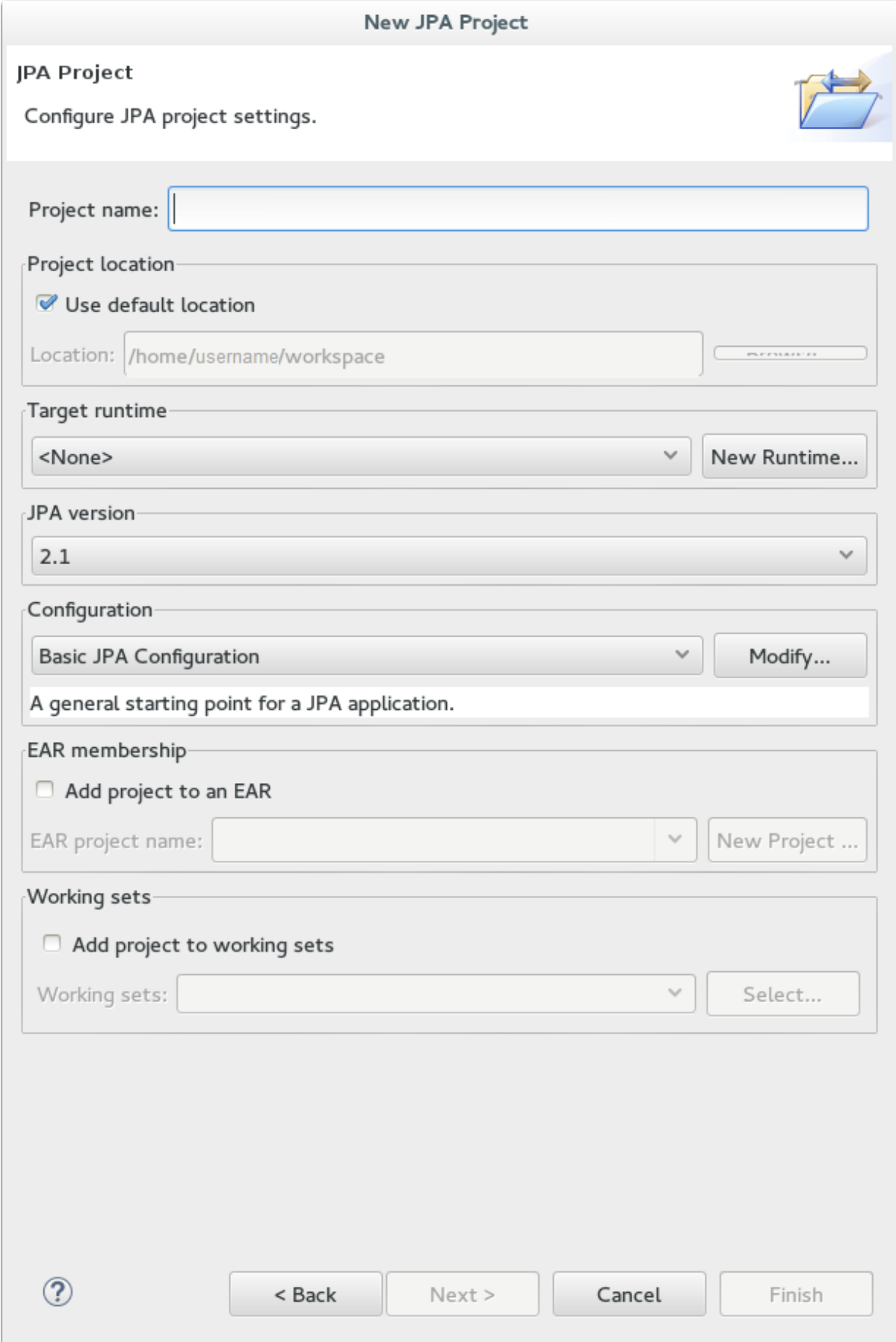
13.2.3.1. Create a JPA project in Red Hat JBoss Developer Studio

Summary

This example covers the steps required to create a JPA project in Red Hat JBoss Developer Studio.

Procedure 13.1. Create a JPA project in Red Hat JBoss Developer Studio

1. In the Red Hat JBoss Developer Studio window, click **File** → **New** → **Project**. Find **JPA** in the list, expand it, and select **JPA Project**. You are presented with the following dialog.



New JPA Project

JPA Project
Configure JPA project settings.

Project name:

Project location

☒ Use default location

Location:

Target runtime

JPA version

Configuration

A general starting point for a JPA application.

EAR membership

☐ Add project to an EAR

EAR project name:

Working sets

☐ Add project to working sets

Working sets:

2. Enter a **Project name**.
3. Select a **Target runtime**. If no target runtime is available, follow these instructions to define a new server and runtime: [Section 1.3.1.5, "Add the JBoss EAP Server Using Define New Server"](#).

4. Under **JPA version**, ensure **2.1** is selected.
5. Under **Configuration**, choose **Basic JPA Configuration**.
6. Click **Finish**.
7. If prompted, choose whether you wish to associate this type of project with the JPA perspective window.

[Report a bug](#)

13.2.3.2. Create the Persistence Settings File in Red Hat JBoss Developer Studio

Summary

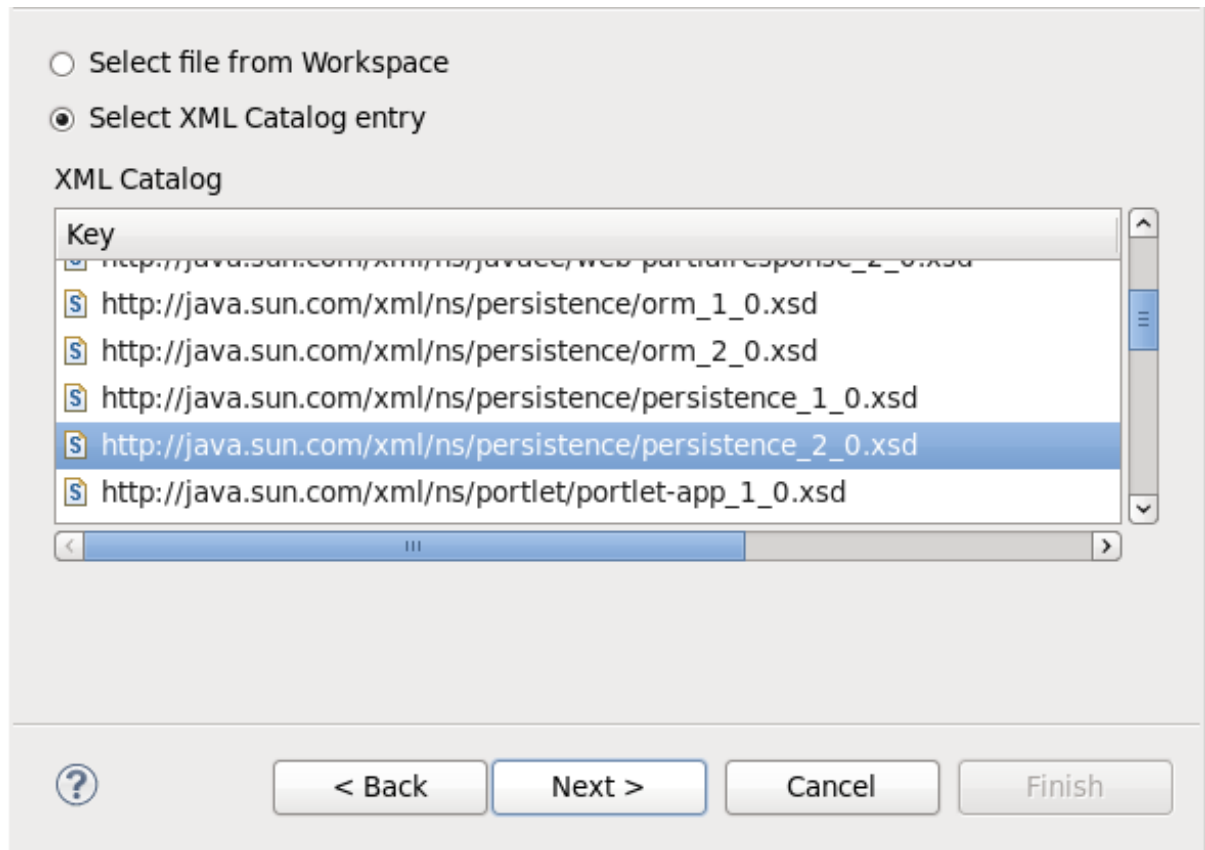
This topic covers the process for creating the **persistence.xml** file in a Java project using Red Hat JBoss Developer Studio.

Prerequisites

- [Section 1.3.1.4, “Start Red Hat JBoss Developer Studio”](#)

Procedure 13.2. Create and Configure a new Persistence Settings File

1. Open an EJB 3.x project in Red Hat JBoss Developer Studio.
2. Right click the project root directory in the **Project Explorer** panel.
3. Select **New** → **Other...**
4. Select **XML File** from the **XML** folder and click **Next**.
5. Select the **ejbModule/META-INF** folder as the parent directory.
6. Name the file **persistence.xml** and click **Next**.
7. Select **Create XML file from an XML schema file** and click **Next**.
8. Select **http://java.sun.com/xml/ns/persistence/persistence_2.0.xsd** from the **Select XML Catalog entry** list and click **Next**.



9. Click **Finish** to create the file.

Result:

The **persistence.xml** has been created in the **META-INF/** folder and is ready to be configured. An example file is available here: [Section 13.2.3.3, "Example Persistence Settings File"](#)

[Report a bug](#)

13.2.3.3. Example Persistence Settings File

Example 13.1. persistence.xml

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">
  <persistence-unit name="example" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-
source>
    <mapping-file>ormap.xml</mapping-file>
    <jar-file>TestApp.jar</jar-file>
    <class>org.test.Test</class>
    <shared-cache-mode>NONE</shared-cache-mode>
    <validation-mode>CALLBACK</validation-mode>
    <properties>
      <property name="hibernate.dialect"
value="org.hibernate.dialect.H2Dialect"/>
    </properties>
  </persistence-unit>
</persistence>
```

```
<property name="hibernate.hbm2ddl.auto" value="create-drop"/>
</properties>
</persistence-unit>
</persistence>
```

[Report a bug](#)

13.2.3.4. Create the Hibernate Configuration File in Red Hat JBoss Developer Studio

Prerequisites

- [Section 1.3.1.4, “Start Red Hat JBoss Developer Studio”](#)

Summary

This topic covers the process for creating the `hibernate.cfg.xml` file in a Java project using Red Hat JBoss Developer Studio.

Procedure 13.3. Create a New Hibernate Configuration File

1. Open a Java project in Red Hat JBoss Developer Studio.
2. Right click the project root directory in the **Project Explorer** panel.
3. Select **New** → **Other...**
4. Select **Hibernate Configuration File** from the **Hibernate** folder and click **Next**.
5. Select the `src/` directory and click **Next**.
6. Configure the following:
 - Session factory name
 - Database dialect
 - Driver class
 - Connection URL
 - Username
 - Password
7. Click **Finish** to create the file.

Result:

The `hibernate.cfg.xml` has been created in the `src/` folder. An example file is available here: [Section 13.2.3.5, “Example Hibernate Configuration File”](#).

[Report a bug](#)

13.2.3.5. Example Hibernate Configuration File

Example 13.2. hibernate.cfg.xml

```

<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <!-- Datasource Name -->
        <property name="connection.datasource">ExampleDS</property>

        <!-- SQL dialect -->
        <property
name="dialect">org.hibernate.dialect.H2Dialect</property>

        <!-- Enable Hibernate's automatic session context management --
>
        <property
name="current_session_context_class">thread</property>

        <!-- Disable the second-level cache -->
        <property
name="cache.region.factory_class">org.hibernate.cache.NoCacheProvider</p
roperty>

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Update the database schema on startup -->
        <property name="hbm2ddl.auto">update</property>

        <mapping
resource="org/hibernate/tutorial/domain/Event.hbm.xml"/>

    </session-factory>

</hibernate-configuration>

```

[Report a bug](#)

13.2.4. Configuration


13.2.4.1. Hibernate Configuration Properties

Table 13.1. Hibernate Java Properties

Property Name	Description
---------------	-------------

Property Name	Description
hibernate.dialect	<p>The classname of a Hibernate org.hibernate.dialect.Dialect. Allows Hibernate to generate SQL optimized for a particular relational database.</p> <p>In most cases Hibernate will be able to choose the correct org.hibernate.dialect.Dialect implementation, based on the JDBC metadata returned by the JDBC driver.</p>
hibernate.show_sql	Boolean. Writes all SQL statements to console. This is an alternative to setting the log category org.hibernate.SQL to debug .
hibernate.format_sql	Boolean. Pretty print the SQL in the log and console.
hibernate.default_schema	Qualify unqualified table names with the given schema/tablespace in generated SQL.
hibernate.default_catalog	Qualifies unqualified table names with the given catalog in generated SQL.
hibernate.session_factory_name	The org.hibernate.SessionFactory will be automatically bound to this name in JNDI after it has been created. For example, jndi/composite/name .
hibernate.max_fetch_depth	Sets a maximum "depth" for the outer join fetch tree for single-ended associations (one-to-one, many-to-one). A 0 disables default outer join fetching. The recommended value is between 0 and 3 .
hibernate.default_batch_fetch_size	Sets a default size for Hibernate batch fetching of associations. The recommended values are 4 , 8 , and 16 .
hibernate.default_entity_mode	Sets a default mode for entity representation for all sessions opened from this SessionFactory . Values include: dynamic-map , dom4j , pojo .
hibernate.order_updates	Boolean. Forces Hibernate to order SQL updates by the primary key value of the items being updated. This will result in fewer transaction deadlocks in highly concurrent systems.
hibernate.generate_statistics	Boolean. If enabled, Hibernate will collect statistics useful for performance tuning.
hibernate.use_identifier_rollback	Boolean. If enabled, generated identifier properties will be reset to default values when objects are deleted.
hibernate.use_sql_comments	Boolean. If turned on, Hibernate will generate comments inside the SQL, for easier debugging. Default value is false .

Property Name	Description
hibernate.id.new_generator_mappings	Boolean. This property is relevant when using @GeneratedValue . It indicates whether or not the new IdentifierGenerator implementations are used for javax.persistence.GenerationType.AUTO , javax.persistence.GenerationType.TABLE and javax.persistence.GenerationType.SEQUENCE . Default value is true .
hibernate.ejb.naming_strategy	<p>Chooses the org.hibernate.cfg.NamingStrategy implementation when using Hibernate EntityManager. This class is deprecated and this property is only provided for backward compatibility. This property must not be used with hibernate.ejb.naming_strategy_delegator.</p> <p>If the application does not use EntityManager, follow the instructions here to configure the NamingStrategy: Hibernate Reference Documentation - Implementing a Naming Strategy.</p>

Property Name	Description
hibernate.ejb.naming_strategy_delegator	<p>Specifies an org.hibernate.cfg.naming.NamingStrategyDelegator implementation for database objects and schema elements when using Hibernate EntityManager. This property has the following possible values.</p> <ul style="list-style-type: none"> • org.hibernate.cfg.naming.LegacyNamingStrategyDelegator: This is the default value. This class is deprecated and is only provided for backward compatibility. • org.hibernate.cfg.naming.ImprovedNamingStrategyDelegator: This is the preferred value. It generates default table and column names that comply with the JPA specification. It allows for specification of both the entity and foreign key class names. This class only affects entities that are mapped using Java annotations or JPA XML descriptors. Entities mapped using hbm.xml are not affected, • If you prefer, you can configure a custom class that implements org.hibernate.cfg.naming.ImprovedNamingStrategyDelegator <div>  <div> <p>NOTE</p> <p>This property must not be used with hibernate.ejb.naming_strategy. It is a temporary replacement for org.hibernate.cfg.NamingStrategy to address its limitations. A more comprehensive solution is planned for Hibernate 5.0 that replaces both org.hibernate.cfg.NamingStrategy and org.hibernate.cfg.naming.NamingStrategyDelegator.</p> </div> </div> <p>If the application does not use EntityManager, follow the instructions here to configure the NamingStrategy: Hibernate Reference Documentation - Implementing a Naming Strategy.</p>



IMPORTANT

For **hibernate.id.new_generator_mappings**, new applications should keep the default value of **true**. Existing applications that used Hibernate 3.3.x may need to change it to **false** to continue using a sequence object or table based generator, and maintain backward compatibility.

[Report a bug](#)

13.2.4.2. Hibernate JDBC and Connection Properties

Table 13.2. Properties

Property Name	Description
hibernate.jdbc.fetch_size	A non-zero value that determines the JDBC fetch size (calls Statement.setFetchSize()).
hibernate.jdbc.batch_size	A non-zero value enables use of JDBC2 batch updates by Hibernate. The recommended values are between 5 and 30 .
hibernate.jdbc.batch_versioned_data	Boolean. Set this property to true if the JDBC driver returns correct row counts from executeBatch() . Hibernate will then use batched DML for automatically versioned data. Default value is to false .
hibernate.jdbc.factory_class	Select a custom org.hibernate.jdbc.Batcher . Most applications will not need this configuration property.
hibernate.jdbc.use_scrollable_resultset	Boolean. Enables use of JDBC2 scrollable resultsets by Hibernate. This property is only necessary when using user-supplied JDBC connections. Hibernate uses connection metadata otherwise.
hibernate.jdbc.use_streams_for_binary	Boolean. This is a system-level property. Use streams when writing/reading binary or serializable types to/from JDBC.
hibernate.jdbc.use_get_generated_keys	Boolean. Enables use of JDBC3 PreparedStatement.getGeneratedKeys() to retrieve natively generated keys after insert. Requires JDBC3+ driver and JRE1.4+. Set to false if JDBC driver has problems with the Hibernate identifier generators. By default, it tries to determine the driver capabilities using connection metadata.
hibernate.connection.provider_class	The classname of a custom org.hibernate.connection.ConnectionProvider which provides JDBC connections to Hibernate.
hibernate.connection.isolation	Sets the JDBC transaction isolation level. Check java.sql.Connection for meaningful values, but note that most databases do not support all isolation levels and some define additional, non-standard isolations. Standard values are 1 , 2 , 4 , 8 .
hibernate.connection.autocommit	Boolean. This property is not recommended for use. Enables autocommit for JDBC pooled connections.

Property Name	Description
<code>hibernate.connection.release_mode</code>	<p>Specifies when Hibernate should release JDBC connections. By default, a JDBC connection is held until the session is explicitly closed or disconnected. The default value auto will choose after_statement for the JTA and CMT transaction strategies, and after_transaction for the JDBC transaction strategy.</p> <p>Available values are auto (default), on_close, after_transaction, after_statement.</p> <p>This setting only affects Session returned from SessionFactory.openSession. For Session obtained through SessionFactory.getCurrentSession, the CurrentSessionContext implementation configured for use controls the connection release mode for that Session.</p>
<code>hibernate.connection.<propertyName></code>	Pass the JDBC property <code><propertyName></code> to DriverManager.getConnection() .
<code>hibernate.jndi.<propertyName></code>	Pass the property <code><propertyName></code> to the JNDI InitialContextFactory .

[Report a bug](#)

13.2.4.3. Hibernate Cache Properties

Table 13.3. Properties

Property Name	Description
<code>hibernate.cache.region.factory_class</code>	The classname of a custom CacheProvider .
<code>hibernate.cache.use_minimal_puts</code>	Boolean. Optimizes second-level cache operation to minimize writes, at the cost of more frequent reads. This setting is most useful for clustered caches and, in Hibernate3, is enabled by default for clustered cache implementations.
<code>hibernate.cache.use_query_cache</code>	Boolean. Enables the query cache. Individual queries still have to be set cacheable.
<code>hibernate.cache.use_second_level_cache</code>	Boolean. Used to completely disable the second level cache, which is enabled by default for classes that specify a <cache> mapping.

Property Name	Description
<code>hibernate.cache.query_cache_factory</code>	The classname of a custom QueryCache interface. The default value is the built-in StandardQueryCache .
<code>hibernate.cache.region_prefix</code>	A prefix to use for second-level cache region names.
<code>hibernate.cache.use_structured_entries</code>	Boolean. Forces Hibernate to store data in the second-level cache in a more human-friendly format.
<code>hibernate.cache.default_cache_concurrency_strategy</code>	Setting used to give the name of the default org.hibernate.annotations.CacheConcurrencyStrategy to use when either @Cacheable or @Cache is used. @Cache(strategy="...") is used to override this default.

[Report a bug](#)

13.2.4.4. Hibernate Transaction Properties

Table 13.4. Properties

Property Name	Description
<code>hibernate.transaction.factory_class</code>	The classname of a TransactionFactory to use with Hibernate Transaction API. Defaults to JDBCTransactionFactory .
<code>jta.UserTransaction</code>	A JNDI name used by JTATransactionFactory to obtain the JTA UserTransaction from the application server.
<code>hibernate.transaction.manager_lookup_class</code>	The classname of a TransactionManagerLookup . It is required when JVM-level caching is enabled or when using hilo generator in a JTA environment.
<code>hibernate.transaction.flush_before_completion</code>	Boolean. If enabled, the session will be automatically flushed during the before completion phase of the transaction. Built-in and automatic session context management is preferred.
<code>hibernate.transaction.auto_close_session</code>	Boolean. If enabled, the session will be automatically closed during the after completion phase of the transaction. Built-in and automatic session context management is preferred.

[Report a bug](#)

13.2.4.5. Miscellaneous Hibernate Properties

Table 13.5. Properties

Property Name	Description
<code>hibernate.current_session_context_class</code>	Supply a custom strategy for the scoping of the "current" Session . Values include jta , thread , managed , custom.Class .
<code>hibernate.query.factory_class</code>	Chooses the HQL parser implementation: org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory or org.hibernate.hql.internal.classic.ClassicQueryTranslatorFactory .
<code>hibernate.query.substitutions</code>	Used to map from tokens in Hibernate queries to SQL tokens (tokens might be function or literal names). For example, hqlLiteral=SQL_LITERAL , hqlFunction=SQLFUNC .
<code>hibernate.hbm2ddl.auto</code>	Automatically validates or exports schema DDL to the database when the SessionFactory is created. With create-drop , the database schema will be dropped when the SessionFactory is closed explicitly. Property value options are validate , update , create , create-drop
<code>hibernate.hbm2ddl.import_files</code>	<p>Comma-separated names of the optional files containing SQL DML statements executed during the SessionFactory creation. This is useful for testing or demonstrating. For example, by adding INSERT statements, the database can be populated with a minimal set of data when it is deployed. An example value is /humans.sql, /dogs.sql.</p> <p>File order matters, as the statements of a given file are executed before the statements of the following files. These statements are only executed if the schema is created (i.e. if hibernate.hbm2ddl.auto is set to create or create-drop).</p>
<code>hibernate.hbm2ddl.import_files_sql_extractor</code>	The classname of a custom ImportSqlCommandExtractor . Defaults to the built-in SingleLineSqlCommandExtractor . This is useful for implementing a dedicated parser that extracts a single SQL statement from each import file. Hibernate also provides MultipleLinesSqlCommandExtractor , which supports instructions/comments and quoted strings spread over multiple lines (mandatory semicolon at the end of each statement).

Property Name	Description
hibernate.bytecode.use_reflection_optimizer	Boolean. This is a system-level property, which cannot be set in the hibernate.cfg.xml file. Enables the use of bytecode manipulation instead of runtime reflection. Reflection can sometimes be useful when troubleshooting. Hibernate always requires either cglib or javassist even if the optimizer is turned off.
hibernate.bytecode.provider	Both javassist or cglib can be used as byte manipulation engines. The default is javassist . Property value is either javassist or cglib

[Report a bug](#)

13.2.4.6. Hibernate SQL Dialects



IMPORTANT

The **hibernate.dialect** property should be set to the correct **org.hibernate.dialect.Dialect** subclass for the application database. If a dialect is specified, Hibernate will use sensible defaults for some of the other properties. This means that they do not have to be specified manually.

Table 13.6. SQL Dialects (hibernate.dialect)

RDBMS	Dialect
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
Firebird	org.hibernate.dialect.FirebirdDialect
FrontBase	org.hibernate.dialect.FrontbaseDialect
H2 Database	org.hibernate.dialect.H2Dialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Informix	org.hibernate.dialect.InformixDialect
Ingres	org.hibernate.dialect.IngresDialect

RDBMS	Dialect
Interbase	<code>org.hibernate.dialect.InterbaseDialect</code>
Mckoi SQL	<code>org.hibernate.dialect.MckoiDialect</code>
Microsoft SQL Server 2000	<code>org.hibernate.dialect.SQLServerDialect</code>
Microsoft SQL Server 2005	<code>org.hibernate.dialect.SQLServer2005Dialect</code>
Microsoft SQL Server 2008	<code>org.hibernate.dialect.SQLServer2008Dialect</code>
Microsoft SQL Server 2012	<code>org.hibernate.dialect.SQLServer2008Dialect</code>
MySQL5	<code>org.hibernate.dialect.MySQL5Dialect</code>
MySQL5 with InnoDB	<code>org.hibernate.dialect.MySQL5InnoDBDialect</code>
MySQL with MyISAM	<code>org.hibernate.dialect.MySQLMyISAMDialect</code>
Oracle (any version)	<code>org.hibernate.dialect.OracleDialect</code>
Oracle 9i	<code>org.hibernate.dialect.Oracle9iDialect</code>
Oracle 10g	<code>org.hibernate.dialect.Oracle10gDialect</code>
Oracle 11g	<code>org.hibernate.dialect.Oracle10gDialect</code>
Pointbase	<code>org.hibernate.dialect.PointbaseDialect</code>
PostgreSQL	<code>org.hibernate.dialect.PostgreSQLDialect</code>
PostgreSQL 9.2	<code>org.hibernate.dialect.PostgreSQL82Dialect</code>
Postgres Plus Advanced Server	<code>org.hibernate.dialect.PostgresPlusDialect</code>

RDBMS	Dialect
Progress	<code>org.hibernate.dialect.ProgressDialect</code>
SAP DB	<code>org.hibernate.dialect.SAPDBDialect</code>
Sybase	<code>org.hibernate.dialect.SybaseASE15Dialect</code>
Sybase 15.7	<code>org.hibernate.dialect.SybaseASE157Dialect</code>
Sybase Anywhere	<code>org.hibernate.dialect.SybaseAnywhereDialect</code>

[Report a bug](#)

13.2.5. Second-Level Caches

13.2.5.1. About Second-Level Caches

A second-level cache is a local data store that holds information persisted outside the application session. The cache is managed by the persistence provider, improving run-time by keeping the data separate from the application.

JBoss EAP 6 supports caching for the following purposes:

- Web Session Clustering
- Stateful Session Bean Clustering
- SSO Clustering
- Hibernate Second Level Cache

Each cache container defines a "repl" and a "dist" cache. These caches should not be used directly by user applications.

[Report a bug](#)

13.2.5.2. Configure a Second Level Cache for Hibernate

This topic covers the configuration requirements for enabling Infinispan to act as the second level cache for Hibernate.

Procedure 13.4. Create and Edit the `hibernate.cfg.xml` file

1. Create the `hibernate.cfg.xml` file

Create the `hibernate.cfg.xml` in the deployment's classpath. For specifics, refer to [Section 13.2.3.4, "Create the Hibernate Configuration File in Red Hat JBoss Developer Studio"](#).

2. Add these lines of XML to the **hibernate.cfg.xml** file in your application. The XML needs to be inside the `<session-factory>` tags:

```
<property
name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.use_query_cache">true</property>
```

3. Add one of the following to the `<session-factory>` section of the **hibernate.cfg.xml** file:

- **If the Infinispan CacheManager is bound to JNDI:**

```
<property name="hibernate.cache.region.factory_class">
    org.hibernate.cache.infinispan.JndiInfinispanRegionFactory
</property>
<property name="hibernate.cache.infinispan.cachemanager">
    java:CacheManager
</property>
```

- **If the Infinispan CacheManager is standalone:**

```
<property name="hibernate.cache.region.factory_class">
    org.hibernate.cache.infinispan.InfinispanRegionFactory
</property>
```

Result

Infinispan is configured as the Second Level Cache for Hibernate.

[Report a bug](#)

13.3. HIBERNATE ANNOTATIONS

13.3.1. Hibernate Annotations

Table 13.7. Hibernate Defined Annotations

Annotation	Description
AccessType	Property Access type.
Any	Defines a ToOne association pointing to several entity types. Matching the according entity type is done through a metadata discriminator column. This kind of mapping should be only marginal.
AnyMetaDef	Defines @Any and @ManyToMany metadata.
AnyMedaDefs	Defines @Any and @ManyToMany set of metadata. Can be defined at the entity level or the package level.

Annotation	Description
BatchSize	Batch size for SQL loading.
Cache	Add caching strategy to a root entity or a collection.
Cascade	Apply a cascade strategy on an association.
Check	Arbitrary SQL check constraints which can be defined at the class, property or collection level.
Columns	Support an array of columns. Useful for component user type mappings.
ColumnTransformer	Custom SQL expression used to read the value from and write a value to a column. Use for direct object loading/saving as well as queries. The write expression must contain exactly one '?' placeholder for the value.
ColumnTransformers	Plural annotation for @ColumnTransformer. Useful when more than one column is using this behavior.
DiscriminatorFormula	Discriminator formula to be placed at the root entity.
DiscriminatorOptions	Optional annotation to express Hibernate specific discriminator properties.
Entity	Extends Entity with Hibernate features.
Fetch	Defines the fetching strategy used for the given association.
FetchProfile	Defines the fetching strategy profile.
FetchProfiles	Plural annotation for @FetchProfile.
Filter	Adds filters to an entity or a target entity of a collection.
FilterDef	Filter definition.
FilterDefs	Array of filter definitions.
FilterJoinTable	Adds filters to a join table collection.
FilterJoinTables	Adds multiple @FilterJoinTable to a collection.

Annotation	Description
Filters	Adds multiple @Filters.
Formula	To be used as a replacement for @Column in most places. The formula has to be a valid SQL fragment.
Generated	This annotated property is generated by the database.
GenericGenerator	Generator annotation describing any kind of Hibernate generator in a detyped manner.
GenericGenerators	Array of generic generator definitions.
Immutable	<p>Mark an Entity or a Collection as immutable. No annotation means the element is mutable.</p> <p>An immutable entity may not be updated by the application. Updates to an immutable entity will be ignored, but no exception is thrown.</p> <p>@Immutable placed on a collection makes the collection immutable, meaning additions and deletions to and from the collection are not allowed. A HibernateException is thrown in this case.</p>
Index	Defines a database index.
JoinFormula	To be used as a replacement for @JoinColumn in most places. The formula has to be a valid SQL fragment.
LazyCollection	Defines the lazy status of a collection.
LazyToOne	Defines the lazy status of a ToOne association (i.e. OneToOne or ManyToOne).
Loader	Overwrites Hibernate default FIND method.
ManyToMany	Defines a ToMany association pointing to different entity types. Matching the according entity type is done through a metadata discriminator column. This kind of mapping should be only marginal.
MapKeyType	Defines the type of key of a persistent map.
MetaValue	Represents a discriminator value associated to a given entity type.

Annotation	Description
NamedNativeQueries	Extends NamedNativeQueries to hold Hibernate NamedNativeQuery objects.
NamedNativeQuery	Extends NamedNativeQuery with Hibernate features.
NamedQueries	Extends NamedQueries to hold Hibernate NamedQuery objects.
NamedQuery	Extends NamedQuery with Hibernate features.
NaturalId	Specifies that a property is part of the natural id of the entity.
NotFound	Action to do when an element is not found on an association.
OnDelete	Strategy to use on collections, arrays and on joined subclasses delete. OnDelete of secondary tables is currently not supported.
OptimisticLock	Whether or not a change of the annotated property will trigger an entity version increment. If the annotation is not present, the property is involved in the optimistic lock strategy (default).
OptimisticLocking	Used to define the style of optimistic locking to be applied to an entity. In a hierarchy, only valid on the root entity.
OrderBy	Order a collection using SQL ordering (not HQL ordering).
ParamDef	A parameter definition.
Parameter	Key/value pattern.
Parent	Reference the property as a pointer back to the owner (generally the owning entity).
Persister	Specify a custom persister.
Polymorphism	Used to define the type of polymorphism Hibernate will apply to entity hierarchies.
Proxy	Lazy and proxy configuration of a particular class.
RowId	Support for ROWID mapping feature of Hibernate.

Annotation	Description
Sort	Collection sort (Java level sorting).
Source	Optional annotation in conjunction with Version and timestamp version properties. The annotation value decides where the timestamp is generated.
SQLDelete	Overwrites the Hibernate default DELETE method.
SQLDeleteAll	Overwrites the Hibernate default DELETE ALL method.
SQLInsert	Overwrites the Hibernate default INSERT INTO method.
SQLUpdate	Overwrites the Hibernate default UPDATE method.
Subselect	Maps an immutable and read-only entity to a given SQL subselect expression.
Synchronize	Ensures that auto-flush happens correctly and that queries against the derived entity do not return stale data. Mostly used with Subselect.
Table	Complementary information to a table either primary or secondary.
Tables	Plural annotation of Table.
Target	Defines an explicit target, avoiding reflection and generics resolving.
Tuplizer	Defines a tuplizer for an entity or a component.
Tuplizers	Defines a set of tuplizers for an entity or a component.
Type	Hibernate Type.
TypeDef	Hibernate Type definition.
TypeDefs	Hibernate Type definition array.
Where	Where clause to add to the element Entity or target entity of a collection. The clause is written in SQL.

Annotation	Description
WhereJoinTable	Where clause to add to the collection join table. The clause is written in SQL.

**NOTE**

The annotation "Entity" is deprecated and scheduled for removal in future releases.

[Report a bug](#)

13.4. HIBERNATE QUERY LANGUAGE

13.4.1. About Hibernate Query Language

The Hibernate Query Language (HQL) and Java Persistence Query Language (JPQL) are both object model focused query languages similar in nature to SQL. HQL is a superset of JPQL. A HQL query is not always a valid JPQL query, but a JPQL query is always a valid HQL query.

Both HQL and JPQL are non-type-safe ways to perform query operations. Criteria queries offer a type-safe approach to querying.

[Report a bug](#)

13.4.2. HQL Statements

HQL allows **SELECT**, **UPDATE**, **DELETE**, and **INSERT** statements. The HQL **INSERT** statement has no equivalent in JPQL.

**IMPORTANT**

Care should be taken as to when an **UPDATE** or **DELETE** statement is executed.

Table 13.8. HQL Statements

Statement	Description
-----------	-------------

Statement	Description
SELECT	<p>The BNF for SELECT statements in HQL is:</p> <pre>select_statement ::= [select_clause] from_clause [where_clause] [groupby_clause] [having_clause] [orderby_clause]</pre> <p>The simplest possible HQL SELECT statement is of the form:</p> <pre>from com.acme.Cat</pre>
UPDATE	The BNF for UPDATE statement in HQL is the same as it is in JPQL
DELETE	The BNF for DELETE statements in HQL is the same as it is in JPQL

[Report a bug](#)

13.4.3. About the INSERT Statement

HQL adds the ability to define **INSERT** statements. There is no JPQL equivalent to this. The BNF for an HQL **INSERT** statement is:

```
insert_statement ::= insert_clause select_statement
insert_clause ::= INSERT INTO entity_name (attribute_list)
attribute_list ::= state_field[, state_field ]*
```

The **attribute_list** is analogous to the **column specification** in the SQL **INSERT** statement. For entities involved in mapped inheritance, only attributes directly defined on the named entity can be used in the **attribute_list**. Superclass properties are not allowed and subclass properties do not make sense. In other words, **INSERT** statements are inherently non-polymorphic.



WARNING

select_statement can be any valid HQL select query, with the caveat that the return types must match the types expected by the insert. Currently, this is checked during query compilation rather than allowing the check to relegate to the database. This may cause problems between Hibernate Types which are *equivalent* as opposed to *equal*. For example, this might cause lead to issues with mismatches between an attribute mapped as a `org.hibernate.type.DateType` and an attribute defined as a `org.hibernate.type.TimestampType`, even though the database might not make a distinction or might be able to handle the conversion.

For the **id** attribute, the insert statement gives you two options. You can either explicitly specify the **id** property in the **attribute_list**, in which case its value is taken from the corresponding select expression, or omit it from the **attribute_list** in which case a generated value is used. This latter option is only available when using **id** generators that operate "in the database"; attempting to use this option with any "in memory" type generators will cause an exception during parsing.

For optimistic locking attributes, the insert statement again gives you two options. You can either specify the attribute in the **attribute_list** in which case its value is taken from the corresponding select expressions, or omit it from the **attribute_list** in which case the **seed value** defined by the corresponding `org.hibernate.type.VersionType` is used.

Example 13.3. INSERT Query Statements

```
String hqlInsert = "insert into DelinquentAccount (id, name) select  
c.id, c.name from Customer c where ...";  
int createdEntities = s.createQuery( hqlInsert ).executeUpdate();
```

[Report a bug](#)

13.4.4. About the FROM Clause

The **FROM** clause is responsible defining the scope of object model types available to the rest of the query. It also is responsible for defining all the "identification variables" available to the rest of the query.

[Report a bug](#)

13.4.5. About the WITH Clause

HQL defines a **WITH** clause to qualify the join conditions. This is specific to HQL; JPQL does not define this feature.

Example 13.4. With Clause

```
select distinct c  
from Customer c  
left join c.orders o
```

```
with o.value > 5000.00
```

The important distinction is that in the generated SQL the conditions of the **with clause** are made part of the **on clause** in the generated SQL as opposed to the other queries in this section where the HQL/JPQL conditions are made part of the **where clause** in the generated SQL. The distinction in this specific example is probably not that significant. The **with clause** is sometimes necessary in more complicated queries.

Explicit joins may reference association or component/embedded attributes. In the case of component/embedded attributes, the join is logical and does not correlate to a physical (SQL) join.

[Report a bug](#)

13.4.6. About Bulk Update, Insert and Delete

Hibernate allows the use of Data Manipulation Language (DML) to bulk insert, update and delete data directly in the mapped database through the Hibernate Query Language.



WARNING

Using DML may violate the object/relational mapping and may affect object state. Object state stays in memory and by using DML, the state of an in-memory object is not affected depending on the operation that is performed on the underlying database. In-memory data must be used with care if DML is used.

The pseudo-syntax for UPDATE and DELETE statements is: (**UPDATE | DELETE**) **FROM?** **EntityName** (**WHERE** **where_conditions**)?.



NOTE

The **FROM** keyword and the **WHERE clause** are optional.

The result of execution of a UPDATE or DELETE statement is the number of rows that are actually affected (updated or deleted).

Example 13.5. Bulk Update Statement

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlUpdate = "update Company set name = :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
    .setString( "newName", newName )
    .setString( "oldName", oldName )
```

```

        .executeUpdate();
tx.commit();
session.close();

```

Example 13.6. Bulk Delete statement

```

Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlDelete = "delete Company where name = :oldName";
int deletedEntities = s.createQuery( hqlDelete )
    .setString( "oldName", oldName )
    .executeUpdate();
tx.commit();
session.close();

```

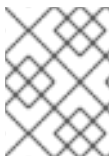
The **int** value returned by the **Query.executeUpdate()** method indicates the number of entities within the database that were affected by the operation.

Internally, the database might use multiple SQL statements to execute the operation in response to a DML Update or Delete request. This might be because of relationships that exist between tables and the join tables that may need to be updated or deleted.

For example, issuing a delete statement (as in the example above) may actually result in deletes being executed against not just the **Company** table for companies that are named with **oldName**, but also against joined tables. Thus, a Company table in a BiDirectional ManyToMany relationship with an Employee table, would lose rows from the corresponding join table **Company_Employee** as a result of the successful execution of the previous example.

The **int deletedEntities** value above will contain a count of all the rows affected due to this operation, including the rows in the join tables.

The pseudo-syntax for INSERT statements is: **INSERT INTO EntityName properties_list select_statement**.



NOTE

Only the INSERT INTO ... SELECT ... form is supported; not the INSERT INTO ... VALUES ... form.

Example 13.7. Bulk Insert statement

```

Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlInsert = "insert into Account (id, name) select c.id, c.name
from Customer c where ...";
int createdEntities = s.createQuery( hqlInsert )
    .executeUpdate();
tx.commit();
session.close();

```

If you do not supply the value for the `id` attribute via the `SELECT` statement, an identifier is generated for you, as long as the underlying database supports auto-generated keys. The return value of this bulk insert operation is the number of entries actually created in the database.

[Report a bug](#)

13.4.7. About Collection Member References

References to collection-valued associations actually refer to the *values* of that collection.

Example 13.8. Collection References

```
select c
from Customer c
    join c.orders o
    join o.lineItems l
    join l.product p
where o.status = 'pending'
    and p.status = 'backorder'

// alternate syntax
select c
from Customer c,
    in(c.orders) o,
    in(o.lineItems) l
    join l.product p
where o.status = 'pending'
    and p.status = 'backorder'
```

In the example, the identification variable `o` actually refers to the object model type `Order` which is the type of the elements of the `Customer#orders` association.

The example also shows the alternate syntax for specifying collection association joins using the `IN` syntax. Both forms are equivalent. Which form an application chooses to use is simply a matter of taste.

[Report a bug](#)

13.4.8. About Qualified Path Expressions

It was previously stated that collection-valued associations actually refer to the *values* of that collection. Based on the type of collection, there are also available a set of explicit qualification expressions.

Table 13.9. Qualified Path Expressions

Expression	Description
VALUE	Refers to the collection value. Same as not specifying a qualifier. Useful to explicitly show intent. Valid for any type of collection-valued reference.

Expression	Description
INDEX	According to HQL rules, this is valid for both Maps and Lists which specify a <code>javax.persistence.OrderColumn</code> annotation to refer to the Map key or the List position (aka the OrderColumn value). JPQL however, reserves this for use in the List case and adds KEY for the MAP case. Applications interested in JPA provider portability should be aware of this distinction.
KEY	Valid only for Maps. Refers to the map's key. If the key is itself an entity, can be further navigated.
ENTRY	Only valid only for Maps. Refers to the Map's logical <code>java.util.Map.Entry</code> tuple (the combination of its key and value). ENTRY is only valid as a terminal path and only valid in the select clause.

Example 13.9. Qualified Collection References

```
// Product.images is a Map<String,String> : key = a name, value = file
path

// select all the image file paths (the map value) for Product#123
select i
from Product p
    join p.images i
where p.id = 123

// same as above
select value(i)
from Product p
    join p.images i
where p.id = 123

// select all the image names (the map key) for Product#123
select key(i)
from Product p
    join p.images i
where p.id = 123

// select all the image names and file paths (the 'Map.Entry') for
Product#123
select entry(i)
from Product p
    join p.images i
where p.id = 123

// total the value of the initial line items for all orders for a
customer
select sum( li.amount )
from Customer c
```

```

        join c.orders o
        join o.lineItems li
    where c.id = 123
        and index(li) = 1

```

[Report a bug](#)

13.4.9. About Scalar Functions

HQL defines some standard functions that are available regardless of the underlying database in use. HQL can also understand additional functions defined by the dialect and the application.

[Report a bug](#)

13.4.10. HQL Standardized Functions

The following functions are available in HQL regardless of the underlying database in use.

Table 13.10. HQL Standardized Functions

Function	Description
BIT_LENGTH	Returns the length of binary data.
CAST	Performs a SQL cast. The cast target should name the Hibernate mapping type to use.
EXTRACT	Performs a SQL extraction on datetime values. An extraction extracts parts of the datetime (the year, for example). See the abbreviated forms below.
SECOND	Abbreviated extract form for extracting the second.
MINUTE	Abbreviated extract form for extracting the minute.
HOURL	Abbreviated extract form for extracting the hour.
DAY	Abbreviated extract form for extracting the day.
MONTH	Abbreviated extract form for extracting the month.
YEAR	Abbreviated extract form for extracting the year.
STR	Abbreviated form for casting a value as character data.

Application developers can also supply their own set of functions. This would usually represent either custom SQL functions or aliases for snippets of SQL. Such function declarations are made by using the **addSqlFunction** method of **org.hibernate.cfg.Configuration**

[Report a bug](#)

13.4.11. About the Concatenation Operation

HQL defines a concatenation operator in addition to supporting the concatenation (**CONCAT**) function. This is not defined by JPQL, so portable applications should avoid using it. The concatenation operator is taken from the SQL concatenation operator - `||`.

Example 13.10. Concatenation Operation Example

```
select 'Mr. ' || c.name.first || ' ' || c.name.last
from Customer c
where c.gender = Gender.MALE
```

[Report a bug](#)

13.4.12. About Dynamic Instantiation

There is a particular expression type that is only valid in the select clause. Hibernate calls this "dynamic instantiation". JPQL supports some of this feature and calls it a "constructor expression".

Example 13.11. Dynamic Instantiation Example - Constructor

```
select new Family( mother, mate, offspr )
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

So rather than dealing with the `Object[]` here we are wrapping the values in a type-safe java object that will be returned as the results of the query. The class reference must be fully qualified and it must have a matching constructor.

The class here need not be mapped. If it does represent an entity, the resulting instances are returned in the NEW state (not managed!).

This is the part JPQL supports as well. HQL supports additional "dynamic instantiation" features. First, the query can specify to return a `List` rather than an `Object[]` for scalar results:

Example 13.12. Dynamic Instantiation Example - List

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

The results from this query will be a `List<List>` as opposed to a `List<Object[]>`

HQL also supports wrapping the scalar results in a `Map`.

Example 13.13. Dynamic Instantiation Example - Map

```

select new map( mother as mother, offspr as offspr, mate as mate )
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr

select new map( max(c.bodyWeight) as max, min(c.bodyWeight) as min,
count(*) as n )
from Cat cxt

```

The results from this query will be a `List<Map<String,Object>>` as opposed to a `List<Object[]>`. The keys of the map are defined by the aliases given to the select expressions.

[Report a bug](#)

13.4.13. About HQL Predicates

Predicates form the basis of the where clause, the having clause and searched case expressions. They are expressions which resolve to a truth value, generally **TRUE** or **FALSE**, although boolean comparisons involving NULLs generally resolve to **UNKNOWN**.

HQL Predicates**Nullness Predicate**

Check a value for nullness. Can be applied to basic attribute references, entity references and parameters. HQL additionally allows it to be applied to component/embeddable types.

Example 13.14. Nullness Checking Examples

```

// select everyone with an associated address
select p
from Person p
where p.address is not null

// select everyone without an associated address
select p
from Person p
where p.address is null

```

Like Predicate

Performs a like comparison on string values. The syntax is:

```

like_expression ::=
    string_expression
    [NOT] LIKE pattern_value
    [ESCAPE escape_character]

```

The semantics follow that of the SQL like expression. The **pattern_value** is the pattern to attempt to match in the **string_expression**. Just like SQL, **pattern_value** can use `"_"` and `"%"` as

wildcards. The meanings are the same. "_" matches any single character. "%" matches any number of characters.

The optional **escape_character** is used to specify an escape character used to escape the special meaning of "_" and "%" in the **pattern_value**. This is useful when needing to search on patterns including either "_" or "%".

Example 13.15. Like Predicate Examples

```
select p
from Person p
where p.name like '%Schmidt'

select p
from Person p
where p.name not like 'Jingleheimer%'

// find any with name starting with "sp_"
select sp
from StoredProcedureMetadata sp
where sp.name like 'sp|_%' escape '|'

```

Between Predicate

Analogous to the SQL **BETWEEN** expression. Perform an evaluation that a value is within the range of 2 other values. All the operands should have comparable types.

Example 13.16. Between Predicate Examples

```
select p
from Customer c
    join c.paymentHistory p
where c.id = 123
    and index(p) between 0 and 9

select c
from Customer c
where c.president.dateOfBirth
    between {d '1945-01-01'}
        and {d '1965-01-01'}

select o
from Order o
where o.total between 500 and 5000

select p
from Person p
where p.name between 'A' and 'E'

```

13.4.14. About Relational Comparisons

Comparisons involve one of the comparison operators - =, >, >=, <, <=, <>. HQL also defines != as a comparison operator synonymous with <>. The operands should be of the same type.

Example 13.17. Relational Comparison Examples

```
// numeric comparison
select c
from Customer c
where c.chiefExecutive.age < 30

// string comparison
select c
from Customer c
where c.name = 'Acme'

// datetime comparison
select c
from Customer c
where c.inceptionDate < {d '2000-01-01'}

// enum comparison
select c
from Customer c
where c.chiefExecutive.gender = com.acme.Gender.MALE

// boolean comparison
select c
from Customer c
where c.sendEmail = true

// entity type comparison
select p
from Payment p
where type(p) = WireTransferPayment

// entity value comparison
select c
from Customer c
where c.chiefExecutive = c.chiefTechnologist
```

Comparisons can also involve subquery qualifiers - **ALL**, **ANY**, **SOME**. **SOME** and **ANY** are synonymous.

The **ALL** qualifier resolves to true if the comparison is true for all of the values in the result of the subquery. It resolves to false if the subquery result is empty.

Example 13.18. ALL Subquery Comparison Qualifier Example

```
// select all players that scored at least 3 points
// in every game.
select p
from Player p
where 3 > all (
```

```

        select spg.points
        from StatsPerGame spg
        where spg.player = p
    )

```

The **ANY/SOME** qualifier resolves to true if the comparison is true for some of (at least one of) the values in the result of the subquery. It resolves to false if the subquery result is empty.

[Report a bug](#)

13.4.15. About the IN Predicate

The **IN** predicate performs a check that a particular value is in a list of values. Its syntax is:

```

in_expression ::= single_valued_expression
                [NOT] IN single_valued_list

single_valued_list ::= constructor_expression |
                     (subquery) |
                     collection_valued_input_parameter

constructor_expression ::= (expression[, expression]*)

```

The types of the **single_valued_expression** and the individual values in the **single_valued_list** must be consistent. JPQL limits the valid types here to string, numeric, date, time, timestamp, and enum types. In JPQL, **single_valued_expression** can only refer to:

- "state fields", which is its term for simple attributes. Specifically this excludes association and component/embedded attributes.
- entity type expressions.

In HQL, **single_valued_expression** can refer to a far more broad set of expression types. Single-valued association are allowed. So are component/embedded attributes, although that feature depends on the level of support for tuple or "row value constructor syntax" in the underlying database. Additionally, HQL does not limit the value type in any way, though application developers should be aware that different types may incur limited support based on the underlying database vendor. This is largely the reason for the JPQL limitations.

The list of values can come from a number of different sources. In the **constructor_expression** and **collection_valued_input_parameter**, the list of values must not be empty; it must contain at least one value.

Example 13.19. In Predicate Examples

```

select p
from Payment p
where type(p) in (CreditCardPayment, WireTransferPayment)

select c
from Customer c
where c.hqAddress.state in ('TX', 'OK', 'LA', 'NM')

```

```

select c
from Customer c
where c.hqAddress.state in ?

select c
from Customer c
where c.hqAddress.state in (
    select dm.state
    from DeliveryMetadata dm
    where dm.salesTax is not null
)

// Not JPQL compliant!
select c
from Customer c
where c.name in (
    ('John', 'Doe'),
    ('Jane', 'Doe')
)

// Not JPQL compliant!
select c
from Customer c
where c.chiefExecutive in (
    select p
    from Person p
    where ...
)

```

[Report a bug](#)

13.4.16. About HQL Ordering

The results of the query can also be ordered. The **ORDER BY** clause is used to specify the selected values to be used to order the result. The types of expressions considered valid as part of the order-by clause include:

- state fields
- component/embeddable attributes
- scalar expressions such as arithmetic operations, functions, etc.
- identification variable declared in the select clause for any of the previous expression types

HQL does not mandate that all values referenced in the order-by clause must be named in the select clause, but it is required by JPQL. Applications desiring database portability should be aware that not all databases support referencing values in the order-by clause that are not referenced in the select clause.

Individual expressions in the order-by can be qualified with either **ASC** (ascending) or **DESC** (descending) to indicated the desired ordering direction.

Example 13.20. Order-by Examples

```
// legal because p.name is implicitly part of p
select p
from Person p
order by p.name

select c.id, sum( o.total ) as t
from Order o
      inner join o.customer c
group by c.id
order by t
```

[Report a bug](#)

13.5. HIBERNATE SERVICES

13.5.1. About Hibernate Services

Services are classes that provide Hibernate with pluggable implementations of various types of functionality. Specifically they are implementations of certain service contract interfaces. The interface is known as the service role; the implementation class is known as the service implementation. Generally speaking, users can plug in alternate implementations of all standard service roles (overriding); they can also define additional services beyond the base set of service roles (extending).

[Report a bug](#)

13.5.2. About Service Contracts

The basic requirement for a service is to implement the marker interface **org.hibernate.service.Service**. Hibernate uses this internally for some basic type safety.

Optionally, the service can also implement the **org.hibernate.service.spi.Startable** and **org.hibernate.service.spi.Stoppable** interfaces to receive notifications of being started and stopped. Another optional service contract is **org.hibernate.service.spi.Manageable** which marks the service as manageable in JMX provided the JMX integration is enabled.

[Report a bug](#)

13.5.3. Types of Service Dependencies

Services are allowed to declare dependencies on other services using either of 2 approaches:

@org.hibernate.service.spi.InjectService

Any method on the service implementation class accepting a single parameter and annotated with **@InjectService** is considered requesting injection of another service.

By default the type of the method parameter is expected to be the service role to be injected. If the parameter type is different than the service role, the **serviceRole** attribute of the **InjectService** should be used to explicitly name the role.

By default injected services are considered required, that is the start up will fail if a named dependent service is missing. If the service to be injected is optional, the **required** attribute of the **InjectService** should be declared as **false** (default is **true**).

org.hibernate.service.spi.ServiceRegistryAwareService

The second approach is a pull approach where the service implements the optional service interface **org.hibernate.service.spi.ServiceRegistryAwareService** which declares a single **injectServices** method.

During startup, Hibernate will inject the **org.hibernate.service.ServiceRegistry** itself into services which implement this interface. The service can then use the **ServiceRegistry** reference to locate any additional services it needs.

[Report a bug](#)

13.5.4. The ServiceRegistry

13.5.4.1. About the ServiceRegistry

The central service API, aside from the services themselves, is the **org.hibernate.service.ServiceRegistry** interface. The main purpose of a service registry is to hold, manage and provide access to services.

Service registries are hierarchical. Services in one registry can depend on and utilize services in that same registry as well as any parent registries.

Use **org.hibernate.service.ServiceRegistryBuilder** to build a **org.hibernate.service.ServiceRegistry** instance.

Example 13.21. Use ServiceRegistryBuilder to create a ServiceRegistry

```
ServiceRegistryBuilder registryBuilder = new ServiceRegistryBuilder(  
    bootstrapServiceRegistry );  
    ServiceRegistry serviceRegistry =  
    registryBuilder.buildServiceRegistry();
```

[Report a bug](#)

13.5.5. Custom Services

13.5.5.1. About Custom Services

Once a **org.hibernate.service.ServiceRegistry** is built it is considered immutable; the services themselves might accept re-configuration, but immutability here means adding/replacing services. So another role provided by the **org.hibernate.service.ServiceRegistryBuilder** is to allow tweaking of the services that will be contained in the **org.hibernate.service.ServiceRegistry** generated from it.

There are two means to tell a **org.hibernate.service.ServiceRegistryBuilder** about custom services.

- Implement a **org.hibernate.service.spi.BasicServiceInitiator** class to control on-demand construction of the service class and add it to the **org.hibernate.service.ServiceRegistryBuilder** via its **addInitiator** method.

- Just instantiate the service class and add it to the `org.hibernate.service.ServiceRegistryBuilder` via its `addService` method.

Either approach the adding a service approach or the adding an initiator approach are valid for extending a registry (adding new service roles) and overriding services (replacing service implementations).

Example 13.22. Use ServiceRegistryBuilder to Replace an Existing Service with a Custom Service

```
ServiceRegistryBuilder registryBuilder = new ServiceRegistryBuilder(
bootstrapServiceRegistry );
registryBuilder.addService( JdbcServices.class, new FakeJdbcService()
);
ServiceRegistry serviceRegistry =
registryBuilder.buildServiceRegistry();

public class FakeJdbcService implements JdbcServices{

    @Override
    public ConnectionProvider getConnectionProvider() {
        return null;
    }

    @Override
    public Dialect getDialect() {
        return null;
    }

    @Override
    public SqlStatementLogger getSqlStatementLogger() {
        return null;
    }

    @Override
    public SQLExceptionHelper getSQLExceptionHelper() {
        return null;
    }

    @Override
    public ExtractedDatabaseMetaData getExtractedMetaDataSupport() {
        return null;
    }

    @Override
    public LobCreator getLobCreator(LobCreationContext
lobCreationContext) {
        return null;
    }

    @Override
    public ResultSetWrapper getResultSetWrapper() {
        return null;
    }
}
```

■ [Report a bug](#)

13.5.6. The Bootstrap Registry

13.5.6.1. About the Boot-strap Registry

The boot-strap registry holds services that absolutely have to be available for most things to work. The main service here is the **ClassLoaderService** which is a perfect example. Even resolving configuration files needs access to class loading services (resource look ups). This is the root registry (no parent) in normal use.

Instances of boot-strap registries are built using the **org.hibernate.service.BootstrapServiceRegistryBuilder** class.

[Report a bug](#)

13.5.6.2. Using BootstrapServiceRegistryBuilder

Example 13.23. Using BootstrapServiceRegistryBuilder

```
BootstrapServiceRegistry bootstrapServiceRegistry = new
BootstrapServiceRegistryBuilder()
    // pass in org.hibernate.integrator.spi.Integrator instances
    which are not
    // auto-discovered (for whatever reason) but which should be
    included
    .with( anExplicitIntegrator )
    // pass in a class loader that Hibernate should use to load
    application classes
    .with( anExplicitClassLoaderForApplicationClasses )
    // pass in a class loader that Hibernate should use to load
    resources
    .with( anExplicitClassLoaderForResources )
    // see BootstrapServiceRegistryBuilder for rest of available
    methods
    ...
    // finally, build the bootstrap registry with all the above
    options
    .build();
```

[Report a bug](#)

13.5.6.3. BootstrapRegistry Services

org.hibernate.service.classloading.spi.ClassLoaderService

Hibernate needs to interact with class loaders. However, the manner in which Hibernate (or any library) should interact with class loaders varies based on the runtime environment which is hosting the application. Application servers, OSGi containers, and other modular class loading systems impose very specific class loading requirements. This service provides Hibernate an abstraction from this environmental complexity. And just as importantly, it does so in a single-swappable-component manner.

In terms of interacting with a class loader, Hibernate needs the following capabilities:

- the ability to locate application classes
- the ability to locate integration classes
- the ability to locate resources (properties files, xml files, etc)
- the ability to load **java.util.ServiceLoader**



NOTE

Currently, the ability to load application classes and the ability to load integration classes are combined into a single "load class" capability on the service. That may change in a later release.

org.hibernate.integrator.spi.IntegratorService

Applications, add-ons and other modules need to integrate with Hibernate. The previous approach required a component, usually an application, to coordinate the registration of each individual module. This registration was conducted on behalf of each module's integrator.

This service focuses on the discovery aspect. It leverages the standard Java **java.util.ServiceLoader** capability provided by the **org.hibernate.service.classloading.spi.ClassLoaderService** in order to discover implementations of the **org.hibernate.integrator.spi.Integrator** contract.

Integrators would simply define a file named **/META-INF/services/org.hibernate.integrator.spi.Integrator** and make it available on the classpath.

This file is used by the **java.util.ServiceLoader** mechanism. It lists, one per line, the fully qualified names of classes which implement the **org.hibernate.integrator.spi.Integrator** interface.

[Report a bug](#)

13.5.7. The SessionFactory Registry

13.5.7.1. SessionFactory Registry

While it is best practice to treat instances of all the registry types as targeting a given **org.hibernate.SessionFactory**, the instances of services in this group explicitly belong to a single **org.hibernate.SessionFactory**.

The difference is a matter of timing in when they need to be initiated. Generally they need access to the **org.hibernate.SessionFactory** to be initiated. This special registry is **org.hibernate.service.spi.SessionFactoryServiceRegistry**

[Report a bug](#)

13.5.7.2. SessionFactory Services

org.hibernate.event.service.spi.EventListenerRegistry

Description

Service for managing event listeners.

Initiator

`org.hibernate.event.service.internal.EventListenerServiceInitiator`

Implementations

`org.hibernate.event.service.internal.EventListenerRegistryImpl`

[Report a bug](#)

13.5.8. Integrators

13.5.8.1. Integrators

The `org.hibernate.integrator.spi.Integrator` is intended to provide a simple means for allowing developers to hook into the process of building a functioning SessionFactory. The `org.hibernate.integrator.spi.Integrator` interface defines 2 methods of interest: **integrate** allows us to hook into the building process; **disintegrate** allows us to hook into a SessionFactory shutting down.



NOTE

There is a 3rd method defined on `org.hibernate.integrator.spi.Integrator`, an overloaded form of **integrate** accepting a `org.hibernate.metamodel.source.MetadataImplementor` instead of `org.hibernate.cfg.Configuration`. This form is intended for use with the new metamodel code scheduled for completion in 5.0.

In addition to the discovery approach provided by the IntegratorService, applications can manually register Integrator implementations when building the BootstrapServiceRegistry.

[Report a bug](#)

13.5.8.2. Integrator use-cases

The main use cases for an `org.hibernate.integrator.spi.Integrator` right now are registering event listeners and providing services (see `org.hibernate.integrator.spi.ServiceContributingIntegrator`). With 5.0 we plan on expanding that to allow altering the metamodel describing the mapping between object and relational models.

Example 13.24. Registering event listeners

```

public class MyIntegrator implements
org.hibernate.integrator.spi.Integrator {

    public void integrate(
        Configuration configuration,
        SessionFactoryImplementor sessionFactory,
        SessionFactoryServiceRegistry serviceRegistry) {

```

```

        // As you might expect, an EventListenerRegistry is the thing
        with which event listeners are registered. It is a
        // service so we look it up using the service registry
        final EventListenerRegistry eventListenerRegistry =
        serviceRegistry.getService( EventListenerRegistry.class );

        // If you wish to have custom determination and handling of
        "duplicate" listeners, you would have to add an
        // implementation of the
        org.hibernate.event.service.spi.DuplicationStrategy contract like this
        eventListenerRegistry.addDuplicationStrategy(
        myDuplicationStrategy );

        // EventListenerRegistry defines 3 ways to register listeners:
        //      1) This form overrides any existing registrations with
        eventListenerRegistry.setListeners( EventType.AUTO_FLUSH,
        myCompleteSetOfListeners );
        //      2) This form adds the specified listener(s) to the
        beginning of the listener chain
        eventListenerRegistry.prependListeners( EventType.AUTO_FLUSH,
        myListenersToBeCalledFirst );
        //      3) This form adds the specified listener(s) to the end of the
        listener chain
        eventListenerRegistry.appendListeners( EventType.AUTO_FLUSH,
        myListenersToBeCalledLast );
    }
}

```

[Report a bug](#)

13.6. BEAN VALIDATION

13.6.1. About Bean Validation

Bean Validation, or JavaBeans Validation, is a model for validating data in Java objects. The model uses built-in and custom annotation constraints to ensure the integrity of application data. The specification is documented here: <http://jcp.org/en/jsr/detail?id=303>.

Hibernate Validator is the JBoss EAP 6 implementation of Bean Validation. It is also the reference implementation of the JSR.

JBoss EAP 6 is 100% compliant with JSR 303 - Bean Validation. Hibernate Validator also provides additional features to the specification.

To get started with Bean Validation, refer to the **bean-validation** quickstart example: [Section 1.4.1.1](#), “Access the Quickstarts”.

[Report a bug](#)

13.6.2. Hibernate Validator

Hibernate Validator is the reference implementation of [JSR 303 - Bean Validation](#).

Bean Validation provides users with a model for validating Java object data. For more information, refer to [Section 13.6.1, “About Bean Validation”](#) and [Section 13.6.3.1, “About Validation Constraints”](#).

[Report a bug](#)

13.6.3. Validation Constraints

13.6.3.1. About Validation Constraints

Validation constraints are rules applied to a Java element, such as a field, property or bean. A constraint will usually have a set of attributes used to set its limits. There are predefined constraints, and custom ones can be created. Each constraint is expressed in the form of an annotation.

The built-in validation constraints for Hibernate Validator are listed here: [Section 13.6.3.3, “Hibernate Validator Constraints”](#)

For more information, refer to [Section 13.6.2, “Hibernate Validator”](#) and [Section 13.6.1, “About Bean Validation”](#).

[Report a bug](#)

13.6.3.2. Create a Constraint Annotation in Red Hat JBoss Developer Studio

Summary

This task covers the process of creating a constraint annotation in Red Hat JBoss Developer Studio, for use within a Java application.

Prerequisites

- [Section 1.3.1.4, “Start Red Hat JBoss Developer Studio”](#)

Procedure 13.5. Create a Constraint Annotation

1. Open a Java project in Red Hat JBoss Developer Studio.
2. **Create a Data Set**
A constraint annotation requires a data set that defines the acceptable values.
 - a. Right click on the project root folder in the **Project Explorer** panel.
 - b. Select **New** → **Enum**.
 - c. Configure the following elements:
 - **Package**:
 - **Name**:
 - d. Click the **Add . . .** button to add any required interfaces.
 - e. Click **Finish** to create the file.
 - f. Add a set of values to the data set and click **Save**.

Example 13.25. Example Data Set

```
package com.example;

public enum CaseMode {
    UPPER,
    LOWER;
}
```

3. Create the Annotation File

Create a new Java class.

4. Configure the constraint annotation and click **Save.****Example 13.26. Example Constraint Annotation File**

```
package com.mycompany;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = CheckCaseValidator.class)
@Documented
public @interface CheckCase {

    String message() default "
{com.mycompany.constraints.checkcase}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    CaseMode value();

}
```

Result

A custom constraint annotation with a set of possible values has been created, ready to be used in the Java project.

[Report a bug](#)

13.6.3.3. Hibernate Validator Constraints

Table 13.11. Built-in Constraints

Annotation	Apply on	Runtime checking	Hibernate Metadata impact
@Length(min=, max=)	property (String)	Check if the string length matches the range.	Column length will be set to max.
@Max(value=)	property (numeric or string representation of a numeric)	Check if the value is less than or equal to max.	Add a check constraint on the column.
@Min(value=)	property (numeric or string representation of a numeric)	Check if the value is more than or equal to Min.	Add a check constraint on the column.
@NotNull	property	Check if the value is not null.	Column(s) are not null.
@NotEmpty	property	Check if the string is not null nor empty. Check if the connection is not null nor empty.	Column(s) are not null (for String).
@Past	property (date or calendar)	Check if the date is in the past.	Add a check constraint on the column.
@Future	property (date or calendar)	Check if the date is in the future.	None.
@Pattern(regex="regexp", flag=) or @Patterns({@Pattern(...)})	property (string)	Check if the property matches the regular expression given a match flag (see java.util.regex.Pattern).	None.
@Range(min=, max=)	property (numeric or string representation of a numeric)	Check if the value is between min and max (included).	Add a check constraint on the column.
@Size(min=, max=)	property (array, collection, map)	Check if the element size is between min and max (included).	None.
@AssertFalse	property	Check that the method evaluates to false (useful for constraints expressed in code rather than annotations).	None.

Annotation	Apply on	Runtime checking	Hibernate Metadata impact
@AssertTrue	property	Check that the method evaluates to true (useful for constraints expressed in code rather than annotations).	None.
@Valid	property (object)	Perform validation recursively on the associated object. If the object is a Collection or an array, the elements are validated recursively. If the object is a Map, the value elements are validated recursively.	None.
@Email	property (String)	Check whether the string is conform to the e-mail address specification.	None.
@CreditCardNumber	property (String)	Check whether the string is a well formatted credit card number (derivative of the Luhn algorithm).	None.
@Digits(integerDigits=1)	property (numeric or string representation of a numeric)	Check whether the property is a number having up to integerDigits integer digits and fractionalDigits fractional digits.	Define column precision and scale.
@EAN	property (string)	Check whether the string is a properly formatted EAN or UPC-A code.	None.

[Report a bug](#)

13.6.4. Configuration

13.6.4.1. Example Validation Configuration File

Example 13.27. validation.xml

```
<validation-config
  xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

  xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/configuration"
  >

  <default-provider>
    org.hibernate.validator.HibernateValidator
  </default-provider>
  <message-interpolator>

    org.hibernate.validator.messageinterpolation.ResourceBundleMessageInterp
    olator
  </message-interpolator>
  <constraint-validator-factory>
    org.hibernate.validator.engine.ConstraintValidatorFactoryImpl
  </constraint-validator-factory>

  <constraint-mapping>
    /constraints-example.xml
  </constraint-mapping>

  <property name="prop1">value1</property>
  <property name="prop2">value2</property>
</validation-config>
```

[Report a bug](#)

13.7. ENVERS

13.7.1. About Hibernate Envers

Hibernate Envers is an auditing and versioning system, providing JBoss EAP 6 with a means to track historical changes to persistent classes. Audit tables are created for entities annotated with **@Audited**, which store the history of changes made to the entity. The data can then be retrieved and queried.

Envers allows developers to:

- audit all mappings defined by the JPA specification,
- audit all hibernate mappings that extend the JPA specification,
- audit entities mapped by or using the native Hibernate API
- log data for each revision using a revision entity, and
- query historical data.

[Report a bug](#)

13.7.2. About Auditing Persistent Classes

Auditing of persistent classes is done in JBoss EAP 6 through Hibernate Envers and the `@Audited` annotation. When the annotation is applied to a class, a table is created, which stores the revision history of the entity.

Each time a change is made to the class, an entry is added to the audit table. The entry contains the changes to the class, and is given a revision number. This means that changes can be rolled back, or previous revisions can be viewed.

[Report a bug](#)

13.7.3. Auditing Strategies

13.7.3.1. About Auditing Strategies

Auditing strategies define how audit information is persisted, queried and stored. There are currently two audit strategies available with Hibernate Envers:

Default Audit Strategy

This strategy persists the audit data together with a start revision. For each row that is inserted, updated or deleted in an audited table, one or more rows are inserted in the audit tables, along with the start revision of its validity.

Rows in the audit tables are never updated after insertion. Queries of audit information use subqueries to select the applicable rows in the audit tables, which are slow and difficult to index.

Validity Audit Strategy

This strategy stores the start revision, as well as the end revision of the audit information. For each row that is inserted, updated or deleted in an audited table, one or more rows are inserted in the audit tables, along with the start revision of its validity.

At the same time, the end revision field of the previous audit rows (if available) is set to this revision. Queries on the audit information can then use *between start and end revision*, instead of subqueries. This means that persisting audit information is a little slower because of the extra updates, but retrieving audit information is a lot faster.

This can also be improved by adding extra indexes.

For more information on auditing, refer to [Section 13.7.2, “About Auditing Persistent Classes”](#). To set the auditing strategy for the application, refer here: [Section 13.7.3.2, “Set the Auditing Strategy”](#).

[Report a bug](#)

13.7.3.2. Set the Auditing Strategy

Summary

There are two audit strategies supported by JBoss EAP 6: the default and validity audit strategies. This task covers the steps required to define the auditing strategy for an application.

Procedure 13.6. Define a Auditing Strategy

- Configure the **org.hibernate.envers.audit_strategy** property in the **persistence.xml** file of the application. If the property is not set in the **persistence.xml** file, then the default audit strategy is used.

Example 13.28. Set the Default Audit Strategy

```
<property name="org.hibernate.envers.audit_strategy"
value="org.hibernate.envers.strategy.DefaultAuditStrategy"/>
```

Example 13.29. Set the Validity Audit Strategy

```
<property name="org.hibernate.envers.audit_strategy"
value="org.hibernate.envers.strategy.ValidityAuditStrategy"/>
```

[Report a bug](#)

13.7.4. Getting Started with Entity Auditing

13.7.4.1. Add Auditing Support to a JPA Entity

JBoss EAP 6 uses entity auditing, through [Section 13.7.1, “About Hibernate Envers”](#), to track the historical changes of a persistent class. This topic covers adding auditing support for a JPA entity.

Procedure 13.7. Add Auditing Support to a JPA Entity

1. Configure the available auditing parameters to suit the deployment: [Section 13.7.5.1, “Configure Envers Parameters”](#).
2. Open the JPA entity to be audited.
3. Import the **org.hibernate.envers.Audited** interface.
4. Apply the **@Audited** annotation to each field or property to be audited, or apply it once to the whole class.

Example 13.30. Audit Two Fields

```
import org.hibernate.envers.Audited;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;

@Entity
public class Person {
    @Id
    @GeneratedValue
    private int id;

    @Audited
```

```

        private String name;

        private String surname;

        @ManyToOne
        @Audited
        private Address address;

        // add getters, setters, constructors, equals and hashCode
here
    }

```

Example 13.31. Audit an entire Class

```

import org.hibernate.envers.Audited;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;

@Entity
@Audited
public class Person {
    @Id
    @GeneratedValue
    private int id;

    private String name;

    private String surname;

    @ManyToOne
    private Address address;

    // add getters, setters, constructors, equals and hashCode
here
}

```

Result

The JPA entity has been configured for auditing. A table called ***Entity_AUD*** will be created to store the historical changes.

[Report a bug](#)

13.7.5. Configuration

13.7.5.1. Configure Envers Parameters

JBoss EAP 6 uses entity auditing, through Hibernate Envers, to track the historical changes of a persistent class. This topic covers configuring the available Envers parameters.

Procedure 13.8. Configure Envers Parameters

1. Open the **persistence.xml** file for the application.
2. Add, remove or configure Envers properties as required. For a list of available properties, refer to [Section 13.7.5.4, “Envers Configuration Properties”](#).

Example 13.32. Example Envers Parameters

```
<persistence-unit name="mypc">
  <description>Persistence Unit.</description>
  <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
  <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
  <properties>
    <property name="hibernate.hbm2ddl.auto" value="create-drop" />
    <property name="hibernate.show_sql" value="true" />
    <property name="hibernate.cache.use_second_level_cache" value="true" />
    <property name="hibernate.cache.use_query_cache" value="true" />
    <property name="hibernate.generate_statistics" value="true" />
    <property name="org.hibernate.envers.versionsTableSuffix" value="_V" />
    <property name="org.hibernate.envers.revisionFieldName" value="ver_rev" />
  </properties>
</persistence-unit>
```

Result

Auditing has been configured for all JPA entities in the application.

[Report a bug](#)

13.7.5.2. Enable or Disable Auditing at Runtime

Summary

This task covers the configuration steps required to enable/disable entity version auditing at runtime.

Procedure 13.9. Enable/Disable Auditing

1. Subclass the **AuditEventListener** class.
2. Override the following methods that are called on Hibernate events:
 - onPostInsert
 - onPostUpdate
 - onPostDelete
 - onPreUpdateCollection
 - onPreRemoveCollection

- `onPostRecreateCollection`
3. Specify the subclass as the listener for the events.
 4. Determine if the change should be audited.
 5. Pass the call to the superclass if the change should be audited.

[Report a bug](#)

13.7.5.3. Configure Conditional Auditing

Summary

Hibernate Envers persists audit data in reaction to various Hibernate events, using a series of event listeners. These listeners are registered automatically if the Envers jar is in the class path. This task covers the steps required to implement conditional auditing, by overriding some of the Envers event listeners.

Procedure 13.10. Implement Conditional Auditing

1. Set the `hibernate.listeners.envers.autoRegister` Hibernate property to false in the `persistence.xml` file.
2. Subclass each event listener to be overridden. Place the conditional auditing logic in the subclass, and call the super method if auditing should be performed.
3. Create a custom implementation of `org.hibernate.integrator.spi.Integrator`, similar to `org.hibernate.envers.event.EnversIntegrator`. Use the event listener subclasses created in step two, rather than the default classes.
4. Add a `META-INF/services/org.hibernate.integrator.spi.Integrator` file to the jar. This file should contain the fully qualified name of the class implementing the interface.

Result

Conditional auditing has been configured, overriding the default Envers event listeners.

[Report a bug](#)

13.7.5.4. Envers Configuration Properties

Table 13.12. Entity Data Versioning Configuration Parameters

Property Name	Default Value	Description
<code>org.hibernate.envers.audit_table_prefix</code>		A string that is prepended to the name of an audited entity, to create the name of the entity that will hold the audit information.

Property Name	Default Value	Description
org.hibernate.envers.audit_table_suffix	_AUD	A string that is appended to the name of an audited entity to create the name of the entity that will hold the audit information. For example, if an entity with a table name of Person is audited, Envers will generate a table called Person_AUD to store the historical data.
org.hibernate.envers.revision_field_name	REV	The name of the field in the audit entity that holds the revision number.
org.hibernate.envers.revision_type_field_name	REVTYPE	The name of the field in the audit entity that holds the type of revision. The current types of revisions possible are: add , mod and del .
org.hibernate.envers.revision_on_collection_change	true	This property determines if a revision should be generated if a relation field that is not owned changes. This can either be a collection in a one-to-many relation, or the field using the mappedBy attribute in a one-to-one relation.
org.hibernate.envers.do_not_audit_optimistic_locking_field	true	When true, properties used for optimistic locking (annotated with @Version) will automatically be excluded from auditing.
org.hibernate.envers.store_data_at_delete	false	This property defines whether or not entity data should be stored in the revision when the entity is deleted, instead of only the ID, with all other properties marked as null. This is not usually necessary, as the data is present in the last-but-one revision. Sometimes, however, it is easier and more efficient to access it in the last revision. However, this means the data the entity contained before deletion is stored twice.
org.hibernate.envers.default_schema	null (same as normal tables)	The default schema name used for audit tables. Can be overridden using the @AuditTable(schema="...") annotation. If not present, the schema will be the same as the schema of the normal tables.

Property Name	Default Value	Description
org.hibernate.envers.default_catalog	null (same as normal tables)	The default catalog name that should be used for audit tables. Can be overridden using the @AuditTable(catalog="...") annotation. If not present, the catalog will be the same as the catalog of the normal tables.
org.hibernate.envers.audit_strategy	org.hibernate.envers.strategy.DefaultAuditStrategy	This property defines the audit strategy that should be used when persisting audit data. By default, only the revision where an entity was modified is stored. Alternatively, org.hibernate.envers.strategy.ValidityAuditStrategy stores both the start revision and the end revision. Together, these define when an audit row was valid.
org.hibernate.envers.audit_strategy_validity_end_rev_field_name	REVEN	The column name that will hold the end revision number in audit entities. This property is only valid if the validity audit strategy is used.
org.hibernate.envers.audit_strategy_validity_store_revend_timestamp	false	This property defines whether the timestamp of the end revision, where the data was last valid, should be stored in addition to the end revision itself. This is useful to be able to purge old audit records out of a relational database by using table partitioning. Partitioning requires a column that exists within the table. This property is only evaluated if the ValidityAuditStrategy is used.
org.hibernate.envers.audit_strategy_validity_revend_timestamp_field_name	REVEN_TIMESTAMP	Column name of the timestamp of the end revision at which point the data was still valid. Only used if the ValidityAuditStrategy is used, and org.hibernate.envers.audit_strategy_validity_store_revend_timestamp evaluates to true.

[Report a bug](#)

13.7.6. Queries

13.7.6.1. Retrieve Auditing Information

Summary

Hibernate Envers provides the functionality to retrieve audit information through queries. This topic provides examples of those queries.



NOTE

Queries on the audited data will be, in many cases, much slower than corresponding queries on **live** data, as they involve correlated subselects.

Example 13.33. Querying for Entities of a Class at a Given Revision

The entry point for this type of query is:

```
AuditQuery query = getAuditReader()
    .createQuery()
    .forEntitiesAtRevision(MyEntity.class, revisionNumber);
```

Constraints can then be specified, using the **AuditEntity** factory class. The query below only selects entities where the **name** property is equal to **John**:

```
query.add(AuditEntity.property("name").eq("John"));
```

The queries below only select entities that are related to a given entity:

```
query.add(AuditEntity.property("address").eq(relatedEntityInstance));
// or
query.add(AuditEntity.relatedId("address").eq(relatedEntityId));
```

The results can then be ordered, limited, and have aggregations and projections (except grouping) set. The example below is a full query.

```
List personsAtAddress = getAuditReader().createQuery()
    .forEntitiesAtRevision(Person.class, 12)
    .addOrder(AuditEntity.property("surname").desc())
    .add(AuditEntity.relatedId("address").eq(addressId))
    .setFirstResult(4)
    .setMaxResults(2)
    .getResultList();
```

Example 13.34. Query Revisions where Entities of a Given Class Changed

The entry point for this type of query is:

```
AuditQuery query = getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true);
```

Constraints can be added to this query in the same way as the previous example. There are additional possibilities for this query:

AuditEntity.revisionNumber()

Specify constraints, projections and order on the revision number in which the audited entity was modified.

AuditEntity.revisionProperty(propertyName)

Specify constraints, projections and order on a property of the revision entity, corresponding to the revision in which the audited entity was modified.

AuditEntity.revisionType()

Provides accesses to the type of the revision (ADD, MOD, DEL).

The query results can then be adjusted as necessary. The query below selects the smallest revision number at which the entity of the **MyEntity** class, with the **entityId** ID has changed, after revision number 42:

```
Number revision = (Number) getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true)
    .setProjection(AuditEntity.revisionNumber().min())
    .add(AuditEntity.id().eq(entityId))
    .add(AuditEntity.revisionNumber().gt(42))
    .getSingleResult();
```

Queries for revisions can also minimize/maximize a property. The query below selects the revision at which the value of the **actualDate** for a given entity was larger than a given value, but as small as possible:

```
Number revision = (Number) getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true)
    // We are only interested in the first revision
    .setProjection(AuditEntity.revisionNumber().min())
    .add(AuditEntity.property("actualDate").minimize()
        .add(AuditEntity.property("actualDate").ge(givenDate))
        .add(AuditEntity.id().eq(givenEntityId)))
    .getSingleResult();
```

The **minimize()** and **maximize()** methods return a criteria, to which constraints can be added, which must be met by the entities with the maximized/minimized properties.

There are two boolean parameters passed when creating the query.

selectEntitiesOnly

This parameter is only valid when an explicit projection is not set.

If true, the result of the query will be a list of entities that changed at revisions satisfying the specified constraints.

If false, the result will be a list of three element arrays. The first element will be the changed entity instance. The second will be an entity containing revision data. If no custom entity is used, this will be an instance of **DefaultRevisionEntity**. The third element array will be the type of the

revision (ADD, MOD, DEL).

selectDeletedEntities

This parameter specifies if revisions in which the entity was deleted must be included in the results. If true, the entities will have the revision type **DEL**, and all fields, except id, will have the value **null**.

Example 13.35. Query Revisions of an Entity that Modified a Given Property

The query below will return all revisions of **MyEntity** with a given id, where the **actualDate** property has been changed.

```
AuditQuery query = getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true)
    .add(AuditEntity.id().eq(id));
    .add(AuditEntity.property("actualDate").hasChanged());
```

The **hasChanged** condition can be combined with additional criteria. The query below will return a horizontal slice for **MyEntity** at the time the *revisionNumber* was generated. It will be limited to the revisions that modified **prop1**, but not **prop2**.

```
AuditQuery query = getAuditReader().createQuery()
    .forEntitiesAtRevision(MyEntity.class, revisionNumber)
    .add(AuditEntity.property("prop1").hasChanged())
    .add(AuditEntity.property("prop2").hasNotChanged());
```

The result set will also contain revisions with numbers lower than the *revisionNumber*. This means that this query cannot be read as "Return all **MyEntities** changed in *revisionNumber* with **prop1** modified and **prop2** untouched."

The query below shows how this result can be returned, using the **forEntitiesModifiedAtRevision** query:

```
AuditQuery query = getAuditReader().createQuery()
    .forEntitiesModifiedAtRevision(MyEntity.class, revisionNumber)
    .add(AuditEntity.property("prop1").hasChanged())
    .add(AuditEntity.property("prop2").hasNotChanged());
```

Example 13.36. Query Entities Modified in a Given Revision

The example below shows the basic query for entities modified in a given revision. It allows entity names and corresponding Java classes changed in a specified revision to be retrieved:

```
Set<Pair<String, Class>> modifiedEntityTypes = getAuditReader()
    .getCrossTypeRevisionChangesReader().findEntityTypes(revisionNumber);
```

There are a number of other queries that are also accessible from **org.hibernate.envers.CrossTypeRevisionChangesReader**:

List<Object> findEntities(Number)

Returns snapshots of all audited entities changed (added, updated and removed) in a given revision. Executes **n+1** SQL queries, where **n** is a number of different entity classes modified within the specified revision.

List<Object> findEntities(Number, RevisionType)

Returns snapshots of all audited entities changed (added, updated or removed) in a given revision filtered by modification type. Executes **n+1** SQL queries, where **n** is a number of different entity classes modified within specified revision.

Map<RevisionType, List<Object>> findEntitiesGroupByRevisionType(Number)

Returns a map containing lists of entity snapshots grouped by modification operation (e.g. addition, update and removal). Executes **3n+1** SQL queries, where **n** is a number of different entity classes modified within specified revision.

[Report a bug](#)

13.8. PERFORMANCE TUNING

13.8.1. Alternative Batch Loading Algorithms

Hibernate allows you to load data for associations using one of four fetching strategies: join, select, subselect and batch. Out of these four strategies, batch loading allows for the biggest performance gains as it is an optimization strategy for select fetching. In this strategy, Hibernate retrieves a batch of entity instances or collections in a single SELECT statement by specifying a list of primary or foreign keys. Batch fetching is an optimization of the lazy select fetching strategy.

There are two ways to configure batch fetching: per-class level or per-collection level.

- Per-Class Level

When Hibernate loads data on a per-class level, it requires the batch size of the association to pre-load when queried. For example, consider that at runtime you have 30 instances of a **car** object loaded in session. Each **car** object belongs to an **owner** object. If you were to iterate through all the **car** objects and request their owners, with **lazy** loading, Hibernate will issue 30 select statements - one for each owner. This is a performance bottleneck.

You can instead, tell Hibernate to pre-load the data for the next batch of owners before they have been sought via a query. When an **owner** object has been queried, Hibernate will query many more of these objects in the same SELECT statement.

The number of **owner** objects to query in advance depends upon the **batch-size** parameter specified at configuration time:

```
<class name="owner" batch-size="10"></class>
```

This tells Hibernate to query at least 10 more **owner** objects in expectation of them being needed in the near future. When a user queries the **owner** of **car A**, the **owner** of **car B** may already have been loaded as part of batch loading. When the user actually needs the **owner** of

car B, instead of going to the database (and issuing a SELECT statement), the value can be retrieved from the current session.

In addition to the **batch-size** parameter, Hibernate 4.2.0 has introduced a new configuration item to improve in batch loading performance. The configuration item is called **Batch Fetch Style** configuration and specified by the **hibernate.batch_fetch_style** parameter.

Three different batch fetch styles are supported: LEGACY, PADDED and DYNAMIC. To specify which style to use, use **org.hibernate.cfg.AvailableSettings#BATCH_FETCH_STYLE**.

- LEGACY: In the legacy style of loading, a set of pre-built batch sizes based on **ArrayHelper.getBatchSizes(int)** are utilized. Batches are loaded using the next-smaller pre-built batch size from the number of existing batchable identifiers.

Continuing with the above example, with a **batch-size** setting of 30, the pre-built batch sizes would be [30, 15, 10, 9, 8, 7, ..., 1]. An attempt to batch load 29 identifiers would result in batches of 15, 10, and 4. There will be 3 corresponding SQL queries, each loading 15, 10 and 4 owners from the database.

- PADDED - Padded is similar to LEGACY style of batch loading. It still utilizes pre-built batch sizes, but uses the next-bigger batch size and pads the extra identifier placeholders.

As with the example above, if 30 owner objects are to be initialized, there will only be one query executed against the database.

However, if 29 owner objects are to be initialized, Hibernate will still execute only 1 SQL select statement of batch size 30, with the extra space padded with a repeated identifier.

- Dynamic - While still conforming to batch-size restrictions, this style of batch loading dynamically builds its SQL SELECT statement using the actual number of objects to be loaded.

For example, for 30 owner objects, and a maximum batch size of 30, a call to retrieve 30 owner objects will result in one SQL SELECT statement. A call to retrieve 35 will result in two SQL statements, of batch sizes 30 and 5 respectively. Hibernate will dynamically alter the second SQL statement to keep at 5, the required number, while still remaining under the restriction of 30 as the batch-size. This is different to the PADDED version, as the second SQL will not get PADDED, and unlike the LEGACY style, there is no fixed size for the second SQL statement - the second SQL is created dynamically.

For a query of less than 30 identifiers, this style will dynamically only load the number of identifiers requested.

- Per-Collection Level

Hibernate can also batch load collections honoring the batch fetch size and styles as listed in the per-class section above.

To reverse the example used in the previous section, consider that you need to load all the **car** objects owned by each **owner** object. If 10 **owner** objects are loaded in the current session iterating through all owners will generate 10 SELECT statements, one for every call to **getCars()** method. If you enable batch fetching for the cars collection in the mapping of Owner, Hibernate can pre-fetch these collections, as shown below.

```
<class name="Owner"><set name="cars" batch-size="5"></set></class>
```

Thus, with a batch-size of 5 and using legacy batch style to load 10 collections, Hibernate will execute two SELECT statements, each retrieving 5 collections.

[Report a bug](#)

13.8.2. Second Level Caching of Object References for Non-mutable Data

Hibernate automatically caches data within memory for improved performance. This is accomplished by an in-memory cache which reduces the number of times that database lookups are required, especially for data that rarely changes.

Hibernate maintains two types of caches. The primary cache (also called the first-level cache) is mandatory. This cache is associated with the current session and all requests must pass through it. The secondary cache (also called the second-level cache) is optional, and is only consulted after the primary cache has been consulted first.

Data is stored in the second-level cache by first disassembling it into a state array. This array is deep copied, and that deep copy is put into the cache. The reverse is done for reading from the cache. This works well for data that changes (mutable data), but is inefficient for immutable data.

Deep copying data is an expensive operation in terms of memory usage and processing speed. For large data sets, memory and processing speed become a performance-limiting factor. Hibernate allows you to specify that immutable data be referenced rather than copied. Instead of copying entire data sets, Hibernate can now store the reference to the data in the cache.

This can be done by changing the value of the configuration setting **hibernate.cache.use_reference_entries** to **true**. By default, **hibernate.cache.use_reference_entries** is set to **false**.

When **hibernate.cache.use_reference_entries** is set to **true**, an immutable data object that does not have any associations is not copied into the second-level cache, and only a reference to it is stored.



WARNING

When **hibernate.cache.use_reference_entries** is set to **true**, immutable data objects with associations are still deep copied into the second-level cache.

[Report a bug](#)

CHAPTER 14. HIBERNATE SEARCH

14.1. GETTING STARTED WITH HIBERNATE SEARCH

14.1.1. About Hibernate Search

Hibernate Search provides full-text search capability to Hibernate applications. It is especially suited to search applications for which SQL-based solutions are not suited, including: full-text, fuzzy and geolocation searches. Hibernate Search uses Apache Lucene as its full-text search engine, but is designed to minimize the maintenance overhead. Once it is configured, indexing, clustering and data synchronization is maintained transparently, allowing you to focus on meeting your business requirements.

[Report a bug](#)

14.1.2. First Steps with Hibernate Search

To get started with Hibernate Search for your application, follow these topics.

- See *Configuration* in the *JBoss EAP Administration and Configuration Guide* to configure Hibernate Search.
- [Section 14.1.3, “Enable Hibernate Search using Maven”](#)
- [Section 14.1.5, “Indexing”](#)
- [Section 14.1.6, “Searching”](#)
- [Section 14.1.7, “Analyzer”](#)

[Report a bug](#)

14.1.3. Enable Hibernate Search using Maven

Use the following configuration in your Maven project to add **hibernate-search-orm** dependencies:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-search-orm</artifactId>
      <version>4.6.0.Final-redhat-2</version>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-search-orm</artifactId>
    <scope>provided</scope>
  </dependency>
</dependencies>
```


[Report a bug](#)

14.1.4. Add Annotations

For this section, consider the example in which you have a database containing details of books. Your application contains the Hibernate managed classes **example.Book** and **example.Author** and you want to add free text search capabilities to your application to enable searching for books.

Example 14.1. Entities Book and Author Before Adding Hibernate Search Specific Annotations

```
package example;
...
@Entity
public class Book {

    @Id
    @GeneratedValue
    private Integer id;

    private String title;

    private String subtitle;

    @ManyToMany
    private Set<Author> authors = new HashSet<Author>();

    private Date publicationDate;

    public Book() {}

    // standard getters/setters follow here
    ...
}

package example;
...
@Entity
public class Author {

    @Id
    @GeneratedValue
    private Integer id;

    private String name;

    public Author() {}

    // standard getters/setters follow here
    ...
}
```

To achieve this you have to add a few annotations to the **Book** and **Author** class. The first annotation **@Indexed** marks **Book** as indexable. By design Hibernate Search stores an untokenized ID in the index

to ensure index unicity for a given entity. **@DocumentId** marks the property to use for this purpose and is in most cases the same as the database primary key. The **@DocumentId** annotation is optional in the case where an **@Id** annotation exists.

Next the fields you want to make searchable must be marked as such. In this example, start with **title** and **subtitle** and annotate both with **@Field**. The parameter **index=Index.YES** will ensure that the text will be indexed, while **analyze=Analyze.YES** ensures that the text will be analyzed using the default Lucene analyzer. Usually, analyzing means chunking a sentence into individual words and potentially excluding common words like 'a' or 'the'. We will talk more about analyzers a little later on. The third parameter we specify within **@Field**, **store=Store.NO**, ensures that the actual data will not be stored in the index. Whether this data is stored in the index or not has nothing to do with the ability to search for it. From Lucene's perspective it is not necessary to keep the data once the index is created. The benefit of storing it is the ability to retrieve it via projections (see [Section 14.3.1.10.5, "Projection"](#)).

Without projections, Hibernate Search will per default execute a Lucene query in order to find the database identifiers of the entities matching the query criteria and use these identifiers to retrieve managed objects from the database. The decision for or against projection has to be made on a case to case basis. The default behavior is recommended since it returns managed objects whereas projections only return object arrays.

Note that **index=Index.YES**, **analyze=Analyze.YES** and **store=Store.NO** are the default values for these parameters and could be omitted.

Another annotation not yet discussed is **@DateBridge**. This annotation is one of the built-in field bridges in Hibernate Search. The Lucene index is purely string based. For this reason Hibernate Search must convert the data types of the indexed fields to strings and vice-versa. A range of predefined bridges are provided, including the **DateBridge** which will convert a **java.util.Date** into a **String** with the specified resolution. For more details see [Section 14.2.4, "Bridges"](#).

This leaves us with **@IndexedEmbedded**. This annotation is used to index associated entities (**@ManyToMany**, **@*ToOne**, **@Embedded** and **@ElementCollection**) as part of the owning entity. This is needed since a Lucene index document is a flat data structure which does not know anything about object relations. To ensure that the authors' name will be searchable you have to ensure that the names are indexed as part of the book itself. On top of **@IndexedEmbedded** you will also have to mark all fields of the associated entity you want to have included in the index with **@Indexed**. For more details see [Section 14.2.1.3, "Embedded and Associated Objects"](#)

These settings should be sufficient for now. For more details on entity mapping see [Section 14.2.1, "Mapping an Entity"](#).

Example 14.2. Entities After Adding Hibernate Search Annotations

```
package example;
...
@Entity
@Indexed
public class Book {

    @Id
    @GeneratedValue
    private Integer id;

    @Field(index=Index.YES, analyze=Analyze.YES, store=Store.NO)
    private String title;
```

```

    @Field(index=Index.YES, analyze=Analyze.YES, store=Store.NO)
    private String subtitle;

    @Field(index = Index.YES, analyze=Analyze.NO, store = Store.YES)
    @DateBridge(resolution = Resolution.DAY)
    private Date publicationDate;

    @IndexedEmbedded
    @ManyToMany
    private Set<Author> authors = new HashSet<Author>();

    public Book() {
    }

    // standard getters/setters follow here
    ...
}

package example;
...
@Entity
public class Author {

    @Id
    @GeneratedValue
    private Integer id;

    @Field
    private String name;

    public Author() {
    }

    // standard getters/setters follow here
    ...
}

```

[Report a bug](#)

14.1.5. Indexing

Hibernate Search will transparently index every entity persisted, updated or removed through Hibernate Core. However, you have to create an initial Lucene index for the data already present in your database. Once you have added the above properties and annotations it is time to trigger an initial batch index of your books. You can achieve this by using one of the following code snippets (see also [Section 14.4.3, “Rebuilding the Index”](#)):

Example 14.3. Using the Hibernate Session to Index Data

```

FullTextSession fullTextSession =
    org.hibernate.search.Search.getFullTextSession(session);
fullTextSession.createIndexer().startAndWait();

```

Example 14.4. Using JPA to Index Data

```
EntityManager em = entityManagerFactory.createEntityManager();
FullTextEntityManager fullTextEntityManager =
    org.hibernate.search.jpa.Search.getFullTextEntityManager(em);
fullTextEntityManager.createIndexer().startAndWait();
```

After executing the above code, you should be able to see a Lucene index under `/var/lucene/indexes/example.Book`. Go ahead and inspect this index with [Luke](#). It will help you to understand how Hibernate Search works.

[Report a bug](#)

14.1.6. Searching

To execute a search, create a Lucene query using either the Lucene API ([Section 14.3.1.1, “Building a Lucene Query Using the Lucene API”](#)) or the Hibernate Search query DSL ([Section 14.3.1.2, “Building a Lucene Query”](#)). Wrap the query in a `org.hibernate.Query` to get the required functionality from the Hibernate API. The following code prepares a query against the indexed fields. Executing the code returns a list of **Books**.

Example 14.5. Using a Hibernate Search Session to Create and Execute a Search

```
FullTextSession fullTextSession = Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();

// create native Lucene query using the query DSL
// alternatively you can write the Lucene query using the Lucene query
// parser
// or the Lucene programmatic API. The Hibernate Search DSL is
// recommended though
QueryBuilder qb = fullTextSession.getSearchFactory()
    .buildQueryBuilder().forEntity( Book.class ).get();
org.apache.lucene.search.Query query = qb
    .keyword()
    .onFields("title", "subtitle", "authors.name", "publicationDate")
    .matching("Java rocks!")
    .createQuery();

// wrap Lucene query in a org.hibernate.Query
org.hibernate.Query hibQuery =
    fullTextSession.createFullTextQuery(query, Book.class);

// execute search
List result = hibQuery.list();

tx.commit();
session.close();
```

Example 14.6. Using JPA to Create and Execute a Search

```

EntityManager em = entityManagerFactory.createEntityManager();
FullTextEntityManager fullTextEntityManager =
    org.hibernate.search.jpa.Search.getFullTextEntityManager(em);
em.getTransaction().begin();

// create native Lucene query using the query DSL
// alternatively you can write the Lucene query using the Lucene query
// parser
// or the Lucene programmatic API. The Hibernate Search DSL is
// recommended though
QueryBuilder qb = fullTextEntityManager.getSearchFactory()
    .buildQueryBuilder().forEntity( Book.class ).get();
org.apache.lucene.search.Query query = qb
    .keyword()
    .onFields("title", "subtitle", "authors.name", "publicationDate")
    .matching("Java rocks!")
    .createQuery();

// wrap Lucene query in a javax.persistence.Query
javax.persistence.Query persistenceQuery =
    fullTextEntityManager.createFullTextQuery(query, Book.class);

// execute search
List result = persistenceQuery.getResultList();

em.getTransaction().commit();
em.close();

```

[Report a bug](#)

14.1.7. Analyzer

Assuming that the title of an indexed book entity is **Refactoring: Improving the Design of Existing Code** and that hits are required for the following queries: **refactor**, **refactors**, **refactored**, and **refactoring**. Select an analyzer class in Lucene that applies word stemming when indexing and searching. Hibernate Search offers several ways to configure the analyzer (see [Section 14.2.3.1, “Default Analyzer and Analyzer by Class”](#) for more information):

- Set the **analyzer** property in the configuration file. The specified class becomes the default analyzer.
- Set the **@Analyzer** annotation at the entity level.
- Set the **@Analyzer** annotation at the field level.

Specify the fully qualified classname or the analyzer to use, or see an analyzer defined by the **@AnalyzerDef** annotation with the **@Analyzer** annotation. The Solr analyzer framework with its factories are utilized for the latter option. For more information about factory classes, see the Solr JavaDoc or read the corresponding section on the Solr Wiki (<http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters>)

In the example, a **StandardTokenizerFactory** is used by two filter factories: **LowerCaseFilterFactory** and **SnowballPorterFilterFactory**. The tokenizer splits words at punctuation characters and hyphens but keeping email addresses and internet hostnames intact. The

standard tokenizer is ideal for this and other general operations. The lowercase filter converts all letters in the token into lowercase and the snowball filter applies language specific stemming.

If using the Solr framework, use the tokenizer with an arbitrary number of filters.

Example 14.7. Using `@AnalyzerDef` and the Solr Framework to Define and Use an Analyzer

```
@Indexed
@AnalyzerDef(
    name = "customanalyzer",
    tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
    filters = {
        @TokenFilterDef(factory = LowerCaseFilterFactory.class),
        @TokenFilterDef(factory = SnowballPorterFilterFactory.class,
            params = { @Parameter(name = "language", value = "English") })
    })
public class Book implements Serializable {

    @Field
    @Analyzer(definition = "customanalyzer")
    private String title;

    @Field
    @Analyzer(definition = "customanalyzer")
    private String subtitle;

    @IndexedEmbedded
    private Set authors = new HashSet();

    @Field(index = Index.YES, analyze = Analyze.NO, store = Store.YES)
    @DateBridge(resolution = Resolution.DAY)
    private Date publicationDate;

    public Book() {
    }

    // standard getters/setters follow here
    ...
}
```

Use `@AnalyzerDef` to define an analyzer, then apply it to entities and properties using `@Analyzer`. In the example, the `customanalyzer` is defined but not applied on the entity. The analyzer is only applied to the `title` and `subtitle` properties. An analyzer definition is global. Define the analyzer for an entity and reuse the definition for other entities as required.

[Report a bug](#)

14.2. MAPPING ENTITIES TO THE INDEX STRUCTURE

14.2.1. Mapping an Entity

All the metadata information required to index entities is described through annotations, so there is no need for XML mapping files. You can still use Hibernate mapping files for the basic Hibernate configuration, but the Hibernate Search specific configuration has to be expressed via annotations.

[Report a bug](#)

14.2.1.1. Basic Mapping

Lets start with the most commonly used annotations for mapping an entity.

The Lucene-based Query API uses the following common annotations to map entities:

- `@Indexed`
- `@Field`
- `@NumericField`
- `@Id`

[Report a bug](#)

14.2.1.1.1. `@Indexed`

Foremost we must declare a persistent class as indexable. This is done by annotating the class with `@Indexed` (all entities not annotated with `@Indexed` will be ignored by the indexing process):

Example 14.8. Making a class indexable with `@Indexed`

```
@Entity
@Indexed
public class Essay {
    ...
}
```

You can optionally specify the `index` attribute of the `@Indexed` annotation to change the default name of the index.

[Report a bug](#)

14.2.1.1.2. `@Field`

For each property (or attribute) of your entity, you have the ability to describe how it will be indexed. The default (no annotation present) means that the property is ignored by the indexing process. `@Field` does declare a property as indexed and allows to configure several aspects of the indexing process by setting one or more of the following attributes:

- **name** : describe under which name, the property should be stored in the Lucene Document. The default value is the property name (following the JavaBeans convention)
- **store** : describe whether or not the property is stored in the Lucene index. You can store the value `Store.YES` (consuming more space in the index but allowing projection, see [Section 14.3.1.10.5, “Projection”](#)), store it in a compressed way `Store.COMPRESS` (this does

consume more CPU), or avoid any storage **Store.NO** (this is the default value). When a property is stored, you can retrieve its original value from the Lucene Document. This is not related to whether the element is indexed or not.

- **index**: describe whether the property is indexed or not. The different values are **Index.NO** (no indexing, ie cannot be found by a query), **Index.YES** (the element gets indexed and is searchable). The default value is **Index.YES**. **Index.NO** can be useful for cases where a property is not required to be searchable, but should be available for projection.



NOTE

Index.NO in combination with **Analyze.YES** or **Norms.YES** is not useful, since **analyze** and **norms** require the property to be indexed

- **analyze**: determines whether the property is analyzed (**Analyze.YES**) or not (**Analyze.NO**). The default value is **Analyze.YES**.



NOTE

Whether or not you want to analyze a property depends on whether you wish to search the element as is, or by the words it contains. It make sense to analyze a text field, but probably not a date field.



NOTE

Fields used for sorting *must not* be analyzed.

- **norms**: describes whether index time boosting information should be stored (**Norms.YES**) or not (**Norms.NO**). Not storing it can save a considerable amount of memory, but there won't be any index time boosting information available. The default value is **Norms.YES**.
- **termVector**: describes collections of term-frequency pairs. This attribute enables the storing of the term vectors within the documents during indexing. The default value is **TermVector.NO**.

The different values of this attribute are:

Value	Definition
TermVector.YES	Store the term vectors of each document. This produces two synchronized arrays, one contains document terms and the other contains the term's frequency.
TermVector.NO	Do not store term vectors.
TermVector.WITH_OFFSETS	Store the term vector and token offset information. This is the same as TermVector.YES plus it contains the starting and ending offset position information for the terms.

Value	Definition
TermVector.WITH_POSITIONS	Store the term vector and token position information. This is the same as TermVector.YES plus it contains the ordinal positions of each occurrence of a term in a document.
TermVector.WITH_POSITION_OFFSETS	Store the term vector, token position and offset information. This is a combination of the YES, WITH_OFFSETS and WITH_POSITIONS.

- **indexNullAs** : Per default null values are ignored and not indexed. However, using **indexNullAs** you can specify a string which will be inserted as token for the **null** value. Per default this value is set to **Field.DO_NOT_INDEX_NULL** indicating that **null** values should not be indexed. You can set this value to **Field.DEFAULT_NULL_TOKEN** to indicate that a default **null** token should be used. This default **null** token can be specified in the configuration using **hibernate.search.default_null_token**. If this property is not set and you specify **Field.DEFAULT_NULL_TOKEN** the string "**_null_**" will be used as default.



NOTE

When the **indexNullAs** parameter is used it is important to use the same token in the search query to search for **null** values. It is also advisable to use this feature only with un-analyzed fields (**analyze=Analyze.NO**).



WARNING

When implementing a custom **FieldBridge** or **TwoWayFieldBridge** it is up to the developer to handle the indexing of null values (see JavaDocs of **LuceneOptions.indexNullAs()**).

[Report a bug](#)

14.2.1.1.3. @NumericField

There is a companion annotation to **@Field** called **@NumericField** that can be specified in the same scope as **@Field** or **@DocumentId**. It can be specified for Integer, Long, Float, and Double properties. At index time the value will be indexed using a Trie structure. When a property is indexed as numeric field, it enables efficient range query and sorting, orders of magnitude faster than doing the same query on standard **@Field** properties. The **@NumericField** annotation accept the following parameters:

Value	Definition
-------	------------

Value	Definition
forField	(Optional) Specify the name of the related @Field that will be indexed as numeric. It's only mandatory when the property contains more than a @Field declaration
precisionStep	(Optional) Change the way that the Trie structure is stored in the index. Smaller precisionSteps lead to more disk space usage and faster range and sort queries. Larger values lead to less space used and range query performance more close to the range query in normal @Fields. Default value is 4.

@NumericField supports only **Double**, **Long**, **Integer** and **Float**. It is not possible to take any advantage from similar functionality in Lucene for the other numeric types, so remaining types should use the string encoding via the default or custom **TwoWayFieldBridge**.

It is possible to use a custom **NumericFieldBridge** assuming you can deal with the approximation during type transformation:

Example 14.9. Defining a custom NumericFieldBridge

```
public class BigDecimalNumericFieldBridge extends NumericFieldBridge {
    private static final BigDecimal storeFactor =
        BigDecimal.valueOf(100);

    @Override
    public void set(String name, Object value, Document document,
        LuceneOptions luceneOptions) {
        if ( value != null ) {
            BigDecimal decimalValue = (BigDecimal) value;
            Long indexedValue = Long.valueOf( decimalValue.multiply(
                storeFactor ).longValue() );
            luceneOptions.addNumericFieldToDocument( name, indexedValue,
                document );
        }
    }

    @Override
    public Object get(String name, Document document) {
        String fromLucene = document.get( name );
        BigDecimal storedBigDecimal = new BigDecimal( fromLucene );
        return storedBigDecimal.divide( storeFactor );
    }
}
```

[Report a bug](#)

14.2.1.1.4. @Id

Finally, the **id** (identifier) property of an entity is a special property used by Hibernate Search to ensure index uniqueness of a given entity. By design, an **id** must be stored and must not be tokenized. To mark a property as an index identifier, use the **@DocumentId** annotation. If you are using JPA and you have specified **@Id** you can omit **@DocumentId**. The chosen entity identifier will also be used as the document identifier.

Example 14.10. Specifying indexed properties

```
@Entity
@Indexed
public class Essay {
    ...

    @Id
    @DocumentId
    public Long getId() { return id; }

    @Field(name="Abstract", store=Store.YES)
    public String getSummary() { return summary; }

    @Lob
    @Field
    public String getText() { return text; }

    @Field @NumericField( precisionStep = 6)
    public float getGrade() { return grade; }
}
```

Example 14.10, “Specifying indexed properties” defines an index with four fields: **id**, **Abstract**, **text** and **grade**. Note that by default the field name is not capitalized, following the JavaBean specification. The **grade** field is annotated as numeric with a slightly larger precision step than the default.

[Report a bug](#)

14.2.1.2. Mapping Properties Multiple Times

Sometimes you need to map a property multiple times per index, with slightly different indexing strategies. For example, sorting a query by field requires the field to be un-analyzed. To search by words on this property and still sort it, it needs to be indexed - once analyzed and once un-analyzed. **@Fields** allows you to achieve this goal.

Example 14.11. Using **@Fields** to map a property multiple times

```
@Entity
@Indexed(index = "Book" )
public class Book {
    @Fields( {
        @Field,
        @Field(name = "summary_forSort", analyze = Analyze.NO,
store = Store.YES)
    } )
    public String getSummary() {
        return summary;
    }
}
```

```

    }
    ...
}

```

In this example the field **summary** is indexed twice, once as **summary** in a tokenized way, and once as **summary_forSort** in an untokenized way.

[Report a bug](#)

14.2.1.3. Embedded and Associated Objects

Associated objects as well as embedded objects can be indexed as part of the root entity index. This is useful if you expect to search a given entity based on properties of associated objects. In [Example 14.12](#), “[Indexing associations](#)” the aim is to return places where the associated city is Atlanta (In the Lucene query parser language, it would translate into **address.city:Atlanta**). The place fields will be indexed in the **Place** index. The **Place** index documents will also contain the fields **address.id**, **address.street**, and **address.city** which you will be able to query.

Example 14.12. Indexing associations

```

@Entity
@Indexed
public class Place {
    @Id
    @GeneratedValue
    @DocumentId
    private Long id;

    @Field
    private String name;

    @OneToOne( cascade = { CascadeType.PERSIST, CascadeType.REMOVE } )
    @IndexedEmbedded
    private Address address;
    ....
}

@Entity
public class Address {
    @Id
    @GeneratedValue
    private Long id;

    @Field
    private String street;

    @Field
    private String city;

    @ContainedIn
    @OneToMany( mappedBy="address" )

```

```

    private Set<Place> places;
    ...
}

```

Because the data is denormalized in the Lucene index when using the **@IndexedEmbedded** technique, Hibernate Search must be aware of any change in the **Place** object and any change in the **Address** object to keep the index up to date. To ensure the **Place** Lucene document is updated when it's **Address** changes, mark the other side of the bidirectional relationship with **@ContainedIn**.



NOTE

@ContainedIn is useful on both associations pointing to entities and on embedded (collection of) objects.

To expand upon this, the following example demonstrates nesting **@IndexedEmbedded**.

Example 14.13. Nested usage of **@IndexedEmbedded** and **@ContainedIn**

```

@Entity
@Indexed
public class Place {
    @Id
    @GeneratedValue
    @DocumentId
    private Long id;

    @Field
    private String name;

    @OneToOne( cascade = { CascadeType.PERSIST, CascadeType.REMOVE } )
    @IndexedEmbedded
    private Address address;
    ....
}

@Entity
public class Address {
    @Id
    @GeneratedValue
    private Long id;

    @Field
    private String street;

    @Field
    private String city;

    @IndexedEmbedded(depth = 1, prefix = "ownedBy_")
    private Owner ownedBy;

    @ContainedIn
    @OneToMany(mappedBy="address")
    private Set<Place> places;
}

```

```

    ...
}

@Embeddable
public class Owner {
    @Field
    private String name;
    ...
}

```

Any **@*ToMany**, **@*ToOne** and **@Embedded** attribute can be annotated with **@IndexedEmbedded**. The attributes of the associated class will then be added to the main entity index. In [Example 14.13](#), “Nested usage of **@IndexedEmbedded** and **@ContainedIn**” the index will contain the following fields:

- id
- name
- address.street
- address.city
- address.ownedBy_name

The default prefix is **propertyName.**, following the traditional object navigation convention. You can override it using the **prefix** attribute as it is shown on the **ownedBy** property.



NOTE

The prefix cannot be set to the empty string.

The **depth** property is necessary when the object graph contains a cyclic dependency of classes (not instances). For example, if **Owner** points to **Place**. Hibernate Search will stop including Indexed embedded attributes after reaching the expected depth (or the object graph boundaries are reached). A class having a self reference is an example of cyclic dependency. In our example, because **depth** is set to 1, any **@IndexedEmbedded** attribute in Owner (if any) will be ignored.

Using **@IndexedEmbedded** for object associations allows you to express queries (using Lucene's query syntax) such as:

- Return places where name contains JBoss and where address city is Atlanta. In Lucene query this would be

```
+name:jboss +address.city:atlanta
```

- Return places where name contains JBoss and where owner's name contain Joe. In Lucene query this would be

```
+name:jboss +address.ownedBy_name:joe
```

This behavior mimics the relational join operation in a more efficient way (at the cost of data duplication). Remember that, out of the box, Lucene indexes have no notion of association, the join operation does

not exist. It might help to keep the relational model normalized while benefiting from the full text index speed and feature richness.



NOTE

An associated object can itself (but does not have to) be **@Indexed**

When **@IndexedEmbedded** points to an entity, the association has to be directional and the other side has to be annotated **@ContainedIn** (as seen in the previous example). If not, Hibernate Search has no way to update the root index when the associated entity is updated (in our example, a **Place** index document has to be updated when the associated **Address** instance is updated).

Sometimes, the object type annotated by **@IndexedEmbedded** is not the object type targeted by Hibernate and Hibernate Search. This is especially the case when interfaces are used in lieu of their implementation. For this reason you can override the object type targeted by Hibernate Search using the **targetElement** parameter.

Example 14.14. Using the **targetElement** property of **@IndexedEmbedded**

```
@Entity
@Indexed
public class Address {
    @Id
    @GeneratedValue
    @DocumentId
    private Long id;

    @Field
    private String street;

    @IndexedEmbedded(depth = 1, prefix = "ownedBy_", targetElement =
Owner.class)
    @Target(Owner.class)
    private Person ownedBy;

    ...
}

@Embeddable
public class Owner implements Person { ... }
```

[Report a bug](#)

14.2.1.4. Limiting Object Embedding to Specific Paths

The **@IndexedEmbedded** annotation provides also an attribute **includePaths** which can be used as an alternative to **depth**, or be combined with it.

When using only **depth** all indexed fields of the embedded type will be added recursively at the same depth. This makes it harder to select only a specific path without adding all other fields as well, which might not be needed.

To avoid unnecessarily loading and indexing entities you can specify exactly which paths are needed. A typical application might need different depths for different paths, or in other words it might need to specify paths explicitly, as shown in [Example 14.15, “Using the `includePaths` property of `@IndexedEmbedded`”](#)

Example 14.15. Using the `includePaths` property of `@IndexedEmbedded`

```
@Entity
@Indexed
public class Person {

    @Id
    public int getId() {
        return id;
    }

    @Field
    public String getName() {
        return name;
    }

    @Field
    public String getSurname() {
        return surname;
    }

    @OneToMany
    @IndexedEmbedded(includePaths = { "name" })
    public Set<Person> getParents() {
        return parents;
    }

    @ContainedIn
    @ManyToOne
    public Human getChild() {
        return child;
    }

    ...//other fields omitted
}
```

Using a mapping as in [Example 14.15, “Using the `includePaths` property of `@IndexedEmbedded`”](#), you would be able to search on a **Person** by **name** and/or **surname**, and/or the **name** of the parent. It will not index the **surname** of the parent, so searching on parent's surnames will not be possible but speeds up indexing, saves space and improve overall performance.

The `@IndexedEmbeddedincludePaths` will include the specified paths *in addition to* what you would index normally specifying a limited value for **depth**. When using `includePaths`, and leaving **depth** undefined, behavior is equivalent to setting **depth=0**: only the included paths are indexed.

Example 14.16. Using the `includePaths` property of `@IndexedEmbedded`

```
@Entity
@Indexed
```



```

public class Human {

    @Id
    public int getId() {
        return id;
    }

    @Field
    public String getName() {
        return name;
    }

    @Field
    public String getSurname() {
        return surname;
    }

    @OneToMany
    @IndexedEmbedded(depth = 2, includePaths = { "parents.parents.name"
})
    public Set<Human> getParents() {
        return parents;
    }

    @ContainedIn
    @ManyToOne
    public Human getChild() {
        return child;
    }

    ...//other fields omitted
}

```

In [Example 14.16](#), “Using the `includePaths` property of `@IndexedEmbedded`”, every human will have its name and surname attributes indexed. The name and surname of parents will also be indexed, recursively up to second line because of the `depth` attribute. It will be possible to search by name or surname, of the person directly, his parents or of his grand parents. Beyond the second level, we will in addition index one more level but only the name, not the surname.

This results in the following fields in the index:

- `id` - as primary key
- `_hibernate_class` - stores entity type
- `name` - as direct field
- `surname` - as direct field
- `parents.name` - as embedded field at depth 1
- `parents.surname` - as embedded field at depth 1
- `parents.parents.name` - as embedded field at depth 2
- `parents.parents.surname` - as embedded field at depth 2

- **parents.parents.parents.name** - as additional path as specified by **includePaths**. The first **parents.** is inferred from the field name, the remaining path is the attribute of **includePaths**

Having explicit control of the indexed paths might be easier if you are designing your application by defining the needed queries first, as at that point you might know exactly which fields you need, and which other fields are unnecessary to implement your use case.

[Report a bug](#)

14.2.2. Boosting

Lucene has the notion of *boosting* which allows you to give certain documents or fields more or less importance than others. Lucene differentiates between index and search time boosting. The following sections show you how you can achieve index time boosting using Hibernate Search.

[Report a bug](#)

14.2.2.1. Static Index Time Boosting

To define a static boost value for an indexed class or property you can use the **@Boost** annotation. You can use this annotation within **@Field** or specify it directly on method or class level.

Example 14.17. Different ways of using @Boost

```
@Entity
@Indexed
@Boost(1.7f)
public class Essay {
    ...

    @Id
    @DocumentId
    public Long getId() { return id; }

    @Field(name="Abstract", store=Store.YES, boost=@Boost(2f))
    @Boost(1.5f)
    public String getSummary() { return summary; }

    @Lob
    @Field(boost=@Boost(1.2f))
    public String getText() { return text; }

    @Field
    public String getISBN() { return isbn; }
}
```

In [Example 14.17](#), “Different ways of using **@Boost**”, **Essay**'s probability to reach the top of the search list will be multiplied by 1.7. The **summary** field will be 3.0 (2 * 1.5, because **@Field.boost** and **@Boost** on a property are cumulative) more important than the **isbn** field. The **text** field will be 1.2 times more important than the **isbn** field. Note that this explanation is wrong in strictest terms, but it is simple and close enough to reality for all practical purposes.

[Report a bug](#)

14.2.2.2. Dynamic Index Time Boosting

The **@Boost** annotation used in [Section 14.2.2.1, “Static Index Time Boosting”](#) defines a static boost factor which is independent of the state of the indexed entity at runtime. However, there are usecases in which the boost factor may depend on the actual state of the entity. In this case you can use the **@DynamicBoost** annotation together with an accompanying custom **BoostStrategy**.

Example 14.18. Dynamic boost example

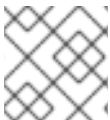
```
public enum PersonType {
    NORMAL,
    VIP
}

@Entity
@Indexed
@DynamicBoost(impl = VIPBoostStrategy.class)
public class Person {
    private PersonType type;

    // ....
}

public class VIPBoostStrategy implements BoostStrategy {
    public float defineBoost(Object value) {
        Person person = ( Person ) value;
        if ( person.getType().equals( PersonType.VIP ) ) {
            return 2.0f;
        }
        else {
            return 1.0f;
        }
    }
}
```

In [Example 14.18, “Dynamic boost example”](#) a dynamic boost is defined on class level specifying **VIPBoostStrategy** as implementation of the **BoostStrategy** interface to be used at indexing time. You can place the **@DynamicBoost** either at class or field level. Depending on the placement of the annotation either the whole entity is passed to the **defineBoost** method or just the annotated field/property value. It's up to you to cast the passed object to the correct type. In the example all indexed values of a VIP person would be double as important as the values of a normal person.



NOTE

The specified **BoostStrategy** implementation must define a public no-arg constructor.

Of course you can mix and match **@Boost** and **@DynamicBoost** annotations in your entity. All defined boost factors are cumulative.

[Report a bug](#)

14.2.3. Analysis

Analysis is the process of converting text into single terms (words) and can be considered as one of the key features of a full-text search engine. Lucene uses the concept of **Analyzers** to control this process. In the following section we cover the multiple ways Hibernate Search offers to configure the analyzers.

[Report a bug](#)

14.2.3.1. Default Analyzer and Analyzer by Class

The default analyzer class used to index tokenized fields is configurable through the **hibernate.search.analyzer** property. The default value for this property is **org.apache.lucene.analysis.standard.StandardAnalyzer**.

You can also define the analyzer class per entity, property and even per `@Field` (useful when multiple fields are indexed from a single property).

Example 14.19. Different ways of using `@Analyzer`

```
@Entity
@Indexed
@Analyzer(impl = EntityAnalyzer.class)
public class MyEntity {
    @Id
    @GeneratedValue
    @DocumentId
    private Integer id;

    @Field
    private String name;

    @Field
    @Analyzer(impl = PropertyAnalyzer.class)
    private String summary;

    @Field( analyzer = @Analyzer(impl = FieldAnalyzer.class) )
    private String body;

    ...
}
```

In this example, **EntityAnalyzer** is used to index tokenized property (**name**), except **summary** and **body** which are indexed with **PropertyAnalyzer** and **FieldAnalyzer** respectively.



WARNING

Mixing different analyzers in the same entity is most of the time a bad practice. It makes query building more complex and results less predictable (for the novice), especially if you are using a `QueryParser` (which uses the same analyzer for the whole query). As a rule of thumb, for any given field the same analyzer should be used for indexing and querying.

[Report a bug](#)

14.2.3.2. Named Analyzers

Analyzers can become quite complex to deal with. For this reason introduces Hibernate Search the notion of analyzer definitions. An analyzer definition can be reused by many `@Analyzer` declarations and is composed of:

- a name: the unique string used to refer to the definition
- a list of char filters: each char filter is responsible to pre-process input characters before the tokenization. Char filters can add, change, or remove characters; one common usage is for characters normalization
- a tokenizer: responsible for tokenizing the input stream into individual words
- a list of filters: each filter is responsible to remove, modify, or sometimes even add words into the stream provided by the tokenizer

This separation of tasks - a list of char filters, and a tokenizer followed by a list of filters - allows for easy reuse of each individual component and let you build your customized analyzer in a very flexible way (like Lego). Generally speaking the char filters do some pre-processing in the character input, then the **Tokenizer** starts the tokenizing process by turning the character input into tokens which are then further processed by the **TokenFilters**. Hibernate Search supports this infrastructure by utilizing the Solr analyzer framework.

NOTE

Some of the analyzers and filters will require additional dependencies. For example to use the snowball stemmer you have to also include the **lucene-snowball** jar and for the **PhoneticFilterFactory** you need the [commons-codec](#) jar. Your distribution of Hibernate Search provides these dependencies in its **lib/optional** directory.

When using Maven all required Solr dependencies are now defined as dependencies of the artifact `org.hibernate:hibernate-search-analyzers`; add the following dependency :

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-search-analyzers</artifactId>
  <version>4.6.0.Final-redhat-2</version>
  <scope>provided</scope>
</dependency>
```

Let's review a concrete example now - [Example 14.20, “@AnalyzerDef and the Solr framework”](#). First a char filter is defined by its factory. In our example, a mapping char filter is used, and will replace characters in the input based on the rules specified in the mapping file. Next a tokenizer is defined. This example uses the standard tokenizer. Last but not least, a list of filters is defined by their factories. In our example, the **StopFilter** filter is built reading the dedicated words property file. The filter is also expected to ignore case.

Example 14.20. @AnalyzerDef and the Solr framework

```
@AnalyzerDef(name="customanalyzer",
    charFilters = {
        @CharFilterDef(factory = MappingCharFilterFactory.class, params = {
            @Parameter(name = "mapping",
                value = "org/hibernate/search/test/analyzer/solr/mapping-
chars.properties")
        })
    },
    tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
    filters = {
        @TokenFilterDef(factory = ISOLatin1AccentFilterFactory.class),
        @TokenFilterDef(factory = LowerCaseFilterFactory.class),
        @TokenFilterDef(factory = StopFilterFactory.class, params = {
            @Parameter(name="words",
                value=
"org/hibernate/search/test/analyzer/solr/stoplist.properties" ),
            @Parameter(name="ignoreCase", value="true")
        })
    })
}

public class Team {
    ...
}
```



NOTE

Filters and char filters are applied in the order they are defined in the **@AnalyzerDef** annotation. Order matters!

Some tokenizers, token filters or char filters load resources like a configuration or metadata file. This is the case for the stop filter and the synonym filter. If the resource charset is not using the VM default, you can explicitly specify it by adding a **resource_charset** parameter.

Example 14.21. Use a specific charset to load the property file

```
@AnalyzerDef(name="customanalyzer",
    charFilters = {
        @CharFilterDef(factory = MappingCharFilterFactory.class, params = {
            @Parameter(name = "mapping",
                value = "org/hibernate/search/test/analyzer/solr/mapping-
chars.properties")
        })
    },
    tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
```

```

filters = {
    @TokenFilterDef(factory = ISOLatin1AccentFilterFactory.class),
    @TokenFilterDef(factory = LowerCaseFilterFactory.class),
    @TokenFilterDef(factory = StopFilterFactory.class, params = {
        @Parameter(name="words",
            value=
"org/hibernate/search/test/analyser/solr/stoplist.properties" ),
        @Parameter(name="resource_charset", value = "UTF-16BE"),
        @Parameter(name="ignoreCase", value="true")
    })
})
})
public class Team {
    ...
}

```

Once defined, an analyzer definition can be reused by an **@Analyzer** declaration as seen in [Example 14.22, “Referencing an analyzer by name”](#).

Example 14.22. Referencing an analyzer by name

```

@Entity
@Indexed
@AnalyzerDef(name="customanalyzer", ... )
public class Team {
    @Id
    @DocumentId
    @GeneratedValue
    private Integer id;

    @Field
    private String name;

    @Field
    private String location;

    @Field
    @Analyzer(definition = "customanalyzer")
    private String description;
}

```

Analyzer instances declared by **@AnalyzerDef** are also available by their name in the **SearchFactory** which is quite useful when building queries.

```

Analyzer analyzer =
fullTextSession.getSearchFactory().getAnalyzer("customanalyzer");

```

Fields in queries must be analyzed with the same analyzer used to index the field so that they speak a common "language": the same tokens are reused between the query and the indexing process. This rule has some exceptions but is true most of the time. Respect it unless you know what you are doing.

[Report a bug](#)

14.2.3.3. Available Analyzers

Solr and Lucene come with a lot of useful default char filters, tokenizers, and filters. You can find a complete list of char filter factories, tokenizer factories and filter factories at <http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters>. Let's check a few of them.

Table 14.1. Example of available char filters

Factory	Description	Parameters	Additional dependencies
MappingCharFilterFactory	Replaces one or more characters with one or more characters, based on mappings specified in the resource file	mapping : points to a resource file containing the mappings using the format: <pre>"á" => "a" "ñ" => "n" "ø" => "o"</pre>	none
HTMLStripCharFilterFactory	Remove HTML standard tags, keeping the text	none	none

Table 14.2. Example of available tokenizers

Factory	Description	Parameters	Additional dependencies
StandardTokenizerFactory	Use the Lucene StandardTokenizer	none	none
HTMLStripCharFilterFactory	Remove HTML tags, keep the text and pass it to a StandardTokenizer.	none	solr-core
PatternTokenizerFactory	Breaks text at the specified regular expression pattern.	pattern : the regular expression to use for tokenizing group : says which pattern group to extract into tokens	solr-core

Table 14.3. Examples of available filters

Factory	Description	Parameters	Additional dependencies
StandardFilterFactory	Remove dots from acronyms and 's from words	none	solr-core

Factory	Description	Parameters	Additional dependencies
LowerCaseFilterFactory	Lowercases all words	none	solr-core
StopFilterFactory	Remove words (tokens) matching a list of stop words	words : points to a resource file containing the stop words ignoreCase : true if case should be ignored when comparing stop words, false otherwise	solr-core
SnowballPorterFilterFactory	Reduces a word to its root in a given language. (example: protect, protects, protection share the same root). Using such a filter allows searches matching related words.	language : Danish, Dutch, English, Finnish, French, German, Italian, Norwegian, Portuguese, Russian, Spanish, Swedish and a few more	solr-core
ISOLatin1AccentFilterFactory	Remove accents for languages like French	none	solr-core
PhoneticFilterFactory	Inserts phonetically similar tokens into the token stream	encoder : One of DoubleMetaphone, Metaphone, Soundex or RefinedSoundex inject : true will add tokens to the stream, false will replace the existing token maxCodeLength : sets the maximum length of the code to be generated. Supported only for Metaphone and DoubleMetaphone encodings	solr-core and commons-codec
CollationKeyFilterFactory	Converts each token into its java.text.CollationKey , and then encodes the CollationKey with IndexableBinaryStringTools , to allow it to be stored as an index term.	custom , language , country , variant , strength , decomposition For more information, see Lucene's CollationKeyFilter javadocs	solr-core and commons-io

We recommend to check all the implementations of `org.apache.solr.analysis.TokenizerFactory` and `org.apache.solr.analysis.TokenFilterFactory` in your IDE to see the implementations available.

[Report a bug](#)

14.2.3.4. Dynamic Analyzer Selection

So far all the introduced ways to specify an analyzer were static. However, there are use cases where it is useful to select an analyzer depending on the current state of the entity to be indexed, for example in a multilingual applications. For an **BlogEntry** class for example the analyzer could depend on the language property of the entry. Depending on this property the correct language specific stemmer should be chosen to index the actual text.

To enable this dynamic analyzer selection Hibernate Search introduces the **AnalyzerDiscriminator** annotation. [Example 14.23, "Usage of @AnalyzerDiscriminator"](#) demonstrates the usage of this annotation.

Example 14.23. Usage of @AnalyzerDiscriminator

```
@Entity
@Indexed
@AnalyzerDefs({
    @AnalyzerDef(name = "en",
        tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
        filters = {
            @TokenFilterDef(factory = LowerCaseFilterFactory.class),
            @TokenFilterDef(factory = EnglishPorterFilterFactory.class)
        }
    ),
    @AnalyzerDef(name = "de",
        tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
        filters = {
            @TokenFilterDef(factory = LowerCaseFilterFactory.class),
            @TokenFilterDef(factory = GermanStemFilterFactory.class)
        }
    )
})
public class BlogEntry {

    @Id
    @GeneratedValue
    @DocumentId
    private Integer id;

    @Field
    @AnalyzerDiscriminator(impl = LanguageDiscriminator.class)
    private String language;

    @Field
    private String text;

    private Set<BlogEntry> references;
```

```

        // standard getter/setter
        ...
    }

    public class LanguageDiscriminator implements Discriminator {

        public String getAnalyzerDefinitionName(Object value, Object entity,
String field) {
            if ( value == null || !( entity instanceof BlogEntry ) ) {
                return null;
            }
            return (String) value;
        }
    }
}

```

The prerequisite for using **@AnalyzerDiscriminator** is that all analyzers which are going to be used dynamically are predefined via **@AnalyzerDef** definitions. If this is the case, one can place the **@AnalyzerDiscriminator** annotation either on the class or on a specific property of the entity for which to dynamically select an analyzer. Via the **impl** parameter of the **AnalyzerDiscriminator** you specify a concrete implementation of the **Discriminator** interface. It is up to you to provide an implementation for this interface. The only method you have to implement is **getAnalyzerDefinitionName()** which gets called for each field added to the Lucene document. The entity which is getting indexed is also passed to the interface method. The **value** parameter is only set if the **AnalyzerDiscriminator** is placed on property level instead of class level. In this case the value represents the current value of this property.

An implementation of the **Discriminator** interface has to return the name of an existing analyzer definition or **null** if the default analyzer should not be overridden. [Example 14.23, “Usage of @AnalyzerDiscriminator”](#) assumes that the language parameter is either 'de' or 'en' which matches the specified names in the **@AnalyzerDefs**.

[Report a bug](#)

14.2.3.5. Retrieving an Analyzer

Retrieving an analyzer can be used when multiple analyzers have been used in a domain model, in order to benefit from stemming or phonetic approximation, etc. In this case, use the same analyzers to building a query. Alternatively, use the Hibernate Search query DSL, which selects the correct analyzer automatically. See [Section 14.3.1.2, “Building a Lucene Query”](#)

Whether you are using the Lucene programmatic API or the Lucene query parser, you can retrieve the scoped analyzer for a given entity. A scoped analyzer is an analyzer which applies the right analyzers depending on the field indexed. Remember, multiple analyzers can be defined on a given entity each one working on an individual field. A scoped analyzer unifies all these analyzers into a context-aware analyzer. While the theory seems a bit complex, using the right analyzer in a query is very easy.



NOTE

When you use programmatic mapping for a child entity, you can only see the fields defined by the child entity. Fields or methods inherited from a parent entity (annotated with `@MappedSuperclass`) are not configurable. To configure properties inherited from a parent entity, either override the property in the child entity or create a programmatic mapping for the parent entity. This mimics the usage of annotations where you cannot annotate a field or method of a parent entity unless it is redefined in the child entity.

Example 14.24. Using the scoped analyzer when building a full-text query

```
org.apache.lucene.queryParser.QueryParser parser = new QueryParser(
    "title",
    fullTextSession.getSearchFactory().getAnalyzer( Song.class )
);

org.apache.lucene.search.Query luceneQuery =
    parser.parse( "title:sky Or title_stemmed:diamond" );

org.hibernate.Query fullTextQuery =
    fullTextSession.createFullTextQuery( luceneQuery, Song.class );

List result = fullTextQuery.list(); //return a list of managed objects
```

In the example above, the song title is indexed in two fields: the standard analyzer is used in the field **title** and a stemming analyzer is used in the field **title_stemmed**. By using the analyzer provided by the search factory, the query uses the appropriate analyzer depending on the field targeted.



NOTE

You can also retrieve analyzers defined via `@AnalyzerDef` by their definition name using `searchFactory.getAnalyzer(String)`.

[Report a bug](#)

14.2.4. Bridges

When discussing the basic mapping for an entity one important fact was so far disregarded. In Lucene all index fields have to be represented as strings. All entity properties annotated with `@Field` have to be converted to strings to be indexed. The reason we have not mentioned it so far is, that for most of your properties Hibernate Search does the translation job for you thanks to set of built-in bridges. However, in some cases you need a more fine grained control over the translation process.

[Report a bug](#)

14.2.4.1. Built-in Bridges

Hibernate Search comes bundled with a set of built-in bridges between a Java property type and its full text representation.

null

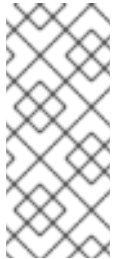
Per default **null** elements are not indexed. Lucene does not support null elements. However, in some situation it can be useful to insert a custom token representing the **null** value. See [Section 14.2.1.1.2, “@Field”](#) for more information.

java.lang.String

Strings are indexed as are

short, Short, integer, Integer, long, Long, float, Float, double, Double, BigInteger, BigDecimal

Numbers are converted into their string representation. Note that numbers cannot be compared by Lucene (that is, used in ranged queries) out of the box: they have to be padded.



NOTE

Using a Range query has drawbacks, an alternative approach is to use a Filter query which will filter the result query to the appropriate range.

Hibernate Search also supports the use of a custom StringBridge as described in [Section 14.2.4.2, “Custom Bridges”](#).

java.util.Date

Dates are stored as yyyyMMddHHmmssSSS in GMT time (200611072203012 for Nov 7th of 2006 4:03PM and 12ms EST). You shouldn't really bother with the internal format. What is important is that when using a TermRangeQuery, you should know that the dates have to be expressed in GMT time.

Usually, storing the date up to the millisecond is not necessary. **@DateBridge** defines the appropriate resolution you are willing to store in the index (**@DateBridge(resolution=Resolution.DAY)**). The date pattern will then be truncated accordingly.

```
@Entity
@Indexed
public class Meeting {
    @Field(analyze=Analyze.NO)
    @DateBridge(resolution=Resolution.MINUTE)
    private Date date;
    ...
}
```



WARNING

A Date whose resolution is lower than **MILLISECOND** cannot be a **@DocumentId**.



IMPORTANT

The default **Date** bridge uses Lucene's **DateTools** to convert from and to **String**. This means that all dates are expressed in GMT time. If your requirements are to store dates in a fixed time zone you have to implement a custom date bridge. Make sure you understand the requirements of your applications regarding to date indexing and searching.

java.net.URI, java.net.URL

URI and URL are converted to their string representation.

java.lang.Class

Class are converted to their fully qualified class name. The thread context class loader is used when the class is rehydrated.

[Report a bug](#)

14.2.4.2. Custom Bridges

Sometimes, the built-in bridges of Hibernate Search do not cover some of your property types, or the String representation used by the bridge does not meet your requirements. The following paragraphs describe several solutions to this problem.

[Report a bug](#)

14.2.4.2.1. StringBridge

The simplest custom solution is to give Hibernate Search an implementation of your expected **Object** to **String** bridge. To do so you need to implement the **org.hibernate.search.bridge.StringBridge** interface. All implementations have to be thread-safe as they are used concurrently.

Example 14.25. Custom StringBridge implementation

```
/**
 * Padding Integer bridge.
 * All numbers will be padded with 0 to match 5 digits
 *
 * @author Emmanuel Bernard
 */
public class PaddedIntegerBridge implements StringBridge {

    private int PADDING = 5;

    public String objectToString(Object object) {
        String rawInteger = ( (Integer) object ).toString();
        if (rawInteger.length() > PADDING)
            throw new IllegalArgumentException( "Try to pad on a number
too big" );
        StringBuilder paddedInteger = new StringBuilder( );
        for ( int padIndex = rawInteger.length() ; padIndex < PADDING ;
padIndex++ ) {
```

```

        paddedInteger.append( '0' );
    }
    return paddedInteger.append( rawInteger ).toString();
}
}

```

Given the string bridge defined in [Example 14.25](#), “Custom **StringBridge** implementation”, any property or field can use this bridge thanks to the **@FieldBridge** annotation:

```

@FieldBridge(impl = PaddedIntegerBridge.class)
private Integer length;

```

[Report a bug](#)

14.2.4.2.2. Parameterized Bridge

Parameters can also be passed to the bridge implementation making it more flexible. [Example 14.26](#), “Passing parameters to your bridge implementation” implements a **ParameterizedBridge** interface and parameters are passed through the **@FieldBridge** annotation.

Example 14.26. Passing parameters to your bridge implementation

```

public class PaddedIntegerBridge implements StringBridge,
ParameterizedBridge {

    public static String PADDING_PROPERTY = "padding";
    private int padding = 5; //default

    public void setParameterValues(Map<String,String> parameters) {
        String padding = parameters.get( PADDING_PROPERTY );
        if (padding != null) this.padding = Integer.parseInt( padding
    );
    }

    public String objectToString(Object object) {
        String rawInteger = ( (Integer) object ).toString();
        if (rawInteger.length() > padding)
            throw new IllegalArgumentException( "Try to pad on a number
too big" );
        StringBuilder paddedInteger = new StringBuilder( );
        for ( int padIndex = rawInteger.length() ; padIndex < padding ;
padIndex++ ) {
            paddedInteger.append( '0' );
        }
        return paddedInteger.append( rawInteger ).toString();
    }
}

//property
@FieldBridge(impl = PaddedIntegerBridge.class,

```

```

        params = @Parameter(name="padding", value="10")
    )
    private Integer length;

```

The **ParameterizedBridge** interface can be implemented by **StringBridge**, **TwoWayStringBridge**, **FieldBridge** implementations.

All implementations have to be thread-safe, but the parameters are set during initialization and no special care is required at this stage.

[Report a bug](#)

14.2.4.2.3. Type Aware Bridge

It is sometimes useful to get the type the bridge is applied on:

- the return type of the property for field/getter-level bridges.
- the class type for class-level bridges.

An example is a bridge that deals with enums in a custom fashion but needs to access the actual enum type. Any bridge implementing **AppliedOnTypeAwareBridge** will get the type the bridge is applied on injected. Like parameters, the type injected needs no particular care with regard to thread-safety.

[Report a bug](#)

14.2.4.2.4. Two-Way Bridge

If you expect to use your bridge implementation on an id property (that is, annotated with **@DocumentId**), you need to use a slightly extended version of **StringBridge** named **TwoWayStringBridge**. Hibernate Search needs to read the string representation of the identifier and generate the object out of it. There is no difference in the way the **@FieldBridge** annotation is used.

Example 14.27. Implementing a TwoWayStringBridge usable for id properties

```

public class PaddedIntegerBridge implements TwoWayStringBridge,
ParameterizedBridge {

    public static String PADDING_PROPERTY = "padding";
    private int padding = 5; //default

    public void setParameterValues(Map parameters) {
        Object padding = parameters.get( PADDING_PROPERTY );
        if (padding != null) this.padding = (Integer) padding;
    }

    public String objectToString(Object object) {
        String rawInteger = ( (Integer) object ).toString();
        if (rawInteger.length() > padding)
            throw new IllegalArgumentException( "Try to pad on a number
too big" );
        StringBuilder paddedInteger = new StringBuilder( );
        for ( int padIndex = rawInteger.length() ; padIndex < padding ;
padIndex++ ) {

```



```

        paddedInteger.append('0');
    }
    return paddedInteger.append( rawInteger ).toString();
}

public Object stringToObject(String stringValue) {
    return new Integer(stringValue);
}
}

//id property
@DocumentId
@FieldBridge(impl = PaddedIntegerBridge.class,
              params = @Parameter(name="padding", value="10"))
private Integer id;

```



IMPORTANT

It is important for the two-way process to be idempotent (ie `object = stringToObject(objectToString(object))`).

[Report a bug](#)

14.2.4.2.5. FieldBridge

Some use cases require more than a simple object to string translation when mapping a property to a Lucene index. To give you the greatest possible flexibility you can also implement a bridge as a **FieldBridge**. This interface gives you a property value and let you map it the way you want in your Lucene **Document**. You can for example store a property in two different document fields. The interface is very similar in its concept to the Hibernate **UserTypes**.

Example 14.28. Implementing the FieldBridge Interface

```

/**
 * Store the date in 3 different fields - year, month, day - to ease
 * Range Query per
 * year, month or day (eg get all the elements of December for the last
 * 5 years).
 * @author Emmanuel Bernard
 */
public class DateSplitBridge implements FieldBridge {
    private final static TimeZone GMT = TimeZone.getTimeZone("GMT");

    public void set(String name, Object value, Document document,
        LuceneOptions luceneOptions) {
        Date date = (Date) value;
        Calendar cal = GregorianCalendar.getInstance(GMT);
        cal.setTime(date);
        int year = cal.get(Calendar.YEAR);
        int month = cal.get(Calendar.MONTH) + 1;
        int day = cal.get(Calendar.DAY_OF_MONTH);
    }
}

```

```

        // set year
        luceneOptions.addFieldToDocument(
            name + ".year",
            String.valueOf( year ),
            document );

        // set month and pad it if needed
        luceneOptions.addFieldToDocument(
            name + ".month",
            month < 10 ? "0" : "" + String.valueOf( month ),
            document );

        // set day and pad it if needed
        luceneOptions.addFieldToDocument(
            name + ".day",
            day < 10 ? "0" : "" + String.valueOf( day ),
            document );
    }
}

//property
@FieldBridge(impl = DateSplitBridge.class)
private Date date;

```

In [Example 14.28, “Implementing the FieldBridge Interface”](#) the fields are not added directly to `Document`. Instead the addition is delegated to the `LuceneOptions` helper; this helper will apply the options you have selected on `@Field`, like `Store` or `TermVector`, or apply the chosen `@Boost` value. It is especially useful to encapsulate the complexity of `COMPRESS` implementations. Even though it is recommended to delegate to `LuceneOptions` to add fields to the `Document`, nothing stops you from editing the `Document` directly and ignore the `LuceneOptions` in case you need to.



NOTE

Classes like `LuceneOptions` are created to shield your application from changes in Lucene API and simplify your code. Use them if you can, but if you need more flexibility you're not required to.

[Report a bug](#)

14.2.4.2.6. ClassBridge

It is sometimes useful to combine more than one property of a given entity and index this combination in a specific way into the Lucene index. The `@ClassBridge` and `@ClassBridges` annotations can be defined at the class level, as opposed to the property level. In this case the custom field bridge implementation receives the entity instance as the value parameter instead of a particular property. Though not shown in [Example 14.29, “Implementing a class bridge”](#), `@ClassBridge` supports the `termVector` attribute discussed in [Section 14.2.1.1, “Basic Mapping”](#).

Example 14.29. Implementing a class bridge

```

@Entity
@Indexed
@ClassBridge(name="branchnetwork",

```

```

        store=Store.YES,
        impl = CatFieldsClassBridge.class,
        params = @Parameter( name="sepChar", value=" " ) )
public class Department {
    private int id;
    private String network;
    private String branchHead;
    private String branch;
    private Integer maxEmployees
    ...
}

public class CatFieldsClassBridge implements FieldBridge,
ParameterizedBridge {
    private String sepChar;

    public void setParameterValues(Map parameters) {
        this.sepChar = (String) parameters.get( "sepChar" );
    }

    public void set( String name, Object value, Document document,
LuceneOptions luceneOptions) {
        // In this particular class the name of the new field was
        // passed
        // from the name field of the ClassBridge Annotation. This is
        // not
        // a requirement. It just works that way in this instance. The
        // actual name could be supplied by hard coding it below.
        Department dep = (Department) value;
        String fieldValue1 = dep.getBranch();
        if ( fieldValue1 == null ) {
            fieldValue1 = "";
        }
        String fieldValue2 = dep.getNetwork();
        if ( fieldValue2 == null ) {
            fieldValue2 = "";
        }
        String fieldValue = fieldValue1 + sepChar + fieldValue2;
        Field field = new Field( name, fieldValue,
luceneOptions.getStore(),
        luceneOptions.getIndex(), luceneOptions.getTermVector() );
        field.setBoost( luceneOptions.getBoost() );
        document.add( field );
    }
}

```

In this example, the particular **CatFieldsClassBridge** is applied to the **department** instance, the field bridge then concatenate both branch and network and index the concatenation.

[Report a bug](#)

14.3. QUERYING

Hibernate Search can execute Lucene queries and retrieve domain objects managed by an Hibernate session. The search provides the power of Lucene without leaving the Hibernate paradigm, giving another dimension to the Hibernate classic search mechanisms (HQL, Criteria query, native SQL query).

Preparing and executing a query consists of following four steps:

- Creating a **FullTextSession**
- Creating a Lucene query using either Hibernate Search query DSL (recommended) or using the Lucene Query API
- Wrapping the Lucene query using an **org.hibernate.Query**
- Executing the search by calling for example **list()** or **scroll()**

To access the querying facilities, use a **FullTextSession**. This Search specific session wraps a regular **org.hibernate.Session** in order to provide query and indexing capabilities.

Example 14.30. Creating a FullTextSession

```
Session session = sessionFactory.openSession();
...
FullTextSession fullTextSession = Search.getFullTextSession(session);
```

Use the **FullTextSession** to build a full-text query using either the Hibernate Search query DSL or the native Lucene query.

Use the following code when using the Hibernate Search query DSL:

```
final QueryBuilder b =
fullTextSession.getSearchFactory().buildQueryBuilder().forEntity(
Myth.class ).get();

org.apache.lucene.search.Query luceneQuery =
    b.keyword()
      .onField("history").boostedTo(3)
      .matching("storm")
      .createQuery();

org.hibernate.Query fullTextQuery = fullTextSession.createFullTextQuery(
luceneQuery );
List result = fullTextQuery.list(); //return a list of managed objects
```

As an alternative, write the Lucene query using either the Lucene query parser or the Lucene programmatic API.

Example 14.31. Creating a Lucene query via the QueryParser

```
SearchFactory searchFactory = fullTextSession.getSearchFactory();
org.apache.lucene.queryParser.QueryParser parser =
    new QueryParser("title", searchFactory.getAnalyzer(Myth.class) );
try {
    org.apache.lucene.search.Query luceneQuery = parser.parse(
```

```

    "history:storm^3" );
}
catch (ParseException e) {
    //handle parsing failure
}

org.hibernate.Query fullTextQuery =
fullTextSession.createFullTextQuery(luceneQuery);
List result = fullTextQuery.list(); //return a list of managed objects

```

A Hibernate query built on the Lucene query is a **org.hibernate.Query**. This query remains in the same paradigm as other Hibernate query facilities, such as HQL (Hibernate Query Language), Native, and Criteria. Use methods such as **list()**, **uniqueResult()**, **iterate()** and **scroll()** with the query.

The same extensions are available with the Hibernate Java Persistence APIs:

Example 14.32. Creating a Search query using the JPA API

```

EntityManager em = entityManagerFactory.createEntityManager();

FullTextEntityManager fullTextEntityManager =
    org.hibernate.search.jpa.Search.getFullTextEntityManager(em);

...
final QueryBuilder b = fullTextEntityManager.getSearchFactory()
    .buildQueryBuilder().forEntity( Myth.class ).get();

org.apache.lucene.search.Query luceneQuery =
    b.keyword()
        .onField("history").boostedTo(3)
        .matching("storm")
        .createQuery();
javax.persistence.Query fullTextQuery =
fullTextEntityManager.createFullTextQuery( luceneQuery );

List result = fullTextQuery.getResultList(); //return a list of managed
objects

```



NOTE

In these examples, the Hibernate API has been used. The same examples can also be written with the Java Persistence API by adjusting the way the **FullTextQuery** is retrieved.

[Report a bug](#)

14.3.1. Building Queries

Hibernate Search queries are built on Lucene queries, allowing users to use any Lucene query type. When the query is built, Hibernate Search uses `org.hibernate.Query` as the query manipulation API for further query processing.

[Report a bug](#)

14.3.1.1. Building a Lucene Query Using the Lucene API

With the Lucene API, use either the query parser (simple queries) or the Lucene programmatic API (complex queries). Building a Lucene query is out of scope for the Hibernate Search documentation. For details, see the online Lucene documentation or a copy of *Lucene in Action* or *Hibernate Search in Action*.

[Report a bug](#)

14.3.1.2. Building a Lucene Query

The Lucene programmatic API enables full-text queries. However, when using the Lucene programmatic API, the parameters must be converted to their string equivalent and must also apply the correct analyzer to the right field. A ngram analyzer for example uses several ngrams as the tokens for a given word and should be searched as such. It is recommended to use the **QueryBuilder** for this task.

The Hibernate Search query API is fluent, with the following key characteristics:

- Method names are in English. As a result, API operations can be read and understood as a series of English phrases and instructions.
- It uses IDE autocompletion which helps possible completions for the current input prefix and allows the user to choose the right option.
- It often uses the chaining method pattern.
- It is easy to use and read the API operations.

To use the API, first create a query builder that is attached to a given **indexedentitytype**. This **QueryBuilder** knows what analyzer to use and what field bridge to apply. Several **QueryBuilder**s (one for each entity type involved in the root of your query) can be created. The **QueryBuilder** is derived from the **SearchFactory**.

```
QueryBuilder mythQB = searchFactory.buildQueryBuilder().forEntity(Myth.class).get();
```

The analyzer used for a given field or fields can also be overridden.

```
QueryBuilder mythQB = searchFactory.buildQueryBuilder()
    .forEntity( Myth.class )
    .overridesForField("history","stem_analyzer_definition")
    .get();
```

The query builder is now used to build Lucene queries. Customized queries generated using Lucene's query parser or **Query** objects assembled using the Lucene programmatic API are used with the Hibernate Search DSL.

[Report a bug](#)

14.3.1.3. Keyword Queries

The following example shows how to search for a specific word:

```
Query luceneQuery =
mythQB.keyword().onField("history").matching("storm").createQuery();
```

Table 14.4. Keyword query parameters

Parameter	Description
keyword()	Use this parameter to find a specific word
onField()	Use this parameter to specify in which lucene field to search the word
matching()	use this parameter to specify the match for search string
createQuery()	creates the Lucene query object

- The value "storm" is passed through the **history FieldBridge**. This is useful when numbers or dates are involved.
- The field bridge value is then passed to the analyzer used to index the field **history**. This ensures that the query uses the same term transformation than the indexing (lower case, ngram, stemming and so on). If the analyzing process generates several terms for a given word, a boolean query is used with the **SHOULD** logic (roughly an **OR** logic).

To search a property that is not of type string.

```
@Indexed
public class Myth {
    @Field(analyze = Analyze.NO)
    @DateBridge(resolution = Resolution.YEAR)
    public Date getCreationDate() { return creationDate; }
    public Date setCreationDate(Date creationDate) { this.creationDate =
creationDate; }
    private Date creationDate;

    ...
}

Date birthdate = ...;
Query luceneQuery =
mythQb.keyword().onField("creationDate").matching(birthdate).createQuery()
;
```



NOTE

In plain Lucene, the **Date** object had to be converted to its string representation (in this case the year)

This conversion works for any object, provided that the **FieldBridge** has an **objectToString** method (and all built-in **FieldBridge** implementations do).

The next example searches a field that uses ngram analyzers. The ngram analyzers index succession of ngrams of words, which helps to avoid user typos. For example, the 3-grams of the word hibernate are hib, ibe, ber, ern, rna, nat, ate.

```
@AnalyzerDef(name = "ngram",
    tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class ),
    filters = {
        @TokenFilterDef(factory = StandardFilterFactory.class),
        @TokenFilterDef(factory = LowerCaseFilterFactory.class),
        @TokenFilterDef(factory = StopFilterFactory.class),
        @TokenFilterDef(factory = NGramFilterFactory.class,
            params = {
                @Parameter(name = "minGramSize", value = "3"),
                @Parameter(name = "maxGramSize", value = "3") } )
    }
)

public class Myth {
    @Field(analyzer=@Analyzer(definition="ngram")
    public String getName() { return name; }
    public String setName(String name) { this.name = name; }
    private String name;

    ...
}

Date birthdate = ...;
Query luceneQuery = mythQb.keyword().onField("name").matching("Sisiphus")
    .createQuery();
```

The matching word "Sisiphus" will be lower-cased and then split into 3-grams: sis, isi, sip, iph, phu, hus. Each of these ngram will be part of the query. The user is then able to find the Sysiphus myth (with a y). All that is transparently done for the user.



NOTE

If the user does not want a specific field to use the field bridge or the analyzer then the **ignoreAnalyzer()** or **ignoreFieldBridge()** functions can be called.

To search for multiple possible words in the same field, add them all in the matching clause.

```
//search document with storm or lightning in their history
Query luceneQuery =
    mythQB.keyword().onField("history").matching("storm
    lightning").createQuery();
```

To search the same word on multiple fields, use the **onFields** method.

```
Query luceneQuery = mythQB
    .keyword()
    .onFields("history", "description", "name")
```



```
.matching("storm")
.createQuery();
```

Sometimes, one field should be treated differently from another field even if searching the same term, use the **andField()** method for that.

```
Query luceneQuery = mythQB.keyword()
    .onField("history")
    .andField("name")
    .boostedTo(5)
    .andField("description")
    .matching("storm")
    .createQuery();
```

In the previous example, only field name is boosted to 5.

[Report a bug](#)

14.3.1.4. Fuzzy Queries

To execute a fuzzy query (based on the Levenshtein distance algorithm), start with a **keyword** query and add the **fuzzy** flag.

```
Query luceneQuery = mythQB
    .keyword()
    .fuzzy()
    .withThreshold( .8f )
    .withPrefixLength( 1 )
    .onField("history")
    .matching("starm")
    .createQuery();
```

The **threshold** is the limit above which two terms are considering matching. It is a decimal between 0 and 1 and the default value is 0.5. The **prefixLength** is the length of the prefix ignored by the "fuzzyness". While the default value is 0, a nonzero value is recommended for indexes containing a huge number of distinct terms.

[Report a bug](#)

14.3.1.5. Wildcard Queries

Wildcard queries are useful in circumstances where only part of the word is known. The **?** represents a single character and ***** represents multiple characters. Note that for performance purposes, it is recommended that the query does not start with either **?** or *****.

```
Query luceneQuery = mythQB
    .keyword()
    .wildcard()
    .onField("history")
    .matching("sto*")
    .createQuery();
```

**NOTE**

Wildcard queries do not apply the analyzer on the matching terms. The risk of * or ? being mangled is too high.

[Report a bug](#)

14.3.1.6. Phrase Queries

So far we have been looking for words or sets of words, the user can also search exact or approximate sentences. Use **phrase()** to do so.

```
Query luceneQuery = mythQB
    .phrase()
    .onField("history")
    .sentence("Thou shalt not kill")
    .createQuery();
```

Approximate sentences can be searched by adding a slop factor. The slop factor represents the number of other words permitted in the sentence: this works like a within or near operator.

```
Query luceneQuery = mythQB
    .phrase()
    .withSlop(3)
    .onField("history")
    .sentence("Thou kill")
    .createQuery();
```

[Report a bug](#)

14.3.1.7. Range Queries

A range query searches for a value in between given boundaries (included or not) or for a value below or above a given boundary (included or not).

```
//look for 0 <= starred < 3
Query luceneQuery = mythQB
    .range()
    .onField("starred")
    .from(0).to(3).excludeLimit()
    .createQuery();

//look for myths strictly BC
Date beforeChrist = ...;
Query luceneQuery = mythQB
    .range()
    .onField("creationDate")
    .below(beforeChrist).excludeLimit()
    .createQuery();
```

[Report a bug](#)

14.3.1.8. Combining Queries

Queries can be aggregated (combined) to create more complex queries. The following aggregation operators are available:

- **SHOULD**: the query should contain the matching elements of the subquery.
- **MUST**: the query must contain the matching elements of the subquery.
- **MUST NOT**: the query must not contain the matching elements of the subquery.

The subqueries can be any Lucene query including a boolean query itself.

Example 14.33. MUST NOT Query

```
//look for popular modern myths that are not urban
Date twentiethCentury = ...;
Query luceneQuery = mythQB
    .bool()
    .must(
        mythQB.keyword().onField("description").matching("urban").createQuery()
    )
    .not()
    .must( mythQB.range().onField("starred").above(4).createQuery() )
    .must( mythQB
        .range()
        .onField("creationDate")
        .above(twentiethCentury)
        .createQuery() )
    .createQuery();
```

Example 14.34. SHOULD Query

```
//look for popular myths that are preferably urban
Query luceneQuery = mythQB
    .bool()
    .should(
        mythQB.keyword().onField("description").matching("urban").createQuery()
    )
    .must( mythQB.range().onField("starred").above(4).createQuery() )
    .createQuery();
```

Example 14.35. NOT Query

```
//look for all myths except religious ones
Query luceneQuery = mythQB
    .all()
    .except( mythQB
        .keyword()
        .onField( "description_stem" )
        .matching( "religion" )
        .createQuery()
    )
    .createQuery();
```

[Report a bug](#)

14.3.1.9. Query Options

The Hibernate Search query DSL is an easy to use and easy to read query API. In accepting and producing Lucene queries, you can incorporate query types not yet supported by the DSL.

The following is a summary of query options for query types and fields:

- **boostedTo** (on query type and on field) boosts the whole query or the specific field to a given factor.
- **withConstantScore** (on query) returns all results that match the query have a constant score equals to the boost.
- **filteredBy(Filter)** (on query) filters query results using the **Filter** instance.
- **ignoreAnalyzer** (on field) ignores the analyzer when processing this field.
- **ignoreFieldBridge** (on field) ignores field bridge when processing this field.

Example 14.36. Combination of Query Options

```
Query luceneQuery = mythQB
    .bool()
    .should(
        mythQB.keyword().onField("description").matching("urban").createQuery()
    )
    .should( mythQB
        .keyword()
        .onField("name")
        .boostedTo(3)
        .ignoreAnalyzer()
        .matching("urban").createQuery() )
    .must( mythQB
        .range()
        .boostedTo(5).withConstantScore()
        .onField("starred").above(4).createQuery() )
    .createQuery();
```

[Report a bug](#)

14.3.1.10. Build a Hibernate Search Query

14.3.1.10.1. Generality

After building the Lucene query, wrap it within a Hibernate query. The query searches all indexed entities and returns all types of indexed classes unless explicitly configured not to do so.

Example 14.37. Wrapping a Lucene Query in a Hibernate Query

```
FullTextSession fullTextSession = Search.getFullTextSession( session );
org.hibernate.Query fullTextQuery = fullTextSession.createFullTextQuery(
    luceneQuery );
```

For improved performance, restrict the returned types as follows:

Example 14.38. Filtering the Search Result by Entity Type

```
fullTextQuery = fullTextSession
    .createFullTextQuery( luceneQuery, Customer.class );

// or

fullTextQuery = fullTextSession
    .createFullTextQuery( luceneQuery, Item.class, Actor.class );
```

The first part of the second example only returns the matching **Customers**. The second part of the same example returns matching **Actors** and **Items**. The type restriction is polymorphic. As a result, if the two subclasses **Salesman** and **Customer** of the base class **Person** return, specify **Person.class** to filter based on result types.

[Report a bug](#)

14.3.1.10.2. Pagination

To avoid performance degradation, it is recommended to restrict the number of returned objects per query. A user navigating from one page to another page is a very common use case. The way to define pagination is similar to defining pagination in a plain HQL or Criteria query.

Example 14.39. Defining pagination for a search query

```
org.hibernate.Query fullTextQuery =
    fullTextSession.createFullTextQuery( luceneQuery, Customer.class );
fullTextQuery.setFirstResult(15); //start from the 15th element
fullTextQuery.setMaxResults(10); //return 10 elements
```



NOTE

It is still possible to get the total number of matching elements regardless of the pagination via **fullTextQuery.getResultSize()**

[Report a bug](#)

14.3.1.10.3. Sorting

Apache Lucene contains a flexible and powerful result sorting mechanism. The default sorting is by relevance and is appropriate for a large variety of use cases. The sorting mechanism can be changed to sort by other properties using the Lucene Sort object to apply a Lucene sorting strategy.

Example 14.40. Specifying a Lucene Sort

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery( query,
    Book.class );
org.apache.lucene.search.Sort sort = new Sort(
    new SortField("title", SortField.STRING));
query.setSort(sort);
List results = query.list();
```

**NOTE**

Fields used for sorting must not be tokenized. For more information about tokenizing, see [Section 14.2.1.1.2, “@Field”](#).

[Report a bug](#)

14.3.1.10.4. Fetching Strategy

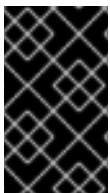
Hibernate Search loads objects using a single query if the return types are restricted to one class. Hibernate Search is restricted by the static fetching strategy defined in the domain model. It is useful to refine the fetching strategy for a specific use case as follows:

Example 14.41. Specifying FetchMode on a query

```
Criteria criteria =
    s.createCriteria( Book.class ).setFetchMode( "authors",
    FetchMode.JOIN );
s.createFullTextQuery( luceneQuery ).setCriteriaQuery( criteria );
```

In this example, the query will return all Books matching the LuceneQuery. The authors collection will be loaded from the same query using an SQL outer join.

In a criteria query definition, the type is guessed based on the provided criteria query. As a result, it is not necessary to restrict the return entity types.

**IMPORTANT**

The fetch mode is the only adjustable property. Do not use a restriction (a where clause) on the **Criteria** query because the **getResultSize()** throws a **SearchException** if used in conjunction with a **Criteria** with restriction.

If more than one entity is expected, do not use **setCriteriaQuery**.

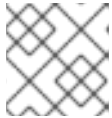
[Report a bug](#)

14.3.1.10.5. Projection

In some cases, only a small subset of the properties is required. Use Hibernate Search to return a subset of properties as follows:

Hibernate Search extracts properties from the Lucene index and converts them to their object representation and returns a list of **Object[]**. Projections prevent a time consuming database round-trip. However, they have following constraints:

- The properties projected must be stored in the index (**@Field(store=Store.YES)**), which increases the index size.
- The properties projected must use a **FieldBridge** implementing **org.hibernate.search.bridge.TwoWayFieldBridge** or **org.hibernate.search.bridge.TwoWayStringBridge**, the latter being the simpler version.



NOTE

All Hibernate Search built-in types are two-way.

- Only the simple properties of the indexed entity or its embedded associations can be projected. Therefore a whole embedded entity cannot be projected.
- Projection does not work on collections or maps which are indexed via **@IndexedEmbedded**

Lucene provides metadata information about query results. Use projection constants to retrieve the metadata.

Example 14.42. Using Projection to Retrieve Metadata

```
org.hibernate.search.FullTextQuery query =
    s.createFullTextQuery( luceneQuery, Book.class );
query.setProjection( FullTextQuery.SCORE, FullTextQuery.THIS,
    "mainAuthor.name" );
List results = query.list();
Object[] firstResult = (Object[]) results.get(0);
float score = firstResult[0];
Book book = firstResult[1];
String authorName = firstResult[2];
```

Fields can be mixed with the following projection constants:

- **FullTextQuery.THIS**: returns the initialized and managed entity (as a non projected query would have done).
- **FullTextQuery.DOCUMENT**: returns the Lucene Document related to the object projected.
- **FullTextQuery.OBJECT_CLASS**: returns the class of the indexed entity.
- **FullTextQuery.SCORE**: returns the document score in the query. Scores are handy to compare one result against an other for a given query but are useless when comparing the result of different queries.
- **FullTextQuery.ID**: the ID property value of the projected object.
- **FullTextQuery.DOCUMENT_ID**: the Lucene document ID. Be careful in using this value as a Lucene document ID can change over time between two different IndexReader opening.

- **FullTextQuery.EXPLANATION**: returns the Lucene Explanation object for the matching object/document in the given query. This is not suitable for retrieving large amounts of data. Running explanation typically is as costly as running the whole Lucene query per matching element. As a result, projection is recommended.

[Report a bug](#)

14.3.1.10.6. Customizing Object Initialization Strategies

By default, Hibernate Search uses the most appropriate strategy to initialize entities matching the full text query. It executes one (or several) queries to retrieve the required entities. This approach minimizes database trips where few of the retrieved entities are present in the persistence context (the session) or the second level cache.

If entities are present in the second level cache, force Hibernate Search to look into the cache before retrieving a database object.

Example 14.43. Check the second-level cache before using a query

```
FullTextQuery query = session.createFullTextQuery(luceneQuery,
User.class);
query.initializeObjectWith(
    ObjectLookupMethod.SECOND_LEVEL_CACHE,
    DatabaseRetrievalMethod.QUERY
);
```

ObjectLookupMethod defines the strategy to check if an object is easily accessible (without fetching it from the database). Other options are:

- **ObjectLookupMethod.PERSISTENCE_CONTEXT** is used if many matching entities are already loaded into the persistence context (loaded in the **Session** or **EntityManager**).
- **ObjectLookupMethod.SECOND_LEVEL_CACHE** checks the persistence context and then the second-level cache.

Set the following to search in the second-level cache:

- Correctly configure and activate the second-level cache.
- Enable the second-level cache for the relevant entity. This is done using annotations such as **@Cacheable**.
- Enable second-level cache read access for either **Session**, **EntityManager** or **Query**. Use **CacheMode.NORMAL** in Hibernate native APIs or **CacheRetrieveMode.USE** in Java Persistence APIs.

**WARNING**

Unless the second-level cache implementation is EHCache or Infinispan, do not use **ObjectLookupMethod.SECOND_LEVEL_CACHE**. Other second-level cache providers do not implement this operation efficiently.

Customize how objects are loaded from the database using **DatabaseRetrievalMethod** as follows:

- **QUERY** (default) uses a set of queries to load several objects in each batch. This approach is recommended.
- **FIND_BY_ID** loads one object at a time using the **Session.get** or **EntityManager.find** semantic. This is recommended if the batch size is set for the entity, which allows Hibernate Core to load entities in batches.

[Report a bug](#)

14.3.1.10.7. Limiting the Time of a Query

Limit the time a query takes in Hibernate Guide as follows:

- Raise an exception when arriving at the limit.
- Limit to the number of results retrieved when the time limit is raised.

[Report a bug](#)

14.3.1.10.8. Raise an Exception on Time Limit

If a query uses more than the defined amount of time, a **QueryTimeoutException** is raised (**org.hibernate.QueryTimeoutException** or **javax.persistence.QueryTimeoutException** depending on the programmatic API).

To define the limit when using the native Hibernate APIs, use one of the following approaches:

Example 14.44. Defining a Timeout in Query Execution

```
Query luceneQuery = ...;
FullTextQuery query = fullTextSession.createFullTextQuery(luceneQuery,
    User.class);

//define the timeout in seconds
query.setTimeout(5);

//alternatively, define the timeout in any given time unit
query.setTimeout(450, TimeUnit.MILLISECONDS);

try {
    query.list();
}
```

```

catch (org.hibernate.QueryTimeoutException e) {
    //do something, too slow
}

```

The **getResultSize()**, **iterate()** and **scroll()** honor the timeout until the end of the method call. As a result, **Iterable** or the **ScrollableResults** ignore the timeout. Additionally, **explain()** does not honor this timeout period. This method is used for debugging and to check the reasons for slow performance of a query.

The following is the standard way to limit execution time using the Java Persistence API (JPA):

Example 14.45. Defining a Timeout in Query Execution

```

Query luceneQuery = ...;
FullTextQuery query = fullTextEM.createFullTextQuery(luceneQuery,
User.class);

//define the timeout in milliseconds
query.setHint( "javax.persistence.query.timeout", 450 );

try {
    query.getResultList();
}
catch (javax.persistence.QueryTimeoutException e) {
    //do something, too slow
}

```



IMPORTANT

The example code does not guarantee that the query stops at the specified results amount.

[Report a bug](#)

14.3.2. Retrieving the Results

After building the Hibernate query, it is executed the same way as a HQL or Criteria query. The same paradigm and object semantic apply to a Lucene Query query and the common operations like: **list()**, **uniqueResult()**, **iterate()**, **scroll()** are available.

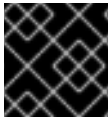
[Report a bug](#)

14.3.2.1. Performance Considerations

If you expect a reasonable number of results (for example using pagination) and expect to work on all of them, **list()** or **uniqueResult()** are recommended. **list()** work best if the entity **batch-size** is set up properly. Note that Hibernate Search has to process all Lucene Hits elements (within the pagination) when using **list()**, **uniqueResult()** and **iterate()**.

If you wish to minimize Lucene document loading, **scroll()** is more appropriate. Don't forget to close the **ScrollableResults** object when you're done, since it keeps Lucene resources. If you expect to

use **scroll**, but wish to load objects in batch, you can use **query.setFetchSize()**. When an object is accessed, and if not already loaded, Hibernate Search will load the next **fetchSize** objects in one pass.



IMPORTANT

Pagination is preferred over scrolling.

[Report a bug](#)

14.3.2.2. Result Size

It is sometimes useful to know the total number of matching documents:

- to provide a total search results feature, as provided by Google searches. For example, "1-10 of about 888,000,000 results"
- to implement a fast pagination navigation
- to implement a multi-step search engine that adds approximation if the restricted query returns zero or not enough results

Of course it would be too costly to retrieve all the matching documents. Hibernate Search allows you to retrieve the total number of matching documents regardless of the pagination parameters. Even more interesting, you can retrieve the number of matching elements without triggering a single object load.

Example 14.46. Determining the Result Size of a Query

```
org.hibernate.search.FullTextQuery query =
    s.createFullTextQuery( luceneQuery, Book.class );
//return the number of matching books without loading a single one
assert 3245 == query.getResultSize();

org.hibernate.search.FullTextQuery query =
    s.createFullTextQuery( luceneQuery, Book.class );
query.setMaxResult(10);
List results = query.list();
//return the total number of matching books regardless of pagination
assert 3245 == query.getResultSize();
```



NOTE

Like Google, the number of results is approximation if the index is not fully up-to-date with the database (asynchronous cluster for example).

[Report a bug](#)

14.3.2.3. ResultTransformer

Projection results are returned as **Object** arrays. If the data structure used for the object does not match the requirements of the application, apply a **ResultTransformer**. The **ResultTransformer** builds the required data structure after the query execution.

Example 14.47. Using ResultTransformer with Projections

```

org.hibernate.search.FullTextQuery query =
    s.createFullTextQuery( luceneQuery, Book.class );
query.setProjection( "title", "mainAuthor.name" );

query.setResultTransformer( new StaticAliasToBeanResultTransformer(
    BookView.class, "title", "author" ) );
List<BookView> results = (List<BookView>) query.list();
for(BookView view : results) {
    log.info( "Book: " + view.getTitle() + ", " + view.getAuthor() );
}

```

Examples of **ResultTransformer** implementations can be found in the Hibernate Core codebase.

[Report a bug](#)

14.3.2.4. Understanding Results

If the results of a query are not what you expected, the **Luke** tool is useful in understanding the outcome. However, Hibernate Search also gives you access to the Lucene **Explanation** object for a given result (in a given query). This class is considered fairly advanced to Lucene users but can provide a good understanding of the scoring of an object. You have two ways to access the Explanation object for a given result:

- Use the **fullTextQuery.explain(int)** method
- Use projection

The first approach takes a document ID as a parameter and return the Explanation object. The document ID can be retrieved using projection and the **FullTextQuery.DOCUMENT_ID** constant.

**WARNING**

The Document ID is unrelated to the entity ID. Be careful not to confuse these concepts.

In the second approach you project the **Explanation** object using the **FullTextQuery.EXPLANATION** constant.

Example 14.48. Retrieving the Lucene Explanation Object Using Projection

```

FullTextQuery ftQuery = s.createFullTextQuery( luceneQuery, Dvd.class )
    .setProjection(
        FullTextQuery.DOCUMENT_ID,
        FullTextQuery.EXPLANATION,
        FullTextQuery.THIS );
@SuppressWarnings("unchecked") List<Object[]> results = ftQuery.list();

```

```

for (Object[] result : results) {
    Explanation e = (Explanation) result[1];
    display( e.toString() );
}

```

Use the `Explanation` object only when required as it is roughly as expensive as running the Lucene query again.

[Report a bug](#)

14.3.3. Filters

Apache Lucene has a powerful feature that allows you to filter query results according to a custom filtering process. This is a very powerful way to apply additional data restrictions, especially since filters can be cached and reused. Use cases include:

- security
- temporal data (example, view only last month's data)
- population filter (example, search limited to a given category)

Hibernate Search pushes the concept further by introducing the notion of parameterizable named filters which are transparently cached. For people familiar with the notion of Hibernate Core filters, the API is very similar:

Example 14.49. Enabling Fulltext Filters for a Query

```

fullTextQuery = s.createFullTextQuery( query, Driver.class );
fullTextQuery.enableFullTextFilter("bestDriver");
fullTextQuery.enableFullTextFilter("security").setParameter( "login",
"andre" );
fullTextQuery.list(); //returns only best drivers where andre has
credentials

```

In this example we enabled two filters on top of the query. You can enable (or disable) as many filters as you like.

Declaring filters is done through the `@FullTextFilterDef` annotation. This annotation can be on any `@Indexed` entity regardless of the query the filter is later applied to. This implies that filter definitions are global and their names must be unique. A `SearchException` is thrown in case two different `@FullTextFilterDef` annotations with the same name are defined. Each named filter has to specify its actual filter implementation.

Example 14.50. Defining and Implementing a Filter

```

@FullTextFilterDefs( {
    @FullTextFilterDef(name = "bestDriver", impl =
BestDriversFilter.class),
    @FullTextFilterDef(name = "security", impl =

```

```
SecurityFilterFactory.class)
```

```
})
public class Driver { ... }
```

```
public class BestDriversFilter extends org.apache.lucene.search.Filter
{
    public DocIdSet getDocIdSet(IndexReader reader) throws IOException
    {
        OpenBitSet bitSet = new OpenBitSet( reader.maxDoc() );
        TermDocs termDocs = reader.termDocs( new Term( "score", "5" )
    );
        while ( termDocs.next() ) {
            bitSet.set( termDocs.doc() );
        }
        return bitSet;
    }
}
```

BestDriversFilter is an example of a simple Lucene filter which reduces the result set to drivers whose score is 5. In this example the specified filter implements the **org.apache.lucene.search.Filter** directly and contains a no-arg constructor.

If your Filter creation requires additional steps or if the filter you want to use does not have a no-arg constructor, you can use the factory pattern:

Example 14.51. Creating a filter using the factory pattern

```
@FullTextFilterDef(name = "bestDriver", impl =
BestDriversFilterFactory.class)
public class Driver { ... }

public class BestDriversFilterFactory {

    @Factory
    public Filter getFilter() {
        //some additional steps to cache the filter results per
        IndexReader
        Filter bestDriversFilter = new BestDriversFilter();
        return new CachingWrapperFilter(bestDriversFilter);
    }
}
```

Hibernate Search will look for a **@Factory** annotated method and use it to build the filter instance. The factory must have a no-arg constructor.

Named filters come in handy where parameters have to be passed to the filter. For example a security filter might want to know which security level you want to apply:

Example 14.52. Passing parameters to a defined filter

```
fullTextQuery = s.createFullTextQuery( query, Driver.class );
fullTextQuery.enableFullTextFilter("security").setParameter( "level", 5
);
```

Each parameter name should have an associated setter on either the filter or filter factory of the targeted named filter definition.

Example 14.53. Using parameters in the actual filter implementation

```
public class SecurityFilterFactory {
    private Integer level;

    /**
     * injected parameter
     */
    public void setLevel(Integer level) {
        this.level = level;
    }

    @Key public FilterKey getKey() {
        StandardFilterKey key = new StandardFilterKey();
        key.addParameter( level );
        return key;
    }

    @Factory
    public Filter getFilter() {
        Query query = new TermQuery( new Term("level", level.toString()
) );
        return new CachingWrapperFilter( new QueryWrapperFilter(query)
);
    }
}
```

Note the method annotated `@Key` returns a **FilterKey** object. The returned object has a special contract: the key object must implement `equals()` / `hashCode()` so that two keys are equal if and only if the given **Filter** types are the same and the set of parameters are the same. In other words, two filter keys are equal if and only if the filters from which the keys are generated can be interchanged. The key object is used as a key in the cache mechanism.

`@Key` methods are needed only if:

- the filter caching system is enabled (enabled by default)
- the filter has parameters

In most cases, using the **StandardFilterKey** implementation will be good enough. It delegates the `equals()` / `hashCode()` implementation to each of the parameters equals and hashCode methods.

As mentioned before the defined filters are per default cached and the cache uses a combination of hard and soft references to allow disposal of memory when needed. The hard reference cache keeps track of the most recently used filters and transforms the ones least used to **SoftReferences** when needed.

Once the limit of the hard reference cache is reached additional filters are cached as **SoftReferences**. To adjust the size of the hard reference cache, use **hibernate.search.filter.cache_strategy.size** (defaults to 128). For advanced use of filter caching, implement your own **FilterCachingStrategy**. The classname is defined by **hibernate.search.filter.cache_strategy**.

This filter caching mechanism should not be confused with caching the actual filter results. In Lucene it is common practice to wrap filters using the **IndexReader** around a **CachingWrapperFilter**. The wrapper will cache the **DocIdSet** returned from the **getDocIdSet(IndexReader reader)** method to avoid expensive recomputation. It is important to mention that the computed **DocIdSet** is only cachable for the same **IndexReader** instance, because the reader effectively represents the state of the index at the moment it was opened. The document list cannot change within an opened **IndexReader**. A different/new **IndexReader** instance, however, works potentially on a different set of **Documents** (either from a different index or simply because the index has changed), hence the cached **DocIdSet** has to be recomputed.

Hibernate Search also helps with this aspect of caching. Per default the **cache** flag of **@FullTextFilterDef** is set to **FilterCacheModeType.INSTANCE_AND_DOCIDSETRESULTS** which will automatically cache the filter instance as well as wrap the specified filter around a Hibernate specific implementation of **CachingWrapperFilter**. In contrast to Lucene's version of this class **SoftReferences** are used together with a hard reference count (see discussion about filter cache). The hard reference count can be adjusted using **hibernate.search.filter.cache_docidresults.size** (defaults to 5). The wrapping behaviour can be controlled using the **@FullTextFilterDef.cache** parameter. There are three different values for this parameter:

Value	Definition
<code>FilterCacheModeType.NONE</code>	No filter instance and no result is cached by Hibernate Search. For every filter call, a new filter instance is created. This setting might be useful for rapidly changing data sets or heavily memory constrained environments.
<code>FilterCacheModeType.INSTANCE_ONLY</code>	The filter instance is cached and reused across concurrent Filter.getDocIdSet() calls. DocIdSet results are not cached. This setting is useful when a filter uses its own specific caching mechanism or the filter results change dynamically due to application specific events making DocIdSet caching in both cases unnecessary.
<code>FilterCacheModeType.INSTANCE_AND_DOCIDSET RESULTS</code>	Both the filter instance and the DocIdSet results are cached. This is the default value.

Last but not least - why should filters be cached? There are two areas where filter caching shines:

Filters should be cached in the following situations:

- the system does not update the targeted entity index often (in other words, the **IndexReader** is reused a lot)
- the Filter's **DocIdSet** is expensive to compute (compared to the time spent to execute the query)

[Report a bug](#)

14.3.3.1. Using Filters in a Sharded Environment

In a sharded environment it is possible to execute queries on a subset of the available shards. This can be done in two steps:

Procedure 14.1. Query a Subset of Index Shards

1. Create a sharding strategy that does select a subset of **IndexManagers** depending on a filter configuration.
2. Activate the filter at query time.

Example 14.54. Query a Subset of Index Shards

In this example the query is run against a specific customer shard if the **customer** filter is activated.

```
public class CustomerShardingStrategy implements IndexShardingStrategy {

    // stored IndexManagers in a array indexed by customerID
    private IndexManager[] indexManagers;

    public void initialize(Properties properties, IndexManager[]
indexManagers) {
        this.indexManagers = indexManagers;
    }

    public IndexManager[] getIndexManagersForAllShards() {
        return indexManagers;
    }

    public IndexManager getIndexManagerForAddition(
        Class<?> entity, Serializable id, String idInString, Document
document) {
        Integer customerID =
Integer.parseInt(document.getFieldable("customerID").stringValue());
        return indexManagers[customerID];
    }

    public IndexManager[] getIndexManagersForDeletion(
        Class<?> entity, Serializable id, String idInString) {
        return getIndexManagersForAllShards();
    }

    /**
     * Optimization; don't search ALL shards and union the results; in
this case, we
     * can be certain that all the data for a particular customer Filter
is in a single
     * shard; simply return that shard by customerID.
     */
    public IndexManager[] getIndexManagersForQuery(
        FullTextFilterImplementor[] filters) {
        FullTextFilter filter = getCustomerFilter(filters, "customer");
```

```

        if (filter == null) {
            return getIndexManagersForAllShards();
        }
        else {
            return new IndexManager[] { indexManagers[Integer.parseInt(
                filter.getParameter("customerID").toString())] };
        }
    }

    private FullTextFilter getCustomerFilter(FullTextFilterImplementor[]
filters, String name) {
        for (FullTextFilterImplementor filter: filters) {
            if (filter.getName().equals(name)) return filter;
        }
        return null;
    }
}

```

In this example, if the filter named **customer** is present, only the shard dedicated to this customer is queried, otherwise, all shards are returned. A given Sharding strategy can react to one or more filters and depends on their parameters.

The second step is to activate the filter at query time. While the filter can be a regular filter (as defined in [Section 14.3.3, “Filters”](#)) which also filters Lucene results after the query, you can make use of a special filter that will only be passed to the sharding strategy (and is otherwise ignored).

To use this feature, specify the **ShardSensitiveOnlyFilter** class when declaring your filter.

```

@Indexed
@FullTextFilterDef(name="customer", impl=ShardSensitiveOnlyFilter.class)
public class Customer {
    ...
}

FullTextQuery query = ftEm.createFullTextQuery(luceneQuery,
Customer.class);
query.enableFulltextFilter("customer").setParameter("CustomerID", 5);
@SuppressWarnings("unchecked")
List<Customer> results = query.getResultList();

```

Note that by using the **ShardSensitiveOnlyFilter**, you do not have to implement any Lucene filter. Using filters and sharding strategy reacting to these filters is recommended to speed up queries in a sharded environment.

[Report a bug](#)


14.3.4. Faceting

Faceted search is a technique which allows the results of a query to be divided into multiple categories. This categorization includes the calculation of hit counts for each category and the ability to further restrict search results based on these facets (categories). [Example 14.55, “Search for Hibernate Search on Amazon”](#) shows a faceting example. The search results in fifteen hits which are displayed on the main part of the page. The navigation bar on the left, however, shows the category *Computers & Internet* with its subcategories *Programming*, *Computer Science*, *Databases*, *Software*, *Web Development*,

Networking and *Home Computing*. For each of these subcategories the number of books is shown matching the main search criteria and belonging to the respective subcategory. This division of the category *Computers & Internet* is one concrete search facet. Another one is for example the average customer review.

Example 14.55. Search for Hibernate Search on Amazon

In Hibernate Search, the classes **QueryBuilder** and **FullTextQuery** are the entry point into the faceting API. The former creates faceting requests and the latter accesses the **FacetManager**. The **FacetManager** applies faceting requests on a query and selects facets that are added to an existing query to refine search results. The examples use the entity **Cd** as shown in [Example 14.56](#), “Entity **Cd**”:

Shop All Departments  Search

Books Advanced Search Browse Subjects New Releases Bestsellers TI

Department
 < Any Department
 < Books
Computers & Internet
 Programming (14)
 Computer Science (4)
 Databases (2)
 Software (2)
 Web Development (2)
 Networking (1)
 Home Computing (1)

Format
☐ Paperback (15)

Author
 Any Author
 Joe Vitale (1)

Shipping Option [\(What's this?\)](#)
 Any Shipping Option
 Free Super Saver Shipping


Avg. Customer Review
 Any Avg. Customer Review
 ★★★★★ & Up (12)
 ★★★★★ & Up (14)
 ★★★★★ & Up (14)
 ★★★★★ & Up (15)


Condition
 Any Condition
 Used (15)
 New (14)

Books > Computers & Internet > "Hibernate Search"

Showing 1 - 12 of 15 Results

- 

Hibernate Search in Action I
 ★★★★★ (3 customer reviews)
Formats
Paperback
 Order in the next **2 hours** to get it by Monday, Apr 18. **\$49.95**
 Only 1 left in stock - order soon.
 Eligible for **FREE** Super Saver Shipping.
Excerpt - Page 1: "... breaking the sus...
Surprise me! See a random page in the book
- 

Spring Persistence with Hib
 (Nov 2, 2010)
 ★★★★★☆ (5 customer reviews)
Formats
Paperback
 Order in the next **19 hours** to get it by Monday, Apr 18. **\$44.95**
Kindle Edition
 Auto-delivered wirelessly
 Other Formats: [Paperback](#)
 Some formats eligible for **FREE** Super Saver Shipping.
Excerpt - Page 11: "... In Chapter 10, you'll be resolving these issues. **Hibernate-Search**...
Surprise me! See a random page in the book
- 

Lucene in Action, Second Ed
 Hatcher and Otis Gospodnetic (

Figure 14.1. Search for Hibernate Search on Amazon

Example 14.56. Entity Cd

```

@Indexed
public class Cd {

    private int id;

    @Fields( {
        @Field,
        @Field(name = "name_un_analyzed", analyze = Analyze.NO)
    })
    private String name;

```

```

@Field(analyze = Analyze.NO)
@NumericField
private int price;

Field(analyze = Analyze.NO)
@DateBridge(resolution = Resolution.YEAR)
private Date releaseYear;

@Field(analyze = Analyze.NO)
private String label;

// setter/getter
...

```

[Report a bug](#)

14.3.4.1. Creating a Faceting Request

The first step towards a faceted search is to create the **FacetingRequest**. Currently two types of faceting requests are supported. The first type is called *discrete faceting* and the second type *range faceting* request. In the case of a discrete faceting request you specify on which index field you want to facet (categorize) and which faceting options to apply. An example for a discrete faceting request can be seen in [Example 14.57, “Creating a discrete faceting request”](#):

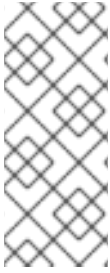
Example 14.57. Creating a discrete faceting request

```

QueryBuilder builder = fullTextSession.getSearchFactory()
    .buildQueryBuilder()
    .forEntity( Cd.class )
    .get();
FacetingRequest labelFacetingRequest = builder.facet()
    .name( "labelFaceting" )
    .onField( "label" )
    .discrete()
    .orderBy( FacetSortOrder.COUNT_DESC )
    .includeZeroCounts( false )
    .maxFacetCount( 1 )
    .createFacetingRequest();

```

When executing this faceting request a **Facet** instance will be created for each discrete value for the indexed field **label**. The **Facet** instance will record the actual field value including how often this particular field value occurs within the original query results. **orderBy**, **includeZeroCounts** and **maxFacetCount** are optional parameters which can be applied on any faceting request. **orderBy** allows to specify in which order the created facets will be returned. The default is **FacetSortOrder.COUNT_DESC**, but you can also sort on the field value or the order in which ranges were specified. **includeZeroCount** determines whether facets with a count of 0 will be included in the result (per default they are) and **maxFacetCount** allows to limit the maximum amount of facets returned.



NOTE

At the moment there are several preconditions an indexed field has to meet in order to apply faceting on it. The indexed property must be of type **String**, **Date** or a subtype of **Number** and **null** values should be avoided. Furthermore the property has to be indexed with **Analyze.NO** and in case of a numeric property **@NumericField** needs to be specified.

The creation of a range faceting request is quite similar except that we have to specify ranges for the field values we are faceting on. A range faceting request can be seen in [Example 14.58, “Creating a range faceting request”](#) where three different price ranges are specified. **below** and **above** can only be specified once, but you can specify as many **from - to** ranges as you want. For each range boundary you can also specify via **excludeLimit** whether it is included into the range or not.

Example 14.58. Creating a range faceting request

```
QueryBuilder builder = fullTextSession.getSearchFactory()
    .buildQueryBuilder()
    .forEntity( Cd.class )
    .get();
FacetingRequest priceFacetingRequest = builder.facet()
    .name( "priceFaceting" )
    .onField( "price" )
    .range()
    .below( 1000 )
    .from( 1001 ).to( 1500 )
    .above( 1500 ).excludeLimit()
    .createFacetingRequest();
```

[Report a bug](#)

14.3.4.2. Applying a Faceting Request

A faceting request is applied to a query via the **FacetManager** class which can be retrieved via the **FullTextQuery** class.

You can enable as many faceting requests as you like and retrieve them afterwards via **getFacets()** specifying the faceting request name. There is also a **disableFaceting()** method which allows you to disable a faceting request by specifying its name.

Example 14.59. Applying a faceting request

```
// create a fulltext query
Query luceneQuery = builder.all().createQuery(); // match all query
FullTextQuery fullTextQuery = fullTextSession.createFullTextQuery(
    luceneQuery, Cd.class );

// retrieve facet manager and apply faceting request
FacetManager facetManager = fullTextQuery.getFacetManager();
facetManager.enableFaceting( priceFacetingRequest );

// get the list of Cds
```

```

List<Cd> cds = fullTextQuery.list();
...

// retrieve the faceting results
List<Facet> facets = facetManager.getFacets( "priceFaceting" );
...

```

[Report a bug](#)

14.3.4.3. Restricting Query Results

Last but not least, you can apply any of the returned **Facets** as additional criteria on your original query in order to implement a "drill-down" functionality. For this purpose **FacetSelection** can be utilized. **FacetSelections** are available via the **FacetManager** and allow you to select a facet as query criteria (**selectFacets**), remove a facet restriction (**deselectFacets**), remove all facet restrictions (**clearSelectedFacets**) and retrieve all currently selected facets (**getSelectedFacets**).

[Example 14.60](#), "Restricting query results via the application of a **FacetSelection**" shows an example.

Example 14.60. Restricting query results via the application of a **FacetSelection**

```

// create a fulltext query
Query luceneQuery = builder.all().createQuery(); // match all query
FullTextQuery fullTextQuery = fullTextSession.createFullTextQuery(
    luceneQuery, clazz );

// retrieve facet manager and apply faceting request
FacetManager facetManager = fullTextQuery.getFacetManager();
facetManager.enableFaceting( priceFacetingRequest );

// get the list of Cd
List<Cd> cds = fullTextQuery.list();
assertTrue(cds.size() == 10);

// retrieve the faceting results
List<Facet> facets = facetManager.getFacets( "priceFaceting" );
assertTrue(facets.get(0).getCount() == 2)

// apply first facet as additional search criteria
facetManager.getFacetGroup( "priceFaceting" ).selectFacets( facets.get(
    0 ) );

// re-execute the query
cds = fullTextQuery.list();
assertTrue(cds.size() == 2);

```

[Report a bug](#)

14.3.5. Optimizing the Query Process

Query performance depends on several criteria:

- The Lucene query.

- The number of objects loaded: use pagination (always) or index projection (if needed).
- The way Hibernate Search interacts with the Lucene readers: defines the appropriate reader strategy.
- Caching frequently extracted values from the index: see [Section 14.3.5.1, “Caching Index Values: FieldCache”](#)

[Report a bug](#)

14.3.5.1. Caching Index Values: FieldCache

The primary function of a Lucene index is to identify matches to your queries. After the query is performed the results must be analyzed to extract useful information. Hibernate Search would typically need to extract the Class type and the primary key.

Extracting the needed values from the index has a performance cost, which in some cases might be very low and not noticeable, but in some other cases might be a good candidate for caching.

The requirements depend on the kind of Projections being used (see [Section 14.3.1.10.5, “Projection”](#)), as in some cases the Class type is not needed as it can be inferred from the query context or other means.

Using the `@CacheFromIndex` annotation you can experiment with different kinds of caching of the main metadata fields required by Hibernate Search:

```
import static org.hibernate.search.annotations.FieldCacheType.CLASS;
import static org.hibernate.search.annotations.FieldCacheType.ID;

@Indexed
@CacheFromIndex( { CLASS, ID } )
public class Essay {
    ...
}
```

It is possible to cache Class types and IDs using this annotation:

- **CLASS:** Hibernate Search will use a Lucene FieldCache to improve performance of the Class type extraction from the index.

This value is enabled by default, and is what Hibernate Search will apply if you don't specify the `@CacheFromIndex` annotation.

- **ID:** Extracting the primary identifier will use a cache. This is likely providing the best performing queries, but will consume much more memory which in turn might reduce performance.



NOTE

Measure the performance and memory consumption impact after warmup (executing some queries). Performance may improve by enabling Field Caches but this is not always the case.

Using a FieldCache has two downsides to consider:

- **Memory usage:** these caches can be quite memory hungry. Typically the CLASS cache has lower requirements than the ID cache.

- Index warmup: when using field caches, the first query on a new index or segment will be slower than when you don't have caching enabled.

With some queries the classtype won't be needed at all, in that case even if you enabled the **CLASS** field cache, this might not be used; for example if you are targeting a single class, obviously all returned values will be of that type (this is evaluated at each Query execution).

For the ID FieldCache to be used, the ids of targeted entities must be using a **TwoWayFieldBridge** (as all building bridges), and all types being loaded in a specific query must use the fieldname for the id, and have ids of the same type (this is evaluated at each Query execution).

[Report a bug](#)

14.4. MANUAL INDEX CHANGES

As Hibernate Core applies changes to the database, Hibernate Search detects these changes and will update the index automatically (unless the EventListeners are disabled). Sometimes changes are made to the database without using Hibernate, as when backup is restored or your data is otherwise affected. In these cases Hibernate Search exposes the Manual Index APIs to explicitly update or remove a single entity from the index, rebuild the index for the whole database, or remove all references to a specific type.

All these methods affect the Lucene Index only, no changes are applied to the database.

[Report a bug](#)

14.4.1. Adding Instances to the Index

Using **FullTextSession.index(T entity)** you can directly add or update a specific object instance to the index. If this entity was already indexed, then the index will be updated. Changes to the index are only applied at transaction commit.

Example 14.61. Indexing an entity via FullTextSession.index(T entity)

```
FullTextSession fullTextSession = Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
Object customer = fullTextSession.load( Customer.class, 8 );
fullTextSession.index(customer);
tx.commit(); //index only updated at commit time
```

In case you want to add all instances for a type, or for all indexed types, the recommended approach is to use a **MassIndexer**: see [Section 14.4.3.2, “Using a MassIndexer”](#) for more details.

[Report a bug](#)

14.4.2. Deleting Instances from the Index

It is equally possible to remove an entity or all entities of a given type from a Lucene index without the need to physically remove them from the database. This operation is named purging and is also done through the **FullTextSession**.

Example 14.62. Purging a specific instance of an entity from the index

```

FullTextSession fullTextSession = Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
for (Customer customer : customers) {
    fullTextSession.purge( Customer.class, customer.getId() );
}
tx.commit(); //index is updated at commit time

```

Purging will remove the entity with the given id from the Lucene index but will not touch the database.

If you need to remove all entities of a given type, you can use the **purgeAll** method. This operation removes all entities of the type passed as a parameter as well as all its subtypes.

Example 14.63. Purging all instances of an entity from the index

```

FullTextSession fullTextSession = Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
fullTextSession.purgeAll( Customer.class );
//optionally optimize the index
//fullTextSession.getSearchFactory().optimize( Customer.class );
tx.commit(); //index changes are applied at commit time

```

It is recommended to optimize the index after such an operation.



NOTE

Methods **index**, **purge**, and **purgeAll** are available on **FullTextEntityManager** as well.



NOTE

All manual indexing methods (**index**, **purge**, and **purgeAll**) only affect the index, not the database, nevertheless they are transactional and as such they won't be applied until the transaction is successfully committed, or you make use of **flushToIndexes**.

[Report a bug](#)

14.4.3. Rebuilding the Index

If you change the entity mapping to the index, chances are that the whole Index needs to be updated; For example if you decide to index a an existing field using a different analyzer you'll need to rebuild the index for affected types. Also if the Database is replaced (like restored from a backup, imported from a legacy system) you'll want to be able to rebuild the index from existing data. Hibernate Search provides two main strategies to choose from:

- Using **FullTextSession.flushToIndexes()** periodically, while using **FullTextSession.index()** on all entities.
- Use a **MassIndexer**.

[Report a bug](#)

14.4.3.1. Using flushToIndexes()

This strategy consists of removing the existing index and then adding all entities back to the index using **FullTextSession.purgeAll()** and **FullTextSession.index()**, however there are some memory and efficiency constraints. For maximum efficiency Hibernate Search batches index operations and executes them at commit time. If you expect to index a lot of data you need to be careful about memory consumption since all documents are kept in a queue until the transaction commit. You can potentially face an **OutOfMemoryException** if you don't empty the queue periodically; to do this use **fullTextSession.flushToIndexes()**. Every time **fullTextSession.flushToIndexes()** is called (or if the transaction is committed), the batch queue is processed, applying all index changes. Be aware that, once flushed, the changes cannot be rolled back.

Example 14.64. Index rebuilding using index() and flushToIndexes()

```
fullTextSession.setFlushMode(FlushMode.MANUAL);
fullTextSession.setCacheMode(CacheMode.IGNORE);
transaction = fullTextSession.beginTransaction();
//Scrollable results will avoid loading too many objects in memory
ScrollableResults results = fullTextSession.createCriteria( Email.class
)
    .setFetchSize(BATCH_SIZE)
    .scroll( ScrollMode.FORWARD_ONLY );
int index = 0;
while( results.next() ) {
    index++;
    fullTextSession.index( results.get(0) ); //index each element
    if (index % BATCH_SIZE == 0) {
        fullTextSession.flushToIndexes(); //apply changes to indexes
        fullTextSession.clear(); //free memory since the queue is
        processed
    }
}
transaction.commit();
```



NOTE

hibernate.search.default.worker.batch_size has been deprecated in favor of this explicit API which provides better control

Try to use a batch size that guarantees that your application will not be out of memory: with a bigger batch size objects are fetched faster from database but more memory is needed.

[Report a bug](#)

14.4.3.2. Using a MassIndexer

Hibernate Search's **MassIndexer** uses several parallel threads to rebuild the index. You can optionally select which entities need to be reloaded or have it reindex all entities. This approach is optimized for best performance but requires to set the application in maintenance mode. Querying the index is not recommended when a **MassIndexer** is busy.

Example 14.65. Rebuild the Index Using a MassIndexer

```
fullTextSession.createIndexer().startAndWait();
```

This will rebuild the index, deleting it and then reloading all entities from the database. Although it is simple to use, some tweaking is recommended to speed up the process.



WARNING

During the progress of a `MassIndexer` the content of the index is undefined! If a query is performed while the `MassIndexer` is working most likely some results will be missing.

Example 14.66. Using a Tuned `MassIndexer`

```
fullTextSession
    .createIndexer( User.class )
    .batchSizeToLoadObjects( 25 )
    .cacheMode( CacheMode.NORMAL )
    .threadsToLoadObjects( 12 )
    .idFetchSize( 150 )
    .progressMonitor( monitor ) //a MassIndexerProgressMonitor
    implementation
    .startAndWait();
```

This will rebuild the index of all `User` instances (and subtypes), and will create 12 parallel threads to load the `User` instances using batches of 25 objects per query. These same 12 threads will also need to process indexed embedded relations and custom **FieldBridges** or **ClassBridges** to output a Lucene document. The threads trigger lazyloading of additional attributes during the conversion process. Because of this, a high number of threads working in parallel is required. The number of threads working on actual index writing is defined by the backend configuration of each index.

It is recommended to leave `cacheMode` to **CacheMode.IGNORE** (the default), as in most reindexing situations the cache will be a useless additional overhead. It might be useful to enable some other **CacheMode** depending on your data as it could increase performance if the main entity is relating to enum-like data included in the index.



NOTE

The ideal of number of threads to achieve best performance is highly dependent on your overall architecture, database design and data values. All internal thread groups have meaningful names so they should be easily identified with most diagnostic tools, including thread dumps.



NOTE

The `MassIndexer` is unaware of transactions, therefore there is no need to begin one or commit afterward. Because it is not transactional it is not recommended to let users use the system during its processing, as it is unlikely people will be able to find results and the system load might be too high anyway.

Other parameters which affect indexing time and memory consumption are:

- `hibernate.search.[default|<indexname>].exclusive_index_use`
- `hibernate.search.[default|<indexname>].indexwriter.max_buffered_docs`
- `hibernate.search.[default|<indexname>].indexwriter.max_merge_docs`
- `hibernate.search.[default|<indexname>].indexwriter.merge_factor`
- `hibernate.search.[default|<indexname>].indexwriter.merge_min_size`
- `hibernate.search.[default|<indexname>].indexwriter.merge_max_size`
- `hibernate.search.[default|<indexname>].indexwriter.merge_max_optimize_size`
- `hibernate.search.[default|<indexname>].indexwriter.merge_calibrate_by_deletes`
- `hibernate.search.[default|<indexname>].indexwriter.ram_buffer_size`
- `hibernate.search.[default|<indexname>].indexwriter.term_index_interval`

Previous versions also had a `max_field_length` but this was removed from Lucene, it's possible to obtain a similar effect by using a `LimitTokenCountAnalyzer`.

All `.indexwriter` parameters are Lucene specific and Hibernate Search passes these parameters through.

The `MassIndexer` uses a forward only scrollable result to iterate on the primary keys to be loaded, but MySQL's JDBC driver will load all values in memory. To avoid this "optimization" set `idFetchSize` to `Integer.MIN_VALUE`.

[Report a bug](#)

14.5. INDEX OPTIMIZATION

From time to time, the Lucene index needs to be optimized. The process is essentially a defragmentation. Until an optimization is triggered Lucene only marks deleted documents as such, no physical are applied. During the optimization process the deletions will be applied which also affects the number of files in the Lucene Directory.

Optimizing the Lucene index speeds up searches but has no effect on the indexation (update) performance. During an optimization, searches can be performed, but will most likely be slowed down. All index updates will be stopped. It is recommended to schedule optimization:

- On an idle system or when searches are least frequent.

- After a large number of index modifications are applied.

MassIndexer (see [Section 14.4.3.2, “Using a MassIndexer”](#)) optimizes indexes by default at the start and at the end of processing. Use **MassIndexer.optimizeAfterPurge** and **MassIndexer.optimizeOnFinish** to change this default behavior.

[Report a bug](#)

14.5.1. Automatic Optimization

Hibernate Search can automatically optimize an index after either:

- a certain amount of operations (insertion or deletion).
- a certain amount of transactions.

The configuration for automatic index optimization can be defined either globally or per index:

Example 14.67. Defining automatic optimization parameters

```
hibernate.search.default.optimizer.operation_limit.max = 1000
hibernate.search.default.optimizer.transaction_limit.max = 100
hibernate.search.Animal.optimizer.transaction_limit.max = 50
```

An optimization will be triggered to the **Animal** index as soon as either:

- the number of additions and deletions reaches **1000**.
- the number of transactions reaches **50**
(**hibernate.search.Animal.optimizer.transaction_limit.max** has priority over **hibernate.search.default.optimizer.transaction_limit.max**)

If none of these parameters are defined, no optimization is processed automatically.

The default implementation of **OptimizerStrategy** can be overridden by implementing **org.hibernate.search.store.optimization.OptimizerStrategy** and setting the **optimizer.implementation** property to the fully qualified name of your implementation. This implementation must implement the interface, be a public class and have a public constructor taking no arguments.

Example 14.68. Loading a custom OptimizerStrategy

```
hibernate.search.default.optimizer.implementation =
com.acme.worlddomination.SmartOptimizer
hibernate.search.default.optimizer.SomeOption = CustomConfigurationValue
hibernate.search.humans.optimizer.implementation = default
```

The keyword **default** can be used to select the Hibernate Search default implementation; all properties after the **.optimizer** key separator will be passed to the implementation's **initialize** method at start.

[Report a bug](#)

14.5.2. Manual Optimization

You can programmatically optimize (defragment) a Lucene index from Hibernate Search through the **SearchFactory**:

Example 14.69. Programmatic Index Optimization

```
FullTextSession fullTextSession =
Search.getFullTextSession(regularSession);
SearchFactory searchFactory = fullTextSession.getSearchFactory();

searchFactory.optimize(Order.class);
// or
searchFactory.optimize();
```

The first example optimizes the Lucene index holding **Orders** and the second optimizes all indexes.



NOTE

searchFactory.optimize() has no effect on a JMS backend. You must apply the optimize operation on the Master node.

[Report a bug](#)

14.5.3. Adjusting Optimization

Apache Lucene has a few parameters to influence how optimization is performed. Hibernate Search exposes those parameters.

Further index optimization parameters include:

- `hibernate.search.[default|<indexname>].indexwriter.max_buffered_docs`
- `hibernate.search.[default|<indexname>].indexwriter.max_merge_docs`
- `hibernate.search.[default|<indexname>].indexwriter.merge_factor`
- `hibernate.search.[default|<indexname>].indexwriter.ram_buffer_size`
- `hibernate.search.[default|<indexname>].indexwriter.term_index_interval`

[Report a bug](#)

14.6. ADVANCED FEATURES

14.6.1. Accessing the SearchFactory

The **SearchFactory** object keeps track of the underlying Lucene resources for Hibernate Search. It is a convenient way to access Lucene natively. The **SearchFactory** can be accessed from a **FullTextSession**:

Example 14.70. Accessing the SearchFactory

```
FullTextSession fullTextSession =
Search.getFullTextSession(regularSession);
SearchFactory searchFactory = fullTextSession.getSearchFactory();
```

[Report a bug](#)

14.6.2. Using an IndexReader

Queries in Lucene are executed on an **IndexReader**. Hibernate Search might cache index readers to maximize performance, or provide other efficient strategies to retrieve an updated **IndexReader** minimizing I/O operations. Your code can access these cached resources, but there are several requirements.

Example 14.71. Accessing an IndexReader

```
IndexReader reader =
searchFactory.getIndexReaderAccessor().open(Order.class);
try {
    //perform read-only operations on the reader
}
finally {
    searchFactory.getIndexReaderAccessor().close(reader);
}
```

In this example the **SearchFactory** determines which indexes are needed to query this entity (considering a Sharding strategy). Using the configured **ReaderProvider** on each index, it returns a compound **IndexReader** on top of all involved indexes. Because this **IndexReader** is shared amongst several clients, you must adhere to the following rules:

- Never call `indexReader.close()`, instead use `readerProvider.closeReader(reader)` when necessary, preferably in a finally block.
- Don not use this **IndexReader** for modification operations (it is a readonly **IndexReader**, and any such attempt will result in an exception).

Aside from those rules, you can use the **IndexReader** freely, especially to do native Lucene queries. Using the shared **IndexReaders** will make most queries more efficient than by opening one directly from, for example, the filesystem.

As an alternative to the method `open(Class... types)` you can use `open(String... indexNames)`, allowing you to pass in one or more index names. Using this strategy you can also select a subset of the indexes for any indexed type if sharding is used.

Example 14.72. Accessing an IndexReader by index names

```
IndexReader reader =
searchFactory.getIndexReaderAccessor().open("Products.1", "Products.3");
```


[Report a bug](#)

14.6.3. Accessing a Lucene Directory

A **Directory** is the most common abstraction used by Lucene to represent the index storage; Hibernate Search doesn't interact directly with a Lucene **Directory** but abstracts these interactions via an **IndexManager**: an index does not necessarily need to be implemented by a **Directory**.

If you know your index is represented as a **Directory** and need to access it, you can get a reference to the **Directory** via the **IndexManager**. Cast the **IndexManager** to a **DirectoryBasedIndexManager** and then use `getDirectoryProvider().getDirectory()` to get a reference to the underlying **Directory**. This is not recommended, we would encourage to use the **IndexReader** instead.

[Report a bug](#)

14.6.4. Sharding Indexes

In some cases it can be useful to split (shard) the indexed data of a given entity into several Lucene indexes.



WARNING

Sharding should only be implemented if the advantages outweigh the disadvantages. Searching sharded indexes will typically be slower as all shards have to be opened for a single search.

Possible use cases for sharding are:

- A single index is so large that index update times are slowing the application down.
- A typical search will only hit a subset of the index, such as when data is naturally segmented by customer, region or application.

By default sharding is not enabled unless the number of shards is configured. To do this use the `hibernate.search.<indexName>.sharding_strategy.nbr_of_shards` property.

Example 14.73. Enabling Index Sharding

In this example 5 shards are enabled.

```
hibernate.search.<indexName>.sharding_strategy.nbr_of_shards = 5
```

Responsible for splitting the data into sub-indexes is the **IndexShardingStrategy**. The default sharding strategy splits the data according to the hash value of the ID string representation (generated by the **FieldBridge**). This ensures a fairly balanced sharding. You can replace the default strategy by implementing a custom **IndexShardingStrategy**. To use your custom strategy you have to set the `hibernate.search.<indexName>.sharding_strategy` property.

Example 14.74. Specifying a Custom Sharding Strategy

```
hibernate.search.<indexName>.sharding_strategy =
my.shardingstrategy.Implementation
```

The **IndexShardingStrategy** property also allows for optimizing searches by selecting which shard to run the query against. By activating a filter a sharding strategy can select a subset of the shards used to answer a query (**IndexShardingStrategy.getIndexManagersForQuery**) and thus speed up the query execution.

Each shard has an independent **IndexManager** and so can be configured to use a different directory provider and back end configuration. The **IndexManager** index names for the **Animal** entity in [Example 14.75, “Sharding Configuration for Entity Animal”](#) are **Animal.0** to **Animal.4**. In other words, each shard has the name of its owning index followed by **.** (dot) and its index number.

Example 14.75. Sharding Configuration for Entity Animal

```
hibernate.search.default.indexBase = /usr/lucene/indexes
hibernate.search.Animal.sharding_strategy.nbr_of_shards = 5
hibernate.search.Animal.directory_provider = filesystem
hibernate.search.Animal.0.indexName = Animal00
hibernate.search.Animal.3.indexBase = /usr/lucene/sharded
hibernate.search.Animal.3.indexName = Animal03
```

In [Example 14.75, “Sharding Configuration for Entity Animal”](#), the configuration uses the default id string hashing strategy and shards the **Animal** index into 5 sub-indexes. All sub-indexes are filesystem instances and the directory where each sub-index is stored is as followed:

- for sub-index 0: **/usr/lucene/indexes/Animal00** (shared indexBase but overridden indexName)
- for sub-index 1: **/usr/lucene/indexes/Animal.1** (shared indexBase, default indexName)
- for sub-index 2: **/usr/lucene/indexes/Animal.2** (shared indexBase, default indexName)
- for sub-index 3: **/usr/lucene/sharded/Animal03** (overridden indexBase, overridden indexName)
- for sub-index 4: **/usr/lucene/indexes/Animal.4** (shared indexBase, default indexName)

When implementing a **IndexShardingStrategy** any field can be used to determine the sharding selection. Consider that to handle deletions, **purge** and **purgeAll** operations, the implementation might need to return one or more indexes without being able to read all the field values or the primary identifier. In that case the information is not enough to pick a single index, all indexes should be returned, so that the delete operation will be propagated to all indexes potentially containing the documents to be deleted.

[Report a bug](#)

14.6.5. Customizing Lucene's Scoring Formula

Lucene allows the user to customize its scoring formula by extending **org.apache.lucene.search.Similarity**. The abstract methods defined in this class match the factors of the following formula calculating the score of query q for document d :

$$\text{score}(q,d) = \text{coord}(q,d) \cdot \text{queryNorm}(q) \cdot \sum_{t \text{ in } q} (\text{tf}(t \text{ in } d) \cdot \text{idf}(t)^2 \cdot t.\text{getBoost}() \cdot \text{norm}(t,d))$$

Factor	Description
$\text{tf}(t \text{ in } d)$	Term frequency factor for the term (t) in the document (d).
$\text{idf}(t)$	Inverse document frequency of the term.
$\text{coord}(q,d)$	Score factor based on how many of the query terms are found in the specified document.
$\text{queryNorm}(q)$	Normalizing factor used to make scores between queries comparable.
$t.\text{getBoost}()$	Field boost.
$\text{norm}(t,d)$	Encapsulates a few (indexing time) boost and length factors.

It is beyond the scope of this manual to explain this formula in more detail. Please refer to **Similarity**'s Javadocs for more information.

Hibernate Search provides three ways to modify Lucene's similarity calculation.

First you can set the default similarity by specifying the fully specified classname of your **Similarity** implementation using the property **hibernate.search.similarity**. The default value is **org.apache.lucene.search.DefaultSimilarity**.

You can also override the similarity used for a specific index by setting the **similarity** property

```
hibernate.search.default.similarity = my.custom.Similarity
```

Finally you can override the default similarity on class level using the **@Similarity** annotation.

```
@Entity
@Indexed
@Similarity(impl = DummySimilarity.class)
public class Book {
    ...
}
```

As an example, let's assume it is not important how often a term appears in a document. Documents with a single occurrence of the term should be scored the same as documents with multiple occurrences. In this case your custom implementation of the method **tf(float freq)** should return 1.0.



WARNING

When two entities share the same index they must declare the same **Similarity** implementation. Classes in the same class hierarchy always share the index, so it's not allowed to override the **Similarity** implementation in a subtype.

Likewise, it does not make sense to define the similarity via the index setting and the class-level setting as they would conflict. Such a configuration will be rejected.

[Report a bug](#)

14.6.6. Exception Handling Configuration

Hibernate Search allows you to configure how exceptions are handled during the indexing process. If no configuration is provided then exceptions are logged to the log output by default. It is possible to explicitly declare the exception logging mechanism as follows:

```
hibernate.search.error_handler = log
```

The default exception handling occurs for both synchronous and asynchronous indexing. Hibernate Search provides an easy mechanism to override the default error handling implementation.

In order to provide your own implementation you must implement the **ErrorHandler** interface, which provides the **handle(ErrorContext context)** method. **ErrorContext** provides a reference to the primary **LuceneWork** instance, the underlying exception and any subsequent **LuceneWork** instances that could not be processed due to the primary exception.

```
public interface ErrorContext {
    List<LuceneWork> getFailingOperations();
    LuceneWork getOperationAtFault();
    Throwable getThrowable();
    boolean hasErrors();
}
```

To register this error handler with Hibernate Search you must declare the fully qualified classname of your **ErrorHandler** implementation in the configuration properties:

```
hibernate.search.error_handler = CustomerErrorHandler
```

[Report a bug](#)

14.6.7. Disable Hibernate Search

Hibernate Search can be partially or completely disabled as required. Hibernate Search's indexing can be disabled, for example, if the index is read-only, or you prefer to perform indexing manually, rather than automatically. It is also possible to completely disable Hibernate Search, preventing indexing and searching.

Disable Indexing

To disable Hibernate Search indexing, change the **indexing_strategy** configuration option to **manual**, then restart JBoss EAP.

```
hibernate.search.indexing_strategy = manual
```

Disable Hibernate Search Completely

To disable Hibernate Search completely, disable all listeners by changing the **autoregister_listeners** configuration option to **false**, then restart JBoss EAP.

```
hibernate.search.autoregister_listeners = false
```

[Report a bug](#)

CHAPTER 15. JAX-RS WEB SERVICES

15.1. ABOUT JAX-RS

JAX-RS is the Java API for RESTful web services. It provides support for building web services using REST, through the use of annotations. These annotations simplify the process of mapping Java objects to web resources. The specification is defined here: <http://www.jcp.org/en/jsr/detail?id=311>.

RESTEasy is the JBoss EAP 6 implementation of JAX-RS. It also provides additional features to the specification.

JBoss EAP 6 is compliant with JSR 311 - JAX-RS.

To get started with JAX-RS and JBoss EAP 6, refer to the **helloworld-rs**, **jax-rs-client**, and **kitchensink** quickstart: [Section 1.4.1.1, “Access the Quickstarts”](#).

[Report a bug](#)

15.2. ABOUT RESTEASY

RESTEasy is a portable implementation of the JAX-RS Java API. It also provides additional features, including a client side framework (the RESTEasy JAX-RS Client Framework) for mapping outgoing requests to remote servers, allowing JAX-RS to operate as a client or server-side specification.

[Report a bug](#)

15.3. ABOUT RESTFUL WEB SERVICES

RESTful web services are designed to expose APIs on the web. They aim to provide better performance, scalability, and flexibility than traditional web services by allowing clients to access data and resources using predictable URLs.

The Java Enterprise Edition 6 specification for RESTful services is JAX-RS. For more information about JAX-RS, refer to [Section 15.1, “About JAX-RS”](#) and [Section 15.2, “About RESTEasy”](#).

[Report a bug](#)

15.4. RESTEASY DEFINED ANNOTATIONS

Table 15.1. JAX-RS/RESTEasy Annotations

Annotation	Usage
ClientResponseType	This is an annotation that you can add to a RESTEasy client interface that has a return type of Response.
ContentEncoding	Meta annotation that specifies a Content-Encoding to be applied via the annotated annotation.
DecorateTypes	Must be placed on a DecoratorProcessor class to specify the supported types.

Annotation	Usage
Decorator	Meta-annotation to be placed on another annotation that triggers decoration.
Form	This can be used as a value object for incoming/outgoing request/responses.
StringParameterUnmarshallerBinder	Meta-annotation to be placed on another annotation that triggers a StringParameterUnmarshaller to be applied to a string based annotation injector.
Cache	Set response Cache-Control header automatically.
NoCache	Set Cache-Control response header of "nocache".
ServerCached	Specifies that the response to this jax-rs method should be cached on the server.
ClientInterceptor	Identifies an interceptor as a client-side interceptor.
DecoderPrecedence	This interceptor is an Content-Encoding decoder.
EncoderPrecedence	This interceptor is an Content-Encoding encoder.
HeaderDecoratorPrecedence	HeaderDecoratorPrecedence interceptors should always come first as they decorate a response (on the server), or an outgoing request (on the client) with special, user-defined, headers.
RedirectPrecedence	Should be placed on a PreProcessInterceptor.
SecurityPrecedence	Should be placed on a PreProcessInterceptor.
ServerInterceptor	Identifies an interceptor as a server-side interceptor.
NoJackson	Placed on class, parameter, field or method when you don't want the Jackson provider to be triggered.
ImageWriterParams	An annotation that a resource class can use to pass parameters to the IIOMImageProvider.
DoNotUseJAXBProvider	Put this on a class or parameter when you do not want the JAXB MessageBodyReader/Writer used but instead have a more specific provider you want to use to marshall the type.
Formatted	Format XML output with indentations and newlines. This is a JAXB Decorator.

Annotation	Usage
IgnoreMediaTypes	Placed on a type, method, parameter, or field to tell JAXRS not to use JAXB provider for a certain media type
Stylesheet	Specifies an XML stylesheet header.
Wrapped	Put this on a method or parameter when you want to marshal or unmarshal a collection or array of JAXB objects.
WrappedMap	Put this on a method or parameter when you want to marshal or unmarshal a map of JAXB objects.
XmlHeader	Sets an XML header for the returned document.
BadgerFish	A JSONConfig.
Mapped	A JSONConfig.
XmlNsMap	A JSONToXml.
MultipartForm	This can be used as a value object for incoming/outgoing request/responses of the multipart/form-data mime type.
PartType	Must be used in conjunction with Multipart providers when writing out a List or Map as a multipart/* type.
XopWithMultipartRelated	This annotation can be used to process/produce incoming/outgoing XOP messages (packaged as multipart/related) to/from JAXB annotated objects.
After	Used to add an expiration attribute when signing or as a stale check for verification.
Signed	Convenience annotation that triggers the signing of a request or response using the DOSETA specification.
Verify	Verification of input signature specified in a signature header.
Path	This must exist either in the class or resource method. If it exists in both, the relative path to the resource method is a concatenation of the class and method.

Annotation	Usage
PathParam	Allows you to map variable URI path fragments into a method call.
QueryParam	Allows you to map URI query string parameter or URL form encoded parameter to the method invocation.
CookieParam	Allows you to specify the value of a cookie or object representation of an HTTP request cookie into the method invocation.
DefaultValue	Can be combined with the other <code>@*Param</code> annotations to define a default value when the HTTP request item does not exist.
Context	Allows you to specify instances of <code>javax.ws.rs.core.HttpHeaders</code> , <code>javax.ws.rs.core.UriInfo</code> , <code>javax.ws.rs.core.Request</code> , <code>javax.servlet.HttpServletRequest</code> , <code>javax.servlet.HttpServletResponse</code> , and <code>javax.ws.rs.core.SecurityContext</code> objects.
Encoded	Can be used on a class, method, or param. By default, inject <code>@PathParam</code> and <code>@QueryParams</code> are decoded. By adding the <code>@Encoded</code> annotation, the value of these params are provided in encoded form.

[Report a bug](#)

15.5. RESTEasy CONFIGURATION

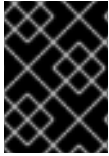
15.5.1. RESTEasy Configuration Parameters

Table 15.2. Elements

Option Name	Default Value	Description
<code>resteasy.servlet.mapping.prefix</code>	No default	If the url-pattern for the Resteasy servlet-mapping is not <code>/*</code> .
<code>resteasy.scan</code>	false	Automatically scan WEB-INF/lib jars and WEB-INF/classes directory for both <code>@Provider</code> and JAX-RS resource classes (<code>@Path</code> , <code>@GET</code> , <code>@POST</code> etc..) and register them.

Option Name	Default Value	Description
resteasy.scan.providers	false	Scan for @Provider classes and register them.
resteasy.scan.resources	false	Scan for JAX-RS resource classes.
resteasy.providers	no default	A comma delimited list of fully qualified @Provider class names you want to register.
resteasy.use.builtin.providers	true	Whether or not to register default, built-in @Provider classes.
resteasy.resources	No default	A comma delimited list of fully qualified JAX-RS resource class names you want to register.
resteasy.jndi.resources	No default	A comma delimited list of JNDI names which reference objects you want to register as JAX-RS resources.
javax.ws.rs.Application	No default	Fully qualified name of Application class to bootstrap in a spec portable way.
resteasy.media.type.mappings	No default	Replaces the need for an Accept header by mapping file name extensions (like .xml or .txt) to a media type. Used when the client is unable to use a Accept header to choose a representation (i.e. a browser).
resteasy.language.mappings	No default	Replaces the need for an Accept-Language header by mapping file name extensions (like .en or .fr) to a language. Used when the client is unable to use a Accept-Language header to choose a language (i.e. a browser).
resteasy.document.expand.entity.references	false	Whether to expand external entities or replace them with an empty string. In JBoss EAP 6, this parameter defaults to false , so it replaces them with an empty string.
resteasy.document.secure.processing.feature	true	Impose security constraints in processing org.w3c.dom.Document documents and JAXB object representations.

Option Name	Default Value	Description
resteasy.document.secure.disableDTDs	true	Prohibit DTDs in org.w3c.dom.Document documents and JAXB object representations.



IMPORTANT

In a Servlet 3.0 container, the **resteasy.scan.*** configurations in the **web.xml** file are ignored, and all JAX-RS annotated components will be automatically scanned.

[Report a bug](#)

15.6. JAX-RS WEB SERVICE SECURITY

15.6.1. Enable Role-Based Security for a RESTEasy JAX-RS Web Service

Summary

RESTEasy supports the `@RolesAllowed`, `@PermitAll`, and `@DenyAll` annotations on JAX-RS methods. However, it does not recognize these annotations by default. Follow these steps to configure the **web.xml** file and enable role-based security.



WARNING

Changing the default values of the following RESTEasy parameters may cause RESTEasy applications to be potentially vulnerable against XXE attacks.

- `resteasy.document.expand.entity.references`
- `resteasy.document.secure.processing.feature`
- `resteasy.document.secure.disableDTDs`

For more information about these parameters, see [Section 15.5.1, “RESTEasy Configuration Parameters”](#).



WARNING

Do not activate role-based security if the application uses EJBs. The EJB container will provide the functionality, instead of RESTEasy.

Procedure 15.1. Enable Role-Based Security for a RESTEasy JAX-RS Web Service

1. Open the **web.xml** file for the application in a text editor.
2. Add the following `<context-param>` to the file, within the **web-app** tags:

```
<context-param>
  <param-name>resteasy.role.based.security</param-name>
  <param-value>true</param-value>
</context-param>
```

3. Declare all roles used within the RESTEasy JAX-RS WAR file, using the `<security-role>` tags:

```
<security-role>
  <role-name>ROLE_NAME</role-name>
</security-role>
<security-role>
  <role-name>ROLE_NAME</role-name>
</security-role>
```

4. Authorize access to all URLs handled by the JAX-RS runtime for all roles:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Resteasy</web-resource-name>
    <url-pattern>/PATH</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>ROLE_NAME</role-name>
    <role-name>ROLE_NAME</role-name>
  </auth-constraint>
</security-constraint>
```

Result

Role-based security has been enabled within the application, with a set of defined roles.

Example 15.1. Example Role-Based Security Configuration

```
<web-app>

  <context-param>
    <param-name>resteasy.role.based.security</param-name>
    <param-value>true</param-value>
  </context-param>

  <servlet-mapping>
    <servlet-name>Resteasy</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Resteasy</web-resource-name>
      <url-pattern>/security</url-pattern>
```

```

</web-resource-collection>
<auth-constraint>
  <role-name>admin</role-name>
  <role-name>user</role-name>
</auth-constraint>
</security-constraint>

  <security-role>
<role-name>admin</role-name>
  </security-role>
  <security-role>
<role-name>user</role-name>
  </security-role>

</web-app>

```

[Report a bug](#)

15.6.2. Secure a JAX-RS Web Service using Annotations

Summary

This topic covers the steps to secure a JAX-RS web service using the supported security annotations

Procedure 15.2. Secure a JAX-RS Web Service using Supported Security Annotations

1. Enable role-based security. For more information, refer to: [Section 15.6.1, “Enable Role-Based Security for a RESTEasy JAX-RS Web Service”](#)
2. Add security annotations to the JAX-RS web service. RESTEasy supports the following annotations:

@RolesAllowed

Defines which roles can access the method. All roles should be defined in the **web.xml** file.

@PermitAll

Allows all roles defined in the **web.xml** file to access the method.

@DenyAll

Denies all access to the method.

[Report a bug](#)

15.7. EXCEPTION HANDLING

15.7.1. Create an Exception Mapper

Summary

Exception mappers are custom, application provided components that catch thrown exceptions and write specific HTTP responses.

Example 15.2. Exception Mapper

An exception mapper is a class that is annotated with the `@Provider` annotation, and implements the **ExceptionHandler** interface.

An example exception mapper is shown below.

```
@Provider
public class EJBExceptionHandler implements
ExceptionHandler<javax.ejb.EJBException>
{
    Response toResponse(EJBException exception) {
        return Response.status(500).build();
    }
}
```

To register an exception mapper, list it in the **web.xml** file under the **resteasy.providers** context-param, or register it programmatically through the **ResteasyProviderFactory** class.

[Report a bug](#)

15.7.2. RESTEasy Internally Thrown Exceptions

Table 15.3. Exception List

Exception	HTTP Code	Description
BadRequestException	400	Bad Request. The request was not formatted correctly, or there was a problem processing the request input.
UnauthorizedException	401	Unauthorized. Security exception thrown if you are using RESTEasy's annotation-based role-based security.
InternalServerErrorException	500	Internal Server Error.
MethodNotAllowedException	405	There is no JAX-RS method for the resource that can handle the invoked HTTP operation.
NotAcceptableException	406	There is no JAX-RS method that can produce the media types listed in the Accept header.
NotFoundException	404	There is no JAX-RS method that serves the request path/resource.

Exception	HTTP Code	Description
ReaderException	400	All exceptions thrown from MessageBodyReaders are wrapped within this exception. If there is no ExceptionHandler for the wrapped exception, or if the exception is not a WebApplicationException , then RESTEasy will return a 400 code by default.
WriterException	500	All exceptions thrown from MessageBodyWriters are wrapped within this exception. If there is no ExceptionHandler for the wrapped exception, or if the exception is not a WebApplicationException , then RESTEasy will return a 400 code by default.
JAXBUnmarshalException	400	The JAXB providers (XML and Jettison) throw this exception on reads. They may be wrapping JAXBExceptions. This class extends ReaderException .
JAXBMarshalException	500	The JAXB providers (XML and Jettison) throw this exception on writes. They may be wrapping JAXBExceptions. This class extends WriterException .
ApplicationException	N/A	Wraps all exceptions thrown from application code. It functions in the same way as InvocationTargetException . If there is an ExceptionHandler for wrapped exception, then that is used to handle the request.
Failure	N/A	Internal RESTEasy error. Not logged.
LoggableFailure	N/A	Internal RESTEasy error. Logged.

Exception	HTTP Code	Description
DefaultOptionsMethodException	N/A	If the user invokes HTTP OPTIONS and no JAX-RS method for it, RESTEasy provides a default behavior by throwing this exception.

[Report a bug](#)

15.8. RESTEASY INTERCEPTORS

15.8.1. Intercept JAX-RS Invocations

Summary

RESTEasy can intercept JAX-RS invocations and route them through listener-like objects called interceptors. This topic covers descriptions of the four types of interceptors.

Example 15.3. MessageBodyReader/Writer Interceptors

MessageBodyReaderInterceptors and MessageBodyWriterInterceptors can be used on either the server or client side. They are annotated with **@Provider**, as well as either **@ServerInterceptor** or **@ClientInterceptor** so that RESTEasy knows whether or not to add them to the interceptor list.

These interceptors wrap around the invocation of **MessageBodyReader.readFrom()** or **MessageBodyWriter.writeTo()**. They can be used to wrap the Output or Input streams.

RESTEasy GZIP support has interceptors that create and override the default Output and Input streams with a GzipOutputStream or GzipInputStream so that gzip encoding can work. They can also be used to append headers to the response, or the outgoing request on the client side.

```
public interface MessageBodyReaderInterceptor
{
    Object read(MessageBodyReaderContext context) throws IOException,
    WebApplicationException;
}

public interface MessageBodyWriterInterceptor
{
    void write(MessageBodyWriterContext context) throws IOException,
    WebApplicationException;
}
```

The interceptors and the MessageBodyReader or Writer is invoked in one big Java call stack. **MessageBodyReaderContext.proceed()** or **MessageBodyWriterContext.proceed()** is called in order to go to the next interceptor or, if there are no more interceptors to invoke, the **readFrom()** or **writeTo()** method of the MessageBodyReader or MessageBodyWriter. This wrapping allows objects to be modified before they get to the Reader or Writer, and then cleaned up after **proceed()** returns.

The example below is a server side interceptor, that adds a header value to the response.

```
@Provider
@ServerInterceptor
public class MyHeaderDecorator implements MessageBodyWriterInterceptor {

    public void write(MessageBodyWriterContext context) throws
IOException, WebApplicationException
    {
        context.getHeaders().add("My-Header", "custom");
        context.proceed();
    }
}
```

Example 15.4. PreProcessInterceptor

PreProcessInterceptors run after a JAX-RS resource method is found to invoke on, but before the actual invocation happens. They are annotated with `@ServerInterceptor`, and run in sequence.

These interfaces are only usable on the server. They can be used to implement security features, or to handle the Java request. The RESTEasy security implementation uses this type of interceptor to abort requests before they occur if the user does not pass authorization. The RESTEasy caching framework also uses this to return cached responses to avoid invoking methods again.

```
public interface PreProcessInterceptor
{
    ServerResponse preProcess(HttpRequest request, ResourceMethod
method) throws Failure, WebApplicationException;
}
```

If the **preProcess()** method returns a `ServerResponse` then the underlying JAX-RS method will not get invoked, and the runtime will process the response and return to the client. If the **preProcess()** method does not return a `ServerResponse`, the underlying JAX-RS method will be invoked.

Example 15.5. PostProcessInterceptors

PostProcessInterceptors run after the JAX-RS method was invoked, but before `MessageBodyWriters` are invoked. They are used if a response header needs to be set when a `MessageBodyWriter` may not be invoked.

They can only be used on the server side. They do not wrap anything, and are invoked in sequence.

```
public interface PostProcessInterceptor
{
    void postProcess(ServerResponse response);
}
```

Example 15.6. ClientExecutionInterceptors

ClientExecutionInterceptors are only usable on the client side. They wrap around the HTTP invocation

that goes to the server. They must be annotated with **@ClientInterceptor** and **@Provider**. These interceptors run after the `MessageBodyWriter`, and after the `ClientRequest` has been built on the client side.

RESTEasy GZIP support uses `ClientExecutionInterceptors` to set the `Accept` header to contain "gzip, deflate" before the request goes out. The RESTEasy client cache uses it to check to see if its cache contains the resource before going over the wire.

```
public interface ClientExecutionInterceptor
{
    ClientResponse execute(ClientExecutionContext ctx) throws Exception;
}

public interface ClientExecutionContext
{
    ClientRequest getRequest();

    ClientResponse proceed() throws Exception;
}
```

[Report a bug](#)

15.8.2. Bind an Interceptor to a JAX-RS Method

Summary

All registered interceptors are invoked for every request by default. The **AcceptedByMethod** interface can be implemented to fine tune this behavior.

Example 15.7. Binding Interceptors Example

RESTEasy will call the **accept()** method for interceptors that implement the **AcceptedByMethod** interface. If the method returns true, the interceptor will be added to the JAX-RS method's call chain; otherwise it will be ignored for that method.

In the example below, **accept()** determines if the `@GET` annotation is present on the JAX-RS method. If it is, the interceptor will be applied to the method's call chain.

```
@Provider
@ServerInterceptor
public class MyHeaderDecorator implements MessageBodyWriterInterceptor,
AcceptedByMethod {

    public boolean accept(Class declaring, Method method) {
        return method.isAnnotationPresent(GET.class);
    }

    public void write(MessageBodyWriterContext context) throws
IOException, WebApplicationException
    {
        context.getHeaders().add("My-Header", "custom");
        context.proceed();
    }
}
```

[Report a bug](#)

15.8.3. Register an Interceptor

Summary

This topic covers how to register a RESTEasy JAX-RS interceptor in an application.

Procedure 15.3. Register an Interceptor

- To register an interceptor, list it in the **web.xml** file under the **resteasy.providers** context-param, or return it as a class or as an object in the **Application.getClasses()** or **Application.getSingletons()** method.

Example 15.8. Registering an interceptor by listing it in the web.xml file:

```
<context-param>
  <param-name>resteasy.providers</param-name>
  <param-value>my.app.CustomInterceptor</paramvalue>
</context-param>
```

Example 15.9. Registering an interceptor using the Application.getClasses() method:

```
package org.jboss.resteasy.example;

import javax.ws.rs.core.Application;
import java.util.HashSet;
import java.util.Set;

public class MyApp extends Application {

    public java.util.Set<java.lang.Class<?>> getClassess() {
        Set<Class<?>> resources = new HashSet<Class<?>>();
        resources.add(MyResource.class);
        resources.add(MyProvider.class);
        return resources;
    }
}
```

Example 15.10. Registering an interceptor using the Application.getSingletons() method:

```
package org.jboss.resteasy.example;

import javax.ws.rs.core.Application;
import java.util.HashSet;
import java.util.Set;

public class MyApp extends Application {
```

```
protected Set<Object> singletons = new HashSet<Object>();

public PubSubApplication() {
    singletons.add(new MyResource());
    singletons.add(new MyProvider());
}

@Override
public Set<Object> getSingletons() {
    return singletons;
}
}
```

[Report a bug](#)

15.8.4. Interceptor Precedence Families

15.8.4.1. About Interceptor Precedence Families

Summary

Interceptors can be sensitive to the order they are invoked. RESTEasy groups interceptors in families to make ordering them simpler. This reference topic covers the built-in interceptor precedence families and the interceptors associated with each.

There are five predefined families. They are invoked in the following order:

SECURITY

SECURITY interceptors are usually `PreProcessInterceptors`. They are invoked first because as little as possible should be done before the invocation is authorized.

HEADER_DECORATOR

HEADER_DECORATOR interceptors add headers to a response or an outgoing request. They follow the security interceptors as the added headers may affect the behavior of other interceptor families.

ENCODER

ENCODER interceptors change the `OutputStream`. For example, the GZIP interceptor creates a `GZIPOutputStream` to wrap the real `OutputStream` for compression.

REDIRECT

REDIRECT interceptors are usually used in `PreProcessInterceptors`, as they may reroute the request and totally bypass the JAX-RS method.

DECODER

DECODER interceptors wrap the `InputStream`. For example, the GZIP interceptor decoder wraps the `InputStream` in a `GzipInputStream` instance.

For complete type safety, there are convenience annotations in the `org.jboss.resteasy.annotations.interception` package: `@DecoredPrecedence`, `@EncoderPrecedence`, `@HeaderDecoratorPrecedence`, `@RedirectPrecedence`,

@SecurityPrecedence. Use these instead of the **@Precedence** annotation. For more information, refer [Section 15.4, “RESTEasy Defined Annotations”](#).

[Report a bug](#)

15.8.4.2. Define a Custom Interceptor Precedence Family

Summary

Custom precedence families can be created and registered in the **web.xml** file. This topic covers examples of the context params available for defining interceptor precedence families.

There are three context params that can be used to define a new precedence family.

Example 15.11. `resteasy.append.interceptor.precedence`

The **`resteasy.append.interceptor.precedence`** context param appends the new precedence family to the default precedence family list.

```
<context-param>
  <param-name>resteasy.append.interceptor.precedence</param-name>
  <param-value>CUSTOM_PRECEDENCE_FAMILY</param-value>
</context-param>
```

Example 15.12. `resteasy.interceptor.before.precedence`

The **`resteasy.interceptor.before.precedence`** context param defines the default precedence family that the custom family is executed before. The parameter value takes the form *DEFAULT_PRECEDENCE_FAMILY/CUSTOM_PRECEDENCE_FAMILY*, delimited by a `'/'`.

```
<context-param>
  <param-name>resteasy.interceptor.before.precedence</param-name>
  <param-value>DEFAULT_PRECEDENCE_FAMILY :
CUSTOM_PRECEDENCE_FAMILY</param-value>
</context-param>
```

Example 15.13. `resteasy.interceptor.after.precedence`

The **`resteasy.interceptor.after.precedence`** context param defines the default precedence family that the custom family is executed after. The parameter value takes the form *DEFAULT_PRECEDENCE_FAMILY/CUSTOM_PRECEDENCE_FAMILY*, delimited by a `':'`.

```
<context-param>
  <param-name>resteasy.interceptor.after.precedence</param-name>
  <param-value>DEFAULT_PRECEDENCE_FAMILY :
CUSTOM_PRECEDENCE_FAMILY</param-value>
</context-param>
```

Precedence families are applied to interceptors using the **@Precedence** annotation. For the default precedence family list, refer to: [Section 15.8.4.1, “About Interceptor Precedence Families”](#).

[Report a bug](#)

15.9. STRING BASED ANNOTATIONS

15.9.1. Convert String Based @*Param Annotations to Objects

JAX-RS @***Param** annotations, including @QueryParam, @MatrixParam, @HeaderParam, @PathParam, and @FormParam, are represented as strings in a raw HTTP request. These types of injected parameters can be converted to objects if these objects have a `valueOf(String)` static method or a constructor that takes one String parameter.

RESTEasy provides two proprietary @**Provider** interfaces to handle this conversion for classes that don't have either a **valueOf(String)** static method, or a string constructor.

Example 15.14. StringConverter

The StringConverter interface is implemented to provide custom string marshalling. It is registered under the `resteasy.providers` context-param in the `web.xml` file. It can also be registered manually by calling the `ResteasyProviderFactory.addStringConverter()` method.

The example below is a simple example of using StringConverter.

```
import org.jboss.resteasy.client.ProxyFactory;
import org.jboss.resteasy.spi.StringConverter;
import org.jboss.resteasy.test.BaseResourceTest;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import javax.ws.rs.HeaderParam;
import javax.ws.rs.MatrixParam;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.QueryParam;
import javax.ws.rs.ext.Provider;

public class StringConverterTest extends BaseResourceTest
{
    public static class POJO
    {
        private String name;

        public String getName()
        {
            return name;
        }

        public void setName(String name)
        {
            this.name = name;
        }
    }

    @Provider
```

```

    public static class POJOConverter implements StringConverter<POJO>
    {
        public POJO fromString(String str)
        {
            System.out.println("FROM STRNG: " + str);
            POJO pojo = new POJO();
            pojo.setName(str);
            return pojo;
        }

        public String toString(POJO value)
        {
            return value.getName();
        }
    }

    @Path("/")
    public static class MyResource
    {
        @Path("{pojo}")
        @PUT
        public void put(@QueryParam("pojo")POJO q, @PathParam("pojo")POJO
pp,
            @MatrixParam("pojo")POJO mp, @HeaderParam("pojo")POJO hp)
        {
            Assert.assertEquals(q.getName(), "pojo");
            Assert.assertEquals(pp.getName(), "pojo");
            Assert.assertEquals(mp.getName(), "pojo");
            Assert.assertEquals(hp.getName(), "pojo");
        }
    }

    @Before
    public void setUp() throws Exception
    {
        dispatcher.getProviderFactory().addStringConverter(POJOConverter.class);
        dispatcher.getRegistry().addPerRequestResource(MyResource.class);
    }

    @Path("/")
    public static interface MyClient
    {
        @Path("{pojo}")
        @PUT
        void put(@QueryParam("pojo")POJO q, @PathParam("pojo")POJO pp,
            @MatrixParam("pojo")POJO mp, @HeaderParam("pojo")POJO hp);
    }

    @Test
    public void testIt() throws Exception
    {
        MyClient client = ProxyFactory.create(MyClient.class,
"http://localhost:8081");
        POJO pojo = new POJO();
        pojo.setName("pojo");

```

```

        client.put(pojo, pojo, pojo, pojo);
    }
}

```

Example 15.15. StringParameterUnmarshaller

The **StringParameterUnmarshaller** interface is sensitive to the annotations placed on the parameter or field you are injecting into. It is created per injector. The `setAnnotations()` method is called by `resteasy` to initialize the unmarshaller.

This interface can be added by creating and registering a provider that implements the interface. It can also be bound using a meta-annotation called

`org.jboss.resteasy.annotations.StringsParameterUnmarshallerBinder`.

The example below formats a **`java.util.Date`** based `@PathParam`.

```

public class StringParamUnmarshallerTest extends BaseResourceTest
{
    @Retention(RetentionPolicy.RUNTIME)
    @StringParameterUnmarshallerBinder(DateFormatter.class)
    public @interface DateFormat
    {
        String value();
    }

    public static class DateFormatter implements
StringParameterUnmarshaller<Date>
    {
        private SimpleDateFormat formatter;

        public void setAnnotations(Annotation[] annotations)
        {
            DateFormat format = FindAnnotation.findAnnotation(annotations,
DateFormat.class);
            formatter = new SimpleDateFormat(format.value());
        }

        public Date fromString(String str)
        {
            try
            {
                return formatter.parse(str);
            }
            catch (ParseException e)
            {
                throw new RuntimeException(e);
            }
        }
    }

    @Path("/datetest")
    public static class Service
    {
        @GET

```



```

        @Produces("text/plain")
        @Path("/{date}")
        public String get(@PathParam("date") @DateFormat("MM-dd-yyyy")
Date date)
        {
            System.out.println(date);
            Calendar c = Calendar.getInstance();
            c.setTime(date);
            Assert.assertEquals(3, c.get(Calendar.MONTH));
            Assert.assertEquals(23, c.get(Calendar.DAY_OF_MONTH));
            Assert.assertEquals(1977, c.get(Calendar.YEAR));
            return date.toString();
        }
    }

    @BeforeClass
    public static void setup() throws Exception
    {
        addPerRequestResource(Service.class);
    }

    @Test
    public void testMe() throws Exception
    {
        ClientRequest request = new
ClientRequest(generateURL("/datetest/04-23-1977"));
        System.out.println(request.getTarget(String.class));
    }
}

```

It defines a new annotation called `@DateFormat`. The annotation is annotated with the meta-annotation `StringParameterUnmarshallerBinder` with a reference to the `DateFormatter` classes.

The `Service.get()` method has a `@PathParam` parameter that is also annotated with `@DateFormat`. The application of `@DateFormat` triggers the binding of the `DateFormatter`. The `DateFormatter` will now be run to unmarshal the path parameter into the date parameter of the `get()` method.

[Report a bug](#)

15.10. CONFIGURE FILE EXTENSIONS

15.10.1. Map File Extensions to Media Types in the `web.xml` File

Summary

Some clients, like browsers, cannot use the `Accept` and `Accept-Language` headers to negotiate the representation's media type or language. `RESTEasy` can map file name suffixes to media types and languages to deal with this issue. Follow these steps to map media types to file extensions, in the `web.xml` file.

Procedure 15.4. Map Media Types to File Extensions

1. Open the `web.xml` file for the application in a text editor.

2. Add the context-param **resteasy.media.type.mappings** to the file, inside the **web-app** tags:

```
<context-param>
  <param-name>resteasy.media.type.mappings</param-name>
</context-param>
```

3. Configure the parameter values. The mappings form a comma delimited list. Each mapping is delimited by a ::

Example 15.16. Example Mapping

```
<context-param>
  <param-name>resteasy.media.type.mappings</param-name>
  <param-value>html : text/html, json : application/json, xml :
application/xml</param-value>
</context-param>
```

[Report a bug](#)

15.10.2. Map File Extensions to Languages in the web.xml File

Summary

Some clients, like browsers, cannot use the Accept and Accept-Language headers to negotiate the representation's media type or language. RESTEasy can map file name suffixes to media types and languages to deal with this issue. Follow these steps to map languages to file extensions, in the **web.xml** file.

Procedure 15.5. Map File Extensions to Languages in the web.xml File

1. Open the **web.xml** file for the application in a text editor.
2. Add the context-param **resteasy.language.mappings** to the file, inside the **web-app** tags:

```
<context-param>
  <param-name>resteasy.language.mappings</param-name>
</context-param>
```

3. Configure the parameter values. The mappings form a comma delimited list. Each mapping is delimited by a ::

Example 15.17. Example Mapping

```
<context-param>
  <param-name>resteasy.language.mappings</param-name>
  <param-value> en : en-US, es : es, fr : fr</param-name>
</context-param>
```

[Report a bug](#)

15.10.3. RESTEasy Supported Media Types

Table 15.4. Media Types

Media Type	Java Type
application/*+xml, text/*+xml, application/*+json, application/*+fastinfoset, application/atom+*	JaxB annotated classes
application/*+xml, text/*+xml	org.w3c.dom.Document
/	java.lang.String
/	java.io.InputStream
text/plain	primitives, java.lang.String, or any type that has a String constructor, or static valueOf(String) method for input, toString() for output
/	javax.activation.DataSource
/	byte[]
/	java.io.File
application/x-www-form-urlencoded	javax.ws.rs.core.MultivaluedMap

[Report a bug](#)

15.11. RESTEASY JAVASCRIPT API

15.11.1. About the RESTEasy JavaScript API

RESTEasy can generate a JavaScript API that uses AJAX calls to invoke JAX-RS operations. Each JAX-RS resource class will generate a JavaScript object of the same name as the declaring class or interface. The JavaScript object contains each JAX-RS method as properties.

Example 15.18. Simple JAX-RS JavaScript API Example

```
@Path("foo")
public class Foo{
    @Path("{id}")
    @GET
    public String get(@QueryParam("order") String order, @HeaderParam("X-Foo") String header,
                     @MatrixParam("colour") String colour,
    @CookieParam("Foo-Cookie") String cookie){
        &
    }
    @POST
```

```
public void post(String text){
}
}
```

We can use the previous JAX-RS API in JavaScript using the following code:

```
var text = Foo.get({order: 'desc', 'X-Foo': 'hello',
                  colour: 'blue', 'Foo-Cookie': 123987235444});
Foo.put({$entity: text});
```

Each JavaScript API method takes an optional object as single parameter where each property is a cookie, header, path, query or form parameter as identified by their name, or the API parameter properties. The properties are available here: [Section 15.11.3, “RESTEasy Javascript API Parameters”](#).

[Report a bug](#)

15.11.2. Enable the RESTEasy JavaScript API Servlet

Summary

The RESTEasy JavaScript API is not enabled by default. Follow these steps to enable it using the `web.xml` file.

Procedure 15.6. Edit `web.xml` to enable RESTEasy JavaScript API

1. Open the `web.xml` file of the application in a text editor.
2. Add the following configuration to the file, inside the `web-app` tags:

```
<servlet>
  <servlet-name>RESTEasy JSAPI</servlet-name>
  <servlet-class>org.jboss.resteasy.jsapi.JSAPIServlet</servlet-
class>
</servlet>

<servlet-mapping>
  <servlet-name>RESTEasy JSAPI</servlet-name>
  <url-pattern>/URL</url-pattern>
</servlet-mapping>
```

[Report a bug](#)

15.11.3. RESTEasy Javascript API Parameters

Table 15.5. Parameter Properties

Property	Default Value	Description
\$entity		The entity to send as a PUT, POST request.

Property	Default Value	Description
\$contentType		The MIME type of the body entity sent as the Content-Type header. Determined by the @Consumes annotation.
\$accepts	*/*	The accepted MIME types sent as the Accept header. Determined by the @Provides annotation.
\$callback		Set to a function (httpCode, xmlHttpRequest, value) for an asynchronous call. If not present, the call will be synchronous and return the value.
\$apiURL		Set to the base URI of the JAX-RS endpoint, not including the last slash.
\$username		If username and password are set, they will be used for credentials for the request.
\$password		If username and password are set, they will be used for credentials for the request.

[Report a bug](#)

15.11.4. Build AJAX Queries with the JavaScript API

Summary

The RESTEasy JavaScript API can be used to manually construct requests. This topic covers examples of this behavior.

Example 15.19. The REST Object

The REST object can be used to override RESTEasy JavaScript API client behavior:

```
// Change the base URL used by the API:
REST.apiURL = "http://api.service.com";

// log everything in a div element
REST.log = function(text){
    jQuery("#log-div").append(text);
};
```

The REST object contains the following read-write properties:

apiURL

Set by default to the JAX-RS root URL. Used by every JavaScript client API functions when constructing the requests.

log

Set to a function(string) in order to receive RESTEasy client API logs. This is useful if you want to debug your client API and place the logs where you can see them.

Example 15.20. The REST.Request Class

The REST.Request class can be used to build custom requests:

```
var r = new REST.Request();
r.setURI("http://api.service.com/orders/23/json");
r.setMethod("PUT");
r.setContentType("application/json");
r.setEntity({id: "23"});
r.addMatrixParameter("JSESSIONID", "12309812378123");
r.execute(function(status, request, entity){
    log("Response is "+status);
});
```

[Report a bug](#)

15.11.5. REST.Request Class Members

Table 15.6. REST.Request Class

Member	Description
execute(callback)	Executes the request with all the information set in the current object. The value is passed to the optional argument callback, not returned.
setAccepts(acceptHeader)	Sets the Accept request header. Defaults to */*.
setCredentials(username, password)	Sets the request credentials.
setEntity(entity)	Sets the request entity.
setContentType(contentTypeHeader)	Sets the Content-Type request header.
setURI(uri)	Sets the request URI. This should be an absolute URI.
setMethod(method)	Sets the request method. Defaults to GET.

Member	Description
<code>setAsync(async)</code>	Controls whether the request should be asynchronous. Defaults to true.
<code>addCookie(name, value)</code>	Sets the given cookie in the current document when executing the request. This will be persistent in the browser.
<code>addQueryParameter(name, value)</code>	Adds a query parameter to the URI query part.
<code>addMatrixParameter(name, value)</code>	Adds a matrix parameter (path parameter) to the last path segment of the request URI.
<code>addHeader(name, value)</code>	Adds a request header.

[Report a bug](#)

15.12. RESTEasy ASYNCHRONOUS JOB SERVICE

15.12.1. About the RESTEasy Asynchronous Job Service

The RESTEasy Asynchronous Job Service is designed to add asynchronous behavior to the HTTP protocol. While HTTP is a synchronous protocol it does have a faint idea of asynchronous invocations. The HTTP 1.1 response code 202, "Accepted" means that the server has received and accepted the response for processing, but the processing has not yet been completed. The Asynchronous Job Service builds around this.

To enable the service, refer to: [Section 15.12.2, "Enable the Asynchronous Job Service"](#). For examples of how the service works, refer to [Section 15.12.3, "Configure Asynchronous Jobs for RESTEasy"](#).

[Report a bug](#)

15.12.2. Enable the Asynchronous Job Service

Procedure 15.7. Modify the web.xml file

- Enable the asynchronous job service in the **web.xml** file:

```
<context-param>
  <param-name>resteasy.async.job.service.enabled</param-name>
  <param-value>true</param-value>
</context-param>
```

Result

The asynchronous job service has been enabled. For configuration options, refer to: [Section 15.12.4, "Asynchronous Job Service Configuration Parameters"](#).

[Report a bug](#)

15.12.3. Configure Asynchronous Jobs for RESTEasy

Summary

This topic covers examples of the query parameters for asynchronous jobs with RESTEasy.



WARNING

Role based security does not work with the Asynchronous Job Service, as it cannot be implemented portably. If the Asynchronous Job Service is used, application security must be done through XML declarations in the **web.xml** file instead.



IMPORTANT

While GET, DELETE, and PUT methods can be invoked asynchronously, this breaks the HTTP 1.1 contract of these methods. While these invocations may not change the state of the resource if invoked more than once, they do change the state of the server as new Job entries with each invocation.

Example 15.21. The Asynch Parameter

The **asynch** query parameter is used to run invocations in the background. A 202 Accepted response is returned, as well as a Location header with a URL pointing to where the response of the background method is located.

```
POST http://example.com/myservice?asynch=true
```

The example above will return a 202 Accepted response. It will also return a Location header with a URL pointing to where the response of the background method is located. An example of the location header is shown below:

```
HTTP/1.1 202 Accepted
Location: http://example.com/asynch/jobs/3332334
```

The URI will take the form of:

```
/asynch/jobs/{job-id}?wait={milliseconds}|nowait=true
```

GET, POST and DELETE operations can be performed on this URL.

- GET returns the JAX-RS resource method invoked as a response if the job was completed. If the job has not been completed, this GET will return a 202 Accepted response code. Invoking GET does not remove the job, so it can be called multiple times.
- POST does a read of the job response and removes the job if it has been completed.
- DELETE is called to manually clean up the job queue.

**NOTE**

When the Job queue is full, it will evict the earliest job from memory automatically, without needing to call DELETE.

Example 15.22. Wait / Nowait

The GET and POST operations allow for the maximum wait time to be defined, using the **wait** and **nowait** query parameters. If the **wait** parameter is not specified, the operation will default to **nowait=true**, and will not wait at all if the job is not complete. The **wait** parameter is defined in milliseconds.

```
POST http://example.com/asynch/jobs/122?wait=3000
```

Example 15.23. The Oneway Parameter

RESTEasy supports fire and forget jobs, using the **oneway** query parameter.

```
POST http://example.com/myservice?oneway=true
```

The example above will return a 202 Accepted response, but no job will be created.

[Report a bug](#)

15.12.4. Asynchronous Job Service Configuration Parameters**Summary**

The table below details the configurable context-params for the Asynchronous Job Service. These parameters can be configured in the **web.xml** file.

Table 15.7. Configuration Parameters

Parameter	Description
resteasy.async.job.service.max.job.results	Number of job results that can be held in the memory at any one time. Default value is 100.
resteasy.async.job.service.max.wait	Maximum wait time on a job when a client is querying for it. Default value is 300000.
resteasy.async.job.service.thread.pool.size	Thread pool size of the background threads that run the job. Default value is 100.
resteasy.async.job.service.base.path	Sets the base path for the job URIs. Default value is /asynch/jobs

Example 15.24. Example Asynchronous Jobs Configuration

```

<web-app>
  <context-param>
    <param-name>resteasy.async.job.service.enabled</param-name>
    <param-value>true</param-value>
  </context-param>

  <context-param>
    <param-name>resteasy.async.job.service.max.job.results</param-
name>
    <param-value>100</param-value>
  </context-param>
  <context-param>
    <param-name>resteasy.async.job.service.max.wait</param-name>
    <param-value>300000</param-value>
  </context-param>
  <context-param>
    <param-name>resteasy.async.job.service.thread.pool.size</param-
name>
    <param-value>100</param-value>
  </context-param>
  <context-param>
    <param-name>resteasy.async.job.service.base.path</param-name>
    <param-value>/asynch/jobs</param-value>
  </context-param>

  <listener>
    <listener-class>
      org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>Resteasy</servlet-name>
    <servlet-class>
      org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Resteasy</servlet-name>
    <url-pattern>*</url-pattern>
  </servlet-mapping>
</web-app>

```

[Report a bug](#)
15.13. RESTEASY JAXB**15.13.1. Create a JAXB Decorator**

Summary

RESTEasy's JAXB providers have a pluggable way to decorate Marshaller and Unmarshaller instances. An annotation is created that can trigger either a Marshaller or Unmarshaller instance. This topic covers the steps to create a JAXB decorator with RESTEasy.

Procedure 15.8. Create a JAXB Decorator with RESTEasy

1. Create the Processor Class

- a. Create a class that implements `DecoratorProcessor<Target, Annotation>`. The target is either the JAXB Marshaller or Unmarshaller class. The annotation is created in step two.
- b. Annotate the class with `@DecorateTypes`, and declare the MIME Types the decorator should decorate.
- c. Set properties or values within the **decorate** function.

Example 15.25. Example Processor Class

```
import org.jboss.resteasy.core.interception.DecoratorProcessor;
import org.jboss.resteasy.annotations.DecorateTypes;

import javax.xml.bind.Marshaller;
import javax.xml.bind.PropertyException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.Produces;
import java.lang.annotation.Annotation;

@DecorateTypes({"text/*+xml", "application/*+xml"})
public class PrettyProcessor implements
DecoratorProcessor<Marshaller, Pretty>
{
    public Marshaller decorate(Marshaller target, Pretty
annotation,
    Class type, Annotation[] annotations, MediaType mediaType)
    {
        target.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
Boolean.TRUE);
    }
}
```

2. Create the Annotation

- a. Create a custom interface that is annotated with the `@Decorator` annotation.
- b. Declare the processor and target for the `@Decorator` annotation. The processor is created in step one. The target is either the JAXB Marshaller or Unmarshaller class.

Example 15.26. Example Annotation

```
import org.jboss.resteasy.annotations.Decorator;

@Target({ElementType.TYPE, ElementType.METHOD,
ElementType.PARAMETER, ElementType.FIELD})
```

```

@Retention(RetentionPolicy.RUNTIME)
@Decorator(processor = PrettyProcessor.class, target =
Marshaller.class)
public @interface Pretty {}

```

3. Add the annotation created in step two to a function so that either the input or output is decorated when it is marshalled.

Result

The JAXB decorator has been created and applied within the JAX-RS web service.

[Report a bug](#)

15.13.2. JAXB and XML Provider

RESTEasy facilitates JAXB provider support for XML.

@XmlHeader and @Stylesheet

RESTEasy provides setting an XML header using the

@org.jboss.resteasy.annotations.providers.jaxb.XmlHeader annotation. For example:

```

@XmlRootElement
public static class Thing
{
    private String name;

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }
}

@Path("/test")
public static class TestService
{
    @GET
    @Path("/header")
    @Produces("application/xml")
    @XmlHeader("<?xml-stylesheet type='text/xsl' href='${baseuri}foo.xsl' ?
>")
    public Thing get()
    {
        Thing thing = new Thing();
        thing.setName("bill");
        return thing;
    }
}

```

The **@XmlHeader** ensures that the XML output has an XML-style sheet header.

RESTEasy has a convenience annotation for style sheet headers. For example:

```
@XmlRootElement
public static class Thing
{
    private String name;

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }
}

@Path("/test")
public static class TestService
{
    @GET
    @Path("/stylesheet")
    @Produces("application/xml")
    @Stylesheet(type="text/css", href="${basepath}foo.xsl")
    @Junk
    public Thing getStyle()
    {
        Thing thing = new Thing();
        thing.setName("bill");
        return thing;
    }
}
```

[Report a bug](#)

15.13.3. JAXB and JSON Provider

RESTEasy allows you to marshal JAXB annotated POJOs to and from JSON. This provider wraps the Jettison JSON library to accomplish this task. For more information about Jettison and how it works, refer to: <http://jettison.codehaus.org/>.

```
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-jettison-provider</artifactId>
  <version>${version.org.jboss.resteasy}</version>
  <scope>provided</scope>
</dependency>
```

Jettison has two mapping formats. One is **BadgerFish** the other is a Jettison mapped convention format. The mapped convention is the default. For more details on the JAXB + JSON Provider integration with Jettison, refer to: http://docs.jboss.org/resteasy/docs/2.3.7.Final/userguide/html_single/index.html

[Report a bug](#)

15.14. RESTEASY ATOM SUPPORT

15.14.1. About the Atom API and Provider

The RESTEasy Atom API and Provider is a simple object model that RESTEasy defines to represent Atom. The main classes for the API are in the **org.jboss.resteasy.plugins.providers.atom** package. RESTEasy uses JAXB to marshal and unmarshal the API. The provider is JAXB based, and is not limited to sending atom objects using XML. All JAXB providers that RESTEasy has can be reused by the Atom API and provider, including JSON. Refer to the javadocs available from the [Customer Service Portal](#) for more information on the API.

```
import org.jboss.resteasy.plugins.providers.atom.Content;
import org.jboss.resteasy.plugins.providers.atom.Entry;
import org.jboss.resteasy.plugins.providers.atom.Feed;
import org.jboss.resteasy.plugins.providers.atom.Link;
import org.jboss.resteasy.plugins.providers.atom.Person;

@Path("atom")
public class MyAtomService
{
    @GET
    @Path("feed")
    @Produces("application/atom+xml")
    public Feed getFeed() throws URISyntaxException
    {
        Feed feed = new Feed();
        feed.setId(new URI("http://example.com/42"));
        feed.setTitle("My Feed");
        feed.setUpdated(new Date());
        Link link = new Link();
        link.setHref(new URI("http://localhost"));
        link.setRel("edit");
        feed.getLinks().add(link);
        feed.getAuthors().add(new Person("John Brown"));
        Entry entry = new Entry();
        entry.setTitle("Hello World");
        Content content = new Content();
        content.setType(MediaType.TEXT_HTML_TYPE);
        content.setText("Nothing much");
        entry.setContent(content);
        feed.getEntries().add(entry);
        return feed;
    }
}
```

[Report a bug](#)

15.14.2. Using JAXB with Atom Provider

The `org.jboss.resteasy.plugins.providers.atom.Content` class allows you to unmarshal and marshal JAXB annotated objects that are the body of the content. You can refer the example of sending an `Entry` with a `Customer` object attached as the body of the entry's content.

```
@XmlRootElement(namespace = "http://jboss.org/Customer")
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer
{
    @XmlElement
    private String name;

    public Customer()
    {
    }

    public Customer(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}

@Path("atom")
public static class AtomServer
{
    @GET
    @Path("entry")
    @Produces("application/atom+xml")
    public Entry getEntry()
    {
        Entry entry = new Entry();
        entry.setTitle("Hello World");
        Content content = new Content();
        content.setJAXBObject(new Customer("bill"));
        entry.setContent(content);
        return entry;
    }
}
```

The `Content.setJAXBObject()` method is used to specify the content object you are sending back to Java. The JAXB object is marshalled appropriately. If you are using a different base format other than XML, i.e. "application/atom+json", the attached JAXB object is marshalled in the same format. If you have an atom document as your input, you can also extract JAXB objects from `Content` using the `Content.getJAXBObject(Class clazz)` method. Here is an example of an input atom document and extracting a `Customer` object from the content.

```
@Path("atom")
public static class AtomServer
{
    @PUT
```

```

    @Path("entry")
    @Produces("application/atom+xml")
    public void putCustomer(Entry entry)
    {
        Content content = entry.getContent();
        Customer cust = content.getJAXBObject(Customer.class);
    }
}

```

[Report a bug](#)

15.15. YAML PROVIDER

RESTEasy comes with built in support for YAML using the **SnakeYAML** library. To enable YAML support, you must insert the following dependencies into the project pom file of your application:

```

<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-yaml-provider</artifactId>
  <version>${version.org.jboss.resteasy}</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>org.yaml</groupId>
  <artifactId>snakeyaml</artifactId>
  <version>${version.org.yaml.snakeyaml}</version>
</dependency>

```

YAML provider recognizes three mime types:

- text/x-yaml
- text/yaml
- application/x-yaml

The following example demonstrates how to use YAML in a resource method:

```

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("/yaml")
public class YamlResource
{
    @GET
    @Produces("text/x-yaml")
    public MyObject getMyObject() {
        return createMyObject();
    }
    ...
}

```


[Report a bug](#)

15.16. EJB INTEGRATION

In order to integrate RESTEasy with EJB, you must first modify the published interfaces of your EJB. Currently, RESTEasy only has simple portable integration with EJBs, so you must also manually configure your RESTEasy war file.

To make an EJB function as a JAX-RS resource, you must annotate an SLSB's **@Remote** or **@Local** interface with JAX-RS annotations:

```
@Local
@Path("/Library")
public interface Library {
    @GET
    @Path("/books/{isbn}")
    public String getBook(@PathParam("isbn") String isbn);
}
@Stateless
public class LibraryBean implements Library {
    ...
}
```

Next, in RESTEasy's **web.xml** file, you must manually register the EJB with RESTEasy using the **resteasy.jndi.resources** **<context-param>**

```
<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <context-param>
    <param-name>resteasy.jndi.resources</param-name>
    <param-value>LibraryBean/local</param-value>
  </context-param>
  <listener>

  <listener-class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap</listener-class>
  </listener>
  <servlet>
    <servlet-name>Resteasy</servlet-name>

  <servlet-class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Resteasy</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

[Report a bug](#)

15.17. JSON SUPPORT VIA JACKSON

Besides the Jettison JAXB adapter for JSON, RESTEasy also supports integration with the Jackson project. Jackson allows you to marshal Java objects to and from JSON. It has a Java bean based model as well as JAXB like APIs.

While Jackson comes with its own JAX-RS integration, RESTEasy expands it. In order to include it in your project, add the following Maven dependency to your build:

```
<repository>
  <id>jboss</id>
  <url>>http://repository.jboss.org/nexus/content/groups/public/</url>
</repository>
...
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-jackson-provider</artifactId>
  <version>${version.org.jboss.resteasy}</version>
  <scope>provided</scope>
</dependency>
```

For more information on JSON support via Jackson project, refer to http://docs.jboss.org/resteasy/docs/2.3.7.Final/userguide/html_single/index.html

[Report a bug](#)

15.18. RESTEASY/SPRING INTEGRATION

15.18.1. RESTEasy/Spring integration

Prerequisites

- Your application must have an existing JAX-WS service and client configuration.

Procedure 15.9. Enable the RESTEasy/Spring integration functionality

- RESTEasy integrates with Spring 3.0.x.

Maven users must use the `resteasy-spring` artifact. Alternatively, the jar is available as a module in JBoss EAP 6.

RESTEasy comes with its own Spring ContextLoaderListener that registers a RESTEasy specific BeanPostProcessor that processes JAX-RS annotations when a bean is created by a BeanFactory. This means that RESTEasy will automatically scan for `@Provider` and JAX-RS resource annotations on your bean class and register them as JAX-RS resources.

Example 15.27. Edit web.xml

Add the following to your web.xml file to enable the RESTEasy/Spring integration functionality:

```
<web-app>
  <display-name>
    Archetype Created Web Application
  </display-name>
  <listener>
```

```

        <listener-class>
            org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap
        </listener-class>
    </listener>

    <listener>
        <listener-class>

org.jboss.resteasy.plugins.spring.SpringContextLoaderListener
        </listener-class>
    </listener>

    <servlet>
        <servlet-name>Resteasy
        </servlet-name>
        <servlet-class>

org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
        </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>
            Resteasy
        </servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>

```

The `SpringContextLoaderListener` must be declared after `ResteasyBootstrap` as it uses `ServletContext` attributes initialized by it.

For more information regarding RestEasy and Spring integration, see http://docs.jboss.org/resteasy/docs/2.3.7.Final/userguide/html_single/

[Report a bug](#)

CHAPTER 16. JAX-WS WEB SERVICES

16.1. ABOUT JAX-WS WEB SERVICES

Java API for XML Web Services (JAX-WS) is an API included in the Java Enterprise Edition (EE) platform, and is used to create Web Services. Web Services are applications designed to communicate with each other over a network, typically exchanging information in XML or other structured text formats. Web Services are platform-independent. A typical JAX-WS application uses a client/server model. The server component is called a *Web Service Endpoint*.

JAX-WS has a counterpart for smaller and simpler Web Services, which use a protocol called JAX-RS. JAX-RS is a protocol for *Representational State Transfer*, or REST. JAX-RS applications are typically light-weight, and rely only on the HTTP protocol itself for communication. JAX-WS makes it easier to support various Web Service oriented protocols, such as **WS-Notification**, **WS-Addressing**, **WS-Policy**, **WS-Security**, and **WS-Trust**. They communicate using a specialized XML language called *Simple Object Access Protocol (SOAP)*, which defines a message architecture and message formats.

A JAX-WS Web Service also includes a machine-readable description of the operations it provides, written in *Web Services Description Language (WSDL)*, which is a specialized XML document type.

A Web Service Endpoint consists of a class which implements **WebService** and **WebMethod** interfaces.

A Web Service Client consists of a client which depends upon several classes called *stubs*, which are generated from the WSDL definition. JBoss EAP 6 includes the tools to generate the classes from WSDL.

In a JAX-WS Web service, a formal contract is established to describe the interface that the Web Service offers. The contract is typically written in WSDL, but may be written in SOAP messages. The architecture of the Web Service typically addresses business requirements, such as transactions, security, messaging, and coordination. JBoss EAP 6 provides mechanisms for handling these business concerns.

Web Services Description Language (WSDL) is an XML-based language used to describe Web Services and how to access them. The Web Service itself is written in Java or another programming language. The WSDL definition consists of references to the interface, port definitions, and instructions for how other Web Services should interact with it over a network. Web Services communicate with each other using *Simple Object Access Protocol (SOAP)*. This type of Web Service contrasts with *RESTful Web Services*, built using *Representative State Transfer (REST)* design principles. These RESTful Web Services do not require the use of WSDL or SOAP, but rely on the structure of the HTTP protocol itself to define how other services interact with them.

JBoss EAP 6 includes support for deploying JAX-WS Web Service endpoints. This support is provided by JBossWS. Configuration of the Web Services subsystem, such as endpoint configuration, handler chains, and handlers, is provided through the **webservices** subsystem.

Working Examples

The JBoss EAP Quickstarts include several fully-functioning JAX-WS Web Service applications. These examples include:

- `wsat-simple`
- `wsba-coordinator-completion-simple`
- `wsba-participant-completion-simple`

[Report a bug](#)

16.2. CONFIGURE THE WEBSERVICES SUBSYSTEM

Many configuration options are available for the **webservices** subsystem, which controls the behavior of Web Services deployed into JBoss EAP 6. The command to modify each element in the Management CLI script (***EAP_HOME/bin/jboss-cli.sh*** or ***EAP_HOME/bin/jboss-cli.bat***) is provided.

Remove the **/profile=default** portion of the command for a standalone server, or replace **default** with the name of profile to configure.

Published Endpoint Address

You can rewrite the **<soap:address>** element in endpoint-published WSDL contracts. This ability can be used to control the server address that is advertised to clients for each endpoint. Each of the following optional elements can be modified to suit your requirements. If there is any active WS deployment then modification of any of these elements requires a server reload.

Table 16.1. Configuration Elements for Published Endpoint Addresses

Element	Description	CLI Command
modify-wsdl-address	Whether to always modify the WSDL address. If true, the content of <soap:address> will always be overwritten. If false, the content of <soap:address> will only be overwritten if it is not a valid URL. The values used will be the wsdl-host , wsdl-port , and wsdl-secure-port described below.	/profile=default/subsystem=webservices/:write-attribute(name=modify-wsdl-address,value=true)
wsdl-host	The hostname / IP address to be used for rewriting <soap:address> . If wsdl-host is set to the string jbossws.undefi ned.host , the requester's host is used when rewriting the <soap:address> .	/profile=default/subsystem=webservices/:write-attribute(name=wsdl-host,value=10.1.1.1)
wsdl-port	An integer which explicitly defines the HTTP port that will be used for rewriting the SOAP address. If undefined, the HTTP port is identified by querying the list of installed HTTP connectors.	/profile=default/subsystem=webservices/:write-attribute(name=wsdl-port,value=8080)
wsdl-secure-port	An integer which explicitly defines the HTTPS port that will be used for rewriting the SOAP address. If undefined, the HTTPS port is identified by querying the list of installed HTTPS connectors.	/profile=default/subsystem=webservices/:write-attribute(name=wsdl-secure-port,value=8443)

Predefined Endpoint Configurations

You can define endpoint configurations which can be referenced by endpoint implementations. One way this might be used is to add a given handler to any WS endpoint that is marked with a given endpoint configuration with the annotation `@org.jboss.ws.api.annotation.EndpointConfig`.

JBoss EAP 6 includes a default **Standard-Endpoint-Config**. An example of a custom configuration, **Recording-Endpoint-Config**, is also included. This provides an example of a recording handler. The **Standard-Endpoint-Config** is automatically used for any endpoint which is not associated with any other configuration.

To read the **Standard-Endpoint-Config** using the Management CLI, use the following command:

```
/profile=default/subsystem=webservices/endpoint-config=Standard-Endpoint-Config/:read-resource(recursive=true,proxies=false,include-runtime=false,include-defaults=true)
```

Endpoint Configurations

An endpoint configuration, referred to as an **endpoint-config** in the Management API, includes a **pre-handler-chain**, **post-handler-chain** and some properties, which are applied to a particular endpoint. The following commands read and add an endpoint config.

Example 16.1. Read an Endpoint Config

```
/profile=default/subsystem=webservices/endpoint-config=Recording-Endpoint-Config:read-resource
```

Example 16.2. Add an Endpoint Config

```
/profile=default/subsystem=webservices/endpoint-config=My-Endpoint-Config:add
```

Handler Chains

Each endpoint config may be associated with **PRE** and **POST** handler chains. Each handler chain may include JAXWS-compliant handlers. For outbound messages, PRE handler chain handlers are executed before any handler attached to the endpoints using standard JAXWS means, such as the `@HandlerChain` annotation. POST handler chain handlers are executed after usual endpoint handlers. For inbound messages, the opposite applies. JAX-WS is a standard API for XML-based web services, and is documented at <http://jcp.org/en/jsr/detail?id=224>.

A handler chain may also include a **protocol-bindings** attribute, which sets the protocols which trigger the chain to start.

Example 16.3. Read a Handler Chain

```
/profile=default/subsystem=webservices/endpoint-config=Recording-Endpoint-Config/pre-handler-chain=recording-handlers:read-resource
```

Example 16.4. Add a Handler Chain

```
/profile=default/subsystem=webservices/endpoint-config=My-Endpoint-
Config/post-handler-chain=my-handlers:add(protocol-
bindings="##SOAP11_HTTP")
```

Handlers

A JAXWS handler is a child element **handler** within a handler chain. The handler takes a **class** attribute, which is the fully-qualified classname of the handler class. When the endpoint is deployed, an instance of that class is created for each referencing deployment. Either the deployment class loader or the class loader for module **org.jboss.as.webservices.server.integration** must be able to load the handler class.

Example 16.5. Read a Handler

```
/profile=default/subsystem=webservices/endpoint-config=Recording-
Endpoint-Config/pre-handler-chain=recording-
handlers/handler=RecordingHandler:read-resource
```

Example 16.6. Add a Handler

```
/profile=default/subsystem=webservices/endpoint-config=My-Endpoint-
Config/post-handler-chain=my-handlers/handler=foo-
handler:add(class="org.jboss.ws.common.invocation.RecordingServerHandler
")
```

Web Services Runtime Information

You can view runtime information about Web Services, such as the web context and the WSDL URL, by querying the endpoints themselves. You can use the ***** character to query all endpoints at once. The following examples show the command for both a server in a managed domain and for a standalone server.

Example 16.7. View Runtime Information about All Web Service Endpoints on A Server in a Managed Domain

This command displays information about all endpoints on a server named **server-one**, which is hosted on physical host **master** and running in a managed domain.

```
/host=master/server=server-
one/deployment="*/subsystem=webservices/endpoint="*":read-resource
```

Example 16.8. View Runtime Information about All Web Service Endpoints on a Standalone Server

This command displays information about all Web Service endpoints on a standalone server.

```
/deployment="*/subsystem=webservices/endpoint="*:read-resource
```

Example 16.9. Example Endpoint Information

The following is an example displaying hypothetical output.

```
{
  "outcome" => "success",
  "result" => [{
    "address" => [
      ("deployment" => "jaxws-samples-handlerchain.war"),
      ("subsystem" => "webservices"),
      ("endpoint" => "jaxws-samples-handlerchain:TestService")
    ],
    "outcome" => "success",
    "result" => {
      "class" =>
"org.jboss.test.ws.jaxws.samples.handlerchain.EndpointImpl",
      "context" => "jaxws-samples-handlerchain",
      "name" => "TestService",
      "type" => "JAXWS_JSE",
      "wsdl-url" => "http://localhost:8080/jaxws-samples-
handlerchain?wsdl"
    }
  ]
}
```

[Report a bug](#)

16.3. CONFIGURE THE HTTP TIMEOUT PER APPLICATION

The HTTP session timeout defines the period after which a HTTP session is considered to have become invalid because there was no activity within the specified period.

The HTTP session timeout can be configured in several places. In order of precedence these are:

- Application - defined in the application's **web.xml** configuration file.
- Server - specified via the **default-session-timeout** attribute.
- Default - 30 minutes.

Procedure 16.1. Configure the HTTP Timeout per Application

1. Edit the application's **WEB-INF/web.xml** file.
2. Add the following configuration XML to the file, changing **30** to the desired timeout (in minutes).

```
<session-config>
  <session-timeout>30</session-timeout>
</session-config>
```


3. If you modified the WAR file, redeploy the application. If you exploded the WAR file, no further action is required because JBoss EAP will automatically undeploy and redeploy the application.

[Report a bug](#)

16.4. JAX-WS WEB SERVICE ENDPOINTS

16.4.1. About JAX-WS Web Service Endpoints

This topic is an overview of JAX-WS web service endpoints and accompanying concepts. A JAX-WS Web Service endpoint is the server component of a Web Service. Clients and other Web Services communicate it over the HTTP protocol using an XML language called *Simple Object Access Protocol (SOAP)*. The endpoint itself is deployed into the JBoss EAP 6 container.

WSDL descriptors can be created in one of two ways:

1. You can write WSDL descriptors manually.
2. You can use JAX-WS annotations that create the WSDL descriptors automatically for you. This is the most common method for creating WSDL descriptors.

An endpoint implementation bean is annotated with JAX-WS annotations and deployed to the server. The server automatically generates and publishes the abstract contract in WSDL format for client consumption. All marshalling and unmarshalling is delegated to the *Java Architecture for XML Binding (JAXB)* service.

The endpoint itself may be a POJO (Plain Old Java Object) or a Java EE Web Application. You can also expose endpoints using an EJB3 stateless session bean. It is packaged into a Web Archive (WAR) file. The specification for packaging the endpoint, called a *Java Service Endpoint (JSE)* is defined in JSR-181, which can be found at <http://jcp.org/aboutJava/communityprocess/mrel/jsr181/index2.html>.

Development Requirements

A Web Service must fulfill the requirements of the JAX-WS API and the Web Services metadata specification at <http://www.jcp.org/en/jsr/summary?id=181>. A valid implementation meets the following requirements:

- It contains a `javax.jws.WebService` annotation.
- All method parameters and return types are compatible with the JAXB 2.0 specification, JSR-222. Refer to <http://www.jcp.org/en/jsr/summary?id=222> for more information.

Example 16.10. Example POJO Endpoint

```
@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class JSEBean01
{
    @WebMethod
    public String echo(String input)
    {
        ...
    }
}
```

Example 16.11. Example Web Services Endpoint

```

<web-app ...>
  <servlet>
    <servlet-name>TestService</servlet-name>
    <servlet-
class>org.jboss.test.ws.jaxws.samples.jsr181pojo.JSEBean01</servlet-
class>
    </servlet>
    <servlet-mapping>
      <servlet-name>TestService</servlet-name>
      <url-pattern>/*</url-pattern>
    </servlet-mapping>
  </web-app>

```

Example 16.12. Exposing an Endpoint in an EJB

This EJB3 stateless session bean exposes the same method on the remote interface and as an endpoint operation.

```

@Stateless
@Remote(EJB3RemoteInterface.class)
@RemoteBinding(jndiBinding = "/ejb3/EJB3EndpointInterface")

@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class EJB3Bean01 implements EJB3RemoteInterface
{
    @WebMethod
    public String echo(String input)
    {
        ...
    }
}

```

Endpoint Providers

JAX-WS services typically implement a Java service endpoint interface (SEI), which may be mapped from a WSDL port type, either directly or using annotations. This SEI provides a high-level abstraction which hides the details between Java objects and their XML representations. However, in some cases, services need the ability to operate at the XML message level. The endpoint **Provider** interface provides this functionality to Web Services which implement it.

Consuming and Accessing the Endpoint

After you deploy your Web Service, you can consume the WSDL to create the component stubs which will be the basis for your application. Your application can then access the endpoint to do its work.

Working Examples

The JBoss EAP Quickstarts include several fully-functioning JAX-WS Web Service applications. These examples include:

- `wsat-simple`
- `wsba-coordinator-completion-simple`
- `wsba-participant-completion-simple`

[Report a bug](#)

16.4.2. Write and Deploy a JAX-WS Web Service Endpoint

Introduction

This topic discusses the development of a simple JAX-WS service endpoint, which is the server-side component, which responds to requests from JAX-WS clients and publishes the WSDL definition for itself. For more in-depth information about JAX-WS service endpoints, refer to [Section 16.6.2, “JAX-WS Common API Reference”](#) and the API documentation bundle in Javadoc format, distributed with JBoss EAP 6.

Development Requirements

A Web Service must fulfill the requirements of the JAXWS API and the Web Services meta data specification at <http://www.jcp.org/en/jsr/summary?id=181>. A valid implementation meets the following requirements:

- It contains a `javax.jws.WebService` annotation.
- All method parameters and return types are compatible with the JAXB 2.0 specification, JSR-222. Refer to <http://www.jcp.org/en/jsr/summary?id=222> for more information.

Example 16.13. Example Service Implementation

```
package org.jboss.test.ws.jaxws.samples.retail.profile;

import javax.ejb.Stateless;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.soap.SOAPBinding;

@Stateless
@WebService(
    name="ProfileMgmt",
    targetNamespace = "http://org.jboss.ws/samples/retail/profile",
    serviceName = "ProfileMgmtService")
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
public class ProfileMgmtBean {

    @WebMethod
    public DiscountResponse getCustomerDiscount(DiscountRequest request)
    {
        return new DiscountResponse(request.getCustomer(), 10.00);
    }
}
```

Example 16.14. Example XML Payload

The following is an example of the **DiscountRequest** class which is used by the **ProfileMgmtBean** bean in the previous example. The annotations are included for verbosity. Typically, the JAXB defaults are reasonable and do not need to be specified.

```
package org.jboss.test.ws.jaxws.samples.retail.profile;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlType;

import org.jboss.test.ws.jaxws.samples.retail.Customer;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(
    name = "discountRequest",
    namespace="http://org.jboss.ws/samples/retail/profile",
    propOrder = { "customer" }
)
public class DiscountRequest {

    protected Customer customer;

    public DiscountRequest() {
    }

    public DiscountRequest(Customer customer) {
        this.customer = customer;
    }

    public Customer getCustomer() {
        return customer;
    }

    public void setCustomer(Customer value) {
        this.customer = value;
    }

}
```

More complex mappings are possible. Refer to the JAXB API specification at <https://jaxb.java.net/> for more information.

Package Your Deployment

The implementation class is wrapped in a **JAR** deployment. Any metadata required for deployment is taken from the annotations on the implementation class and the service endpoint interface. Deploy the JAR using the Management CLI or the Management Interface, and the HTTP endpoint is created automatically.

The following listing shows an example of the correct structure for JAR deployment of an EJB Web Service.

Example 16.15. Example JAR Structure for a Web Service Deployment

```
[user@host ~]$ jar -tf jaxws-samples-retail.jar
org/jboss/test/ws/jaxws/samples/retail/profile/DiscountRequest.class
org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.class
org/jboss/test/ws/jaxws/samples/retail/profile/ObjectFactory.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmt.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtBean.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtService.class
org/jboss/test/ws/jaxws/samples/retail/profile/package-info.class
```

[Report a bug](#)

16.5. JAX-WS WEB SERVICE CLIENTS

16.5.1. Consume and Access a JAX-WS Web Service

After creating a Web Service endpoint, either manually or using JAX-WS annotations, you can access its WSDL, which can be used to create the basic client application which will communicate with the Web Service. The process of generating Java code from the published WSDL is called consuming the Web service. This happens in the following phases:

1. Create the client artifacts.
2. Construct a service stub.

Create the Client Artifacts

Before you can create client artifacts, you need to create your WSDL contract. The following WSDL contract is used for the examples presented in the rest of this topic.

The examples below rely on having this WSDL contract in the **ProfileMgmtService.wsdl** file.

Example 16.16. Example WSDL Contract

```
<definitions
  name='ProfileMgmtService'
  targetNamespace='http://org.jboss.ws/samples/retail/profile'
  xmlns='http://schemas.xmlsoap.org/wsdl/'
  xmlns:ns1='http://org.jboss.ws/samples/retail'
  xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
  xmlns:tns='http://org.jboss.ws/samples/retail/profile'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'>

  <types>

    <xs:schema targetNamespace='http://org.jboss.ws/samples/retail'
      version='1.0'
      xmlns:xs='http://www.w3.org/2001/XMLSchema'>
```

```

        <xs:complexType name='customer'>
            <xs:sequence>
                <xs:element minOccurs='0' name='creditCardDetails'
type='xs:string' />
                <xs:element minOccurs='0' name='firstName'
type='xs:string' />
                <xs:element minOccurs='0' name='lastName'
type='xs:string' />
            </xs:sequence>
        </xs:complexType>
    </xs:schema>

    <xs:schema
        targetNamespace='http://org.jboss.ws/samples/retail/profile'
        version='1.0'
        xmlns:ns1='http://org.jboss.ws/samples/retail'
        xmlns:tns='http://org.jboss.ws/samples/retail/profile'
        xmlns:xs='http://www.w3.org/2001/XMLSchema'>

        <xs:import namespace='http://org.jboss.ws/samples/retail' />
        <xs:element name='getCustomerDiscount'
            nillable='true' type='tns:discountRequest' />
        <xs:element name='getCustomerDiscountResponse'
            nillable='true' type='tns:discountResponse' />
        <xs:complexType name='discountRequest'>
            <xs:sequence>
                <xs:element minOccurs='0' name='customer'
type='ns1:customer' />
            </xs:sequence>
        </xs:complexType>
        <xs:complexType name='discountResponse'>
            <xs:sequence>
                <xs:element minOccurs='0' name='customer'
type='ns1:customer' />
                <xs:element name='discount' type='xs:double' />
            </xs:sequence>
        </xs:complexType>
    </xs:schema>

</types>

    <message name='ProfileMgmt_getCustomerDiscount'>
        <part element='tns:getCustomerDiscount'
name='getCustomerDiscount' />
    </message>
    <message name='ProfileMgmt_getCustomerDiscountResponse'>
        <part element='tns:getCustomerDiscountResponse'
name='getCustomerDiscountResponse' />
    </message>
    <portType name='ProfileMgmt'>
        <operation name='getCustomerDiscount'
            parameterOrder='getCustomerDiscount'>

            <input message='tns:ProfileMgmt_getCustomerDiscount' />
            <output

```

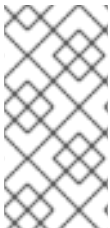
```

message='tns:ProfileMgmt_getCustomerDiscountResponse' />
  </operation>
</portType>
<binding name='ProfileMgmtBinding' type='tns:ProfileMgmt'>
  <soap:binding style='document'
    transport='http://schemas.xmlsoap.org/soap/http' />
  <operation name='getCustomerDiscount'>
    <soap:operation soapAction='' />
    <input>

      <soap:body use='literal' />
    </input>
    <output>
      <soap:body use='literal' />
    </output>
  </operation>
</binding>
<service name='ProfileMgmtService'>
  <port binding='tns:ProfileMgmtBinding' name='ProfileMgmtPort'>

    <soap:address
      location='SERVER:PORT/jaxws-samples-
retail/ProfileMgmtBean' />
  </port>
</service>
</definitions>

```



NOTE

If you use JAX-WS annotations to create your Web Service endpoint, the WSDL contract is generated automatically, and you only need its URL. You can get this URL from the **Webservices** section of the **Runtime** section of the web-based Management Console, after the endpoint is deployed.

The **wsconsume.sh** or **wsconsume.bat** tool is used to consume the abstract contract (WSDL) and produce annotated Java classes and optional sources that define it. The command is located in the **EAP_HOME/bin/** directory of the JBoss EAP 6 installation.

Example 16.17. Syntax of the wsconsume.sh Command

```

[user@host bin]$ ./wsconsume.sh --help
WSConsumeTask is a cmd line tool that generates portable JAX-WS
artifacts from a WSDL file.

usage: org.jboss.ws.tools.cmd.WSConsume [options] <wsdl-url>

options:
  -h, --help                Show this help message
  -b, --binding=<file>      One or more JAX-WS or JAXB binding
files
  -k, --keep                Keep/Generate Java source
  -c --catalog=<file>       Oasis XML Catalog file for entity

```

```

resolution
  -p --package=<name>           The target package for generated source
  -w --wsdlLocation=<loc>       Value to use for
@WebService.wsdlLocation
  -o, --output=<directory>      The directory to put generated artifacts
  -s, --source=<directory>      The directory to put Java source
  -t, --target=<2.0|2.1|2.2>    The JAX-WS specification target
  -q, --quiet                   Be somewhat more quiet
  -v, --verbose                 Show full exception stack traces
  -l, --load-consumer           Load the consumer and exit (debug
utility)
  -e, --extension               Enable SOAP 1.2 binding extension
  -a, --additionalHeaders       Enable processing of implicit SOAP
headers
  -n, --nocompile               Do not compile generated sources

```

The following command generates the source **.java** files listed in the output, from the **ProfileMgmtService.wsdl** file. The sources use the directory structure of the package, which is specified with the **-p** switch.

```

[user@host bin]$ wsconsume.sh -k -p
org.jboss.test.ws.jaxws.samples.retail.profile ProfileMgmtService.wsdl
output/org/jboss/test/ws/jaxws/samples/retail/profile/Customer.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountRequest.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.java
a
output/org/jboss/test/ws/jaxws/samples/retail/profile/ObjectFactory.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmt.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtService.j
ava
output/org/jboss/test/ws/jaxws/samples/retail/profile/package-info.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/Customer.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountRequest.clas
s
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.cla
ss
output/org/jboss/test/ws/jaxws/samples/retail/profile/ObjectFactory.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmt.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtService.c
lass
output/org/jboss/test/ws/jaxws/samples/retail/profile/package-info.class

```

Both **.java** source files and compiled **.class** files are generated into the **output/** directory within the directory where you run the command.

Table 16.2. Descriptions of Artifacts Created by wsconsume.sh

File	Description
ProfileMgmt.java	Service endpoint interface.
Customer.java	Custom data type.

File	Description
Discount*.java	Custom data types.
ObjectFactory.java	JAXB XML registry.
package-info.java	JAXB package annotations.
ProfileMgmtService.java	Service factory.

The `wsconsume.sh` command generates all custom data types (JAXB annotated classes), the service endpoint interface and a service factory class. These artifacts are used to build web service client implementations.

Construct a Service Stub

Web service clients use service stubs to abstract the details of a remote web service invocation. To a client application, a WS invocation looks like an invocation of any other business component. In this case the service endpoint interface acts as the business interface, and a service factory class is not used to construct it as a service stub.

Example 16.18. Constructing a Service Stub and Accessing the Endpoint

The following example first creates a service factory using the WSDL location and the service name. Next, it uses the service endpoint interface created by the `wsconsume.sh` command to build the service stub. Finally, the stub can be used just as any other business interface would be.

You can find the WSDL URL for your endpoint in the web-based Management Console. Choose the **Runtime** menu item in the top bar then the **Webservices** entry under **Subsystems** in the left pane. View the **Attributes** tab to review your deployments details.

```
import javax.xml.ws.Service;
[...]
```

```
Service service = Service.create(
    new URL("http://example.org/service?wsdl"),
    new QName("MyService")
);
ProfileMgmt profileMgmt = service.getPort(ProfileMgmt.class);

// Use the service stub in your application
```

[Report a bug](#)

16.5.2. Develop a JAX-WS Client Application

This topic discusses JAX-WS Web Service clients in general. The client communicates with, and requests work from, the JAX-WS endpoint, which is deployed in the Java Enterprise Edition 6 container. For detailed information about the classes, methods, and other implementation details mentioned below, refer to [Section 16.6.2, “JAX-WS Common API Reference”](#) and the relevant sections of the Javadocs bundle included with JBoss EAP 6.

Service

Overview

A **Service** is an abstraction which represents a WSDL service. A WSDL service is a collection of related ports, each of which includes a port type bound to a particular protocol and a particular endpoint address.

Usually, the Service is generated when the rest of the component stubs are generated from an existing WSDL contract. The WSDL contract is available via the WSDL URL of the deployed endpoint, or can be created from the endpoint source using the **wsprovide.sh** command in the **EAP_HOME/bin/** directory.

This type of usage is referred to as the *static* use case. In this case, you create instances of the **Service** class which is created as one of the component stubs.

You can also create the service manually, using the **Service.create** method. This is referred to as the *dynamic* use case.

Usage

Static Use Case

The *static* use case for a JAX-WS client assumes that you already have a WSDL contract. This may be generated by an external tool or generated by using the correct JAX-WS annotations when you create your JAX-WS endpoint.

To generate your component stubs, you use the **wsconsume.sh** or **wsconsume.bat** script which is included in **EAP_HOME/bin/**. The script takes the WSDL URL or file as a parameter, and generates multiple of files, structured in a directory tree. The source and class files representing your **Service** are named **CLASSNAME_Service.java** and **CLASSNAME_Service.class**, respectively.

The generated implementation class has two public constructors, one with no arguments and one with two arguments. The two arguments represent the WSDL location (a **java.net.URL**) and the service name (a **javax.xml.namespace.QName**) respectively.

The no-argument constructor is the one used most often. In this case the WSDL location and service name are those found in the WSDL. These are set implicitly from the **@WebServiceClient** annotation that decorates the generated class.

Example 16.19. Example Generated Service Class

```
@WebServiceClient(name="StockQuoteService",
    targetNamespace="http://example.com/stocks",
    wsdlLocation="http://example.com/stocks.wsdl")
public class StockQuoteService extends javax.xml.ws.Service
{
    public StockQuoteService()
    {
        super(new URL("http://example.com/stocks.wsdl"), new
QName("http://example.com/stocks", "StockQuoteService"));
    }

    public StockQuoteService(String wsdlLocation, QName serviceName)
    {
        super(wsdlLocation, serviceName);
    }
}
```

```

    }
    ...
}

```

Dynamic Use Case

In the dynamic case, no stubs are generated automatically. Instead, a web service client uses the **Service.create** method to create **Service** instances. The following code fragment illustrates this process.

Example 16.20. Creating Services Manually

```

URL wsdlLocation = new URL("http://example.org/my.wsdl");
QName serviceName = new QName("http://example.org/sample",
    "MyService");
Service service = Service.create(wsdlLocation, serviceName);

```

Handler Resolver

JAX-WS provides a flexible plug-in framework for message processing modules, known as *handlers*. These handlers extend the capabilities of a JAX-WS runtime system. A **Service** instance provides access to a **HandlerResolver** via a pair of **getHandlerResolver** and **setHandlerResolver** methods that can configure a set of handlers on a per-service, per-port or per-protocol binding basis.

When a **Service** instance creates a proxy or a **Dispatch** instance, the handler resolver currently registered with the service creates the required handler chain. Subsequent changes to the handler resolver configured for a **Service** instance do not affect the handlers on previously created proxies or **Dispatch** instances.

Executor

Service instances can be configured with a **java.util.concurrent.Executor**. The **Executor** invokes any asynchronous callbacks requested by the application. The **setExecutor** and **getExecutor** methods of **Service** can modify and retrieve the **Executor** configured for a service.

Dynamic Proxy

A *dynamic proxy* is an instance of a client proxy using one of the **getPort** methods provided in the **Service**. The **portName** specifies the name of the WSDL port the service uses. The **serviceEndpointInterface** specifies the service endpoint interface supported by the created dynamic proxy instance.

Example 16.21. getPort Methods

```

public <T> T getPort(QName portName, Class<T> serviceEndpointInterface)
public <T> T getPort(Class<T> serviceEndpointInterface)

```

The *Service Endpoint Interface* is usually generated using the `wsconsume.sh` command, which parses the WSDL and creates Java classes from it.

A typed method which returns a port is also provided. These methods also return dynamic proxies that implement the SEI. See the following example.

Example 16.22. Returning the Port of a Service

```
@WebServiceClient(name = "TestEndpointService", targetNamespace =
    "http://org.jboss.ws/wsref",
    wsdlLocation = "http://localhost.localdomain:8080/jaxws-samples-
    webserviceref?wsdl")

public class TestEndpointService extends Service
{
    ...

    public TestEndpointService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }

    @WebEndpoint(name = "TestEndpointPort")
    public TestEndpoint getTestEndpointPort()
    {
        return (TestEndpoint)super.getPort(TESTENDPOINTPORT,
        TestEndpoint.class);
    }
}
```

@WebServiceRef

The `@WebServiceRef` annotation declares a reference to a Web Service. It follows the resource pattern shown by the `javax.annotation.Resource` annotation defined in <http://www.jcp.org/en/jsr/summary?id=250>.

Use Cases for @WebServiceRef

- You can use it to define a reference whose type is a generated **Service** class. In this case, the type and value element each refer to the generated **Service** class type. Moreover, if the reference type can be inferred by the field or method declaration the annotation is applied to, the type and value elements may (but are not required to) have the default value of **Object.class**. If the type cannot be inferred, then at least the type element must be present with a non-default value.
- You can use it to define a reference whose type is an SEI. In this case, the type element may (but is not required to) be present with its default value if the type of the reference can be inferred from the annotated field or method declaration. However, the value element must always be present and refer to a generated service class type, which is a subtype of **javax.xml.ws.Service**. The **wsdlLocation** element, if present, overrides the WSDL location information specified in the `@WebService` annotation of the referenced generated service class.

Example 16.23. @WebServiceRef Examples

```

public class EJB3Client implements EJB3Remote
{
    @WebServiceRef
    public TestEndpointService service4;

    @WebServiceRef
    public TestEndpoint port3;
}

```

Dispatch

XML Web Services use XML messages for communication between the endpoint, which is deployed in the Java EE container, and any clients. The XML messages use an XML language called *Simple Object Access Protocol (SOAP)*. The JAX-WS API provides the mechanisms for the endpoint and clients to each be able to send and receive SOAP messages. Marshalling is the process of converting a Java Object into a SOAP XML message. Unmarshalling is the process of converting the SOAP XML message back into a Java Object.

In some cases, you need access to the raw SOAP messages themselves, rather than the result of the conversion. The **Dispatch** class provides this functionality. **Dispatch** operates in one of two usage modes, which are identified by one of the following constants.

- **javax.xml.ws.Service.Mode.MESSAGE** - This mode directs client applications to work directly with protocol-specific message structures. When used with a SOAP protocol binding, a client application works directly with a SOAP message.
- **javax.xml.ws.Service.Mode.PAYLOAD** - This mode causes the client to work with the payload itself. For instance, if it is used with a SOAP protocol binding, a client application would work with the contents of the SOAP body rather than the entire SOAP message.

Dispatch is a low-level API which requires clients to structure messages or payloads as XML, with strict adherence to the standards of the individual protocol and a detailed knowledge of message or payload structure. **Dispatch** is a generic class which supports input and output of messages or message payloads of any type.

Example 16.24. Dispatch Usage

```

Service service = Service.create(wsdlURL, serviceName);
Dispatch dispatch = service.createDispatch(portName, StreamSource.class,
Mode.PAYLOAD);

String payload = "<ns1:ping
xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
dispatch.invokeOneWay(new StreamSource(new StringReader(payload)));

payload = "<ns1:feedback
xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
Source retObj = (Source)dispatch.invoke(new StreamSource(new
StringReader(payload)));

```

Asynchronous Invocations

The **BindingProvider** interface represents a component that provides a protocol binding which clients can use. It is implemented by proxies and is extended by the **Dispatch** interface.

BindingProvider instances may provide asynchronous operation capabilities. Asynchronous operation invocations are decoupled from the **BindingProvider** instance at invocation time. The response context is not updated when the operation completes. Instead, a separate response context is made available using the **Response** interface.

Example 16.25. Example Asynchronous Invocation

```
public void testInvokeAsync() throws Exception
{
    URL wsdlURL = new URL("http://" + getServerHost() + ":8080/jaxws-
samples-asynchronous?wsdl");
    QName serviceName = new QName(targetNS, "TestEndpointService");
    Service service = Service.create(wsdlURL, serviceName);
    TestEndpoint port = service.getPort(TestEndpoint.class);
    Response response = port.echoAsync("Async");
    // access future
    String retStr = (String) response.get();
    assertEquals("Async", retStr);
}
```

@Oneway Invocations

The **@Oneway** annotation indicates that the given web method takes an input message but returns no output message. Usually, a **@Oneway** method returns the thread of control to the calling application before the business method is executed.

Example 16.26. Example @Oneway Invocation

```
@WebService (name="PingEndpoint")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class PingEndpointImpl
{
    private static String feedback;

    @WebMethod
    @Oneway
    public void ping()
    {
        log.info("ping");
        feedback = "ok";
    }

    @WebMethod
    public String feedback()
    {
        log.info("feedback");
        return feedback;
    }
}
```

Timeout Configuration

Two different properties control the timeout behavior of the HTTP connection and the timeout of a client

which is waiting to receive a message. The first is `javax.xml.ws.client.connectionTimeout` and the second is `javax.xml.ws.client.receiveTimeout`. Each is expressed in milliseconds, and the correct syntax is shown below.

Example 16.27. JAX-WS Timeout Configuration

```
public void testConfigureTimeout() throws Exception
{
    //Set timeout until a connection is established

    ((BindingProvider)port).getRequestContext().put("javax.xml.ws.client.con
nectionTimeout", "6000");

    //Set timeout until the response is received
    ((BindingProvider)
port).getRequestContext().put("javax.xml.ws.client.receiveTimeout",
"1000");

    port.echo("testTimeout");
}
```

[Report a bug](#)

16.6. JAX-WS DEVELOPMENT REFERENCE

16.6.1. Enable Web Services Addressing (WS-Addressing)

Prerequisites

- Your application must have an existing JAX-WS service and client configuration.

Procedure 16.2. Annotate and Update client code

1. Annotate the service endpoint

Add the `@Addressing` annotation to the application's endpoint code.

Example 16.28. `@Addressing` annotation

This example demonstrates a regular JAX-WS endpoint with the `@Addressing` annotation added.

```
package org.jboss.test.ws.jaxws.samples.wsa;

import javax.jws.WebService;
import javax.xml.ws.soap.Addressing;

@WebService
(
    portName = "AddressingServicePort",
    serviceName = "AddressingService",
    wsdlLocation = "WEB-INF/wsdl/AddressingService.wsdl",
    targetNamespace = "http://www.jboss.org/jbossws/ws-
```

```

        extensions/wsaddressing",
        endpointInterface =
        "org.jboss.test.ws.jaxws.samples.wsa.ServiceIface"
    )
    @Addressing(enabled=true, required=true)
    public class ServiceImpl implements ServiceIface
    {
        public String sayHello()
        {
            return "Hello World!";
        }
    }

```

2. Update client code

Update the client code in the application so that it configures WS-Addressing.

Example 16.29. Client configuration for WS-Addressing

This example demonstrates a regular JAX-WS client updated to configure WS-Addressing.

```

package org.jboss.test.ws.jaxws.samples.wsa;

import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.soap.AddressingFeature;

public final class AddressingTestCase
{
    private final String serviceURL =
        "http://localhost:8080/jaxws-samples-
        wsa/AddressingService";

    public static void main(String[] args) throws Exception
    {
        // construct proxy
        QName serviceName =
            new QName("http://www.jboss.org/jbossws/ws-
            extensions/wsaddressing",
                "AddressingService");
        URL wsdlURL = new URL(serviceURL + "?wsdl");
        Service service = Service.create(wsdlURL, serviceName);
        ServiceIface proxy =
            (ServiceIface)service.getPort(ServiceIface.class,
                new
                AddressingFeature());
        // invoke method
        proxy.sayHello();
    }
}

```

Result

The client and endpoint are now communicating using WS-Addressing.

[Report a bug](#)

16.6.2. JAX-WS Common API Reference

Several JAX-WS development concepts are shared between Web Service endpoints and clients. These include the handler framework, message context, and fault handling.

Handler Framework

The handler framework is implemented by a JAX-WS protocol binding in the runtime of the client and the endpoint, which is the server component. Proxies and **Dispatch** instances, known collectively as *binding providers*, each use protocol bindings to bind their abstract functionality to specific protocols.

Client and server-side handlers are organized into an ordered list known as a *handler chain*. The handlers within a handler chain are invoked each time a message is sent or received. Inbound messages are processed by handlers before the binding provider processes them. Outbound messages are processed by handlers after the binding provider processes them.

Handlers are invoked with a message context which provides methods to access and modify inbound and outbound messages and to manage a set of properties. Message context properties facilitate communication between individual handlers, as well as between handlers and client and service implementations. Different types of handlers are invoked with different types of message contexts.

Types of Message Handlers

Logical Handler

Logical handlers only operate on message context properties and message payloads. Logical handlers are protocol-independent and cannot affect protocol-specific parts of a message. Logical handlers implement interface `javax.xml.ws.handler.LogicalHandler`.

Protocol Handler

Protocol handlers operate on message context properties and protocol-specific messages. Protocol handlers are specific to a particular protocol and may access and change protocol-specific aspects of a message. Protocol handlers implement any interface derived from `javax.xml.ws.handler.Handler` except `javax.xml.ws.handler.LogicalHandler`.

Service Endpoint Handler

On a service endpoint, handlers are defined using the `@HandlerChain` annotation. The location of the handler chain file can be either an absolute `java.net.URL` in `externalForm` or a relative path from the source file or class file.

Example 16.30. Example Service Endpoint Handler

```
@WebService
@HandlerChain(file = "jaxws-server-source-handlers.xml")
public class SOAPEndpointSourceImpl
{
    ...
}
```

Service Client Handler

On a JAX-WS client, handlers are defined either by using the **@HandlerChain** annotation, as in service endpoints, or dynamically, using the JAX-WS API.

Example 16.31. Defining a Service Client Handler Using the API

```
Service service = Service.create(wsdlURL, serviceName);
Endpoint port = (Endpoint)service.getPort(Endpoint.class);

BindingProvider bindingProvider = (BindingProvider)port;
List<Handler> handlerChain = new ArrayList<Handler>();
handlerChain.add(new LogHandler());
handlerChain.add(new AuthorizationHandler());
handlerChain.add(new RoutingHandler());
bindingProvider.getBinding().setHandlerChain(handlerChain);
```

The call to the **setHandlerChain** method is required.

Message Context

The **MessageContext** interface is the super interface for all JAX-WS message contexts. It extends **Map<String, Object>** with additional methods and constants to manage a set of properties that enable handlers in a handler chain to share processing related state. For example, a handler may use the **put** method to insert a property into the message context. One or more other handlers in the handler chain may subsequently obtain the message via the **get** method.

Properties are scoped as either **APPLICATION** or **HANDLER**. All properties are available to all handlers for an instance of a *message exchange pattern (MEP)* of a particular endpoint. For instance, if a logical handler puts a property into the message context, that property is also available to any protocol handlers in the chain during the execution of an MEP instance.



NOTE

An asynchronous Message Exchange Pattern (MEP) allows for sending and receiving messages asynchronously at the HTTP connection level. You can enable it by setting additional properties in the request context.

Properties scoped at the **APPLICATION** level are also made available to client applications and service endpoint implementations. The **defaultscope** for a property is **HANDLER**.

Logical and SOAP messages use different contexts.

Logical Message Context

When logical handlers are invoked, they receive a message context of type **LogicalMessageContext**. **LogicalMessageContext** extends **MessageContext** with methods which obtain and modify the message payload. It does not provide access to the protocol-specific aspects of a message. A protocol binding defines which components of a message are available via a logical message context. A logical handler deployed in a SOAP binding can access the contents of the SOAP body but not the SOAP headers. On the other hand, the XML/HTTP binding defines that a logical handler can access the entire XML payload of a message.

SOAP Message Context

When SOAP handlers are invoked, they receive a **SOAPMessageContext**. **SOAPMessageContext** extends **MessageContext** with methods which obtain and modify the SOAP message payload.

Fault Handling

An application may throw a **SOAPFaultException** or an application-specific user exception. In the case of the latter, the required fault wrapper beans are generated at run-time if they are not already part of the deployment.

Example 16.32. Fault Handling Examples

```
public void throwSoapFaultException()
{
    SOAPFactory factory = SOAPFactory.newInstance();
    SOAPFault fault = factory.createFault("this is a fault string!", new
    QName("http://foo", "FooCode"));
    fault.setFaultActor("mr. actor");
    fault.addDetail().addChildElement("test");
    throw new SOAPFaultException(fault);
}

public void throwApplicationException() throws UserException
{
    throw new UserException("validation", 123, "Some validation error");
}
```

JAX-WS Annotations

The annotations available via the JAX-WS API are defined in JSR-224, which can be found at <http://www.jcp.org/en/jsr/detail?id=224>. These annotations are in package **javax.xml.ws**.

The annotations available via the JWS API are defined in JSR-181, which can be found at <http://www.jcp.org/en/jsr/detail?id=181>. These annotations are in package **javax.jws**.

[Report a bug](#)

CHAPTER 17. WEBSOCKETS

17.1. ABOUT WEBSOCKETS

The WebSocket protocol provides two way communication between web clients and servers. Communications between clients and the server are event-based, allowing for faster processing and smaller bandwidth compared with polling-based methods. WebSocket is available for use in web applications via a JavaScript API.

A connection is first established between client and server as an HTTP connection. The client then requests a WebSocket connection using the **Upgrade** header. All communications are then full-duplex over the same TCP/IP connection, with minimal data overhead. Because each message does not include unnecessary HTTP header content, Websocket communications require smaller bandwidth. The result is a low latency communications path, suited to applications which require real-time responsiveness.

The JBoss EAP 6 WebSocket implementation provides full dependency injection support for server endpoints, however, it does not provide CDI services for client endpoints. CDI support is limited to that required by the Java EE 6 platform, and as a result, Java EE 7 features such as interceptors on endpoints are not supported.

[Report a bug](#)

17.2. CREATE A WEBSOCKET APPLICATION

A WebSocket application requires the following components and configuration changes:

- A Java client or a WebSocket enabled HTML client. You can verify HTML client browser support at this location: <http://caniuse.com/websockets>
- A WebSocket server endpoint class.
- A **jboss-web.xml** file configured to enable WebSockets.
- Project dependencies configured to declare a dependency on the WebSocket API.
- Enable the **NIOS** connector in the **web** subsystem of the Red Hat JBoss Enterprise Application Platform server configuration file.



NOTE

WebSocket applications require Java Runtime Environment version 7 or greater. Otherwise the WebSocket will not be enabled.

Procedure 17.1. Create the WebSocket Application

The following is a simple example of a WebSocket application. It provides buttons to open a connection, send a message, and close a connection. It does not implement any other functions or include any error handling, which would be required for a real world application.

1. Create the JavaScript HTML client.

The following is an example of a WebSocket client. It contains these JavaScript functions:

- **connect()**: This function creates the WebSocket connection passing the WebSocket URI.

The resource location matches the resource defined in the server endpoint class. This function also intercepts and handles the WebSocket **onopen**, **onmessage**, **onerror**, and **onclose**.

- **sendMessage()**: This function gets the name entered in the form, creates a message, and sends it using a `WebSocket.send()` command.
- **disconnect()**: This function issues the `WebSocket.close()` command.
- **displayMessage()**: This function sets the display message on the page to the value returned by the WebSocket endpoint method.
- **displayStatus()**: This function displays the WebSocket connection status.

```

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <title>WebSocket: Say Hello</title>
    <link rel="stylesheet" type="text/css"
href="resources/css/hello.css" />
    <script type="text/javascript">
      var websocket = null;

      function connect() {
        var wsURI = 'ws://' + window.location.host +
'/jboss-websocket-hello/websocket/helloName';
        websocket = new WebSocket(wsURI);

        websocket.onopen = function() {
          displayStatus('Open');
          document.getElementById('sayHello').disabled =
false;
          displayMessage('Connection is now open. Type a
name and click Say Hello to send a message.');
```

```

        if (websocket !== null) {
            websocket.close();
            websocket = null;
        }
        message.setAttribute("class", "message");
        message.value = 'WebSocket closed.';
        // log the event
    }

    function sendMessage() {
        if (websocket !== null) {
            var content =
document.getElementById('name').value;
            websocket.send(content);
        } else {
            displayMessage('WebSocket connection is not
established. Please click the Open Connection button.', 'error');
        }
    }

    function displayMessage(data, style) {
        var message =
document.getElementById('hellomessage');
        message.setAttribute("class", style);
        message.value = data;
    }

    function displayStatus(status) {
        var currentStatus =
document.getElementById('currentstatus');
        currentStatus.value = status;
    }

</script>
</head>
<body>

    <div>
        <h1>Welcome to JBoss!</h1>
        <div>This is a simple example of a WebSocket
implementation.</div>
        <div id="connect-container">
            <div>
                <fieldset>
                    <legend>Connect or disconnect using
WebSocket :</legend>
                    <input type="button" id="connect"
onclick="connect();" value="Open Connection" />
                    <input type="button" id="disconnect"
onclick="disconnect();" value="Close Connection" />
                </fieldset>
            </div>
            <div>
                <fieldset>
                    <legend>Type your name below. then click
the `Say Hello` button :</legend>

```

```

        <input id="name" type="text" size="40"
style="width: 40%"/>
        <input type="button" id="sayHello"
onclick="sendMessage();" value="Say Hello" disabled="disabled"/>
    </fieldset>
</div>
<div>Current WebSocket Connection Status: <output
id="currentstatus" class="message">Closed</output></div>
<div>
    <output id="hellomessage" />
</div>
</div>
</div>
</body>
</html>

```

2. Create the WebSocket server endpoint.

You can create a WebSocket server endpoint using either of the following methods.

- **Programmatic Endpoint:** The endpoint extends the Endpoint class.
- **Annotated Endpoint:** The endpoint class uses annotations to interact with the WebSocket events. It is simpler to code than the programmatic endpoint

The code example below uses the annotated endpoint approach and handles the following events.

- The **@ServerEndpoint** annotation identifies this class as a WebSocket server endpoint and specifies the path.
- The **@OnOpen** annotation is triggered when the WebSocket connection is opened.
- The **@OnMessage** annotation is triggered when a message is sent to the WebSocket connection.
- The **@OnClose** annotation is triggered when the WebSocket connection is closed.

```

package org.jboss.as.quickstarts.websocket_hello;

import javax.websocket.CloseReason;
import javax.websocket.OnClose;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/websocket/helloName")
public class HelloName {

    @OnMessage
    public String sayHello(String name) {
        System.out.println("Say hello to '" + name + "'");
        return ("Hello" + name);
    }

    @OnOpen

```

```

    public void helloOnOpen(Session session) {
        System.out.println("WebSocket opened: " + session.getId());
    }

    @OnClose
    public void helloOnClose(CloseReason reason) {
        System.out.println("Closing a WebSocket due to " +
            reason.getReasonPhrase());
    }
}

```

3. Configure the `jboss-web.xml` file.

You must create the `<enable-websockets>` element in the application `WEB-INF/jboss-web.xml` and set it to `true`.

```

<?xml version="1.0" encoding="UTF-8"?>
<!--Enable WebSockets -->
<jboss-web>
    <enable-websockets>true</enable-websockets>
</jboss-web>

```

4. Declare the WebSocket API dependency in your project POM file.

If you use Maven, you add the following dependency to the project `pom.xml` file.

```

<dependency>
    <groupId>org.jboss.spec.java.websocket</groupId>
    <artifactId>jboss-websocket-api_1.0_spec</artifactId>
    <version>1.0.0.Final</version>
    <scope>provided</scope>
</dependency>

```

5. Configure the JBoss EAP server.

Configure the `http<connector>` in the `web` subsystem of the server configuration file to use the `NIO2` protocol.

- a. Start the JBoss EAP server.
- b. Launch the Management CLI using the command for your operating system.

For Linux:

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

For Windows:

```
EAP_HOME\bin\jboss-cli.bat --connect
```

- c. To enable the non blocking Java `NIO2` connector protocol in the `web` subsystem of the JBoss EAP server configuration file, type the following command .

```

/subsystem=web/connector=http/write-
attribute(name=protocol,value=org.apache.coyote.http11.Http11NioP
rotocol)

```


For either command, you should see the following result:

```
{
  "outcome" => "success",
  "response-headers" => {
    "operation-requires-reload" => true,
    "process-state" => "reload-required"
  }
}
```

- d. Notify the server to reload the configuration.

```
reload
```

You should see the following result:

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

- e. Review the changes to the JBoss EAP server configuration file. The **web** subsystem should now contain the following XML for the **http <connector>**.

```
<subsystem xmlns="urn:jboss:domain:web:2.1" default-virtual-
server="default-host" native="false">
  <connector name="http"
protocol="org.apache.coyote.http11.Http11NioProtocol"
scheme="http" socket-binding="http"/>
  <virtual-server name="default-host" enable-welcome-
root="true">
    <alias name="localhost"/>
    <alias name="example.com"/>
  </virtual-server>
</subsystem>
```

[Report a bug](#)

CHAPTER 18. APPLICATION SECURITY

18.1. FOUNDATIONAL CONCEPTS

18.1.1. About Encryption

Encryption refers to obfuscating sensitive information by applying mathematical algorithms to it. Encryption is one of the foundations of securing your infrastructure from data breaches, system outages, and other risks.

Encryption can be applied to simple string data, such as passwords. It can also be applied to data communication streams. The HTTPS protocol, for instance, encrypts all data before transferring it from one party to another. If you connect from one server to another using the Secure Shell (SSH) protocol, all of your communication is sent in an encrypted *tunnel*.

[Report a bug](#)

18.1.2. About Security Domains

Security domains are part of the JBoss EAP 6 security subsystem. All security configuration is now managed centrally, by the domain controller of a managed domain, or by the standalone server.

A security domain consists of configurations for authentication, authorization, security mapping, and auditing. It implements *Java Authentication and Authorization Service (JAAS)* declarative security.

Authentication refers to verifying the identity of a user. In security terminology, this user is referred to as a *principal*. Although authentication and authorization are different, many of the included authentication modules also handle authorization.

Authorization is a process by which the server determines if an authenticated user has permission or privileges to access specific resources in the system or operation.

Security mapping refers to the ability to add, modify, or delete information from a principal, role, or attribute before passing the information to your application.

The auditing manager allows you to configure *provider modules* to control the way that security events are reported.

If you use security domains, you can remove all specific security configuration from your application itself. This allows you to change security parameters centrally. One common scenario that benefits from this type of configuration structure is the process of moving applications between testing and production environments.

[Report a bug](#)

18.1.3. About SSL Encryption

Secure Sockets Layer (SSL) encrypts network traffic between two systems. Traffic between the two systems is encrypted using a two-way key, generated during the *handshake* phase of the connection and known only by those two systems.

For secure exchange of the two-way encryption key, SSL makes use of Public Key Infrastructure (PKI), a method of encryption that utilizes a *key pair*. A *key pair* consists of two separate but matching cryptographic keys - a public key and a private key. The public key is shared with others and is used to

encrypt data, and the private key is kept secret and is used to decrypt data that has been encrypted using the public key.

When a client requests a secure connection, a handshake phase takes place before secure communication can begin. During the SSL handshake the server passes its public key to the client in the form of a certificate. The certificate contains the identity of the server (its URL), the public key of the server, and a digital signature that validates the certificate. The client then validates the certificate and makes a decision about whether the certificate is trusted or not. If the certificate is trusted, the client generates the two-way encryption key for the SSL connection, encrypts it using the public key of the server, and sends it back to the server. The server decrypts the two-way encryption key, using its private key, and further communication between the two machines over this connection is encrypted using the two-way encryption key.



WARNING

Red Hat recommends that you explicitly disable SSL in favor of TLSv1.1 or TLSv1.2 in all affected packages.

[Report a bug](#)

18.1.4. About Declarative Security

Declarative security is a method to separate security concerns from your application code by using the container to manage security. The container provides an authorization system based on either file permissions or users, groups, and roles. This approach is usually superior to *programmatic* security, which gives the application itself all of the responsibility for security.

JBoss EAP 6 provides declarative security via security domains.

[Report a bug](#)

18.2. ROLE-BASED SECURITY IN APPLICATIONS

18.2.1. About Application Security

Securing your applications is a multi-faceted and important concern for every application developer. JBoss EAP 6 provides all the tools you need to write secure applications, including the following abilities:

- [Section 18.2.2, “About Authentication”](#)
- [Section 18.2.3, “About Authorization”](#)
- [Section 18.2.4, “About Security Auditing”](#)
- [Section 18.2.5, “About Security Mapping”](#)
- [Section 18.1.4, “About Declarative Security”](#)
- [Section 18.4.2.1, “About EJB Method Permissions”](#)

- [Section 18.4.3.1, “About EJB Security Annotations”](#)

See also [Section 18.2.8, “Use a Security Domain in Your Application”](#).

[Report a bug](#)

18.2.2. About Authentication

Authentication refers to identifying a subject and verifying the authenticity of the identification. The most common authentication mechanism is a username and password combination. Other common authentication mechanisms use shared keys, smart cards, or fingerprints. The outcome of a successful authentication is referred to as a principal, in terms of Java Enterprise Edition declarative security.

JBoss EAP 6 uses a pluggable system of authentication modules to provide flexibility and integration with the authentication systems you already use in your organization. Each security domain may contain one or more configured authentication modules. Each module includes additional configuration parameters to customize its behavior. The easiest way to configure the authentication subsystem is within the web-based management console.

Authentication is not the same as authorization, although they are often linked. Many of the included authentication modules can also handle authorization.

[Report a bug](#)

18.2.3. About Authorization

Authorization is a mechanism for granting or denying access to a resource based on identity. It is implemented as a set of declarative security roles which can be added to principals.

JBoss EAP 6 uses a modular system to configure authorization. Each security domain may contain one or more authorization policies. Each policy has a basic module which defines its behavior. It is configured through specific flags and attributes. The easiest way to configure the authorization subsystem is by using the web-based management console.

Authorization is different from authentication, and usually happens after authentication. Many of the authentication modules also handle authorization.

[Report a bug](#)

18.2.4. About Security Auditing

Security auditing refers to triggering events, such as writing to a log, in response to an event that happens within the security subsystem. Auditing mechanisms are configured as part of a security domain, along with authentication, authorization, and security mapping details.

Auditing uses *provider modules*. You can use one of the included ones, or implement your own.

[Report a bug](#)

18.2.5. About Security Mapping

Security mapping allows you to combine authentication and authorization information after the authentication or authorization happens, but before the information is passed to your application.

You can map principals (authentication), roles (authorization), or credentials (attributes which are not principals or roles).

Role Mapping is used to add, replace, or remove roles to the subject after authentication.

Principal mapping is used to modify a principal after authentication.

Attribute mapping is used to convert attributes from an external system to be used by your application, and vice versa.

[Report a bug](#)

18.2.6. Java Authentication and Authorization Service (JAAS)

Java Authentication and Authorization Service (JAAS) is a security API which consists of a set of Java packages designed for user authentication and authorization. The API is a Java implementation of the standard Pluggable Authentication Modules (PAM) framework. It extends the Java Enterprise Edition access control architecture to support user-based authorization.

In JBoss EAP 6, JAAS only provides declarative role-based security. For more information about declarative security, refer to [Section 18.1.4, “About Declarative Security”](#).

JAAS is independent of any underlying authentication technologies, such as Kerberos or LDAP. You can change your underlying security structure without changing your application. You only need to change the JAAS configuration.

[Report a bug](#)

18.2.7. About Java Authentication and Authorization Service (JAAS)

The security architecture of JBoss EAP 6 is comprised of the security configuration subsystem, and application-specific security configurations which are included in several configuration files within the application.

Domain, Server Group, and Server Specific Configuration

Server groups (in a managed domain) and servers (in a standalone server) include the configuration for security domains. A security domain includes information about a combination of authentication, authorization, mapping, and auditing modules, with configuration details. An application specifies which security domain it requires, by name, in its **jboss-web.xml**.

Application-specific Configuration

Application-specific configuration takes place in one or more of the following four files.

Table 18.1. Application-Specific Configuration Files

File	Description
ejb-jar.xml	The deployment descriptor for an Enterprise JavaBean (EJB) application, located in the META-INF directory of the archive. Use the ejb-jar.xml to specify roles and map them to principals, at the application level. You can also limit specific methods and classes to certain roles. It is also used for other EJB-specific configuration not related to security.

File	Description
web.xml	The deployment descriptor for a Java Enterprise Edition (EE) web application. Use the web.xml to declare the resource and transport constraints for the application, such as limiting the type of HTTP requests that are allowed. You can also configure simple web-based authentication in this file. It is also used for other application-specific configuration not related to security. The security domain the application uses for authentication and authorization is defined in jboss-web.xml .
jboss-ejb3.xml	Contains JBoss-specific extensions to the ejb-jar.xml descriptor.
jboss-web.xml	Contains JBoss-specific extensions to the web.xml descriptor.



NOTE

The **ejb-jar.xml** and **web.xml** are defined in the Java Enterprise Edition (Java EE) specification. The **jboss-ejb3.xml** provides JBoss-specific extensions for the **ejb-jar.xml**, and the **jboss-web.xml** provides JBoss-specific extensions for the **web.xml**.

[Report a bug](#)

18.2.8. Use a Security Domain in Your Application

Overview

To use a security domain in your application, first you need to define the security domain in the server's configuration and then enable it for an application in the application's deployment descriptor. Then you must add the required annotations to the EJB that uses it. This topic covers the steps required to use a security domain in your application.



WARNING

If an application is part of a security domain that uses an authentication cache, user authentications for that application will also be available to other applications in that security domain.

Procedure 18.1. Configure Your Application to Use a Security Domain

1. Define the Security Domain

You need to define the security domain in the server's configuration file, and then enable it for an application in the application's descriptor file.

a. **Configure the security domain in the server's configuration file**

The security domain is configured in the **security** subsystem of the server's configuration file. If the JBoss EAP 6 instance is running in a managed domain, this is the **domain/configuration/domain.xml** file. If the JBoss EAP 6 instance is running as a standalone server, this is the **standalone/configuration/standalone.xml** file.

The **other**, **jboss-web-policy**, and **jboss-ejb-policy** security domains are provided by default in JBoss EAP 6. The following XML example was copied from the **security** subsystem in the server's configuration file.

The **cache-type** attribute of a security domain specifies a cache for faster authentication checks. Allowed values are **default** to use a simple map as the cache, or **infinispan** to use an Infinispan cache.

```
<subsystem xmlns="urn:jboss:domain:security:1.2">
  <security-domains>
    <security-domain name="other" cache-type="default">
      <authentication>
        <login-module code="Remoting" flag="optional">
          <module-option name="password-stacking"
value="useFirstPass"/>
        </login-module>
        <login-module code="RealmDirect"
flag="required">
          <module-option name="password-stacking"
value="useFirstPass"/>
        </login-module>
      </authentication>
    </security-domain>
    <security-domain name="jboss-web-policy" cache-
type="default">
      <authorization>
        <policy-module code="Delegating"
flag="required"/>
      </authorization>
    </security-domain>
    <security-domain name="jboss-ejb-policy" cache-
type="default">
      <authorization>
        <policy-module code="Delegating"
flag="required"/>
      </authorization>
    </security-domain>
  </security-domains>
</subsystem>
```

You can configure additional security domains as needed using the Management Console or CLI.

b. **Enable the security domain in the application's descriptor file**

The security domain is specified in the **<security-domain>** child element of the **<jboss-web>** element in the application's **WEB-INF/jboss-web.xml** file. The following example configures a security domain named **my-domain**.

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

This is only one of many settings which you can specify in the **WEB-INF/jboss-web.xml** descriptor.

2. Add the Required Annotation to the EJB

You configure security in the EJB using the **@SecurityDomain** and **@RolesAllowed** annotations. The following EJB code example limits access to the **other** security domain by users in the **guest** role.

```
package example.ejb3;

import java.security.Principal;

import javax.annotation.Resource;
import javax.annotation.security.RolesAllowed;
import javax.ejb.SessionContext;
import javax.ejb.Stateless;

import org.jboss.ejb3.annotation.SecurityDomain;

/**
 * Simple secured EJB using EJB security annotations
 * Allow access to "other" security domain by users in a "guest"
 * role.
 */
@Stateless
@RolesAllowed({ "guest" })
@SecurityDomain("other")
public class SecuredEJB {

    // Inject the Session Context
    @Resource
    private SessionContext ctx;

    /**
     * Secured EJB method using security annotations
     */
    public String getSecurityInfo() {
        // Session context injected using the resource annotation
        Principal principal = ctx.getCallerPrincipal();
        return principal.toString();
    }
}
```

For more code examples, see the **ejb-security** quickstart in the JBoss EAP 6 Quickstarts bundle, which is available from the Red Hat Customer Portal.

**NOTE**

The security domain for an EJB can also be set using the **jboss-ejb3.xml** deployment descriptor. See [Section 8.8.4, “jboss-ejb3.xml Deployment Descriptor Reference”](#) for details.

Procedure 18.2. Configure JBoss EAP 6 to access custom principal in EJB 3 bean

1. Configure the ApplicationRealm to defer to JAAS:

```
<security-realm name="MyDomainRealm">
  <authentication>
    <jaas name="my-security-domain"/>
  </authentication>
</security-realm>
```

2. Configure the JAAS security-domain to use the custom principal:

```
<security-domain name="my-security-domain" cache-type="default">
  <authentication>
    <login-module code="UsersRoles" flag="required">
      <module-option name="usersProperties"
value="file:///${jboss.server.config.dir}/users.properties"/>
      <module-option name="rolesProperties"
value="file:///${jboss.server.config.dir}/roles.properties"/>
      <module-option name="principalClass"
value="org.jboss.example.CustomPrincipalImpl"/>
    </login-module>
  </authentication>
</security-domain>
```

3. Deploy the custom principal as a JBoss module.
4. Configure the **org.jboss.as.remoting** module (**modules/org/jboss/as/remoting/main/module.xml**) to depend on the module that contains the custom principal:

```
<resources>
  <resource-root path="jboss-as-remoting-7.1.2.Final-redhat-
1.jar"/>
  <!-- Insert resources here -->
</resources>

<dependencies>
  <module name="org.jboss.staxmapper"/>
  <module name="org.jboss.as.controller"/>
  <module name="org.jboss.as.domain-management"/>
  <module name="org.jboss.as.network"/>
  <module name="org.jboss.as.protocol"/>
  <module name="org.jboss.as.server"/>
  <module name="org.jboss.as.security" optional="true"/>
  <module name="org.jboss.as.threads"/>
  <module name="org.jboss.logging"/>
  <module name="org.jboss.modules"/>
```

```

<module name="org.jboss.msc"/>
<module name="org.jboss.remoting3"/>
<module name="org.jboss.sasl"/>
<module name="org.jboss.threads"/>
<module name="org.picketbox" optional="true"/>
<module name="javax.api" />
<module name="org.jboss.example" /> <!--FIXME: dependency on
custom principal added here -->
</dependencies>

```

5. Configure the client to use **org.jboss.ejb.client.naming**, the **jboss-ejb-client.properties** file should look like the following:

```

remote.connections=default
endpoint.name=client-endpoint
remote.connection.default.port=4447
remote.connection.default.host=localhost
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLE
D=false
remote.connection.default.connect.options.org.xnio.Options.SASL_POLI
CY_NOANONYMOUS=false
# The following setting is required when deferring to JAAS
remote.connection.default.connect.options.org.xnio.Options.SASL_POLI
CY_NOPLAINTEXT=false

remote.connection.default.username=admin
remote.connection.default.password=testing

```

[Report a bug](#)

18.2.9. Use Role-Based Security In Servlets

To add security to a servlet, you map each servlet to a URL pattern, and create security constraints on the URL patterns which need to be secured. The security constraints limit access to the URLs to roles. The authentication and authorization are handled by the security domain specified in the WAR's **jboss-web.xml**.

Prerequisites

Before you use role-based security in a servlet, the security domain used to authenticate and authorize access needs to be configured in the JBoss EAP 6 container.

Procedure 18.3. Add Role-Based Security to Servlets

1. **Add mappings between servlets and URL patterns.**

Use **<servlet-mapping>** elements in the **web.xml** to map individual servlets to URL patterns. The following example maps the servlet called **DisplayOpResult** to the URL pattern **/DisplayOpResult**.

```

<servlet-mapping>
  <servlet-name>DisplayOpResult</servlet-name>
  <url-pattern>/DisplayOpResult</url-pattern>
</servlet-mapping>

```

2. Add security constraints to the URL patterns.

To map the URL pattern to a security constraint, use a `<security-constraint>`. The following example constrains access from the URL pattern `/DisplayOpResult` to be accessed by principals with the role `eap_admin`. The role needs to be present in the security domain.

```
<security-constraint>
  <display-name>Restrict access to role eap_admin</display-name>
  <web-resource-collection>
    <web-resource-name>Restrict access to role eap_admin</web-
resource-name>
    <url-pattern>/DisplayOpResult/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>eap_admin</role-name>
  </auth-constraint>
</security-constraint>

<security-role>
  <role-name>eap_admin</role-name>
</security-role>

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

You need to specify the authentication method, which can be any of the following: **BASIC**, **FORM**, **DIGEST**, **CLIENT-CERT**, **SPNEGO**. This example uses **BASIC** authentication.

3. Specify the security domain in the WAR's `jboss-web.xml`

Add the security domain to the WAR's `jboss-web.xml` in order to connect the servlets to the configured security domain, which knows how to authenticate and authorize principals against the security constraints. The following example uses the security domain called `acme_domain`.

```
<jboss-web>
  ...
  <security-domain>acme_domain</security-domain>
  ...
</jboss-web>
```

Example 18.1. Example `web.xml` with Role-Based Security Configured

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0">

  <display-name>Use Role-Based Security In Servlets</display-name>

  <welcome-file-list>
    <welcome-file>/index.jsp</welcome-file>
```

```

</welcome-file-list>

<servlet-mapping>
  <servlet-name>DisplayOpResult</servlet-name>
  <url-pattern>/DisplayOpResult</url-pattern>
</servlet-mapping>

<security-constraint>
  <display-name>Restrict access to role eap_admin</display-name>
  <web-resource-collection>
    <web-resource-name>Restrict access to role eap_admin</web-
resource-name>
    <url-pattern>/DisplayOpResult/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>eap_admin</role-name>
  </auth-constraint>
</security-constraint>

<security-role>
  <role-name>eap_admin</role-name>
</security-role>

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

</web-app>

```

[Report a bug](#)

18.2.10. Use A Third-Party Authentication System In Your Application

You can integrate third-party security systems with JBoss EAP 6. These types of systems are usually token-based. The external system performs the authentication and passes a token back to the Web application through the request headers. This is often referred to as *perimeter authentication*. To configure perimeter authentication in your application, add a custom authentication valve. If you have a valve from a third-party provider, be sure it is in your classpath and follow the examples below, along with the documentation for your third-party authentication module.



NOTE

The location for configuring valves has changed in JBoss EAP 6. There is no longer a **context.xml** deployment descriptor. Valves are configured directly in the **jboss-web.xml** descriptor instead. The **context.xml** is now ignored.

Example 18.2. Basic Authentication Valve

```

<jboss-web>
  <valve>
    <class-
name>org.jboss.security.negotiation.NegotiationAuthenticator</class-

```

```

name>
  </valve>
</jboss-web>

```

This valve is used for Kerberos-based SSO. It also shows the most simple pattern for specifying a third-party authenticator for your Web application.

Example 18.3. Custom Valve With Header Attributes Set

```

<jboss-web>
  <valve>
    <class-
name>org.jboss.web.tomcat.security.GenericHeaderAuthenticator</class-
name>
    <param>
      <param-name>httpHeaderForSSOAuth</param-name>
      <param-value>sm_ssoid,ct-remote-user,HTTP_OBLIX_UID</param-value>
    </param>
    <param>
      <param-name>sessionCookieForSSOAuth</param-name>
      <param-value>SMSESSION,CTSESSION,ObSSOCookie</param-value>
    </param>
  </valve>
</jboss-web>

```

This example shows how to set custom attributes on your valve. The authenticator checks for the presence of the header ID and the session key, and passes them into the JAAS framework which drives the security layer, as the username and password value. You need a custom JAAS login module which can process the username and password and populate the subject with the correct roles. If no header values match the configured values, regular form-based authentication semantics apply.

Writing a Custom Authenticator

Writing your own authenticator is out of scope of this document. However, the following Java code is provided as an example.

Example 18.4. GenericHeaderAuthenticator.java

```

/*
 * JBoss, Home of Professional Open Source.
 * Copyright 2006, Red Hat Middleware LLC, and individual contributors
 * as indicated by the @author tags. See the copyright.txt file in the
 * distribution for a full listing of individual contributors.
 *
 * This is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as
 * published by the Free Software Foundation; either version 2.1 of
 * the License, or (at your option) any later version.
 *
 * This software is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU

```

```

* Lesser General Public License for more details.
*
* You should have received a copy of the GNU Lesser General Public
* License along with this software; if not, write to the Free
* Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA
* 02110-1301 USA, or see the FSF site: http://www.fsf.org.
*/

package org.jboss.web.tomcat.security;

import java.io.IOException;
import java.security.Principal;
import java.util.StringTokenizer;

import javax.management.JMException;
import javax.management.ObjectName;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.catalina.Realm;
import org.apache.catalina.Session;
import org.apache.catalina.authenticator.Constants;
import org.apache.catalina.connector.Request;
import org.apache.catalina.connector.Response;
import org.apache.catalina.deploy.LoginConfig;
import org.jboss.logging.Logger;

import org.jboss.as.web.security.ExtendedFormAuthenticator;

/**
 * JBAS-2283: Provide custom header based authentication support
 *
 * Header Authenticator that deals with userid from the request header
 * Requires
 * two attributes configured on the Tomcat Service - one for the http
 * header
 * denoting the authenticated identity and the other is the SESSION
 * cookie
 *
 * @author <a href="mailto:Anil.Saldhana@jboss.org">Anil Saldhana</a>
 * @author <a href="mailto:sguilhen@redhat.com">Stefan Guilhen</a>
 * @version $Revision$
 * @since Sep 11, 2006
 */
public class GenericHeaderAuthenticator extends
ExtendedFormAuthenticator {
    protected static Logger log = Logger
        .getLogger(GenericHeaderAuthenticator.class);

    protected boolean trace = log.isTraceEnabled();

    // JBAS-4804: GenericHeaderAuthenticator injection of ssoid and
    // sessioncookie name.
    private String httpHeaderForSSOAuth = null;

```

```

private String sessionCookieForSSOAuth = null;

/**
 * <p>
 * Obtain the value of the <code>httpHeaderForSSOAuth</code>
attribute. This
 * attribute is used to indicate the request header ids that have to
be
 * checked in order to retrieve the SSO identity set by a third party
 * security system.
 * </p>
 *
 * @return a <code>String</code> containing the value of the
 *         <code>httpHeaderForSSOAuth</code> attribute.
 */
public String getHttpHeaderForSSOAuth() {
    return httpHeaderForSSOAuth;
}

/**
 * <p>
 * Set the value of the <code>httpHeaderForSSOAuth</code> attribute.
This
 * attribute is used to indicate the request header ids that have to
be
 * checked in order to retrieve the SSO identity set by a third party
 * security system.
 * </p>
 *
 * @param httpHeaderForSSOAuth
 *        a <code>String</code> containing the value of the
 *        <code>httpHeaderForSSOAuth</code> attribute.
 */
public void setHttpHeaderForSSOAuth(String httpHeaderForSSOAuth) {
    this.httpHeaderForSSOAuth = httpHeaderForSSOAuth;
}

/**
 * <p>
 * Obtain the value of the <code>sessionCookieForSSOAuth</code>
attribute.
 * This attribute is used to indicate the names of the SSO cookies
that may
 * be present in the request object.
 * </p>
 *
 * @return a <code>String</code> containing the names (separated by a
 *         <code>','</code>) of the SSO cookies that may have been
set by a
 *         third party security system in the request.
 */
public String getSessionCookieForSSOAuth() {
    return sessionCookieForSSOAuth;
}

/**

```

```

    * <p>
    * Set the value of the <code>sessionCookieForSSOAuth</code>
attribute. This
    * attribute is used to indicate the names of the SSO cookies that may
be
    * present in the request object.
    * </p>
    *
    * @param sessionCookieForSSOAuth
    *         a <code>String</code> containing the names (separated
by a
    *         <code>','</code>) of the SSO cookies that may have been
set by
    *         a third party security system in the request.
    */
    public void setSessionCookieForSSOAuth(String sessionCookieForSSOAuth)
    {
        this.sessionCookieForSSOAuth = sessionCookieForSSOAuth;
    }

    /**
    * <p>
    * Creates an instance of <code>GenericHeaderAuthenticator</code>.
    * </p>
    * </p>
    */
    public GenericHeaderAuthenticator() {
        super();
    }

    public boolean authenticate(Request request, HttpServletResponse
response,
        LoginConfig config) throws IOException {
        log.trace("Authenticating user");

        Principal principal = request.getUserPrincipal();
        if (principal != null) {
            if (trace)
                log.trace("Already authenticated '" + principal.getName() +
""");
            return true;
        }

        Realm realm = context.getRealm();
        Session session = request.getSessionInternal(true);

        String username = getUserId(request);
        String password = getSessionCookie(request);

        // Check if there is sso id as well as sessionkey
        if (username == null || password == null) {
            log.trace("Username is null or password(sessionkey) is
null: fallback to form auth");
            return super.authenticate(request, response, config);
        }
        principal = realm.authenticate(username, password);

```



```

    if (principal == null) {
        forwardToErrorPage(request, response, config);
        return false;
    }

    session.setNote(Constants.SESS_USERNAME_NOTE, username);
    session.setNote(Constants.SESS_PASSWORD_NOTE, password);
    request.setUserPrincipal(principal);

    register(request, response, principal, HttpServletRequest.FORM_AUTH,
        username, password);
    return true;
}

/**
 * Get the username from the request header
 *
 * @param request
 * @return
 */
protected String getUserId(Request request) {
    String ssoId = null;
    // We can have a comma-separated ids
    String ids = "";
    try {
        ids = this.getIdentityHeaderId();
    } catch (JMEException e) {
        if (trace)
            log.trace("getUserId exception", e);
    }
    if (ids == null || ids.length() == 0)
        throw new IllegalStateException(
            "Http headers configuration in tomcat service missing");

    StringTokenizer st = new StringTokenizer(ids, ",");
    while (st.hasMoreTokens()) {
        ssoId = request.getHeader(st.nextToken());
        if (ssoId != null)
            break;
    }
    if (trace)
        log.trace("SSOID-" + ssoId);
    return ssoId;
}

/**
 * Obtain the session cookie from the request
 *
 * @param request
 * @return
 */
protected String getSessionCookie(Request request) {
    Cookie[] cookies = request.getCookies();
    log.trace("Cookies:" + cookies);
    int numCookies = cookies != null ? cookies.length : 0;

```

```

// We can have comma-separated ids
String ids = "";
try {
    ids = this.getSessionCookieId();
    log.trace("Session Cookie Ids=" + ids);
} catch (JMException e) {
    if (trace)
        log.trace("checkSessionCookie exception", e);
}
if (ids == null || ids.length() == 0)
    throw new IllegalStateException(
        "Session cookies configuration in tomcat service missing");

StringTokenizer st = new StringTokenizer(ids, ",");
while (st.hasMoreTokens()) {
    String cookieToken = st.nextToken();
    String val = getCookieValue(cookies, numCookies, cookieToken);
    if (val != null)
        return val;
}
if (trace)
    log.trace("Session Cookie not found");
return null;
}

/**
 * Get the configured header identity id in the tomcat service
 *
 * @return
 * @throws JMException
 */
protected String getIdentityHeaderId() throws JMException {
    if (this.httpHeaderForSSOAuth != null)
        return this.httpHeaderForSSOAuth;
    return (String) mserver.getAttribute(new ObjectName(
        "jboss.web:service=WebServer"), "HttpHeaderForSSOAuth");
}

/**
 * Get the configured session cookie id in the tomcat service
 *
 * @return
 * @throws JMException
 */
protected String getSessionCookieId() throws JMException {
    if (this.sessionCookieForSSOAuth != null)
        return this.sessionCookieForSSOAuth;
    return (String) mserver.getAttribute(new ObjectName(
        "jboss.web:service=WebServer"), "SessionCookieForSSOAuth");
}

/**
 * Get the value of a cookie if the name matches the token
 *
 * @param cookies
 *         array of cookies

```

```

    * @param numCookies
    *         number of cookies in the array
    * @param token
    *         Key
    * @return value of cookie
    */
    protected String getCookieValue(Cookie[] cookies, int numCookies,
        String token) {
        for (int i = 0; i < numCookies; i++) {
            Cookie cookie = cookies[i];
            log.trace("Matching cookieToken:" + token + " with cookie name="
                + cookie.getName());
            if (token.equals(cookie.getName())) {
                if (trace)
                    log.trace("Cookie-" + token + " value=" + cookie.getValue());
                return cookie.getValue();
            }
        }
        return null;
    }
}

```

[Report a bug](#)

18.3. LOGIN MODULES

[Report a bug](#)

18.3.1. Using Modules

JBoss EAP 6 includes several bundled login modules suitable for most user management needs. JBoss EAP 6 can read user information from a relational database, an LDAP server, or flat files. In addition to these core login modules, JBoss EAP 6 provides other login modules that provide user information for very customized needs.

More login modules and their options can be found in Appendix A.1.

[Report a bug](#)

18.3.1.1. Password Stacking

Multiple login modules can be chained together in a stack, with each login module providing both the credentials verification and role assignment during authentication. This works for many use cases, but sometimes credentials verification and role assignment are split across multiple user management stores.

[Section 18.3.1.4, “Ldap Login Module”](#) describes how to combine LDAP and a relational database, allowing a user to be authenticated by either system. Consider the case where users are managed in a central LDAP server but application-specific roles are stored in the application's relational database. The password-stacking module option captures this relationship.

To use password stacking, each login module should set the <module-option> **password-stacking** attribute to **useFirstPass**. If a previous module configured for password stacking has authenticated

the user, all the other stacking modules will consider the user authenticated and only attempt to provide a set of roles for the authorization step.

When **password-stacking** option is set to **useFirstPass**, this module first looks for a shared user name and password under the property names `javax.security.auth.login.name` and `javax.security.auth.login.password` respectively in the login module shared state map.

If found, these properties are used as the principal name and password. If not found, the principal name and password are set by this login module and stored under the property names `javax.security.auth.login.name` and `javax.security.auth.login.password` respectively.



NOTE

When using password stacking, set all modules to be required. This ensures that all modules are considered, and have the chance to contribute roles to the authorization process.

Example 18.5. Password Stacking Sample

This management CLI example shows how password stacking could be used.

```
/subsystem=security/security-
domain=pwdStack/authentication=classic/login-module=Ldap:add( \
  code=Ldap, \
  flag=required, \
  module-options=[ \
    ("password-stacking"=>"useFirstPass"), \
    ... Ldap login module configuration
  ])
/subsystem=security/security-
domain=pwdStack/authentication=classic/login-module=Database:add( \
  code=Database, \
  flag=required, \
  module-options=[ \
    ("password-stacking"=>"useFirstPass"), \
    ... Database login module configuration
  ])
```

[Report a bug](#)

18.3.1.2. Password Hashing

Most login modules must compare a client-supplied password to a password stored in a user management system. These modules generally work with plain text passwords, but can be configured to support hashed passwords to prevent plain text passwords from being stored on the server side.



IMPORTANT

Red Hat JBoss Enterprise Application Platform Common Criteria certified release only supports SHA-256 for password hashing.

Example 18.6. Password Hashing

The following is a login module configuration that assigns unauthenticated users the principal name **nobody** and contains base64-encoded, SHA-256 hashes of the passwords in a **usersb64.properties** file. The **usersb64.properties** file is part of the deployment classpath.

```
/subsystem=security/security-domain=testUsersRoles:add
/subsystem=security/security-
domain=testUsersRoles/authentication=classic:add
/subsystem=security/security-
domain=testUsersRoles/authentication=classic/login-
module=UsersRoles:add( \
    code=UsersRoles, \
    flag=required, \
    module-options=[ \
        ("usersProperties"=>"usersb64.properties"), \
        ("rolesProperties"=>"test-users-roles.properties"), \
        ("unauthenticatedIdentity"=>"nobody"), \
        ("hashAlgorithm"=>"SHA-256"), \
        ("hashEncoding"=>"base64") \
    ])

```

hashAlgorithm

Name of the **java.security.MessageDigest** algorithm to use to hash the password. There is no default so this option must be specified to enable hashing. Typical values are **SHA-256**, **SHA-1** and **MD5**.

hashEncoding

String that specifies one of three encoding types: **base64**, **hex** or **rfc2617**. The default is **base64**.

hashCharset

Encoding character set used to convert the clear text password to a byte array. The platform default encoding is the default.

hashUserPassword

Specifies the hashing algorithm must be applied to the password the user submits. The hashed user password is compared against the value in the login module, which is expected to be a hash of the password. The default is **true**.

hashStorePassword

Specifies the hashing algorithm must be applied to the password stored on the server side. This is used for digest authentication, where the user submits a hash of the user password along with a request-specific tokens from the server to be compare. The hash algorithm (for digest, this would be **rfc2617**) is utilized to compute a server-side hash, which should match the hashed value sent from the client.

If you must generate passwords in code, the **org.jboss.security.auth.spi.Util** class provides a static helper method that will hash a password using the specified encoding. The following example produces a base64-encoded, MD5 hashed password.

```
String hashedPassword = Util.createPasswordHash("SHA-256",
    Util.BASE64_ENCODING, null, null, "password");

```

OpenSSL provides an alternative way to quickly generate hashed passwords at the command-line. The following example also produces a base64-encoded, SHA-256 hashed password. Here the password in plain text - **password** - is piped into the OpenSSL digest function then piped into another OpenSSL function to convert into base64-encoded format.

```
echo -n password | openssl dgst -sha256 -binary | openssl base64
```

In both cases, the hashed version of the password is the same:

XohImNooBHFR00VvjcyPj3NgPQ1qq73WKhHvch0VQtg=. This value must be stored in the users' properties file specified in the security domain - **usersb64.properties** - in the example above.

[Report a bug](#)

18.3.1.3. Unauthenticated Identity

Not all requests are received in an authenticated format. **unauthenticatedIdentity** is a login module configuration option that assigns a specific identity (guest, for example) to requests that are made with no associated authentication information. This can be used to allow unprotected servlets to invoke methods on EJBs that do not require a specific role. Such a principal has no associated roles and so can only access either unsecured EJBs or EJB methods that are associated with the unchecked permission constraint.

- **unauthenticatedIdentity**: This defines the principal name that should be assigned to requests that contain no authentication information.

[Report a bug](#)

18.3.1.4. Ldap Login Module

Ldap login module is a **LoginModule** implementation that authenticates against a Lightweight Directory Access Protocol (LDAP) server. Use the **Ldap** login module if your user name and credentials are stored in an LDAP server that is accessible using a Java Naming and Directory Interface (JNDI) LDAP provider.

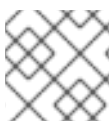


NOTE

If you wish to use LDAP with the SPNEGO authentication or skip some of the authentication phases while using an LDAP server, consider using the **AdvancedLdap** login module chained with the SPNEGO login module or only the **AdvancedLdap** login module.

Distinguished Name (DN)

In Lightweight Directory Access Protocol (LDAP), the distinguished name uniquely identifies an object in a directory. Each distinguished name must have a unique name and location from all other objects, which is achieved using a number of attribute-value pairs (AVPs). The AVPs define information such as common names, organization unit, among others. The combination of these values results in a unique string required by the LDAP.



NOTE

This login module also supports unauthenticated identity and password stacking.

The LDAP connectivity information is provided as configuration options that are passed through to the environment object used to create JNDI initial context. The standard LDAP JNDI properties used include the following:

java.naming.factory.initial

InitialContextFactory implementation class name. This defaults to the Sun LDAP provider implementation **com.sun.jndi.ldap.LdapCtxFactory**.

java.naming.provider.url

LDAP URL for the LDAP server.

java.naming.security.authentication

Security protocol level to use. The available values include **none**, **simple**, and **strong**. If the property is undefined, the behavior is determined by the service provider.

java.naming.security.protocol

Transport protocol to use for secure access. Set this configuration option to the type of service provider (for example, SSL). If the property is undefined, the behavior is determined by the service provider.

java.naming.security.principal

Specifies the identity of the Principal for authenticating the caller to the service. This is built from other properties as described below.

java.naming.security.credentials

Specifies the credentials of the Principal for authenticating the caller to the service. Credentials can take the form of a hashed password, a clear-text password, a key, or a certificate. If the property is undefined, the behavior is determined by the service provider.

For details of Ldap login module configuration options, see the *Included Authentication Modules* reference in the *Security Guide* for JBoss EAP.



NOTE

In certain directory schemas (e.g., Microsoft Active Directory), role attributes in the user object are stored as DN's to role objects instead of simple names. For implementations that use this schema type, `roleAttributesDN` must be set to **true**.

User authentication is performed by connecting to the LDAP server, based on the login module configuration options. Connecting to the LDAP server is done by creating an **InitialLdapContext** with an environment composed of the LDAP JNDI properties described previously in this section.

The `Context.SECURITY_PRINCIPAL` is set to the distinguished name of the user obtained by the callback handler in combination with the `principalDNPrefix` and `principalDNSuffix` option values, and the `Context.SECURITY_CREDENTIALS` property is set to the respective String password.

Once authentication has succeeded (**InitialLdapContext** instance is created), the user's roles are queried by performing a search on the **rolesCtxDN** location with search attributes set to the `roleAttributeName` and `uidAttributeName` option values. The roles names are obtained by invoking the **toString** method on the role attributes in the search result set.

Example 18.7. LDAP Login Module Security Domain

This management CLI example shows how to use the parameters in a security domain authentication configuration.

```
/subsystem=security/security-domain=testLDAP:add(cache-type=default)
/subsystem=security/security-domain=testLDAP/authentication=classic:add
/subsystem=security/security-
domain=testLDAP/authentication=classic/login-module=Ldap:add( \
  code=Ldap, \
  flag=required, \
  module-options=[ \
    ("java.naming.factory.initial"=>"com.sun.jndi.ldap.LdapCtxFactory"),
  \
    ("java.naming.provider.url"=>"ldap://ldaphost.jboss.org:1389/"), \
    ("java.naming.security.authentication"=>"simple"), \
    ("principalDNPrefix"=>"uid="), \
    ("principalDNSuffix"=>","ou=People,dc=jboss,dc=org"), \
    ("rolesCtxDN"=>"ou=Roles,dc=jboss,dc=org"), \
    ("uidAttributeID"=>"member"), \
    ("matchOnUserDN"=>true), \
    ("roleAttributeID"=>"cn"), \
    ("roleAttributeIsDN"=>false) \
  ])
])
```

The *java.naming.factory.initial*, *java.naming.factory.url* and *java.naming.security* options in the testLDAP security domain configuration indicate the following conditions:

- The Sun LDAP JNDI provider implementation will be used
- The LDAP server is located on host **ldaphost.jboss.org** on port 1389
- The LDAP simple authentication method will be use to connect to the LDAP server.

The login module attempts to connect to the LDAP server using a Distinguished Name (DN) representing the user it is trying to authenticate. This DN is constructed from the passed principalDNPrefix, the user name of the user and the principalDNSuffix as described above. In [Example 18.8, “LDIF File Example”](#), the user name **jsmith** would map to **uid=jsmith,ou=People,dc=jboss,dc=org**.



NOTE

The example assumes the LDAP server authenticates users using the **userPassword** attribute of the user's entry (**theduke** in this example). Most LDAP servers operate in this manner, however if your LDAP server handles authentication differently you must ensure LDAP is configured according to your production environment requirements.

Once authentication succeeds, the roles on which authorization will be based are retrieved by performing a subtree search of the **rolesCtxDN** for entries whose **uidAttributeID** match the user. If **matchOnUserDN** is true, the search will be based on the full DN of the user. Otherwise the search will be based on the actual user name entered. In this example, the search is under

ou=Roles,dc=jboss,dc=org for any entries that have a **member** attribute equal to **uid=jsmith,ou=People,dc=jboss,dc=org**. The search would locate **cn=JBossAdmin** under the roles entry.

The search returns the attribute specified in the `roleAttributeID` option. In this example, the attribute is **cn**. The value returned would be **JBossAdmin**, so the **jsmith** user is assigned to the **JBossAdmin** role.

A local LDAP server often provides identity and authentication services, but is unable to use authorization services. This is because application roles do not always map well onto LDAP groups, and LDAP administrators are often hesitant to allow external application-specific data in central LDAP servers. The LDAP authentication module is often paired with another login module, such as the database login module, that can provide roles more suitable to the application being developed.

An LDAP Data Interchange Format (LDIF) file representing the structure of the directory this data operates against is shown in [Example 18.8, “LDIF File Example”](#).

LDAP Data Interchange Format (LDIF)

Plain text data interchange format used to represent LDAP directory content and update requests. Directory content is represented as one record for each object or update request. Content consists of add, modify, delete, and rename requests.

Example 18.8. LDIF File Example

```
dn: dc=jboss,dc=org
objectclass: top
objectclass: dcObject
objectclass: organization
dc: jboss
o: JBoss

dn: ou=People,dc=jboss,dc=org
objectclass: top
objectclass: organizationalUnit
ou: People

dn: uid=jsmith,ou=People,dc=jboss,dc=org
objectclass: top
objectclass: uidObject
objectclass: person
uid: jsmith
cn: John
sn: Smith
userPassword: theduke

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

dn: cn=JBossAdmin,ou=Roles,dc=jboss,dc=org
objectclass: top
objectclass: groupOfNames
cn: JBossAdmin
member: uid=jsmith,ou=People,dc=jboss,dc=org
description: the JBossAdmin group
```

[Report a bug](#)

18.3.1.5. LdapExtended Login Module

Distinguished Name (DN)

In Lightweight Directory Access Protocol (LDAP), the distinguished name uniquely identifies an object in a directory. Each distinguished name must have a unique name and location from all other objects, which is achieved using a number of attribute-value pairs (AVPs). The AVPs define information such as common names, organization unit, among others. The combination of these values results in a unique string required by the LDAP.

The LdapExtended (**`org.jboss.security.auth.spi.LdapExtLoginModule`**) searches for the user to bind, as well as the associated roles, for authentication. The roles query recursively follows DN's to navigate a hierarchical role structure.

The LoginModule options include whatever options are supported by the chosen LDAP JNDI provider supports. Examples of standard property names are:

- `Context.INITIAL_CONTEXT_FACTORY = "java.naming.factory.initial"`
- `Context.SECURITY_PROTOCOL = "java.naming.security.protocol"`
- `Context.PROVIDER_URL = "java.naming.provider.url"`
- `Context.SECURITY_AUTHENTICATION = "java.naming.security.authentication"`
- `Context.REFERRAL = "java.naming.referral"`

Login module implementation logic follows the order below:

1. The initial LDAP server bind is authenticated using the `bindDN` and `bindCredential` properties. The `bindDN` is a user with permissions 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` property.
2. The resulting `userDN` is authenticated by binding to the LDAP server using the `userDN` as the `InitialLdapContext` environment `Context.SECURITY_PRINCIPAL`. The `Context.SECURITY_CREDENTIALS` property is either set to the String password obtained by the callback handler.
3. If this is successful, the associated user roles are queried using the `rolesCtxDN`, `roleAttributeID`, `roleAttributeIsDN`, `roleNameAttributeID`, and `roleFilter` options.

NOTE

AdvancedLdap Login Module differs from LdapExtended Login Module in the following ways:

- The top level role is queried only for roleAttributeID and not for roleNameAttributeID.
- When the roleAttributesDN module property is set to false, the recursive role search is disabled even if the recurseRoles module option is set to true.

For details of LdapExtended login module options, see the *Included Authentication Modules* reference in the *Security Guide* for JBoss EAP.

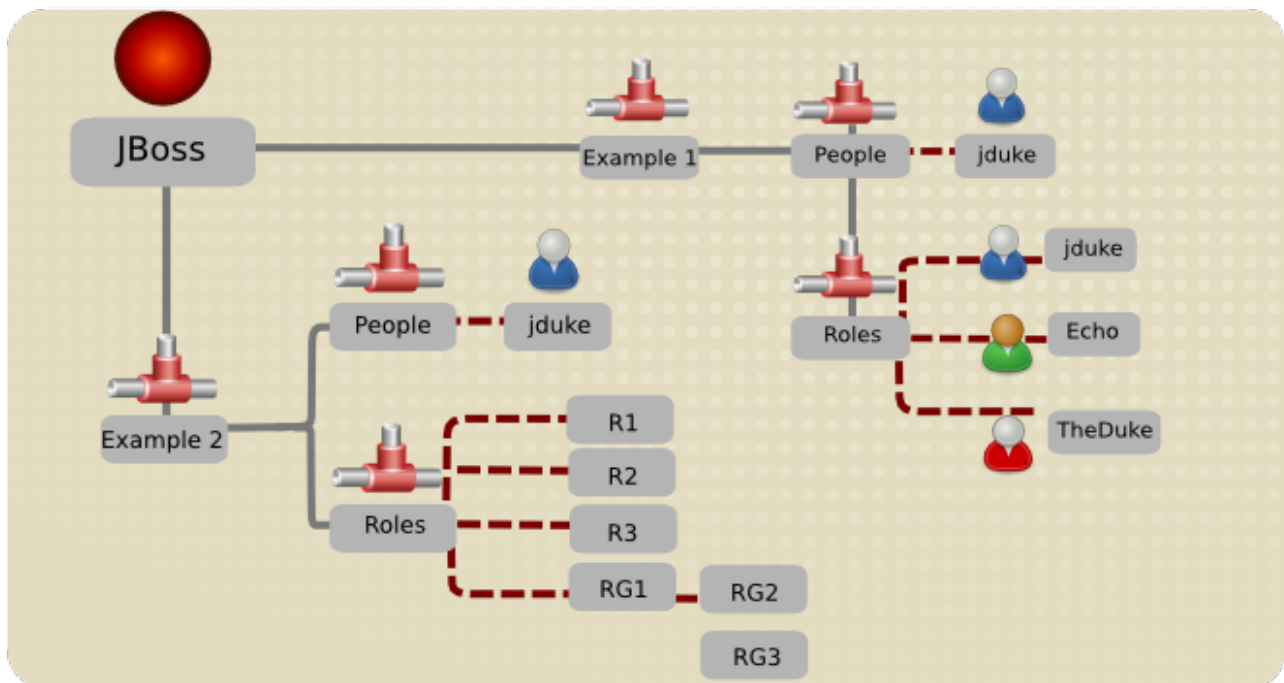


Figure 18.1. LDAP Structure Example

Example 18.9. Example 2 LDAP Configuration

```

version: 1
dn: o=example2,dc=jboss,dc=org
objectClass: top
objectClass: organization
o: example2

dn: ou=People,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: organizationalUnit
ou: People

dn: uid=jduke,ou=People,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: uidObject
objectClass: person
objectClass: inetOrgPerson
cn: Java Duke
  
```

```
employeeNumber: judke-123
sn: Duke
uid: jduke
userPassword:: dGhlZHVrZQ==

dn: uid=jduke2,ou=People,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: uidObject
objectClass: person
objectClass: inetOrgPerson
cn: Java Duke2
employeeNumber: judke2-123
sn: Duke2
uid: jduke2
userPassword:: dGhlZHVrZTI=

dn: ou=Roles,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: organizationalUnit
ou: Roles

dn: uid=jduke,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: groupUserEx
memberOf: cn=Echo,ou=Roles,o=example2,dc=jboss,dc=org
memberOf: cn=TheDuke,ou=Roles,o=example2,dc=jboss,dc=org
uid: jduke

dn: uid=jduke2,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: groupUserEx
memberOf: cn=Echo2,ou=Roles,o=example2,dc=jboss,dc=org
memberOf: cn=TheDuke2,ou=Roles,o=example2,dc=jboss,dc=org
uid: jduke2

dn: cn=Echo,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: groupOfNames
cn: Echo
description: the echo role
member: uid=jduke,ou=People,dc=jboss,dc=org

dn: cn=TheDuke,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: TheDuke
description: the duke role
member: uid=jduke,ou=People,o=example2,dc=jboss,dc=org

dn: cn=Echo2,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: groupOfNames
cn: Echo2
description: the Echo2 role
member: uid=jduke2,ou=People,dc=jboss,dc=org
```

```

dn: cn=TheDuke2,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: TheDuke2
description: the duke2 role
member: uid=jduke2,ou=People,o=example2,dc=jboss,dc=org

dn: cn=JBossAdmin,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: groupOfNames
cn: JBossAdmin
description: the JBossAdmin group
member: uid=jduke,ou=People,dc=jboss,dc=org

```

The module configuration for this LDAP structure example is outlined in the following management CLI command.

```

/subsystem=security/security-
domain=testLdapExample2/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://ldaphost.jboss.org"), \
        ("java.naming.security.authentication"=>"simple"), \
        ("bindDN"=>"cn=Root,dc=jboss,dc=org"), \
        ("bindCredential"=>"secret1"), \
        ("baseCtxDN"=>"ou=People,o=example2,dc=jboss,dc=org"), \
        ("baseFilter"=>"(uid={0})"), \
        ("rolesCtxDN"=>"ou=Roles,o=example2,dc=jboss,dc=org"), \
        ("roleFilter"=>"(uid={0})"), \
        ("roleAttributeIsDN"=>"true"), \
        ("roleAttributeID"=>"memberOf"), \
        ("roleNameAttributeID"=>"cn") \
    ])

```

Example 18.10. Example 3 LDAP Configuration

```

dn: o=example3,dc=jboss,dc=org
objectclass: top
objectclass: organization
o: example3

dn: ou=People,o=example3,dc=jboss,dc=org
objectclass: top
objectclass: organizationalUnit
ou: People

dn: uid=jduke,ou=People,o=example3,dc=jboss,dc=org
objectclass: top

```

```

objectclass: uidObject
objectclass: person
objectClass: inetOrgPerson
uid: jduke
employeeNumber: judke-123
cn: Java Duke
sn: Duke
userPassword: theduke

dn: ou=Roles,o=example3,dc=jboss,dc=org
objectClass: top
objectClass: organizationalUnit
ou: Roles

dn: uid=jduke,ou=Roles,o=example3,dc=jboss,dc=org
objectClass: top
objectClass: groupUserEx
memberOf: cn=Echo,ou=Roles,o=example3,dc=jboss,dc=org
memberOf: cn=TheDuke,ou=Roles,o=example3,dc=jboss,dc=org
uid: jduke

dn: cn=Echo,ou=Roles,o=example3,dc=jboss,dc=org
objectClass: top
objectClass: groupOfNames
cn: Echo
description: the JBossAdmin group
member: uid=jduke,ou=People,o=example3,dc=jboss,dc=org

dn: cn=TheDuke,ou=Roles,o=example3,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: TheDuke
member: uid=jduke,ou=People,o=example3,dc=jboss,dc=org

```

The module configuration for this LDAP structure example is outlined in the following management CLI command.

```

/subsystem=security/security-
domain=testLdapExample3/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://ldaphost.jboss.org"), \
    ("java.naming.security.authentication"=>"simple"), \
    ("bindDN"=>"cn=Root,dc=jboss,dc=org"), \
    ("bindCredential"=>"secret1"), \
    ("baseCtxDN"=>"ou=People,o=example3,dc=jboss,dc=org"), \
    ("baseFilter"=>"(cn={0})"), \
    ("rolesCtxDN"=>"ou=Roles,o=example3,dc=jboss,dc=org"), \
    ("roleFilter"=>"(member={1})"), \
    ("roleAttributeID"=>"cn") \
  ]
)

```

])

Example 18.11. Example 4 LDAP Configuration

```

dn: o=example4,dc=jboss,dc=org
objectclass: top
objectclass: organization
o: example4

dn: ou=People,o=example4,dc=jboss,dc=org
objectclass: top
objectclass: organizationalUnit
ou: People

dn: uid=jduke,ou=People,o=example4,dc=jboss,dc=org
objectClass: top
objectClass: uidObject
objectClass: person
objectClass: inetOrgPerson
cn: Java Duke
employeeNumber: jduke-123
sn: Duke
uid: jduke
userPassword:: dGhlZHVrZQ==

dn: ou=Roles,o=example4,dc=jboss,dc=org
objectClass: top
objectClass: organizationalUnit
ou: Roles

dn: cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: RG1
member: cn=empty

dn: cn=RG2,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: RG2
member: cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org
member: uid=jduke,ou=People,o=example4,dc=jboss,dc=org

dn: cn=RG3,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: RG3
member: cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org

dn: cn=R1,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: R1
member: cn=RG2,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org

```

```

dn: cn=R2,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: R2
member: cn=RG2,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org

dn: cn=R3,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: R3
member: cn=RG2,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org
member: cn=RG3,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org

dn: cn=R4,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: R4
member: cn=RG3,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org

dn: cn=R5,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: R5
member: cn=RG3,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org
member: uid=jduke,ou=People,o=example4,dc=jboss,dc=org

```

The module configuration for this LDAP structure example is outlined in the code sample.

```

/subsystem=security/security-
domain=testLdapExample4/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://ldaphost.jboss.org"), \
    ("java.naming.security.authentication"=>"simple"), \
    ("bindDN"=>"cn=Root,dc=jboss,dc=org"), \
    ("bindCredential"=>"secret1"), \
    ("baseCtxDN"=>"ou=People,o=example4,dc=jboss,dc=org"), \
    ("baseFilter"=>"(cn={0})"), \
    ("rolesCtxDN"=>"ou=Roles,o=example4,dc=jboss,dc=org"), \
    ("roleFilter"=>"(member={1})"), \
    ("roleRecursion"=>"1"), \
    ("roleAttributeID"=>"memberOf") \
  ])

```

Example 18.12. Default Active Directory Configuration

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.

```
/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"=>"JBOSS\searchuser"), \
    ("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") \
  ])
)
```

Example 18.13. Recursive Roles Active Directory Configuration

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.

```
/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"=>"JBOSS\searchuser"), \
    ("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") \
])
```

[Report a bug](#)

18.3.1.6. UsersRoles Login Module

UsersRoles login module is a simple login module that supports multiple users and user roles loaded from Java properties files. The default username-to-password mapping filename is **users.properties** and the default username-to-roles mapping filename is **roles.properties**.

For details of UsersRoles login module options, see the *Included Authentication Modules* reference in the *Security Guide* for JBoss EAP.

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 Java EE deployment (for example, into the **WEB-INF/classes** folder in the **WAR** archive), or into any directory on the server classpath. 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.

Example 18.14. UsersRoles Login Module

```
/subsystem=security/security-domain=ejb3-
sampleapp/authentication=classic/login-module=UsersRoles:add( \
  code=UsersRoles, \
  flag=required, \
  module-options=[ \
    ("usersProperties"=>"ejb3-sampleapp-users.properties"), \
    ("rolesProperties"=>"ejb3-sampleapp-roles.properties") \
  ])
```

In [Example 18.14, “UsersRoles Login Module”](#), the **ejb3-sampleapp-users.properties** file uses a **username=password** format with each user entry on a separate line:

```
username1=password1
username2=password2
...
```

The **ejb3-sampleapp-roles.properties** file referenced in [Example 18.14, “UsersRoles Login Module”](#) uses the pattern **username=role1, role2**, with an optional group name value. For example:

```
username1=role1,role2,...
username1.RoleGroup1=role3,role4,...
username2=role1,role3,...
```

The user name.XXX property name pattern present in **ejb3-sampleapp-roles.properties** is used to assign the user name roles to a particular named group of roles where the **XXX** portion of the property name is the group name. The user name=... form is an abbreviation for user name.Roles=..., where the

Roles group name is the standard name the **JBossAuthorizationManager** expects to contain the roles which define the permissions of users.

The following would be equivalent definitions for the **jduke** user name:

```
jduke=TheDuke,AnimatedCharacter
jduke.Roles=TheDuke,AnimatedCharacter
```

[Report a bug](#)

18.3.1.7. Database Login Module

The **Database** login module is a Java Database Connectivity-based (JDBC) login module that supports authentication and role mapping. Use this login module if you have your user name, password and role information stored in a relational database.



NOTE

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

The **Database** login module is based on two logical tables:

```
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 you 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 notion, 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

```
INSERT INTO Principals VALUES('java', 'echoman')
INSERT INTO Roles VALUES('java', 'Echo', 'Roles')
INSERT INTO Roles VALUES('java', 'caller_java', 'CallerPrincipal')
```

For details of Database login module options, see the *Included Authentication Modules* reference in the *Security Guide* for JBoss EAP.

An example **Database login module** configuration could be constructed as follows:

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

A corresponding login module configuration in a security domain:

```
/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=?")
  \
  ])
```

[Report a bug](#)

18.3.1.8. Certificate Login Module

Certificate login module authenticates users based on X509 certificates. A typical use case for this login module is **CLIENT-CERT** authentication in the web tier.

This login module only performs authentication: you must combine it with another login module capable of acquiring authorization roles to completely define access to a secured web or EJB component. Two subclasses of this login module, **CertRolesLoginModule** and **DatabaseCertLoginModule** extend the behavior to obtain the authorization roles from either a properties file or database.

For details of **Certificate** login module options, see the *Included Authentication Modules* reference in the *Security Guide* for JBoss EAP.

The **Certificate** login module needs a **KeyStore** to perform user validation. This is obtained from a JSSE configuration of linked security domain as shown in the following configuration fragment:

```
/subsystem=security/security-domain=trust-domain:add
/subsystem=security/security-domain=trust-domain/jsse=classic:add( \
  truststore={ \
    password=>pass1234, \
    url=>/home/jbosseap/trusted-clients.jks \
  })

/subsystem=security/security-domain=testCert:add
/subsystem=security/security-domain=testCert/authentication=classic:add
/subsystem=security/security-domain=testCert/authentication=classic/login-
module=Certificate:add( \
  code=Certificate, \
  flag=required, \
  module-options=[ \
    ("securityDomain"=>"trust-domain"), \
  ])
```

Procedure 18.4. Secure Web Applications with Certificates and Role-based Authorization

This procedure describes how to secure a web application, such as the **user-app.war**, using client

certificates and role-based authorization. In this example the **CertificateRoles** login module is used for authentication and authorization. Both the **trusted-clients.keystore** and the **app-roles.properties** require an entry that maps to the principal associated with the client certificate.

By default, the principal is created using the client certificate distinguished name, such as the DN specified in [Example 18.15, “Certificate Example”](#).

1. Declare Resources and Roles

Modify **web.xml** to declare the resources to be secured along with the allowed roles and security domain to be used for authentication and authorization.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Protect App</web-
resource-name>
            <url-pattern>/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>Admin</role-name>
        </auth-constraint>
    </security-constraint>

    <login-config>
        <auth-method>CLIENT-CERT</auth-method>
        <realm-name>Secured area</realm-name>
    </login-config>

    <security-role>
        <role-name>Admin</role-name>
    </security-role>
</web-app>
```

2. Specify the Security Domain

In the **jboss-web.xml** file, specify the required security domain.

```
<jboss-web>
  <security-domain>app-sec-domain</security-domain>
</jboss-web>
```

3. Configure Login Module

Define the login module configuration for the **app-sec-domain** domain you just specified using the management CLI.

```
[
/subsystem=security/security-domain=trust-domain:add
/subsystem=security/security-domain=trust-domain/jsse=classic:add( \
  truststore={ \
    password=>pass1234, \
    url=>/home/jbosseap/trusted-clients.jks \
  })

/subsystem=security/security-domain=app-sec-domain:add
/subsystem=security/security-domain=app-sec-
domain/authentication=classic:add
/subsystem=security/security-domain=app-sec-
domain/authentication=classic/login-module=CertificateRoles:add( \
  code=CertificateRoles, \
  flag=required, \
  module-options=[ \
    ("securityDomain"=>"trust-domain"), \
    ("rolesProperties"=>"app-roles.properties") \
  ])
]
```

Example 18.15. Certificate Example

```
[conf]$ keytool -printcert -file valid-client-cert.crt
Owner: CN=valid-client, OU=Security QE, OU=JBoss, O=Red Hat, C=CZ
Issuer: CN=EAP Certification Authority, OU=Security QE, OU=JBoss, O=Red
Hat, C=CZ
Serial number: 2
Valid from: Mon Mar 24 18:21:55 CET 2014 until: Tue Mar 24 18:21:55 CET
2015
Certificate fingerprints:
    MD5: 0C:54:AE:6E:29:ED:E4:EF:46:B5:14:30:F2:E0:2A:CB
    SHA1:
D6:FB:19:E7:11:28:6C:DE:01:F2:92:2F:22:EF:BB:5D:BF:73:25:3D
    SHA256:
CD:B7:B1:72:A3:02:42:55:A3:1C:30:E1:A6:F0:20:B0:2C:0F:23:4F:7A:8E:2F:2D:
FA:AF:55:3E:A7:9B:2B:F4
    Signature algorithm name: SHA1withRSA
    Version: 3
```

The **trusted-clients.keystore** would need the certificate in [Example 18.15, “Certificate Example”](#) stored with an alias of **CN=valid-client, OU=Security QE, OU=JBoss, O=Red Hat, C=CZ**. The **app-roles.properties** must have the same entry. Since the DN contains characters that are normally treated as delimiters, you must escape the problem characters using a backslash ('\') as illustrated below.

```
# A sample app-roles.properties file
CN\=valid-client,\ OU\=Security\ QE,\ OU\=JBoss,\ O\=Red\ Hat,\ C\=CZ
```

[Report a bug](#)

18.3.1.9. Identity Login Module

Identity login module is a simple login module that associates a hard-coded user name to any subject authenticated against the module. It creates a **SimplePrincipal** instance using the name specified by the **principal** option.



NOTE

This module supports password stacking.

This login module is useful when you need to provide a fixed identity to a service, and in development environments when you want to test the security associated with a given principal and associated roles.

For details of Identity login module options, see the *Included Authentication Modules* reference in the *Security Guide* for JBoss EAP.

A sample security domain configuration is described below. It authenticates all users as the principal named **jduke** and assigns role names of **TheDuke**, and **AnimatedCharacter**:

```
/subsystem=security/security-domain=testIdentity:add
/subsystem=security/security-
domain=testIdentity/authentication=classic:add
/subsystem=security/security-
domain=testIdentity/authentication=classic/login-module=Identity:add( \
  code=Identity, \
  flag=required, \
  module-options=[ \
    ("principal"=>"jduke"), \
    ("roles"=>"TheDuke,AnimatedCharacter") \
  ])
```

[Report a bug](#)

18.3.1.10. RunAs Login Module

RunAs login module is a helper module that pushes a **run as** role onto the stack for the duration of the login phase of authentication, then pops the **run as** role from the stack in either the commit or abort phase.

The purpose of this login module is to provide a role for other login modules that must access secured resources in order to perform their authentication (for example, a login module that accesses a secured EJB). **RunAs** login module must be configured ahead of the login modules that require a **run as** role established.

For details of RunAs login module options, see the *Included Authentication Modules* reference in the *Security Guide* for JBoss EAP.

[Report a bug](#)

18.3.1.10.1. RunAsIdentity Creation

In order for JBoss EAP 6 to secure access to EJB methods, the identity of the user must be known at the time the method call is made.

A user's identity in the server is represented either by a **javax.security.auth.Subject** instance or

an `org.jboss.security.RunAsIdentity` instance. Both these classes store one or more principals that represent the identity and a list of roles that the identity possesses. In the case of the `javax.security.auth.Subject` a list of credentials is also stored.

In the `<assembly-descriptor>` section of the `ejb-jar.xml` deployment descriptor, you specify one or more roles that a user must have to access the various EJB methods. A comparison of these lists reveals whether the user has one of the roles necessary to access the EJB method.

Example 18.16. `org.jboss.security.RunAsIdentity` Creation

In the `ejb-jar.xml` file, you specify a `<security-identity>` element with a `<run-as>` role defined as a child of the `<session>` element.

```
<session>
  ...
  <security-identity>
    <run-as>
      <role-name>Admin</role-name>
    </run-as>
  </security-identity>
  ...
</session>
```

This declaration signifies that an **Admin** `RunAsIdentity` role must be created.

To name a principal for the **Admin** role, you define a `<run-as-principal>` element in the `jboss-ejb3.xml` file.

```
<jboss:ejb-jar
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
  xmlns:s="urn:security:1.1"
  version="3.1" impl-version="2.0">
  <assembly-descriptor>
    <s:security>
      <ejb-name>WhoAmIBean</ejb-name>
      <s:run-as-principal>John</s:run-as-principal>
    </s:security>
  </assembly-descriptor>
</jboss:ejb-jar>
```

The `<security-identity>` element in both the `ejb-jar.xml` and `<security>` element in the `jboss-ejb3.xml` files are parsed at deployment time. The `<run-as>` role name and the `<run-as-principal>` name are then stored in the `org.jboss.metadata.ejb.spec.SecurityIdentityMetaData` class.

Example 18.17. Assigning multiple roles to a `RunAsIdentity`

You can assign more roles to `RunAsIdentity` by mapping roles to principals in the `jboss-ejb3.xml` deployment descriptor `<assembly-descriptor>` element group.


```

<jboss:ejb-jar xmlns:sr="urn:security-role"
...>
  <assembly-descriptor>
    ...
    <sr:security-role>
      <sr:role-name>Support</sr:role-name>
      <sr:principal-name>John</sr:principal-name>
      <sr:principal-name>Jill</sr:principal-name>
      <sr:principal-name>Tony</sr:principal-name>
    </sr:security-role>
  </assembly-descriptor>
</jboss:ejb-jar>

```

In [Example 18.16](#), “[org.jboss.security.RunAsIdentity Creation](#)”, the **<run-as-principal>** of **John** was created. The configuration in this example extends the **Admin** role, by adding the **Support** role. The new role contains extra principals, including the originally defined principal **John**.

The **<security-role>** element in both the **ejb-jar.xml** and **jboss-ejb3.xml** files are parsed at deployment time. The **<role-name>** and the **<principal-name>** data is stored in the **org.jboss.metadata.ejb.spec.SecurityIdentityMetaData** class.

[Report a bug](#)

18.3.1.11. Client Login Module

Client login module (**org.jboss.security.ClientLoginModule**) is an implementation of **LoginModule** for use by JBoss clients when establishing caller identity and credentials. This creates a new **SecurityContext** assigns it a principal and a credential and sets the **SecurityContext** to the **ThreadLocal** security context.

Client login module is the only supported mechanism for a client to establish the current thread's caller. Both stand-alone client applications, and server environments (acting as JBoss EJB clients where the security environment has not been configured to use the EAP security subsystem transparently) must use **Client** login module.

Note that this login module does not perform any authentication. It merely copies the login information provided to it into the server EJB invocation layer for subsequent authentication on the server. If you need to perform client-side authentication of users you would need to configure another login module in addition to the **Client** login module.

For details of Client login module options, see the *Included Authentication Modules* reference in the *Security Guide* for JBoss EAP.

[Report a bug](#)

18.3.1.12. SPNEGO Login Module

SPNEGO login module (**org.jboss.security.negotiation.spnego.SPNEGOLoginModule**) is an implementation of **LoginModule** that establishes caller identity and credentials with a KDC. The module implements SPNEGO (Simple and Protected GSSAPI Negotiation mechanism) and is a part of the JBoss

Negotiation project. This authentication can be used in the chained configuration with the **AdvancedLdap** login module to allow cooperation with an LDAP server.

For details of SPNEGO login module options, see the *Included Authentication Modules* reference in the *Security Guide* for JBoss EAP.

The JBoss Negotiation module is not included as a standard dependency for deployed applications. To use the **SPNEGO** or **AdvancedLdap** login modules in your project, you must add the dependency manually by editing the **META-INF/jboss-deployment-structure.xml** deployment descriptor file.

Example 18.18. Add JBoss Negotiation Module as a Dependency

```
<jboss-deployment-structure>
  <deployment>
    <dependencies>
      <module name="org.jboss.security.negotiation" />
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

[Report a bug](#)

18.3.1.13. RoleMapping Login Module

RoleMapping login module supports mapping roles, that are the end result of the authentication process, to one or more declarative roles. For example, if the authentication process has determined that the user "A" has the roles "ldapAdmin" and "testAdmin", and the declarative role defined in the **web.xml** or **ejb-jar.xml** file for access is **admin**, then this login module maps the **admin** roles to the user A.

For details of **RoleMapping** login module options, see the *Included Authentication Modules* reference in the *Security Guide* for JBoss EAP.

The **RoleMapping** login module must be defined as an optional module to a login module configuration as it alters mapping of the previously mapped roles.

Example 18.19. Defining mapped roles

```
/subsystem=security/security-domain=test-domain-2/:add
/subsystem=security/security-domain=test-domain-2/authentication=classic:add
/subsystem=security/security-domain=test-domain-2/authentication=classic/login-module=test-2-lm/:add(\
flag=required,\
code=UsersRoles,\
module-options=[("usersProperties"=>"users.properties"),\
("rolesProperties"=>"roles.properties")]\
)
/subsystem=security/security-domain=test-domain-2/authentication=classic/login-module=test2-map/:add(\
flag=optional,\
```

```
code=RoleMapping,\
module-options=[("rolesProperties"=>"rolesMapping-roles.properties")]\\
)
```

Another example achieving the same result, but using the mapping module. This is the preferred method of role mapping:

Example 18.20. Preferred method of defining mapped roles

```
/subsystem=security/security-domain=test-domain-2/:add
/subsystem=security/security-domain=test-domain-
2/authentication=classic:add
/subsystem=security/security-domain=test-domain-
2/authentication=classic/login-module=test-2-lm/:add(\\
flag=required,\\
code=UsersRoles,\\
module-options=[("usersProperties"=>"users.properties"),
("rolesProperties"=>"roles.properties")]\\
)
/subsystem=security/security-domain=test-domain-
2/mapping=classic/mapping-module=test2-map/:add(\\
code=PropertiesRoles,type=role,\\
module-options=[("rolesProperties"=>"rolesMapping-roles.properties")]\\
)
```

Example 18.21. Properties File used by a RoleMappingLoginModule

```
ldapAdmin=admin, testAdmin
```

If the authenticated subject contains role **ldapAdmin**, then the roles **admin** and **testAdmin** are added to or substitute the authenticated subject depending on the `replaceRole` property value.

[Report a bug](#)

18.3.1.14. bindCredential Module Option

The **bindCredential** module option is used to store the credentials for the DN and can be used by several login and mapping modules. There are several methods for obtaining the password.

Plaintext in a management CLI command.

The password for the **bindCredential** module may be provided in plaintext, in a management CLI command. For example: (**"bindCredential"=>"secret1"**). For security reasons, the password should be encrypted using the JBoss EAP vault mechanism.

Use an external command.

To obtain the password from the output of an external command, use the format **{EXT} . . .** where the **. . .** is the external command. The first line of the command output is used as the password.

To improve performance, the `{EXTC[:expiration_in_millis]}` variant caches the password for a specified number of milliseconds. By default the cached password does not expire. If the value `0` (zero) is specified, the cached credentials do not expire.

The **EXTC** variant is only supported by the **LdapExtended** login module.

Example 18.22. Obtain a password from an external command

```
{EXT}cat /mysecretpasswordfile
```

Example 18.23. Obtain a password from an external file and cache it for 500 milliseconds

```
{EXTC:500}cat /mysecretpasswordfile
```

[Report a bug](#)

18.3.2. Custom Modules

If the login modules bundled with the EAP security framework do not work with your security environment, you can write your own custom login module implementation. The **AuthenticationManager** requires a particular usage pattern of the **Subject** principals set. You must understand the JAAS Subject class's information storage features and the expected usage of these features to write a login module that works with the **AuthenticationManager**.

This section examines this requirement and introduces two abstract base **LoginModule** implementations that can help you implement custom login modules.

You can obtain security information associated with a **Subject** by using the following methods:

```
java.util.Set getPrincipals()
java.util.Set getPrincipals(java.lang.Class c)
java.util.Set getPrivateCredentials()
java.util.Set getPrivateCredentials(java.lang.Class c)
java.util.Set getPublicCredentials()
java.util.Set getPublicCredentials(java.lang.Class c)
```

For **Subject** identities and roles, EAP has selected the most logical choice: the principals sets obtained via **getPrincipals()** and **getPrincipals(java.lang.Class)**. The usage pattern is as follows:

- User identities (for example; user name, social security number, employee ID) are stored as **java.security.Principal** objects in the **SubjectPrincipals** set. The **Principal** implementation that represents the user identity must base comparisons and equality on the name of the principal. A suitable implementation is available as the **org.jboss.security.SimplePrincipal** class. Other **Principal** instances may be added to the **SubjectPrincipals** set as needed.
- Assigned user roles are also stored in the **Principals** set, and are grouped in named role sets using **java.security.acl.Group** instances. The **Group** interface defines a collection of **Principals** and/or **Groups**, and is a subinterface of **java.security.Principal**.

- Any number of role sets can be assigned to a **Subject**.
- The EAP security framework uses two well-known role sets with the names **Roles** and **CallerPrincipal**.
 - The **Roles** group is the collection of **Principals** for the named roles as known in the application domain under which the **Subject** has been authenticated. This role set is used by methods like the `EJBContext.isCallerInRole(String)`, which EJBs can use to see if the current caller belongs to the named application domain role. The security interceptor logic that performs method permission checks also uses this role set.
 - The **CallerPrincipal Group** consists of the single **Principal** identity assigned to the user in the application domain. The `EJBContext.getCallerPrincipal()` method uses the **CallerPrincipal** to allow the application domain to map from the operation environment identity to a user identity suitable for the application. If a **Subject** does not have a **CallerPrincipal Group**, the application identity is the same as operational environment identity.

[Report a bug](#)

18.3.2.1. Subject Usage Pattern Support

To simplify correct implementation of the **Subject** usage patterns described in [Section 18.3.2, “Custom Modules”](#), EAP includes login modules that populate the authenticated **Subject** with a template pattern that enforces correct **Subject** usage.

AbstractServerLoginModule

The most generic of the two is the `org.jboss.security.auth.spi.AbstractServerLoginModule` class.

It provides an implementation of the `javax.security.auth.spi.LoginModule` interface and offers abstract methods for the key tasks specific to an operation environment security infrastructure. The key details of the class are highlighted in [Example 18.24, “AbstractServerLoginModule Class Fragment”](#). The JavaDoc comments detail the responsibilities of subclasses.



IMPORTANT

The `loginOk` instance variable is pivotal. This must be set to `true` if the log in succeeds, or `false` by any subclasses that override the log in method. If this variable is incorrectly set, the commit method will not correctly update the subject.

Tracking the log in phase outcomes allows login modules to be chained together with control flags. These control flags do not require the login modules to succeed as part of the authentication process.

Example 18.24. AbstractServerLoginModule Class Fragment

```
package org.jboss.security.auth.spi;
/**
 * This class implements the common functionality required for a JAAS
 * server-side LoginModule and implements the PicketBox standard
 * Subject usage pattern of storing identities and roles. Subclass
 * this module to create your own custom LoginModule and override the
 * login(), getRoleSets(), and getIdentity() methods.
```

```

    */
    public abstract class AbstractServerLoginModule
        implements javax.security.auth.spi.LoginModule
    {
        protected Subject subject;
        protected CallbackHandler callbackHandler;
        protected Map sharedState;
        protected Map options;
        protected Logger log;

        /** Flag indicating if the shared credential should be used */
        protected boolean useFirstPass;
        /**
         * Flag indicating if the login phase succeeded. Subclasses that
         * override the login method must set this to true on successful
         * completion of login
         */
        protected boolean loginOk;

        // ...
        /**
         * Initialize the login module. This stores the subject,
         * callbackHandler and sharedState and options for the login
         * session. Subclasses should override if they need to process
         * their own options. A call to super.initialize(...) must be
         * made in the case of an override.
         *
         * <p>
         * The options are checked for the <em>password-stacking</em>
         parameter.
         * If this is set to "useFirstPass", the login identity will be
         taken from the
         * <code>javax.security.auth.login.name</code> value of the
         sharedState map,
         * and the proof of identity from the
         * <code>javax.security.auth.login.password</code> value of the
         sharedState map.
         *
         * @param subject the Subject to update after a successful login.
         * @param callbackHandler the CallbackHandler that will be used to
         obtain the
         * the user identity and credentials.
         * @param sharedState a Map shared between all configured login
         module instances
         * @param options the parameters passed to the login module.
         */
        public void initialize(Subject subject,
                               CallbackHandler callbackHandler,
                               Map sharedState,
                               Map options)
        {
            // ...
        }

        /**

```

```

    * Looks for javax.security.auth.login.name and
    * javax.security.auth.login.password values in the sharedState
    * map if the useFirstPass option was true and returns true if
    * they exist. If they do not or are null this method returns
    * false.
    * Note that subclasses that override the login method
    * must set the loginOk var to true if the login succeeds in
    * order for the commit phase to populate the Subject. This
    * implementation sets loginOk to true if the login() method
    * returns true, otherwise, it sets loginOk to false.
    */
    public boolean login()
        throws LoginException
    {
        // ...
    }

    /**
     * Overridden by subclasses to return the Principal that
     * corresponds to the user primary identity.
     */
    abstract protected Principal getIdentity();

    /**
     * Overridden by subclasses to return the Groups that correspond
     * to the role sets assigned to the user. Subclasses should
     * create at least a Group named "Roles" that contains the roles
     * assigned to the user. A second common group is
     * "CallerPrincipal," which provides the application identity of
     * the user rather than the security domain identity.
     *
     * @return Group[] containing the sets of roles
     */
    abstract protected Group[] getRoleSets() throws LoginException;
}

```

UsernamePasswordLoginModule

The second abstract base login module suitable for custom login modules is the **org.jboss.security.auth.spi.UsernamePasswordLoginModule**.

This login module further simplifies custom login module implementation by enforcing a string-based user name as the user identity and a **char[]** password as the authentication credentials. It also supports the mapping of anonymous users (indicated by a null user name and password) to a principal with no roles. The key details of the class are highlighted in the following class fragment. The JavaDoc comments detail the responsibilities of subclasses.

Example 18.25. UsernamePasswordLoginModule Class Fragment

```

package org.jboss.security.auth.spi;

/**
 * An abstract subclass of AbstractServerLoginModule that imposes a
 * an identity == String username, credentials == String password
 * view on the login process. Subclasses override the

```

```

* getUsersPassword() and getUsersRoles() methods to return the
* expected password and roles for the user.
*/
public abstract class UsernamePasswordLoginModule
    extends AbstractServerLoginModule
{
    /** The login identity */
    private Principal identity;
    /** The proof of login identity */
    private char[] credential;
    /** The principal to use when a null username and password are seen
    */
    private Principal unauthenticatedIdentity;

    /**
     * The message digest algorithm used to hash passwords. If null then
     * plain passwords will be used. */
    private String hashAlgorithm = null;

    /**
     * The name of the charset/encoding to use when converting the
     * password String to a byte array. Default is the platform's
     * default encoding.
     */
    private String hashCharset = null;

    /** The string encoding format to use. Defaults to base64. */
    private String hashEncoding = null;

    // ...

    /**
     * Override the superclass method to look for an
     * unauthenticatedIdentity property. This method first invokes
     * the super version.
     *
     * @param options,
     * @option unauthenticatedIdentity: the name of the principal to
     * assign and authenticate when a null username and password are
     * seen.
     */
    public void initialize(Subject subject,
                          CallbackHandler callbackHandler,
                          Map sharedState,
                          Map options)
    {
        super.initialize(subject, callbackHandler, sharedState,
                        options);
        // Check for unauthenticatedIdentity option.
        Object option = options.get("unauthenticatedIdentity");
        String name = (String) option;
        if (name != null) {
            unauthenticatedIdentity = new SimplePrincipal(name);
        }
    }
}

```



```

// ...

/**
 * A hook that allows subclasses to change the validation of the
 * input password against the expected password. This version
 * checks that neither inputPassword or expectedPassword are null
 * and that inputPassword.equals(expectedPassword) is true;
 *
 * @return true if the inputPassword is valid, false otherwise.
 */
protected boolean validatePassword(String inputPassword,
                                   String expectedPassword)
{
    if (inputPassword == null || expectedPassword == null) {
        return false;
    }
    return inputPassword.equals(expectedPassword);
}

/**
 * Get the expected password for the current username available
 * via the getUsername() method. This is called from within the
 * login() method after the CallbackHandler has returned the
 * username and candidate password.
 *
 * @return the valid password String
 */
abstract protected String getUsersPassword()
    throws LoginException;
}

```

Subclassing Login Modules

The choice of sub-classing the **AbstractServerLoginModule** versus **UsernamePasswordLoginModule** is based on whether a string-based user name and credentials are usable for the authentication technology you are writing the login module for. If the string-based semantic is valid, then subclass **UsernamePasswordLoginModule**, otherwise subclass **AbstractServerLoginModule**.

Subclassing Steps

The steps your custom login module must execute depend on which base login module class you choose. When writing a custom login module that integrates with your security infrastructure, you should start by sub-classing **AbstractServerLoginModule** or **UsernamePasswordLoginModule** to ensure that your login module provides the authenticated **Principal** information in the form expected by the EAP security manager.

When sub-classing the **AbstractServerLoginModule**, you must override the following:

- **void initialize(Subject, CallbackHandler, Map, Map):** if you have custom options to parse.
- **boolean login():** to perform the authentication activity. Be sure to set the **loginOk** instance variable to true if log in succeeds, false if it fails.

- **Principal getIdentity()**: to return the **Principal** object for the user authenticated by the **log()** step.
- **Group[] getRoleSets()**: to return at least one **Group** named **Roles** that contains the roles assigned to the **Principal** authenticated during **login()**. A second common **Group** is named **CallerPrincipal** and provides the user's application identity rather than the security domain identity.

When sub-classing the **UsernamePasswordLoginModule**, you must override the following:

- **void initialize(Subject, CallbackHandler, Map, Map)**: if you have custom options to parse.
- **Group[] getRoleSets()**: to return at least one **Group** named **Roles** that contains the roles assigned to the **Principal** authenticated during **login()**. A second common **Group** is named **CallerPrincipal** and provides the user's application identity rather than the security domain identity.
- **String getUsersPassword()**: to return the expected password for the current user name available via the **getUsername()** method. The **getUsersPassword()** method is called from within **login()** after the **callbackhandler** returns the user name and candidate password.

[Report a bug](#)

18.3.2.2. Custom LoginModule Example

The following information will help you to create a custom Login Module example that extends the **UsernamePasswordLoginModule** and obtains a user's password and role names from a JNDI lookup.

At the end of this section you will have created a custom JNDI context login module that will return a user's password if you perform a lookup on the context using a name of the form **password/<username>** (where **<username>** is the current user being authenticated). Similarly, a lookup of the form **roles/<username>** returns the requested user's roles. In [Example 18.26](#), "[JndiUserAndPassLoginModule Custom Login Module](#)" is the source code for the **JndiUserAndPassLoginModule** custom login module.

Note that because this extends the JBoss **UsernamePasswordLoginModule**, the **JndiUserAndPassLoginModule** obtains the user's password and roles from the JNDI store. The **JndiUserAndPassLoginModule** does not interact with the JAAS LoginModule operations.

Example 18.26. JndiUserAndPassLoginModule Custom Login Module

```
package org.jboss.book.security.ex2;

import java.security.acl.Group;
import java.util.Map;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginException;
import org.jboss.logging.Logger;
import org.jboss.security.SimpleGroup;
import org.jboss.security.SimplePrincipal;
import org.jboss.security.auth.spi.UsernamePasswordLoginModule;
```

```

/**
 * An example custom login module that obtains passwords and roles for a
 * user from a JNDI lookup.
 *
 * @author Scott.Stark@jboss.org
 */
public class JndiUserAndPassLoginModule extends
UsernamePasswordLoginModule {
    /** The JNDI name to the context that handles the password/username
lookup */
    private String userPathPrefix;
    /** The JNDI name to the context that handles the roles/username
lookup */
    private String rolesPathPrefix;
    private static Logger log =
Logger.getLogger(JndiUserAndPassLoginModule.class);
    /**
     * Override to obtain the userPathPrefix and rolesPathPrefix options.
     */
    @Override
    public void initialize(Subject subject, CallbackHandler
callbackHandler, Map sharedState, Map options) {
        super.initialize(subject, callbackHandler, sharedState, options);
        userPathPrefix = (String) options.get("userPathPrefix");
        rolesPathPrefix = (String) options.get("rolesPathPrefix");
    }
    /**
     * Get the roles the current user belongs to by querying the
rolesPathPrefix + '/' + super.getUsername() JNDI location.
     */
    @Override
    protected Group[] getRoleSets() throws LoginException {
        try {
            InitialContext ctx = new InitialContext();
            String rolesPath = rolesPathPrefix + '/' + super.getUsername();
            String[] roles = (String[]) ctx.lookup(rolesPath);
            Group[] groups = { new SimpleGroup("Roles") };
            log.info("Getting roles for user=" + super.getUsername());
            for (int r = 0; r < roles.length; r++) {
                SimplePrincipal role = new SimplePrincipal(roles[r]);
                log.info("Found role=" + roles[r]);
                groups[0].addMember(role);
            }
            return groups;
        } catch (NamingException e) {
            log.error("Failed to obtain groups for user=" +
super.getUsername(), e);
            throw new LoginException(e.toString(true));
        }
    }
    /**
     * Get the password of the current user by querying the userPathPrefix
+ '/' + super.getUsername() JNDI location.
     */
    @Override
    protected String getUsersPassword() throws LoginException {

```

```

    try {
        InitialContext ctx = new InitialContext();
        String userPath = userPathPrefix + '/' + super.getUsername();
        log.info("Getting password for user=" + super.getUsername());
        String passwd = (String) ctx.lookup(userPath);
        log.info("Found password=" + passwd);
        return passwd;
    } catch (NamingException e) {
        log.error("Failed to obtain password for user=" +
super.getUsername(), e);
        throw new LoginException(e.toString(true));
    }
}
}

```

Example 18.27. Definition of security-ex2 security domain with the newly-created custom login module

```

/subsystem=security/security-domain=security-ex2/:add
/subsystem=security/security-domain=security-
ex2/authentication=classic:add
/subsystem=security/security-domain=security-
ex2/authentication=classic/login-module=ex2/:add(\
flag=required,\
code=org.jboss.book.security.ex2.JndiUserAndPassLoginModule,\
module-options=[("userPathPrefix"=>"/security/store/password"),\
("rolesPathPrefix"=>"/security/store/roles")]\
)

```

The choice of using the **JndiUserAndPassLoginModule** custom login module for the server side authentication of the user is determined by the login configuration for the example security domain. The EJB JAR **META-INF/jboss-ejb3.xml** descriptor sets the security domain. For a web application it is part of the **WEB-INF/jboss-web.xml** file.

Example 18.28. jboss-ejb3.xml Example

```

<?xml version="1.0"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
xmlns="http://java.sun.com/xml/ns/javaee" xmlns:s="urn:security"
version="3.1" impl-version="2.0">
  <assembly-descriptor>
    <s:security>
      <ejb-name>*</ejb-name>
      <s:security-domain>security-ex2</s:security-domain>
    </s:security>
  </assembly-descriptor>
</jboss:ejb-jar>

```

Example 18.29. jboss-web.xml example

```
<?xml version="1.0"?>
<jboss-web>
  <security-domain>security-ex2</security-domain>
</jboss-web>
```

[Report a bug](#)

18.4. EJB APPLICATION SECURITY

18.4.1. Security Identity

18.4.1.1. About EJB Security Identity

An EJB can specify an identity to use when invoking methods on other components. This is the EJB's *security identity* (also known as *invocation identity*).

By default, the EJB uses its own caller identity. The identity can alternatively be set to a specific security role. Using specific security roles is useful when you want to construct a segmented security model - for example, restricting access to a set of components to internal EJBs only.

[Report a bug](#)

18.4.1.2. Set the Security Identity of an EJB

The security identity of the EJB is specified through the **<security-identity>** tag in the security configuration.

By default - if no **<security-identity>** tag is present - the EJB's own caller identity is used.

Example 18.30. Set the security identity of an EJB to be the same as its caller

This example sets the security identity for method invocations made by an EJB to be the same as the current caller's identity. This behavior is the default if you do not specify a **<security-identity>** element declaration.

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>ASessionBean</ejb-name>
      <!-- ... -->
      <security-identity>
        <use-caller-identity/>
      </security-identity>
    </session>
    <!-- ... -->
  </enterprise-beans>
</ejb-jar>
```

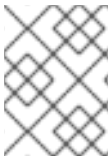
Example 18.31. Set the security identity of an EJB to a specific role

To set the security identity to a specific role, use the `<run-as>` and `<role-name>` tags inside the `<security-identity>` tag.

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>RunAsBean</ejb-name>
      <!-- ... -->
      <security-identity>
        <run-as>
          <description>A private internal role</description>
          <role-name>InternalRole</role-name>
        </run-as>
      </security-identity>
    </session>
  </enterprise-beans>
  <!-- ... -->
</ejb-jar>
```

By default, when you use `<run-as>`, a principal named **anonymous** is assigned to outgoing calls. To assign a different principal, use the `<run-as-principal>`.

```
<session>
  <ejb-name>RunAsBean</ejb-name>
  <security-identity>
    <run-as-principal>internal</run-as-principal>
  </security-identity>
</session>
```



NOTE

You can also use the `<run-as>` and `<run-as-principal>` elements inside a servlet element.

See also:

- [Section 18.4.1.1, “About EJB Security Identity”](#)
- [Section 20.1, “EJB Security Parameter Reference”](#)

[Report a bug](#)

18.4.2. EJB Method Permissions

18.4.2.1. About EJB Method Permissions

EJBs can restrict access to their methods to specific security roles.

The EJB `<method-permission>` element declaration specifies the roles that can invoke the EJB's interface methods. You can specify permissions for the following combinations:

- All home and component interface methods of the named EJB

- A specified method of the home or component interface of the named EJB
- A specified method within a set of methods with an overloaded name

[Report a bug](#)

18.4.2.2. Use EJB Method Permissions

Overview

The `<method-permission>` element defines the logical roles that are allowed to access the EJB methods defined by `<method>` elements. Several examples demonstrate the syntax of the XML. Multiple method permission statements may be present, and they have a cumulative effect. The `<method-permission>` element is a child of the `<assembly-descriptor>` element of the `<ejb-jar>` descriptor.

The XML syntax is an alternative to using annotations for EJB method permissions.

Example 18.32. Allow roles to access all methods of an EJB

```
<method-permission>
  <description>The employee and temp-employee roles may access any
method
of the EmployeeService bean </description>
  <role-name>employee</role-name>
  <role-name>temp-employee</role-name>
  <method>
    <ejb-name>EmployeeService</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
```

Example 18.33. Allow roles to access only specific methods of an EJB, and limiting which method parameters can be passed.

```
<method-permission>
  <description>The employee role may access the findByPrimaryKey,
getEmployeeInfo, and the updateEmployeeInfo(String) method of
the AcmePayroll bean </description>
  <role-name>employee</role-name>
  <method>
    <ejb-name>AcmePayroll</ejb-name>
    <method-name>findByPrimaryKey</method-name>
  </method>
  <method>
    <ejb-name>AcmePayroll</ejb-name>
    <method-name>getEmployeeInfo</method-name>
  </method>
  <method>
    <ejb-name>AcmePayroll</ejb-name>
    <method-name>updateEmployeeInfo</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
```

```

    </method-params>
  </method>
</method-permission>

```

Example 18.34. Allow any authenticated user to access methods of EJBs

Using the `<unchecked/>` element allows any authenticated user to use the specified methods.

```

<method-permission>
  <description>Any authenticated user may access any method of the
  EmployeeServiceHelp bean</description>
  <unchecked/>
  <method>
    <ejb-name>EmployeeServiceHelp</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>

```

Example 18.35. Completely exclude specific EJB methods from being used

```

<exclude-list>
  <description>No fireTheCTO methods of the EmployeeFiring bean may be
  used in this deployment</description>
  <method>
    <ejb-name>EmployeeFiring</ejb-name>
    <method-name>fireTheCTO</method-name>
  </method>
</exclude-list>

```

Example 18.36. A complete `<assembly-descriptor>` containing several `<method-permission>` blocks

```

<ejb-jar>
  <assembly-descriptor>
    <method-permission>
      <description>The employee and temp-employee roles may
access any
      method of the EmployeeService bean </description>
      <role-name>employee</role-name>
      <role-name>temp-employee</role-name>
      <method>
        <ejb-name>EmployeeService</ejb-name>
        <method-name>*</method-name>
      </method>
    </method-permission>
    <method-permission>
      <description>The employee role may access the
findByPrimaryKey,
      getEmployeeInfo, and the updateEmployeeInfo(String)
method of

```



```

        the AcmePayroll bean </description>
<role-name>employee</role-name>
<method>
    <ejb-name>AcmePayroll</ejb-name>
    <method-name>findByPrimaryKey</method-name>
</method>
<method>
    <ejb-name>AcmePayroll</ejb-name>
    <method-name>getEmployeeInfo</method-name>
</method>
<method>
    <ejb-name>AcmePayroll</ejb-name>
    <method-name>updateEmployeeInfo</method-name>
    <method-params>
        <method-param>java.lang.String</method-param>
    </method-params>
</method>
</method-permission>
<method-permission>
    <description>The admin role may access any method of the
        EmployeeServiceAdmin bean </description>
    <role-name>admin</role-name>
    <method>
        <ejb-name>EmployeeServiceAdmin</ejb-name>
        <method-name>*</method-name>
    </method>
</method-permission>
<method-permission>
    <description>Any authenticated user may access any method
of the
        EmployeeServiceHelp bean</description>
    <unchecked/>
    <method>
        <ejb-name>EmployeeServiceHelp</ejb-name>
        <method-name>*</method-name>
    </method>
</method-permission>
<exclude-list>
    <description>No fireTheCTO methods of the EmployeeFiring
bean may be
        used in this deployment</description>
    <method>
        <ejb-name>EmployeeFiring</ejb-name>
        <method-name>fireTheCTO</method-name>
    </method>
</exclude-list>
</assembly-descriptor>
</ejb-jar>

```

[Report a bug](#)

18.4.3. EJB Security Annotations

18.4.3.1. About EJB Security Annotations

EJB `javax.annotation.security` annotations are defined in JSR250.

EJBs use security annotations to pass information about security to the deployer. These include:

@DeclareRoles

Declares which roles are available.

@RunAs

Configures the propagated security identity of a component.

[Report a bug](#)

18.4.3.2. Use EJB Security Annotations

Overview

You can use either XML descriptors or annotations to control which security roles are able to call methods in your Enterprise JavaBeans (EJBs). For information on using XML descriptors, refer to [Section 18.4.2.2, “Use EJB Method Permissions”](#).

Any method values explicitly specified in the deployment descriptor override annotation values. If a method value is not specified in the deployment descriptor, those values set using annotations are used. The overriding granularity is on a per-method basis.

Annotations for Controlling Security Permissions of EJBs

@DeclareRoles

Use `@DeclareRoles` to define which security roles to check permissions against. If no `@DeclareRoles` is present, the list is built automatically from the `@RolesAllowed` annotation. For information about configuring roles, refer to the *Java EE 6 Tutorial* [Specifying Authorized Users by Declaring Security Roles](#).

@RolesAllowed, @PermitAll, @DenyAll

Use `@RolesAllowed` to list which roles are allowed to access a method or methods. Use `@PermitAll` or `@DenyAll` to either permit or deny all roles from using a method or methods. For information about configuring annotation method permissions, refer to the *Java EE 6 Tutorial* [Specifying Authorized Users by Declaring Security Roles](#).

@RunAs

Use `@RunAs` to specify a role a method uses when making calls from the annotated method. For information about configuring propagated security identities using annotations, refer to the *Java EE 6 Tutorial* [Propagating a Security Identity \(Run-As\)](#).

Example 18.37. Security Annotations Example

```
@Stateless
@RolesAllowed({"admin"})
@SecurityDomain("other")
public class WelcomeEJB implements Welcome {
    @PermitAll
    public String WelcomeEveryone(String msg) {
```

```

    return "Welcome to " + msg;
}
@RunAs("tempemployee")
public String GoodBye(String msg) {
    return "Goodbye, " + msg;
}
public String GoodbyeAdmin(String msg) {
    return "See you later, " + msg;
}
}

```

In this code, all roles can access method **WelcomeEveryone**. The **GoodBye** method uses the **tempemployee** role when making calls. Only the **admin** role can access method **GoodbyeAdmin**, and any other methods with no security annotation.

[Report a bug](#)

18.4.4. Remote Access to EJBs

18.4.4.1. About Remote Method Access

JBoss Remoting is the framework which provides remote access to EJBs, JMX MBeans, and other similar services. It works within the following transport types, with or without SSL:

Supported Transport Types

- Socket / Secure Socket
- RMI / RMI over SSL
- HTTP / HTTPS
- Servlet / Secure Servlet
- Bisocket / Secure Bisocket



WARNING

Red Hat recommends that you explicitly disable SSL in favor of TLSv1.1 or TLSv1.2 in all affected packages.

JBoss Remoting also provides automatic discovery via Multicast or JNDI.

It is used by many of the subsystems within JBoss EAP 6, and also enables you to design, implement, and deploy services that can be remotely invoked by clients over several different transport mechanisms. It also allows you to access existing services in JBoss EAP 6.

Data Marshalling

The Remoting system also provides data marshalling and unmarshalling services. Data marshalling refers to the ability to safely move data across network and platform boundaries, so that a separate system can perform work on it. The work is then sent back to the original system and behaves as though it were handled locally.

Architecture Overview

When you design a client application which uses Remoting, you direct your application to communicate with the server by configuring it to use a special type of resource locator called an **InvokerLocator**, which is a simple String with a URL-type format. The server listens for requests for remote resources on a **connector**, which is configured as part of the **remoting** subsystem. The **connector** hands the request off to a configured **ServerInvocationHandler**. Each **ServerInvocationHandler** implements a method **invoke(InvocationRequest)**, which knows how to handle the request.

The JBoss Remoting framework contains three layers that mirror each other on the client and server side.

JBoss Remoting Framework Layers

- The user interacts with the outer layer. On the client side, the outer layer is the **Client** class, which sends invocation requests. On the server side, it is the **InvocationHandler**, which is implemented by the user and receives invocation requests.
- The transport is controlled by the invoker layer.
- The lowest layer contains the marshaller and unmarshaller, which convert data formats to wire formats.

[Report a bug](#)

18.4.4.2. About Remoting Callbacks

When a Remoting client requests information from the server, it can block and wait for the server to reply, but this is often not the ideal behavior. To allow the client to listen for asynchronous events on the server, and continue doing other work while waiting for the server to finish the request, your application can ask the server to send a notification when it has finished. This is referred to as a callback. One client can add itself as a listener for asynchronous events generated on behalf of another client, as well. There are two different choices for how to receive callbacks: pull callbacks or push callbacks. Clients check for pull callbacks synchronously, but passively listen for push callbacks.

In essence, a callback works by the server sending an **InvocationRequest** to the client. Your server-side code works the same regardless of whether the callback is synchronous or asynchronous. Only the client needs to know the difference. The server's **InvocationRequest** sends a **responseObject** to the client. This is the payload that the client has requested. This may be a direct response to a request or an event notification.

Your server also tracks listeners using an **m_listeners** object. It contains a list of all listeners that have been added to your server handler. The **ServerInvocationHandler** interface includes methods that allow you to manage this list.

The client handles pull and push callback in different ways. In either case, it must implement a callback handler. A callback handler is an implementation of interface **org.jboss.remoting.InvokerCallbackHandler**, which processes the callback data. After implementing the callback handler, you either add yourself as a listener for a pull callback, or implement a callback server for a push callback.

Pull Callbacks

For a pull callback, your client adds itself to the server's list of listeners using the **Client.addListener()** method. It then polls the server periodically for synchronous delivery of callback data. This poll is performed using the **Client.getCallbacks()**.

Push Callback

A push callback requires your client application to run its own `InvocationHandler`. To do this, you need to run a Remoting service on the client itself. This is referred to as a *callback server*. The callback server accepts incoming requests asynchronously and processes them for the requester (in this case, the server). To register your client's callback server with the main server, pass the callback server's **InvokerLocator** as the second argument to the **addListener** method.

[Report a bug](#)

18.4.4.3. About Remoting Server Detection

Remoting servers and clients can automatically detect each other using JNDI or Multicast. A Remoting Detector is added to both the client and server, and a `NetworkRegistry` is added to the client.

The Detector on the server side periodically scans the `InvokerRegistry` and pulls all server invokers it has created. It uses this information to publish a detection message which contains the locator and subsystems supported by each server invoker. It publishes this message via a multicast broadcast or a binding into a JNDI server.

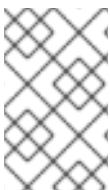
On the client side, the Detector receives the multicast message or periodically polls the JNDI server to retrieve detection messages. If the Detector notices that a detection message is for a newly-detected remoting server, it registers it into the `NetworkRegistry`. The Detector also updates the `NetworkRegistry` if it detects that a server is no longer available.

[Report a bug](#)

18.4.4.4. Configure the Remoting Subsystem

Overview

JBoss Remoting has three top-level configurable elements: the worker thread pool, one or more connectors, and a series of local and remote connection URIs. This topic presents an explanation of each configurable item, example CLI commands for how to configure each item, and an XML example of a fully-configured subsystem. This configuration only applies to the server. Most people will not need to configure the Remoting subsystem at all, unless they use custom connectors for their own applications. Applications which act as Remoting clients, such as EJBs, need separate configuration to connect to a specific connector.



NOTE

The Remoting subsystem configuration is not exposed to the web-based Management Console, but it is fully configurable from the command-line based Management CLI. Editing the XML by hand is not recommended.

Adapting the CLI Commands

The CLI commands are formulated for a managed domain, when configuring the **default** profile. To configure a different profile, substitute its name. For a standalone server, omit the **/profile=default** part of the command.

Configuration Outside the Remoting Subsystem

There are a few configuration aspects which are outside of the **remoting** subsystem:

Network Interface

The network interface used by the **remoting** subsystem is the **public** interface defined in the **domain/configuration/domain.xml** or **standalone/configuration/standalone.xml**.

```
<interfaces>
  <interface name="management"/>
  <interface name="public"/>
  <interface name="unsecure"/>
</interfaces>
```

The per-host definition of the **public** interface is defined in the **host.xml** in the same directory as the **domain.xml** or **standalone.xml**. This interface is also used by several other subsystems. Exercise caution when modifying it.

```
<interfaces>
  <interface name="management">
    <inet-address
value="${jboss.bind.address.management:127.0.0.1}"/>
  </interface>
  <interface name="public">
    <inet-address value="${jboss.bind.address:127.0.0.1}"/>
  </interface>
  <interface name="unsecure">
    <!-- Used for IIOP sockets in the standard configuration.
         To secure JacORB you need to setup SSL -->
    <inet-address value="${jboss.bind.address.unsecure:127.0.0.1}"/>
  </interface>
</interfaces>
```

socket-binding

The default socket-binding used by the **remoting** subsystem binds to TCP port 4447. Refer to the documentation about socket bindings and socket binding groups for more information if you need to change this.

Information about socket binding and socket binding groups can be found in the *Socket Binding Groups* chapter of JBoss EAP's *Administration and Configuration Guide* available at https://access.redhat.com/documentation/en-us/red_hat_jboss_enterprise_application_platform/?version=6.4

Remoting Connector Reference for EJB

The EJB subsystem contains a reference to the remoting connector for remote method invocations. The following is the default configuration:

```
<remote connector-ref="remoting-connector" thread-pool-name="default"/>
```

Secure Transport Configuration

Remoting transports use StartTLS to use a secure (HTTPS, Secure Servlet, etc) connection if the client requests it. The same socket binding (network port) is used for secured and unsecured connections, so no additional server-side configuration is necessary. The client requests the secure or unsecured transport, as its needs dictate. JBoss EAP 6 components which use Remoting, such as EJBs, the ORB, and the JMS provider, request secured interfaces by default.



WARNING

StartTLS works by activating a secure connection if the client requests it, and otherwise defaulting to an unsecured connection. It is inherently susceptible to a *Man in the Middle* style exploit, wherein an attacker intercepts the client's request and modifies it to request an unsecured connection. Clients must be written to fail appropriately if they do not receive a secure connection, unless an unsecured connection actually is an appropriate fall-back.

Worker Thread Pool

The worker thread pool is the group of threads which are available to process work which comes in through the Remoting connectors. It is a single element `<worker-thread-pool>`, and takes several attributes. Tune these attributes if you get network timeouts, run out of threads, or need to limit memory usage. Specific recommendations depend on your specific situation. Contact Red Hat Global Support Services for more information.

Table 18.2. Worker Thread Pool Attributes

Attribute	Description	CLI Command
read-threads	The number of read threads to create for the remoting worker. Defaults to 1 .	<code>/profile=default/subsystem=remoting/:write-attribute(name=worker-read-threads,value=1)</code>
write-threads	The number of write threads to create for the remoting worker. Defaults to 1 .	<code>/profile=default/subsystem=remoting/:write-attribute(name=worker-write-threads,value=1)</code>
task-keepalive	The number of milliseconds to keep non-core remoting worker task threads alive. Defaults to 60 .	<code>/profile=default/subsystem=remoting/:write-attribute(name=worker-task-keepalive,value=60)</code>
task-max-threads	The maximum number of threads for the remoting worker task thread pool. Defaults to 16 .	<code>/profile=default/subsystem=remoting/:write-attribute(name=worker-task-max-threads,value=16)</code>

Attribute	Description	CLI Command
task-core-threads	The number of core threads for the remoting worker task thread pool. Defaults to 4 .	<code>/profile=default/subsystem=remoting/:write-attribute(name=worker-task-core-threads,value=4)</code>
task-limit	The maximum number of remoting worker tasks to allow before rejecting. Defaults to 16384 .	<code>/profile=default/subsystem=remoting/:write-attribute(name=worker-task-limit,value=16384)</code>

Connector

The connector is the main Remoting configuration element. Multiple connectors are allowed. Each consists of a element **<connector>** element with several sub-elements, as well as a few possible attributes. The default connector is used by several subsystems of JBoss EAP 6. Specific settings for the elements and attributes of your custom connectors depend on your applications, so contact Red Hat Global Support Services for more information.

Table 18.3. Connector Attributes

Attribute	Description	CLI Command
socket-binding	The name of the socket binding to use for this connector.	<code>/profile=default/subsystem=remoting/connector=remoting-connector/:write-attribute(name=socket-binding,value=remoting)</code>
authentication-provider	The Java Authentication Service Provider Interface for Containers (JASPIC) module to use with this connector. The module must be in the classpath.	<code>/profile=default/subsystem=remoting/connector=remoting-connector/:write-attribute(name=authentication-provider,value=myProvider)</code>
security-realm	Optional. The security realm which contains your application's users, passwords, and roles. An EJB or Web Application can authenticate against a security realm. ApplicationRealm is available in a default JBoss EAP 6 installation.	<code>/profile=default/subsystem=remoting/connector=remoting-connector/:write-attribute(name=security-realm,value=ApplicationRealm)</code>

Table 18.4. Connector Elements

Attribute	Description	CLI Command
sasl	Enclosing element for Simple Authentication and Security Layer (SASL) authentication mechanisms	N/A
properties	Contains one or more <property> elements, each with a name attribute and an optional value attribute.	<code>/profile=default/subsystem=remoting/connector=remoting-connector/property=myProp/:add(value=myPropValue)</code>

Outbound Connections

You can specify three different types of outbound connection:

- Outbound connection to a URI.
- Local outbound connection – connects to a local resource such as a socket.
- Remote outbound connection – connects to a remote resource and authenticates using a security realm.

All of the outbound connections are enclosed in an **<outbound-connections>** element. Each of these connection types takes an **outbound-socket-binding-ref** attribute. The outbound-connection takes a **uri** attribute. The remote outbound connection takes optional **username** and **security-realm** attributes to use for authorization.

Table 18.5. Outbound Connection Elements

Attribute	Description	CLI Command
outbound-connection	Generic outbound connection.	<code>/profile=default/subsystem=remoting/outbound-connection=my-connection/:add(uri=http://my-connection)</code>
local-outbound-connection	Outbound connection with a implicit local:// URI scheme.	<code>/profile=default/subsystem=remoting/local-outbound-connection=my-connection/:add(outbound-socket-binding-ref=remoting2)</code>
remote-outbound-connection	Outbound connections for remote:// URI scheme, using basic/digest authentication with a security realm.	<code>/profile=default/subsystem=remoting/remote-outbound-connection=my-connection/:add(outbound-socket-binding-ref=remoting,username=myUser,security-realm=ApplicationRealm)</code>

SASL Elements

Before defining the SASL child elements, you need to create the initial SASL element. Use the following command:

```
/profile=default/subsystem=remoting/connector=remoting-
connector/security=sasl:add
```

The child elements of the SASL element are described in the table below.

Table 18.6. SASL child elements

Attribute	Description	CLI Command
include-mechanisms	Contains a value attribute, which is a list of SASL mechanisms.	<pre>/profile=default/sub system=remoting/conn ector=remoting- connector/security=s asl:write- attribute(name=inclu de-mechanisms,value= ["DIGEST","PLAIN","G SSAPI"])</pre>
qop	Contains a value attribute, which is a list of SASL Quality of protection values, in decreasing order of preference.	<pre>/profile=default/sub system=remoting/conn ector=remoting- connector/security=s asl:write- attribute(name=qop,v alue=["auth"])</pre>
strength	Contains a value attribute, which is a list of SASL cipher strength values, in decreasing order of preference.	<pre>/profile=default/sub system=remoting/conn ector=remoting- connector/security=s asl:write- attribute(name=stren gth,value= ["medium"])</pre>

Attribute	Description	CLI Command
reuse-session	Contains a value attribute which is a boolean value. If true, attempt to reuse sessions.	<pre>/profile=default/sub system=remoting/conn ector=remoting- connector/security=s asl:write- attribute(name=reuse - session,value=false)</pre>
server-auth	Contains a value attribute which is a boolean value. If true, the server authenticates to the client.	<pre>/profile=default/sub system=remoting/conn ector=remoting- connector/security=s asl:write- attribute(name=serve r-auth,value=false)</pre>
policy	<p>An enclosing element which contains zero or more of the following elements, which each take a single value.</p> <ul style="list-style-type: none"> • forward-secrecy – whether mechanisms are required to implement forward secrecy (breaking into one session will not automatically provide information for breaking into future sessions) • no-active – whether mechanisms susceptible to non-dictionary attacks are permitted. A value of false permits, and true denies. • no-anonymous – whether mechanisms that accept anonymous login are permitted. A value of false permits, and true denies. • no-dictionary – whether mechanisms susceptible to passive dictionary attacks are allowed. A value of false permits, and true denies. • no-plain-text – whether 	<pre>/profile=default/sub system=remoting/conn ector=remoting- connector/security=s asl/sasl- policy=policy:add /profile=default/sub system=remoting/conn ector=remoting- connector/security=s asl/sasl- policy=policy:write- attribute(name=forwa rd- secrecy,value=true) /profile=default/sub system=remoting/conn ector=remoting- connector/security=s asl/sasl- policy=policy:write- attribute(name=no- active,value=false) /profile=default/sub system=remoting/conn ector=remoting- connector/security=s</pre>

Attribute	Description	CLI Command
	<p>mechanisms which are acceptable to simple plain passive attacks are allowed. A value of false permits, and true denies.</p> <ul style="list-style-type: none"> pass-credentials – whether mechanisms which pass client credentials are allowed. 	<pre>asl/sasl-policy:write-attribute(name=no-anonymous,value=false)</pre> <pre>/profile=default/subsystem=remoting/connector=remoting-connector/security=ssl/sasl-policy=policy:write-attribute(name=no-dictionary,value=true)</pre> <pre>/profile=default/subsystem=remoting/connector=remoting-connector/security=ssl/sasl-policy=policy:write-attribute(name=no-plaintext,value=false)</pre> <pre>/profile=default/subsystem=remoting/connector=remoting-connector/security=ssl/sasl-policy=policy:write-attribute(name=pass-credentials,value=true)</pre>
properties	<p>Contains one or more <property> elements, each with a name attribute and an optional value attribute.</p>	<pre>/profile=default/subsystem=remoting/connector=remoting-connector/security=ssl/property=myprop:add(value=1)</pre> <pre>/profile=default/subsystem=remoting/connector=remoting-connector/security=ssl/property=myprop2:add(value=2)</pre>

Example 18.38. Example Configurations

This example shows the default remoting subsystem that ships with JBoss EAP 6.

```
<subsystem xmlns="urn:jboss:domain:remoting:1.1">
  <connector name="remoting-connector" socket-binding="remoting"
security-realm="ApplicationRealm"/>
</subsystem>
```

This example contains many hypothetical values, and is presented to put the elements and attributes discussed previously into context.

```
<subsystem xmlns="urn:jboss:domain:remoting:1.1">
  <worker-thread-pool read-threads="1" task-keepalive="60" task-max-
threads="16" task-core-thread="4" task-limit="16384" write-threads="1"
/>
  <connector name="remoting-connector" socket-binding="remoting"
security-realm="ApplicationRealm">
    <sasl>
      <include-mechanisms value="GSSAPI PLAIN DIGEST-MD5" />
      <qop value="auth" />
      <strength value="medium" />
      <reuse-session value="false" />
      <server-auth value="false" />
      <policy>
        <forward-secrecy value="true" />
        <no-active value="false" />
        <no-anonymous value="false" />
        <no-dictionary value="true" />
        <no-plain-text value="false" />
        <pass-credentials value="true" />
      </policy>
      <properties>
        <property name="myprop1" value="1" />
        <property name="myprop2" value="2" />
      </properties>
    </sasl>
    <authentication-provider name="myprovider" />
    <properties>
      <property name="myprop3" value="propValue" />
    </properties>
  </connector>
  <outbound-connections>
    <outbound-connection name="my-outbound-connection"
uri="http://myhost:7777"/>
    <remote-outbound-connection name="my-remote-connection"
outbound-socket-binding-ref="my-remote-socket" username="myUser"
security-realm="ApplicationRealm"/>
    <local-outbound-connection name="myLocalConnection" outbound-
socket-binding-ref="my-outbound-socket"/>
  </outbound-connections>
</subsystem>
```

Configuration Aspects Not Yet Documented

- JNDI and Multicast Automatic Detection

[Report a bug](#)

18.4.4.5. Use Security Realms with Remote EJB Clients

One way to add security to clients which invoke EJBs remotely is to use security realms. A security realm is a simple database of username/password pairs and username/role pairs. The terminology is also used in the context of web containers, with a slightly different meaning.

To authenticate a specific username/password pair that exists in a security realm against an EJB, follow these steps:

- Add a new security realm to the domain controller or standalone server.
- Add the following parameters to the **jboss-ejb-client.properties** file, which is in the classpath of the application. This example assumes the connection is referred to as **default** by the other parameters in the file.

```
remote.connection.default.username=appuser
remote.connection.default.password=apppassword
```

- Create a custom Remoting connector on the domain or standalone server, which uses your new security realm.
- Deploy your EJB to the server group which is configured to use the profile with the custom Remoting connector, or to your standalone server if you are not using a managed domain.

[Report a bug](#)

18.4.4.6. Add a New Security Realm

1. Run the Management CLI.

Start the **jboss-cli.sh** or **jboss-cli.bat** command and connect to the server.

2. Create the new security realm itself.

Run the following command to create a new security realm named **MyDomainRealm** on a domain controller or a standalone server.

For a domain instance, use this command:

```
/host=master/core-service=management/security-
realm=MyDomainRealm:add()
```

For a standalone instance, use this command:

```
/core-service=management/security-realm=MyDomainRealm:add()
```

3. Create the references to the properties file which will store information about the new role.

Run the following command to create a pointer a file named **myfile.properties**, which will contain the properties pertaining to the new role.

**NOTE**

The newly created properties file is not managed by the included **add-user.sh** and **add-user.bat** scripts. It must be managed externally.

For a domain instance, use this command:

```
/host=master/core-service=management/security-  
realm=MyDomainRealm/authentication=properties:add(path=myfile.proper  
ties)
```

For a standalone instance, use this command:

```
/core-service=management/security-  
realm=MyDomainRealm/authentication=properties:add(path=myfile.proper  
ties)
```

Result

Your new security realm is created. When you add users and roles to this new realm, the information will be stored in a separate file from the default security realms. You can manage this new file using your own applications or procedures.

[Report a bug](#)

18.4.4.7. Add a User to a Security Realm

1. **Run the `add-user.sh` or `add-user.bat` command.**

Open a terminal and change directories to the **EAP_HOME/bin/** directory. If you run Red Hat Enterprise Linux or another UNIX-like operating system, run **add-user.sh**. If you run Microsoft Windows Server, run **add-user.bat**.

2. **Choose whether to add a Management User or Application User.**

For this procedure, type **b** to add an Application User.

3. **Choose the realm the user will be added to.**

By default, the only available realm is **ApplicationRealm**. If you have added a custom realm, you can type its name instead.

4. **Type the username, password, and roles, when prompted.**

Type the desired username, password, and optional roles when prompted. Verify your choice by typing **yes**, or type **no** to cancel the changes. The changes are written to each of the properties files for the security realm.

[Report a bug](#)

18.4.4.8. About Remote EJB Access Using SSL Encryption

By default, the network traffic for Remote Method Invocation (RMI) of EJB2 and EJB3 Beans is not encrypted. In instances where encryption is required, Secure Sockets Layer (SSL) can be utilized so that the connection between the client and server is encrypted. Using SSL also has the added benefit of allowing the network traffic to traverse some firewalls, depending on the firewall configuration.

**WARNING**

Red Hat recommends that you explicitly disable SSL in favor of TLSv1.1 or TLSv1.2 in all affected packages.

[Report a bug](#)

18.5. JAX-RS APPLICATION SECURITY

18.5.1. Enable Role-Based Security for a RESTEasy JAX-RS Web Service

Summary

RESTEasy supports the `@RolesAllowed`, `@PermitAll`, and `@DenyAll` annotations on JAX-RS methods. However, it does not recognize these annotations by default. Follow these steps to configure the `web.xml` file and enable role-based security.

**WARNING**

Changing the default values of the following RESTEasy parameters may cause RESTEasy applications to be potentially vulnerable against XXE attacks.

- `resteasy.document.expand.entity.references`
- `resteasy.document.secure.processing.feature`
- `resteasy.document.secure.disableDTDs`

For more information about these parameters, see [Section 15.5.1, “RESTEasy Configuration Parameters”](#).

**WARNING**

Do not activate role-based security if the application uses EJBs. The EJB container will provide the functionality, instead of RESTEasy.

Procedure 18.5. Enable Role-Based Security for a RESTEasy JAX-RS Web Service

1. Open the `web.xml` file for the application in a text editor.

2. Add the following `<context-param>` to the file, within the **web-app** tags:

```
<context-param>
  <param-name>resteasy.role.based.security</param-name>
  <param-value>true</param-value>
</context-param>
```

3. Declare all roles used within the RESTEasy JAX-RS WAR file, using the `<security-role>` tags:

```
<security-role>
  <role-name>ROLE_NAME</role-name>
</security-role>
<security-role>
  <role-name>ROLE_NAME</role-name>
</security-role>
```

4. Authorize access to all URLs handled by the JAX-RS runtime for all roles:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Resteasy</web-resource-name>
    <url-pattern>/PATH</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>ROLE_NAME</role-name>
    <role-name>ROLE_NAME</role-name>
  </auth-constraint>
</security-constraint>
```

Result

Role-based security has been enabled within the application, with a set of defined roles.

Example 18.39. Example Role-Based Security Configuration

```
<web-app>

  <context-param>
    <param-name>resteasy.role.based.security</param-name>
    <param-value>true</param-value>
  </context-param>

  <servlet-mapping>
    <servlet-name>Resteasy</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Resteasy</web-resource-name>
      <url-pattern>/security</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>admin</role-name>
      <role-name>user</role-name>
```

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

  <security-role>
<role-name>admin</role-name>
  </security-role>
  <security-role>
<role-name>user</role-name>
  </security-role>

</web-app>
```

[Report a bug](#)

18.5.2. Secure a JAX-RS Web Service using Annotations

Summary

This topic covers the steps to secure a JAX-RS web service using the supported security annotations

Procedure 18.6. Secure a JAX-RS Web Service using Supported Security Annotations

1. Enable role-based security. For more information, refer to: [Section 18.5.1, “Enable Role-Based Security for a RESTEasy JAX-RS Web Service”](#)
2. Add security annotations to the JAX-RS web service. RESTEasy supports the following annotations:

@RolesAllowed

Defines which roles can access the method. All roles should be defined in the **web.xml** file.

@PermitAll

Allows all roles defined in the **web.xml** file to access the method.

@DenyAll

Denies all access to the method.

[Report a bug](#)

18.6. PASSWORD VAULTS FOR SENSITIVE STRINGS

18.6.1. Password Vault System

Configuration of JBoss EAP and associated applications requires potentially sensitive information, such as usernames and passwords.

The Password Vault provides a feature to mask the password information and store it in an encrypted keystore. You can include references of the encrypted keystore in Management CLI commands or applications. The Password Vault uses the Java Keystore as its storage mechanism. The Password Vault consists of two parts: storage and key storage. Java Keystore is used to store the key, which is used to encrypt or decrypt sensitive strings in Vault storage.

[Report a bug](#)

18.6.2. Configure and Use Password Vault

The masked keystore password feature provided in Password Vault provides the option to obtain the masked keystore password from Password Vault, which is stored on the JBoss EAP server. The Password Vault uses the Java Keystore as its storage mechanism.

Procedure 18.7. Basic steps to configure and use Password Vault

1. Setup a Java Keystore to store key for password encryption.

For information on creating a keystore, refer [Section 18.6.4, “Create a Java Keystore to Store Sensitive Strings”](#).

2. Initialize the Password Vault.

For information on masking the password and initialize the password vault, refer [Section 18.6.5, “Initialize the Password Vault”](#).

3. Store a Sensitive String in the Password Vault.

For information on storing sensitive string in Password Vault, refer [Section 18.6.8, “Store a Sensitive String in the Password Vault”](#).

4. Configure JBoss EAP 6 to use the Password Vault.

For information on configuring JBoss EAP 6 to use the Password Vault, refer [Section 18.6.6, “Configure JBoss EAP 6 to Use the Password Vault”](#). For custom implementation, refer [Section 18.6.7, “Configure JBoss EAP 6 to Use a Custom Implementation of the Password Vault”](#).



NOTE

To use an encrypted sensitive string in configuration, refer [Section 18.6.9, “Use an Encrypted Sensitive String in Configuration”](#).

To use an encrypted sensitive string in an application, refer [Section 18.6.10, “Use an Encrypted Sensitive String in an Application”](#).

To verify a sensitive string in Password Vault, refer [Section 18.6.11, “Check if a Sensitive String is in the Password Vault”](#).

To remove a sensitive string from Password Vault, refer [Section 18.6.12, “Remove a Sensitive String from the Password Vault”](#).

[Report a bug](#)

18.6.3. Obtain Keystore Password From External Source

You can also use the EXT, EXTC, CMD, CMDC or CLASS methods in Vault configuration for obtaining the Java keystore password.

```
<vault-option name="KEYSTORE_PASSWORD" value="[here]"
```

The description for the methods are listed as:

- **{EXT}...**: Refers to the exact command, where '...' is the exact command. For example: **{EXT}/usr/bin/getmypassword --section 1 --query company**, run the **/usr/bin/getmypassword** command, which displays the password on standard output and use it as password for Security Vault's keystore. In this example, the command is using two options: **--section 1** and **--query company**.
- **{EXTC[:expiration_in_millis]}...**: Refers to the exact command, where the '...' is the exact command line that is passed to the `Runtime.exec(String)` method to execute a platform command. The first line of the command output is used as the password. EXTC variant caches the passwords for `expiration_in_millis` milliseconds. Default cache expiration is 0 (zero), meaning items in the cache never expire. For example:
{EXTC:120000}/usr/bin/getmypassword --section 1 --query company Verify if cache contains **/usr/bin/getmypassword** output, if it contains the output then use it. If it does not contain the output, run the command to output it to cache and use it. In this example, the cache expires in 2 minute (120000 milliseconds).
- **{CMD}...** or **{CMDC[:expiration_in_millis]}...**: The general command is a string delimited by ',' where the first part is the actual command and further parts represents the parameters. The comma can be backslashed to keep it as a part of the parameter. For example, **{CMD}/usr/bin/getmypassword, --section,1, --query,company**
- **{CLASS[@jboss_module_spec]}classname[:ctorargs]**: Where the '[:ctorargs]' is an optional string delimited by the ':' from the classname is passed to the classname ctor. The ctorargs is a comma delimited list of strings. For example, **{CLASS@org.test.passwd}org.test.passwd.ExternamPassworProvider**. In this example, we load **org.test.passwd.ExternamPassworProvider** class from **org.test.passwd** module and use the **toCharArray()** method to get the password. If **toCharArray()** is not available use **toString()** method. The **org.test.passwd.ExternamPassworProvider** class must have the default constructor.

[Report a bug](#)

18.6.4. Create a Java Keystore to Store Sensitive Strings

Prerequisites

- The **keytool** utility, provided by the Java Runtime Environment (JRE). Locate the path for the file, which on Red Hat Enterprise Linux is **/usr/bin/keytool**.



WARNING

JCEKS keystore implementations differ between Java vendors so you must generate the keystore using the **keytool** utility from the same vendor as the Java development kit you use.

Using a keystore generated by the **keytool** from one vendor's Java development kit in a JBoss EAP instance running on a Java development kit from a different vendor results in the following exception:

```
java.io.IOException:
com.sun.crypto.provider.SealedObjectForKeyProtector
```

Procedure 18.8. Set up a Java Keystore

1. Create a directory to store your keystore and other encrypted information.

Create a directory to store your keystore and other important information. The rest of this procedure assumes that the directory is **EAP_HOME/vault/**. Since this directory will contain sensitive information it should be accessible to only limited users. At a minimum the user account under which JBoss EAP is running requires read-write access.

2. Determine the parameters to use with **keytool** utility.

Decide on values for the following parameters:

alias

The alias is a unique identifier for the vault or other data stored in the keystore. Aliases are case-insensitive.

storetype

The storetype specifies the keystore type. The value **jceks** is recommended.

keyalg

The algorithm to use for encryption. Use the documentation for your JRE and operating system to see which other choices may be available to you.

keysize

The size of an encryption key impacts how difficult it is to decrypt through brute force. For information on appropriate values, see the documentation distributed with the **keytool** utility.

storepass

The value of **storepass** is the password is used to authenticate to the keystore so that the key can be read. The password must be at least 6 characters long and must be provided when the keystore is accessed. If you omit this parameter, you will be prompted to enter it when you execute the command.

keypass

The value of **keypass** is the password used to access the specific key and must match the value of the **storepass** parameter.

validity

The value of **validity** is the period (in days) for which the key will be valid.

keystore

The value of **keystore** is the filepath and filename in which the keystore's values are to be stored. The keystore file is created when data is first added to it.

Ensure you use the correct file path separator: / (forward slash) for Red Hat Enterprise Linux and similar operating systems, \ (backslash) for Microsoft Windows Server.

The **keytool** utility has many other options. See the documentation for your JRE or your operating system for more details.

3. Run the keytool command

Launch your operating system's command line interface and run the **keytool** utility, supplying the information that you gathered.

Example 18.40. Create a Java Keystore

```
$ keytool -genseckey -alias vault -storetype jceks -keyalg AES -keysize  
128 -storepass vault22 -keypass vault22 -validity 730 -keystore  
EAP_HOME/vault/vault.keystore
```

Result

In this a keystore has been created in the file **EAP_HOME/vault/vault.keystore**. It stores a single key, with the alias **vault**, which will be used to store encrypted strings, such as passwords, for JBoss EAP.

[Report a bug](#)

18.6.5. Initialize the Password Vault**Prerequisites**

- [Section 18.6.4, “Create a Java Keystore to Store Sensitive Strings”](#)

Overview

The Password Vault can be initialized either interactively, where you are prompted for each parameter's value, or non-interactively, where you provide all parameters' values on the command line. Each method gives the same result, so choose whichever method you prefer.

Refer to the following list when using either method.

Parameter Values**Keystore URL (KEYSTORE_URL)**

The file system path or URI of the keystore file. The examples use ***EAP_HOME/vault/vault.keystore***.

Keystore password (KEYSTORE_PASSWORD)

The password used to access the keystore.

Salt (SALT)

The **salt** value is a random string of eight characters used, together with the iteration count, to encrypt the content of the keystore.

Keystore Alias (KEYSTORE_ALIAS)

The alias by which the keystore is known.

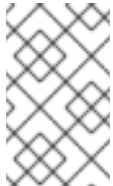
Iteration Count (ITERATION_COUNT)

The number of times the encryption algorithm is run.

Directory to store encrypted files (ENC_FILE_DIR)

The path in which the encrypted files are to be stored. This is typically the directory containing the password vault.

It is convenient but not mandatory to store all of your encrypted information in the same place as the key store. This directory should be only accessible to limited users. At a minimum the user account under which JBoss EAP is running requires read-write access. If you followed [Section 18.6.4, “Create a Java Keystore to Store Sensitive Strings”](#), your keystore is in a directory called ***EAP_HOME/vault/***.



NOTE

The trailing backslash or forward slash on the directory name is required. Ensure you use the correct file path separator: / (forward slash) for Red Hat Enterprise Linux and similar operating systems, \ (backslash) for Microsoft Windows Server.

Vault Block (VAULT_BLOCK)

The name to be given to this block in the password vault. Choose a value which is significant to you.

Attribute (ATTRIBUTE)

The name to be given to the attribute being stored. Choose a value which is significant to you. For example, you could choose a name which you associate with a datasource.

Security Attribute (SEC-ATTR)

The password which is being stored in the password vault.

Procedure 18.9. Run the Password Vault Command Interactively

Use this method if you would prefer to be prompted for the value of each parameter.

1. **Launch the Password Vault command interactively.**

Launch your operating system's command line interface and run **EAP_HOME/bin/vault.sh** (on Red Hat Enterprise Linux and similar operating systems) or **EAP_HOME\bin\vault.bat** (on Microsoft Windows Server). Start a new interactive session by typing **0** (zero).

2. Complete the prompted parameters.

Follow the prompts to input the required parameters.

3. Make a note of the masked password information.

The masked password, salt, and iteration count are printed to standard output. Make a note of them in a secure location. They are required to add entries to the Password Vault. Access to the keystore file and these values could allow an attacker access to obtain access to sensitive information in the Password Vault.

4. Exit the interactive console.

Type **3** (three) to exit the interactive console.

Example 18.41. Run the Password Vault command interactively

```
Please enter a Digit:: 0: Start Interactive Session 1: Remove
Interactive Session 2: Exit
0
Starting an interactive session
Enter directory to store encrypted files:EAP_HOME/vault/
Enter Keystore URL:EAP_HOME/vault/vault.keystore
Enter Keystore password: vault22
Enter Keystore password again: vault22
Values match
Enter 8 character salt:1234abcd
Enter iteration count as a number (Eg: 44):120
Enter Keystore Alias:vault
Initializing Vault
Oct 17, 2014 12:58:11 PM
org.picketbox.plugins.vault.PicketBoxSecurityVault init
INFO: PBOX000361: Default Security Vault Implementation Initialized and
Ready
Vault Configuration in AS7 config file:
*****
...
</extensions>
<vault>
  <vault-option name="KEYSTORE_URL"
value="EAP_HOME/vault/vault.keystore"/>
  <vault-option name="KEYSTORE_PASSWORD" value="MASK-5d0aAVafCSd"/>
  <vault-option name="KEYSTORE_ALIAS" value="vault"/>
  <vault-option name="SALT" value="1234abcd"/>
  <vault-option name="ITERATION_COUNT" value="120"/>
  <vault-option name="ENC_FILE_DIR" value="EAP_HOME/vault"/>
</vault><management> ...
*****
Vault is initialized and ready for use
Handshake with Vault complete
```

Procedure 18.10. Run the Password Vault Command Non-interactively

Use this method if you would prefer to provide all parameters' values at once.

- Launch your operating system's command line interface and run the Password Vault command. Refer to the [Parameter Values](#) list, substituting the placeholder values with your preferred values.

Use ***EAP_HOME/bin/vault.sh*** (on Red Hat Enterprise Linux and similar operating systems) or ***EAP_HOME\bin\vault.bat*** (on Microsoft Windows Server).

```
vault.sh --keystore KEYSTORE_URL --keystore-password
KEYSTORE_PASSWORD --alias KEYSTORE_ALIAS --vault-block VAULT_BLOCK -
-attribute ATTRIBUTE --sec-attr SEC-ATTR --enc-dir ENC_FILE_DIR --
iteration ITERATION_COUNT --salt SALT
```

Example 18.42. Run the Password Vault command non-interactively

```
vault.sh --keystore EAP_HOME/vault/vault.keystore --keystore-
password vault22 --alias vault --vault-block vb --attribute
password --sec-attr openS3sam3 --enc-dir EAP_HOME/vault/ --
iteration 120 --salt 1234abcd
```

Command output

```
=====
=====

JBoss Vault

JBOSS_HOME: EAP_HOME

JAVA: java

=====
=====

Oct 17, 2014 2:23:43 PM
org.picketbox.plugins.vault.PicketBoxSecurityVault init
INFO: PBOX000361: Default Security Vault Implementation
Initialized and Ready
Secured attribute value has been stored in vault.
Please make note of the following:
*****
Vault Block:vb
Attribute Name:password
Configuration should be done as follows:
VAULT::vb::password::1
*****
Vault Configuration in AS7 config file:
*****
...
</extensions>
<vault>
  <vault-option name="KEYSTORE_URL"
value="EAP_HOME/vault/vault.keystore"/>
  <vault-option name="KEYSTORE_PASSWORD" value="MASK-
```

```
5d0aAVafCSd"/>
<vault-option name="KEYSTORE_ALIAS" value="vault"/>
<vault-option name="SALT" value="1234abcd"/>
<vault-option name="ITERATION_COUNT" value="120"/>
<vault-option name="ENC_FILE_DIR" value="EAP_HOME/vault"/>
</vault><management> ...
*****
```

Result

Your keystore password has been masked for use in configuration files and deployments. In addition, your vault is initialized and ready to use.

[Report a bug](#)

18.6.6. Configure JBoss EAP 6 to Use the Password Vault

Overview

Before you can mask passwords and other sensitive attributes in configuration files, you need to make JBoss EAP 6 aware of the password vault which stores and decrypts them.

Prerequisites

- [Section 18.6.5, “Initialize the Password Vault”](#)

Procedure 18.11. Enable the Password Vault

- Run the following Management CLI command, substituting the placeholder values with those from the output of the Password Vault command in [Section 18.6.5, “Initialize the Password Vault”](#).



NOTE

If you use Microsoft Windows Server, use two backslashes (\\) in the file path where you would normally use one. For example, **C:\\data\\vault\\vault.keystore**. This is because a single backslash character (\\) is used for character escaping.

```
/core-service=vault:add(vault-options=[("KEYSTORE_URL" =>
"PATH_TO_KEYSTORE"), ("KEYSTORE_PASSWORD" => "MASKED_PASSWORD"),
("KEYSTORE_ALIAS" => "ALIAS"), ("SALT" => "SALT"), ("ITERATION_COUNT"
=> "ITERATION_COUNT"), ("ENC_FILE_DIR" => "ENC_FILE_DIR")])
```

Example 18.43. Enable the Password Vault

```
/core-service=vault:add(vault-options=[("KEYSTORE_URL" =>
"EAP_HOME/vault/vault.keystore"), ("KEYSTORE_PASSWORD" => "MASK-
5d0aAVafCSd"), ("KEYSTORE_ALIAS" => "vault"), ("SALT" => "1234abcd"),
("ITERATION_COUNT" => "120"), ("ENC_FILE_DIR" => "EAP_HOME/vault/")])
```

Result

JBoss EAP 6 is configured to decrypt masked strings stored in the Password Vault. To add strings to the Password Vault and use them in your configuration, see [Section 18.6.8, “Store a Sensitive String in the Password Vault”](#).

[Report a bug](#)

18.6.7. Configure JBoss EAP 6 to Use a Custom Implementation of the Password Vault

Overview

You can use your own implementation of **SecurityVault** to mask passwords and other sensitive attributes in configuration files.

Prerequisites

- [Section 18.6.5, “Initialize the Password Vault”](#)

Procedure 18.12. Use a Custom Implementation of the Password Vault

1. Create a class that implements the interface **SecurityVault**.
2. Create a module containing the class from the previous step, and specify a dependency on **org.picketbox** where the interface is **SecurityVault**.
3. Enable the custom Password Vault in the JBoss EAP server configuration by adding the vault element with the following attributes:

code

The fully qualified name of class that implements **SecurityVault**.

module

The name of the module that contains the custom class.

Optionally, you can use **vault-options** parameters to initialize the custom class for a Password Vault.

Example 18.44. Use vault-options Parameters to Initialize the Custom Class

```
/core-
service=vault:add(code="custom.vault.implementation.CustomSecurity
Vault", module="custom.vault.module", vault-options=
[("KEYSTORE_URL" => "PATH_TO_KEYSTORE"), ("KEYSTORE_PASSWORD" =>
"MASKED_PASSWORD"), ("KEYSTORE_ALIAS" => "ALIAS"), ("SALT" =>
"SALT"), ("ITERATION_COUNT" => "ITERATION_COUNT"), ("ENC_FILE_DIR"
=> "ENC_FILE_DIR")])
```

Result

JBoss EAP 6 is configured to decrypt masked strings using a custom implementation of the password vault.

[Report a bug](#)

18.6.8. Store a Sensitive String in the Password Vault

Overview

Including passwords and other sensitive strings in plaintext configuration files is a security risk. Store these strings instead in the Password Vault for improved security, where they can then be referenced in configuration files, Management CLI commands and applications in their masked form.

Sensitive strings can be store in the Password Vault either interactively, where you are prompted for each parameter's value, or non-interactively, where you provide all parameters' values on the command line. Each method gives the same result, so choose whichever method you prefer. For a description of all parameters, see [Section 18.6.5, "Initialize the Password Vault"](#).

Prerequisites

- [Section 18.6.6, "Configure JBoss EAP 6 to Use the Password Vault"](#)

Procedure 18.13. Store a Sensitive String Interactively

Use this method if you would prefer to be prompted for the value of each parameter.

1. Run the Password Vault command

Launch your operating system's command line interface and run the Password Vault command. Use `EAP_HOME/bin/vault.sh` (on Red Hat Enterprise Linux and similar operating systems) or `EAP_HOME\bin\vault.bat` (on Microsoft Windows Server). Start a new interactive session by typing `0` (zero).

2. Complete the prompted parameters about the Password Vault

Follow the prompts to input the required authentication parameters. These values must match those provided when the Password Vault was created.



NOTE

The keystore password must be given in plaintext form, not masked form.

3. Complete the prompted parameters about the sensitive string

Enter `0` (zero) to start storing the sensitive string. Follow the prompts to input the required parameters.

4. Make note of the information about the masked string

A message prints to standard output, showing the vault block, attribute name, masked string, and advice about using the string in your configuration. Make note of this information in a secure location. An extract of sample output is as follows:

```
Vault Block:ds_Example1
Attribute Name:password
Configuration should be done as follows:
VAULT::ds_Example1::password::1
```

5. Exit the interactive console

Enter `3` (three) to exit the interactive console.

Example 18.45. Store a Sensitive String Interactively

```

=====
=

JBoss Vault

JBOSS_HOME: EAP_HOME/jboss-eap-6.4

JAVA: java

=====
=

*****
****   JBoss Vault   *****
*****

Please enter a Digit::  0: Start Interactive Session  1: Remove
Interactive Session  2: Exit
0
Starting an interactive session
Enter directory to store encrypted files:11:18:46,086 INFO
[org.jboss.security] (management-handler-thread - 4) PBOX0
Enter directory to store encrypted files:EAP_HOME/vault/
Enter Keystore URL:EAP_HOME/vault/vault.keystore
Enter Keystore password:
Enter Keystore password again:
Values match
Enter 8 character salt:1234abcd
Enter iteration count as a number (Eg: 44):120
Enter Keystore Alias:vault
Initializing Vault
Oct 21, 2014 11:20:49 AM
org.picketbox.plugins.vault.PicketBoxSecurityVault init
INFO: PBOX000361: Default Security Vault Implementation Initialized and
Ready
Vault Configuration in AS7 config file:
*****

...
</extensions>
<vault>
  <vault-option name="KEYSTORE_URL"
value="EAP_HOME/vault/vault.keystore"/>
  <vault-option name="KEYSTORE_PASSWORD" value="MASK-5d0aAVafCSd"/>
  <vault-option name="KEYSTORE_ALIAS" value="vault"/>
  <vault-option name="SALT" value="1234abcd"/>
  <vault-option name="ITERATION_COUNT" value="120"/>
  <vault-option name="ENC_FILE_DIR" value="EAP_HOME/vault"/>
</vault><management> ...
*****

Vault is initialized and ready for use
Handshake with Vault complete
Please enter a Digit::  0: Store a secured attribute  1: Check whether
a secured attribute exists  2: Remove secured attribute  3: Exit
0
Task: Store a secured attribute

```

```

Please enter secured attribute value (such as password):
Please enter secured attribute value (such as password) again:
Values match
Enter Vault Block:ds_Example1
Enter Attribute Name:password
Secured attribute value has been stored in vault.
Please make note of the following:
*****
Vault Block:ds_Example1
Attribute Name:password
Configuration should be done as follows:
VAULT::ds_Example1::password::1
*****
Please enter a Digit:: 0: Store a secured attribute 1: Check whether
a secured attribute exists 2: Remove secured attribute 3: Exit

```

Procedure 18.14. Store a Sensitive String Non-interactively

Use this method if you would prefer to provide all parameters' values at once.

1. Launch your operating system's command line interface and run the Password Vault command. Use ***EAP_HOME/bin/vault.sh*** (on Red Hat Enterprise Linux and similar operating systems) or ***EAP_HOME\bin\vault.bat*** (on Microsoft Windows Server).

Substitute the placeholder values with your own values. The values for parameters ***KEYSTORE_URL***, ***KEYSTORE_PASSWORD*** and ***KEYSTORE_ALIAS*** must match those provided when the Password Vault was created.



NOTE

The keystore password must be given in plaintext form, not masked form.

```

EAP_HOME/bin/vault.sh --keystore KEYSTORE_URL --keystore-password
KEYSTORE_PASSWORD --alias KEYSTORE_ALIAS --vault-block VAULT_BLOCK -
-attribute ATTRIBUTE --sec-attr SEC-ATTR --enc-dir ENC_FILE_DIR --
iteration ITERATION_COUNT --salt SALT

```

2. **Make note of the information about the masked string**

A message prints to standard output, showing the vault block, attribute name, masked string, and advice about using the string in your configuration. Make note of this information in a secure location. An extract of sample output is as follows:

```

Vault Block:vb
Attribute Name:password
Configuration should be done as follows:
VAULT::vb::password::1

```

Example 18.46. Run the Password Vault command non-interactively

```

EAP_HOME/bin/vault.sh --keystore EAP_HOME/vault/vault.keystore --
keystore-password vault22 --alias vault --vault-block vb --attribute
password --sec-attr 0pens3sam3 --enc-dir EAP_HOME/vault/ --iteration 120

```

```
--salt 1234abcd
```

Command output

```
=====
=

JBoss Vault

JBOSS_HOME: EAP_HOME

JAVA: java

=====
=

Oct 22, 2014 9:24:43 AM
org.picketbox.plugins.vault.PicketBoxSecurityVault init
INFO: PBOX000361: Default Security Vault Implementation Initialized and
Ready
Secured attribute value has been stored in vault.
Please make note of the following:
*****
Vault Block:vb
Attribute Name:password
Configuration should be done as follows:
VAULT::vb::password::1
*****
Vault Configuration in AS7 config file:
*****
...
</extensions>
<vault>
  <vault-option name="KEYSTORE_URL"
value="EAP_HOME/vault/vault.keystore"/>
  <vault-option name="KEYSTORE_PASSWORD" value="vault22"/>
  <vault-option name="KEYSTORE_ALIAS" value="vault"/>
  <vault-option name="SALT" value="1234abcd"/>
  <vault-option name="ITERATION_COUNT" value="120"/>
  <vault-option name="ENC_FILE_DIR" value="EAP_HOME/vault/vault/">
</vault><management> ...
*****
```

Result

The sensitive string has now been stored in the Password Vault and can be used in configuration files, Management CLI commands and applications in its masked form.

[Report a bug](#)

18.6.9. Use an Encrypted Sensitive String in Configuration

Prerequisites

- [Section 18.6.8, “Store a Sensitive String in the Password Vault”](#)

Any sensitive string which has been encrypted can be used in a configuration file or Management CLI command in its masked form, providing expressions are allowed.

To confirm if expressions are allowed within a particular subsystem, run the following Management CLI command against that subsystem.

**NOTE**

Add the prefix `/host=HOST_NAME` to the command for a managed domain.

```
/core-service=SUBSYSTEM:read-resource-description(recursive=true)
```

Example 18.47. List the Description of all Resources in the Management Subsystem

```
/core-service=management:read-resource-description(recursive=true)
```

From the output of running this command, look for the value of the **expressions-allowed** parameter. If this is **true**, then you can use expressions within the configuration of this subsystem.

Use the following syntax to replace any plaintext string with the masked form.

```
${VAULT::VAULT_BLOCK::ATTRIBUTE_NAME::MASKED_STRING}
```

Example 18.48. Datasource Definition Using a Password in Masked Form

In this example the vault block is **ds_ExampleDS** and the attribute is **password**.

```
...
<subsystem xmlns="urn:jboss:domain:datasources:1.0">
  <datasources>
    <datasource jndi-name="java:jboss/datasources/ExampleDS"
enabled="true" use-java-context="true" pool-name="H2DS">
      <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-1</connection-
url>
      <driver>h2</driver>
      <pool></pool>
      <security>
        <user-name>sa</user-name>
        <password>${VAULT::ds_ExampleDS::password::1}</password>
      </security>
    </datasource>
    <drivers>
      <driver name="h2" module="com.h2database.h2">
        <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-
datasource-class>
      </driver>
    </drivers>
  </datasources>
</subsystem>
...
```


[Report a bug](#)

18.6.10. Use an Encrypted Sensitive String in an Application

Prerequisites

- [Section 18.6.8, “Store a Sensitive String in the Password Vault”](#)

Encrypted strings stored in the Password Vault can be used in your application's source code.

Example 18.49. Servlet Using a Vaulted Password

This example is an extract of a servlet's source code, illustrating the use of a masked password in a datasource definition, instead of the plaintext password. The plaintext version is commented out so that you can see the difference.

```

/*@DataSourceDefinition(
    name = "java:jboss/datasources/LoginDS",
    user = "sa",
    password = "sa",
    className = "org.h2.jdbcx.JdbcDataSource",
    url = "jdbc:h2:tcp://localhost/mem:test"
)*/
@DataSourceDefinition(
    name = "java:jboss/datasources/LoginDS",
    user = "sa",
    password = "VAULT::DS::thePass::1",
    className = "org.h2.jdbcx.JdbcDataSource",
    url = "jdbc:h2:tcp://localhost/mem:test"
)

```

[Report a bug](#)

18.6.11. Check if a Sensitive String is in the Password Vault

Overview

Before attempting to store or use a sensitive string in the Password Vault it can be useful to first confirm if it is already stored.

This check can be done either interactively, where you are prompted for each parameter's value, or non-interactively, where you provide all parameters' values on the command line. Each method gives the same result, so choose whichever method you prefer.

Procedure 18.15. Check For a Sensitive String Interactively

Use this method if you would prefer to be prompted for the value of each parameter.

1. Run the Password Vault command

Launch your operating system's command line interface and run the Password Vault command. Use **EAP_HOME/bin/vault.sh** (on Red Hat Enterprise Linux and similar operating systems)

or **EAP_HOME\bin\vault.bat** (on Microsoft Windows Server). Start a new interactive session by typing **0** (zero).

2. Complete the prompted parameters about the Password Vault

Follow the prompts to input the required authentication parameters. These values must match those provided when the Password Vault was created.



NOTE

The keystore password must be given in plaintext form, not masked form.

3. Enter **1** (one) to select “Check whether a secured attribute exists”.
4. Enter the name of the vault block in which the sensitive string is stored.
5. Enter the name of the sensitive string to be checked.

Result

If the sensitive string is stored in the vault block specified, a confirmation message like the following will be output.

```
A value exists for (VAULT_BLOCK, ATTRIBUTE)
```

If the sensitive string is *not* stored in the specified block, a message like the following will be output.

```
No value has been store for (VAULT_BLOCK, ATTRIBUTE)
```

Example 18.50. Check For a Sensitive String Interactively

```
=====
=
JBoss Vault

JBOSS_HOME: EAP_HOME

JAVA: java

=====
=
*****
****  JBoss Vault  *****
*****
Please enter a Digit::  0: Start Interactive Session  1: Remove
Interactive Session  2: Exit
0
Starting an interactive session
Enter directory to store encrypted files:EAP_HOME/vault
Enter Keystore URL:EAP_HOME/vault/vault.keystore
Enter Keystore password:
Enter Keystore password again:
Values match
```

```

Enter 8 character salt:1234abcd
Enter iteration count as a number (Eg: 44):120
Enter Keystore Alias:vault
Initializing Vault
Oct 22, 2014 12:53:56 PM
org.picketbox.plugins.vault.PicketBoxSecurityVault init
INFO: PBOX000361: Default Security Vault Implementation Initialized and
Ready
Vault Configuration in AS7 config file:
*****

...
</extensions>
<vault>
  <vault-option name="KEYSTORE_URL"
value="EAP_HOME/vault/vault.keystore"/>
  <vault-option name="KEYSTORE_PASSWORD" value="MASK-5dOaAVafCSd"/>
  <vault-option name="KEYSTORE_ALIAS" value="vault"/>
  <vault-option name="SALT" value="1234abcd"/>
  <vault-option name="ITERATION_COUNT" value="120"/>
  <vault-option name="ENC_FILE_DIR" value="EAP_HOME/vault"/>
</vault><management> ...
*****

Vault is initialized and ready for use
Handshake with Vault complete
Please enter a Digit:: 0: Store a secured attribute 1: Check whether
a secured attribute exists 2: Remove secured attribute 3: Exit
1
Task: Verify whether a secured attribute exists
Enter Vault Block:vb
Enter Attribute Name:password
A value exists for (vb, password)
Please enter a Digit:: 0: Store a secured attribute 1: Check whether
a secured attribute exists 2: Remove secured attribute 3: Exit

```

Procedure 18.16. Check For a Sensitive String Non-Interactively

Use this method if you would prefer to provide all parameters' values at once. For a description of all parameters, see [Section 18.6.5, "Initialize the Password Vault"](#).

- Launch your operating system's command line interface and run the Password Vault command. Use ***EAP_HOME/bin/vault.sh*** (on Red Hat Enterprise Linux and similar operating systems) or ***EAP_HOME\bin\vault.bat*** (on Microsoft Windows Server).

Substitute the placeholder values with your own values. The values for parameters ***KEYSTORE_URL***, ***KEYSTORE_PASSWORD-password*** and ***KEYSTORE_ALIAS*** must match those provided when the Password Vault was created.



NOTE

The keystore password must be given in plaintext form, not masked form.

```

EAP_HOME/bin/vault.sh --keystore KEYSTORE_URL --keystore-password
KEYSTORE_PASSWORD --alias KEYSTORE_ALIAS --check-sec-attr --vault-

```

```
block VAULT_BLOCK --attribute ATTRIBUTE --enc-dir ENC_FILE_DIR --
iteration ITERATION_COUNT --salt SALT
```

Result

If the sensitive string is stored in the vault block specified, the following message will be output.

```
Password already exists.
```

If the value is *not* stored in the specified block, the following message will be output.

```
Password doesn't exist.
```

[Report a bug](#)

18.6.12. Remove a Sensitive String from the Password Vault

Overview

For security reasons it is best to remove sensitive strings from the Password Vault when they are no longer required. For example, if you are decommissioning an application, any sensitive strings used in datasource definitions should be removed at the same time.

Prerequisite

Before removing a sensitive string from the Password Vault, confirm if it is used in the configuration of JBoss EAP. One method of doing this is to use the 'grep' utility to search configuration files for instances of the masked string. On Red Hat Enterprise Linux (and similar operating systems), **grep** is installed by default but for Microsoft Windows Server it must be installed manually.

The Password Vault utility provides two modes: interactive and non-interactive. Interactive mode prompts you for each parameter's value, where non-interactive mode requires you to provide all parameters' values in a single command.

Procedure 18.17. Remove a Sensitive String Interactively

Use this method if you would prefer to be prompted for the value of each parameter.

1. Run the Password Vault command

Launch your operating system's command line interface and run **EAP_HOME/bin/vault.sh** (on Red Hat Enterprise Linux and similar operating systems) or **EAP_HOME\bin\vault.bat** (on Microsoft Windows Server). Start a new interactive session by typing **0** (zero).

2. Provide Authentication Details

Follow the prompts to input the required authentication parameters. These values must match those provided when the Password Vault was created.



NOTE

The keystore password must be given in plaintext form, not masked form.

3. Enter **2** (two) to choose option **Remove secured attribute**.

4. Enter the name of the vault block in which the sensitive string is stored.

5. Enter the name of the sensitive string to be removed.

Result

If the sensitive string is successfully removed, a confirmation message like the following will be output.

```
Secured attribute [VAULT_BLOCK::ATTRIBUTE] has been successfully removed
from vault
```

If the sensitive string is *not removed*, a message like the following will be output.

```
Secured attribute [VAULT_BLOCK::ATTRIBUTE] was not removed from vault,
check whether it exist
```

Example 18.51. Remove a Sensitive String Interactively

```
*****
****   JBoss Vault   *****
*****

Please enter a Digit::  0: Start Interactive Session  1: Remove
Interactive Session  2: Exit
0
Starting an interactive session
Enter directory to store encrypted files:EAP_HOME/vault/
Enter Keystore URL:EAP_HOME/vault/vault.keystore
Enter Keystore password:
Enter Keystore password again:
Values match
Enter 8 character salt:1234abcd
Enter iteration count as a number (Eg: 44):120
Enter Keystore Alias:vault
Initializing Vault
Dec 23, 2014 1:40:56 PM
org.picketbox.plugins.vault.PicketBoxSecurityVault init
INFO: PBOX000361: Default Security Vault Implementation Initialized and
Ready
Vault Configuration in configuration file:
*****

...
</extensions>
<vault>
  <vault-option name="KEYSTORE_URL"
value="EAP_HOME/vault/vault.keystore"/>
  <vault-option name="KEYSTORE_PASSWORD" value="MASK-5d0aAVafCSd"/>
  <vault-option name="KEYSTORE_ALIAS" value="vault"/>
  <vault-option name="SALT" value="1234abcd"/>
  <vault-option name="ITERATION_COUNT" value="120"/>
  <vault-option name="ENC_FILE_DIR" value="EAP_HOME/vault"/>
</vault><management> ...
*****

Vault is initialized and ready for use
Handshake with Vault complete
Please enter a Digit::  0: Store a secured attribute  1: Check whether
a secured attribute exists  2: Remove secured attribute  3: Exit
2
Task: Remove secured attribute
```

```
Enter Vault Block:craft
Enter Attribute Name:password
Secured attribute [craft::password] has been successfully removed from
vault
```

Procedure 18.18. Remove a Sensitive String Non-interactively

Use this method if you would prefer to provide all parameters' values at once. For a description of all parameters, see [Section 18.6.5, "Initialize the Password Vault"](#).

- Launch your operating system's command line interface and run the Password Vault command. Use **`EAP_HOME/bin/vault.sh`** (on Red Hat Enterprise Linux and similar operating systems) or **`EAP_HOME\bin\vault.bat`** (on Microsoft Windows Server).

Substitute the placeholder values with your own values. The values for parameters **`KEYSTORE_URL`**, **`KEYSTORE_PASSWORD`** and **`KEYSTORE_ALIAS`** must match those provided when the Password Vault was created.



NOTE

The keystore password must be given in plaintext form, not masked form.

```
EAP_HOME/bin/vault.sh --keystore KEYSTORE_URL --keystore-password
KEYSTORE_PASSWORD --alias KEYSTORE_ALIAS --remove-sec-attr --vault-
block VAULT_BLOCK --attribute ATTRIBUTE --enc-dir ENC_FILE_DIR --
iteration ITERATION_COUNT --salt SALT
```

Result

If the sensitive string is successfully removed, a confirmation message like the following will be output.

```
Secured attribute [VAULT_BLOCK::ATTRIBUTE] has been successfully removed
from vault
```

If the sensitive string is *not removed*, a message like the following will be output.

```
Secured attribute [VAULT_BLOCK::ATTRIBUTE] was not removed from vault,
check whether it exist
```

Example 18.52. Remove a Sensitive String Non-interactively

```
./vault.sh --keystore EAP_HOME/vault/vault.keystore --keystore-password
vault22 --alias vault --remove-sec-attr --vault-block craft --attribute
password --enc-dir ../vault/ --iteration 120 --salt 1234abcd
=====
=

JBoss Vault

JBOSS_HOME: EAP_HOME

JAVA: java
```

```
=====
=
Dec 23, 2014 1:54:24 PM
org.picketbox.plugins.vault.PicketBoxSecurityVault init
INFO: PBOX000361: Default Security Vault Implementation Initialized and
Ready
Secured attribute [craft::password] has been successfully removed from
vault
```

[Report a bug](#)

18.7. JAVA AUTHORIZATION CONTRACT FOR CONTAINERS (JACC)

18.7.1. About Java Authorization Contract for Containers (JACC)

Java Authorization Contract for Containers (JACC) is a standard which defines a contract between containers and authorization service providers, which results in the implementation of providers for use by containers. It was defined in JSR-115, which can be found on the Java Community Process website at <http://jcp.org/en/jsr/detail?id=115>. It has been part of the core Java Enterprise Edition (Java EE) specification since Java EE version 1.3.

JBoss EAP 6 implements support for JACC within the security functionality of the security subsystem.

[Report a bug](#)

18.7.2. Configure Java Authorization Contract for Containers (JACC) Security

To configure Java Authorization Contract for Containers (JACC), you need to configure your security domain with the correct module, and then modify your `jboss-web.xml` to include the correct parameters.

Add JACC Support to the Security Domain

To add JACC support to the security domain, add the **JACC** authorization policy to the authorization stack of the security domain, with the **required** flag set. The following is an example of a security domain with JACC support. However, the security domain is configured in the Management Console or Management CLI, rather than directly in the XML.

```
<security-domain name="jacc" cache-type="default">
  <authentication>
    <login-module code="UsersRoles" flag="required">
    </login-module>
  </authentication>
  <authorization>
    <policy-module code="JACC" flag="required"/>
  </authorization>
</security-domain>
```

Configure a Web Application to Use JACC

The `jboss-web.xml` is located in the **WEB-INF/** directory of your deployment, and contains overrides and additional JBoss-specific configuration for the web container. To use your JACC-enabled security

domain, you need to include the `<security-domain>` element, and also set the `<use-jboss-authorization>` element to `true`. The following application is properly configured to use the JACC security domain above.

```
<jboss-web>
  <security-domain>jacc</security-domain>
  <use-jboss-authorization>true</use-jboss-authorization>
</jboss-web>
```

Configure an EJB Application to Use JACC

Configuring EJBs to use a security domain and to use JACC differs from Web Applications. For an EJB, you can declare *method permissions* on a method or group of methods, in the `ejb-jar.xml` descriptor. Within the `<ejb-jar>` element, any child `<method-permission>` elements contain information about JACC roles. Refer to the example configuration for more details. The `EJBMethodPermission` class is part of the Java Enterprise Edition 6 API, and is documented at <http://docs.oracle.com/javaee/6/api/javax/security/jacc/EJBMethodPermission.html>.

Example 18.53. Example JACC Method Permissions in an EJB

```
<ejb-jar>
  <assembly-descriptor>
    <method-permission>
      <description>The employee and temp-employee roles may access any
method of the EmployeeService bean </description>
      <role-name>employee</role-name>
      <role-name>temp-employee</role-name>
      <method>
        <ejb-name>EmployeeService</ejb-name>
        <method-name>*</method-name>
      </method>
    </method-permission>
  </assembly-descriptor>
</ejb-jar>
```

You can also constrain the authentication and authorization mechanisms for an EJB by using a security domain, just as you can do for a web application. Security domains are declared in the `jboss-ejb3.xml` descriptor, in the `<security>` child element. In addition to the security domain, you can also specify the `<run-as-principal>`, which changes the principal the EJB runs as.

Example 18.54. Example Security Domain Declaration in an EJB

```
<ejb-jar>
  <assembly-descriptor>
    <security>
      <ejb-name>*</ejb-name>
      <security-domain>myDomain</security-domain>
      <run-as-principal>myPrincipal</run-as-principal>
    </security>
  </assembly-descriptor>
</ejb-jar>
```


[Report a bug](#)

18.8. JAVA AUTHENTICATION SPI FOR CONTAINERS (JASPI)

18.8.1. About Java Authentication SPI for Containers (JASPI) Security

Java Authentication SPI for Containers (JASPI or JASPIC) is a pluggable interface for Java applications. It is defined in JSR-196 of the Java Community Process. Refer to <http://www.jcp.org/en/jsr/detail?id=196> for details about the specification.

[Report a bug](#)

18.8.2. Configure Java Authentication SPI for Containers (JASPI) Security

To authenticate against a JASPI provider, add a `<authentication-jaspi>` element to your security domain. The configuration is similar to a standard authentication module, but login module elements are enclosed in a `<login-module-stack>` element. The structure of the configuration is:

Example 18.55. Structure of the `authentication-jaspi` element

```
<authentication-jaspi>
  <login-module-stack name="...">
    <login-module code="..." flag="...">
      <module-option name="..." value="..." />
    </login-module>
  </login-module-stack>
  <auth-module code="..." login-module-stack-ref="...">
    <module-option name="..." value="..." />
  </auth-module>
</authentication-jaspi>
```

The login module itself is configured in exactly the same way as a standard authentication module.

Because the web-based management console does not expose the configuration of JASPI authentication modules, you need to stop JBoss EAP 6 completely before adding the configuration directly to **`EAP_HOME/domain/configuration/domain.xml`** or **`EAP_HOME/standalone/configuration/standalone.xml`**.

[Report a bug](#)

CHAPTER 19. SINGLE SIGN ON (SSO)

19.1. ABOUT SINGLE SIGN ON (SSO) FOR WEB APPLICATIONS

Overview

Single Sign On (SSO) allows authentication to one resource to implicitly allow access to other resources.

Clustered and Non-Clustered SSO

Non-clustered SSO limits the sharing of access information to applications on the same virtual host. In addition, there is no resiliency in the event of a host failure. Clustered SSO data can be shared between applications in multiple hosts, and is resilient to failover. In addition, clustered SSO is able to receive requests from a load balancer.

How SSO Works

If a resource is unprotected, a user is not challenged to authenticate at all. If a user accesses a protected resource, the user is required to authenticate.

Upon successful authentication, the roles associated with the user are stored and used for authentication of all other associated resources.

If the user logs out of an application, or an application invalidates the session programmatically, all persisted authentication data is removed, and the process starts over.

A session timeout does not invalidate the SSO session if other sessions are still valid.

[Report a bug](#)

19.2. ABOUT CLUSTERED SINGLE SIGN ON (SSO) FOR WEB APPLICATIONS

Single Sign On (SSO) is the ability for users to authenticate to a single web application, and by means of a successful authentication, will successfully authenticate to multiple other applications without needing to be prompted at each one. Clustered SSO stores the authentication information in a clustered cache. This allows for applications on multiple different servers to share the information, and also makes the information resilient to a failure of one of the hosts.

Some of the supported SSO mechanisms (for example, Kerberos, PicketLink SAML) need valves to work correctly. Valves have a similar function as the servlet filters, but they are processed before the container managed authentication. Valves for web applications can be defined in the `jboss-web.xml` deployment descriptor.

[Report a bug](#)

19.3. CHOOSE THE RIGHT SSO IMPLEMENTATION

JBoss EAP 6 runs Java Enterprise Edition (EE) applications, which may be web applications, EJB applications, web services, or other types. Single Sign On (SSO) allows you to propagate security context and identity information between these applications. Several SSO solutions are available but choosing the right solution depends on your requirements.

Note that there is a distinct difference between a clustered web application and clustered SSO. A clustered web application is one which is distributed across the nodes of a cluster to spread the load of hosting that application. If marked as distributable, all new sessions, and changes to existing sessions

are replicated to other members of the cluster. An application is marked as able to be distributed across cluster nodes with the `<distributed/>` tag in the `web.xml` deployment descriptor. Clustered SSO allows for replication of security context and identity information, regardless of whether or not the applications are themselves clustered. Although these technologies may be used together they are separate concepts.

Kerberos-Based Desktop SSO

If your organization already uses a Kerberos-based authentication and authorization system, such as Microsoft Active Directory, you can use the same systems to transparently authenticate to your enterprise applications running on JBoss EAP 6.

Non-Clustered Web Application SSO

If you are running multiple applications on a single instance and need to enable SSO session replication for those applications, non-clustered SSO will meet your requirements.

Clustered Web Application SSO

If you are running either a single application, or multiple applications, across a cluster and need to enable SSO session replication for those applications, clustered SSO will meet your requirements.

[Report a bug](#)

19.4. USE SINGLE SIGN ON (SSO) IN A WEB APPLICATION

Overview

Single Sign On (SSO) capabilities are provided by the web and Infinispan subsystems. Use this procedure to configure SSO in web applications.

Prerequisites

- A configured security domain which handles authentication and access.
- The **infinispan** subsystem. By default, it is present in all the profiles for managed domain and standalone server.
- The **web cache-container** and SSO replicated-cache. The initial configuration files already contain the **web** cache-container, and some of the configurations already contain the SSO replicated-cache as well. Use the following commands to check for and enable the SSO replicated-cache. Note that these commands modify the **ha** profile of a managed domain. You can change the commands to use a different profile, or remove the **/profile=ha** portion of the command, for a standalone server.

Example 19.1. Check for the web cache-container

The profiles and configurations mentioned above include the **web** cache-container by default. Use the following command to verify its presence. If you use a different profile, substitute its name instead of **ha**.

```
/profile=ha/subsystem=infinispan/cache-container=web/:read-
resource(recursive=false,proxies=false,include-
runtime=false,include-defaults=true)
```

If the result is **success** the subsystem is present. Otherwise, you need to add it.

Example 19.2. Add the web cache-container

Use the following three commands to enable the **web** cache-container to your configuration. Modify the name of the profile as appropriate, as well as the other parameters. The parameters here are the ones used in a default configuration.

```
/profile=ha/subsystem=infinispan/cache-container=web:add(aliases=[
"standard-session-cache"], default-cache="repl", module="org.jboss.as.clustering.web.infinispan")
```

```
/profile=ha/subsystem=infinispan/cache-container=web/transport=TRANSPORT:add(lock-timeout=60000)
```

```
/profile=ha/subsystem=infinispan/cache-container=web/replicated-cache=repl:add(mode="ASYNC", batching=true)
```

Example 19.3. Check for the SSO replicated-cache

Run the following Management CLI command:

```
/profile=ha/subsystem=infinispan/cache-container=web/:read-resource(recursive=true, proxies=false, include-runtime=false, include-defaults=true)
```

Look for output like the following: **"sso" => {**

If you do not find it, the SSO replicated-cache is not present in your configuration.

Example 19.4. Add the SSO replicated-cache

```
/profile=ha/subsystem=infinispan/cache-container=web/replicated-cache=sso:add(mode="SYNC", batching=true)
```

Configure Clustered SSO for a Managed Domain

The **web** subsystem needs to be configured to use SSO. The following command enables SSO on the virtual server called **default-host**, and the cookie domain **domain.com**. The cache name is **sso**, and reauthentication is disabled.

```
/profile=ha/subsystem=web/virtual-server=default-host/sso=configuration:add(cache-container="web", cache-name="sso", reauthenticate="false", domain="domain.com")
```

Each application which will share the SSO information must be configured to use the same `<security-domain>` in its **jboss-web.xml** deployment descriptor and the same Realm in its **web.xml** configuration file.

Configure Clustered or Non-Clustered SSO for a Standalone Server

Configure **sso** under the web subsystem in the server profile. The **ClusteredSingleSignOn** version is used when attribute **cache-container** is present, otherwise standard **SingleSignOn** class is used.

Example 19.5. Example Non-Clustered SSO Configuration

```
/subsystem=web/virtual-server=default-  
host/sso=configuration:add(reauthenticate="false")
```

Invalidate a Session

An application can programmatically invalidate a session by invoking method `javax.servlet.http.HttpSession.invalidate()`.

[Report a bug](#)

19.5. ABOUT KERBEROS

Kerberos is a network authentication protocol for client/server applications. It allows authentication across a non-secure network in a secure way, using secret-key symmetric cryptography.

Kerberos uses security tokens called tickets. To use a secured service, you need to obtain a ticket from the Ticket Granting Service (TGS), which is a service running on a server on the network. After obtaining the ticket, you request a Service Ticket (ST) from an Authentication Service (AS), which is another service running on the network. You then use the ST to authenticate to the service you want to use. The TGS and the AS both run inside an enclosing service called the Key Distribution Center (KDC).

Kerberos is designed to be used in a client-server environment, and is rarely used in Web applications or thin client environments. However, many organizations already use a Kerberos system for desktop authentication, and prefer to reuse their existing system rather than create a second one for their Web Applications. Kerberos is an integral part of Microsoft Active Directory, and is also used in many Red Hat Enterprise Linux environments.

[Report a bug](#)

19.6. ABOUT SPNEGO

Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) provides a mechanism for extending a Kerberos-based Single Sign On (SSO) environment for use in Web applications.

When an application on a client computer, such as a web browser, attempts to access a protect page on the web server, the server responds that authorization is required. The application then requests a service ticket from the Kerberos Key Distribution Center (KDC). After the ticket is obtained, the application wraps it in a request formatted for SPNEGO, and sends it back to the Web application, via the browser. The web container running the deployed Web application unpacks the request and authenticates the ticket. Upon successful authentication, access is granted.

SPNEGO works with all types of Kerberos providers, including the Kerberos service included in Red Hat Enterprise Linux and the Kerberos server which is an integral part of Microsoft Active Directory.

[Report a bug](#)

19.7. ABOUT MICROSOFT ACTIVE DIRECTORY

Microsoft Active Directory is a directory service developed by Microsoft to authenticate users and computers in a Microsoft Windows domain. It is included as part of Microsoft Windows Server. The computer in the Microsoft Windows Server is referred to as the domain controller. Red Hat Enterprise Linux servers running the Samba service can also act as the domain controller in this type of network.

Active Directory relies on three core technologies which work together:

- Lightweight Directory Access Protocol (LDAP), for storing information about users, computers, passwords, and other resources.
- Kerberos, for providing secure authentication over the network.
- Domain Name Service (DNS) for providing mappings between IP addresses and host names of computers and other devices on the network.

[Report a bug](#)

19.8. CONFIGURE KERBEROS OR MICROSOFT ACTIVE DIRECTORY DESKTOP SSO FOR WEB APPLICATIONS

Introduction

To authenticate your web or EJB applications using your organization's existing Kerberos-based authentication and authorization infrastructure, such as Microsoft Active Directory, you can use the JBoss Negotiation capabilities built into JBoss EAP 6. If you configure your web application properly, a successful desktop or network login is sufficient to transparently authenticate against your web application, so no additional login prompt is required.

Difference from Previous Versions of the Platform

There are a few noticeable differences between JBoss EAP 6 and earlier versions:

- Security domains are configured for each profile of a managed domain, or for each standalone server. They are not part of the deployment itself. The security domain a deployment should use is named in the deployment's **jboss-web.xml** or **jboss-ejb3.xml** file.
- Security properties are configured as part of a security domain. They are not part of the deployment.
- You can no longer override the authenticators as part of your deployment. However, you can add a NegotiationAuthenticator valve to your **jboss-web.xml** descriptor to achieve the same effect. The valve still requires the **<security-constraint>** and **<login-config>** elements to be defined in the **web.xml**. These are used to decide which resources are secured. However, the chosen auth-method will be overridden by the NegotiationAuthenticator valve in the **jboss-web.xml**.
- The **CODE** attributes in security domains now use a simple name instead of a fully-qualified class name. The following table shows the mappings between the classes used for JBoss Negotiation, and their classes.

Table 19.1. Login Module Codes and Class Names

Simple Name	Class Name	Purpose
Kerberos	com.sun.security.auth.module.Krb5LoginModule	Kerberos login module when using the Oracle JDK
	com.ibm.security.auth.module.Krb5LoginModule	Kerberos login module when using the IBM Java development kit
SPNEGO	org.jboss.security.negotiation.spnego.SPNEGOLoginModule	The mechanism which enables your Web applications to authenticate to your Kerberos authentication server.
AdvancedLdap	org.jboss.security.negotiation.AdvancedLdapLoginModule	Used with LDAP servers other than Microsoft Active Directory.
AdvancedAdLdap	org.jboss.security.negotiation.AdvancedAdLoginModule	Used with Microsoft Active Directory LDAP servers.

JBoss Negotiation Toolkit

The **JBoss Negotiation Toolkit** is a debugging tool which is available for download from <https://community.jboss.org/servlet/JiveServlet/download/16876-2-34629/jboss-negotiation-toolkit.war>. It is provided as an extra tool to help you to debug and test the authentication mechanisms before introducing your application into production. It is an unsupported tool, but is considered to be very helpful, as SPNEGO can be difficult to configure for web applications.

Procedure 19.1. Setup SSO Authentication for your Web or EJB Applications

1. **Configure one security domain to represent the identity of the server. Set system properties if necessary.**

The first security domain authenticates the container itself to the directory service. It needs to use a login module which accepts some type of static login mechanism, because a real user is not involved. This example uses a static principal and references a keytab file which contains the credential.

The XML code is given here for clarity, but you should use the Management Console or Management CLI to configure your security domains.

```
<security-domain name="host" cache-type="default">
  <authentication>
    <login-module code="Kerberos" flag="required">
      <module-option name="storeKey" value="true"/>
      <module-option name="useKeyTab" value="true"/>
      <module-option name="principal"
value="host/testserver@MY_REALM"/>
      <module-option name="keyTab"
value="/home/username/service.keytab"/>
      <module-option name="doNotPrompt" value="true"/>
      <module-option name="debug" value="false"/>
    </login-module>
  </authentication>
</security-domain>
```

2. Configure a second security domain to secure the web application or applications. Set system properties if necessary.

The second security domain is used to authenticate the individual user to the Kerberos or SPNEGO authentication server. You need at least one login module to authenticate the user, and another to search for the roles to apply to the user. The following XML code shows an example SPNEGO security domain. It includes an authorization module to map roles to individual users. You can also use a module which searches for the roles on the authentication server itself.

```
<security-domain name="SPNEGO" cache-type="default">
  <authentication>
    <!-- Check the username and password -->
    <login-module code="SPNEGO" flag="requisite">
      <module-option name="password-stacking"
value="useFirstPass"/>
      <module-option name="serverSecurityDomain" value="host"/>
    </login-module>
    <!-- Search for roles -->
    <login-module code="UsersRoles" flag="required">
      <module-option name="password-stacking"
value="useFirstPass" />
      <module-option name="usersProperties" value="spnego-
users.properties" />
      <module-option name="rolesProperties" value="spnego-
roles.properties" />
    </login-module>
  </authentication>
</security-domain>
```

3. Specify the security-constraint and login-config in the web.xml

The **web.xml** descriptor contain information about security constraints and login configuration. The following are example values for each.

```
<security-constraint>
  <display-name>Security Constraint on Conversation</display-name>
  <web-resource-collection>
    <web-resource-name>examplesWebApp</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>RequiredRole</role-name>
  </auth-constraint>
</security-constraint>

<login-config>
  <auth-method>SPNEGO</auth-method>
  <realm-name>SPNEGO</realm-name>
</login-config>

<security-role>
  <description> role required to log in to the
Application</description>
  <role-name>RequiredRole</role-name>
</security-role>
```


4. Specify the security domain and other settings in the `jboss-web.xml` descriptor.

Specify the name of the client-side security domain (the second one in this example) in the `jboss-web.xml` descriptor of your deployment, to direct your application to use this security domain.

You can no longer override authenticators directly. Instead, you can add the `NegotiationAuthenticator` as a valve to your `jboss-web.xml` descriptor, if you need to. The `<jacc-star-role-allow>` allows you to use the asterisk (*) character to match multiple role names, and is optional.

```
<jboss-web>
  <security-domain>SPNEGO</security-domain>
  <valve>
    <class-
name>org.jboss.security.negotiation.NegotiationAuthenticator</class-
name>
  </valve>
  <jacc-star-role-allow>true</jacc-star-role-allow>
</jboss-web>
```

5. Add a dependency to your application's `MANIFEST.MF`, to locate the Negotiation classes.

The web application needs a dependency on class `org.jboss.security.negotiation` to be added to the deployment's `META-INF/MANIFEST.MF` manifest, in order to locate the JBoss Negotiation classes. The following shows a properly-formatted entry.

```
Manifest-Version: 1.0
Build-Jdk: 1.6.0_24
Dependencies: org.jboss.security.negotiation
```

- o As an alternative, add a dependency to your application by editing the `META-INF/jboss-deployment-structure.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-deployment-structure>
  <deployment>
    <dependencies>
      <module name='org.jboss.security.negotiation' />
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```



NOTE

You can use only one Kerberos login module in a security domain.

Result

Your web application accepts and authenticates credentials against your Kerberos, Microsoft Active Directory, or other SPNEGO-compatible directory service. If the user runs the application from a system which is already logged into the directory service, and where the required roles are already applied to the user, the web application does not prompt for authentication, and SSO capabilities are achieved.

[Report a bug](#)

19.9. CONFIGURE SPNEGO FALL BACK TO FORM AUTHENTICATION

Follow the procedure below to setup a SPNEGO fall back to form authentication.

Procedure 19.2. SPNEGO security with fall back to form authentication

1. Set up SPNEGO

Refer the procedure described in [Section 19.8, “Configure Kerberos or Microsoft Active Directory Desktop SSO for Web Applications”](#)

2. Modify web.xml

Add a **login-config** element to your application and setup the login and error pages in web.xml:

```
<login-config>
  <auth-method>SPNEGO</auth-method>
  <realm-name>SPNEGO</realm-name>
  <form-login-config>
    <form-login-page>/login.jsp</form-login-page>
    <form-error-page>/error.jsp</form-error-page>
  </form-login-config>
</login-config>
```

3. Add web content

Add references of **login.html** and **error.html** to **web.xml**. These files are added to web application archive to the place specified in **form-login-config** configuration. For more information refer *Enable Form-based Authentication* section in the *Security Guide* for JBoss EAP 6. A typical **login.html** looks like this:

```
<html>
  <head>
    <title>Vault Form Authentication</title>
  </head>
  <body>
    <h1>Vault Login Page</h1>
    <p>
      <form method="post" action="j_security_check">
        <table>
          <tr>
            <td>Username</td><td>-</td>
            <td><input type="text" name="j_username"></td>
          </tr>
          <tr>
            <td>Password</td><td>-</td>
            <td><input type="password" name="j_password"></td>
          </tr>
          <tr>
            <td colspan="2"><input type="submit"></td>
          </tr>
        </table>
      </form>
    </p>
    <hr>
  </body>
</html>
```

**NOTE**

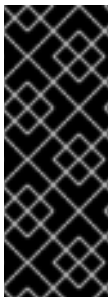
The fallback to FORM logic is only available in the case when no SPNEGO (or NTLM) tokens are present. As a result, a login form is not presented to the browser if the browser sends an NTLM token.

[Report a bug](#)

19.10. ABOUT SAML WEB BROWSER BASED SSO

PicketLink in JBoss EAP provides a platform to implement federated identity based services. This includes centralized identity services and Single Sign-On (SSO) for applications.

The SAML profile has support for both the HTTP/POST and the HTTP/Redirect bindings with centralized identity services to enable web SSO for your applications. The architecture for the SAML v2 based Web SSO follows the hub and spoke architecture of identity management. In this architecture an identity provider (IDP) acts as the central source (hub) for identity and role information to all the applications (Service Providers). The spokes are the service providers (SP).

**IMPORTANT**

If one HTTP client (web browser) connects to more SPs pointing to the same IDP, the IDP does not distinguish between the different SPs. If more requests from one client come simultaneously, the IDP handles the most recent request from an SP and sends back SAML assertion about the authenticated user. It means the SAML response from the IDP can be in such case forwarded to incorrect SP. To get back to the older SP, you will need to reenter the SP URL in the browser.

**NOTE**

For more information, refer *Red Hat JBoss Enterprise Application Platform 6.4 How to Setup SSO with SAML V2* document and *Browser-based SSO Using SAML* section in the *Red Hat JBoss Enterprise Application Platform 6.4 Security Architecture* document.

[Report a bug](#)

19.11. COOKIE DOMAIN

19.11.1. About the Cookie Domain

The *cookie domain* refers to the set of hosts able to read a cookie from the client browser which is accessing your application. It is a configuration mechanism to minimize the risk of third parties accessing information your application stores in browser cookies.

The default value for the cookie domain is `/`. This means that only the issuing host can read the contents of a cookie. Setting a specific cookie domain makes the contents of the cookie available to a wider range of hosts. To set the cookie domain, refer to [Section 19.11.2, “Configure the Cookie Domain for Single Sign On”](#).

[Report a bug](#)

19.11.2. Configure the Cookie Domain for Single Sign On

To enable your SSO valve to share a SSO context, configure the cookie domain in the valve configuration. The following configuration would allow applications on **http://app1.xyz.com** and **http://app2.xyz.com** to share an SSO context, even if these applications run on different servers in a cluster or the virtual host with which they are associated has multiple aliases.

Clustered SSO (shared against clustered JBoss EAP instances)

Using the CLI (in Standalone mode):

```
/subsystem=web/virtual-server=default-host/sso=configuration:add(cache-
container="web",cache-name="sso")
```

Editing **standalone.xml** or **domain.xml** and append the below to the relevant web subsystem:

```
<subsystem xmlns="urn:jboss:domain:web:1.1" default-virtual-
server="default-host" native="false">
  <connector name="http" protocol="HTTP/1.1" scheme="http" socket-
binding="http"/>
  <virtual-server name="default-host" enable-welcome-root="true">
    <alias name="localhost"/>
    <alias name="example.com"/>
    <sso cache-container="web" cache-name="sso"/> <!--FIXME: ADD this Line-->
  </virtual-server>
</subsystem>
```

Non-Clustered SSO (SSO only shared against instances within the Jboss EAP instances)

Using the CLI (in Standalone mode):

```
/subsystem=web/virtual-server=default-host/sso=configuration:add()
```

Editing **standalone.xml** or **domain.xml** and append the below to the relevant web subsystem:

```
<subsystem xmlns="urn:jboss:domain:web:1.1" default-virtual-
server="default-host" native="false">
  <connector name="http" protocol="HTTP/1.1" scheme="http" socket-
binding="http"/>
  <virtual-server name="default-host" enable-welcome-root="true">
    <alias name="localhost"/>
    <alias name="example.com"/>
    <sso/> <!--FIXME: ADD this Line-->
  </virtual-server>
</subsystem>
```

The Single Sign On (SSO) configuration in JBoss EAP 6 includes a domain attribute that can be specified. For example:

```
/subsystem=web/virtual-server=default-
host/sso=configuration:add(domain="example.com",...)
```

Which adds the following SSO configuration:

■

```
<sso domain="example.com"/>
```

[Report a bug](#)

CHAPTER 20. DEVELOPMENT SECURITY REFERENCES

20.1. EJB SECURITY PARAMETER REFERENCE

Table 20.1. EJB security parameter elements

Element	Description
<code><security-identity></code>	Contains child elements pertaining to the security identity of an EJB.
<code><use-caller-identity /></code>	Indicates that the EJB uses the same security identity as the caller.
<code><run-as></code>	Contains a <code><role-name></code> element.
<code><run-as-principal></code>	If present, indicates the principal assigned to outgoing calls. If not present, outgoing calls are assigned to a principal named anonymous .
<code><role-name></code>	Specifies the role the EJB should run as.
<code><description></code>	Describes the role named in <code><role-name></code> .

Example 20.1. Security identity examples

The example `ejb-jar.xml` file below shows each tag described in [Table 20.1](#), “EJB security parameter elements”. They can also be used inside a `<session>`.

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>ASessionBean</ejb-name>
      <security-identity>
        <use-caller-identity/>
      </security-identity>
    </session>
    <session>
      <ejb-name>RunAsBean</ejb-name>
      <security-identity>
        <run-as>
          <description>A private internal role</description>
          <role-name>InternalRole</role-name>
        </run-as>
      </security-identity>
    </session>
    <session>
      <ejb-name>RunAsBean</ejb-name>
      <security-identity>
        <run-as-principal>internal</run-as-principal>
      </security-identity>
```

```
</session>
</enterprise-beans>
</ejb-jar>
```

The above parameters can also be included in the **jboss-ejb3.xml** file which is discussed in more detail in [Section 8.8.4, “jboss-ejb3.xml Deployment Descriptor Reference”](#).

[Report a bug](#)

CHAPTER 21. CONFIGURATION REFERENCES

21.1. JBOSS-WEB.XML CONFIGURATION REFERENCE

Introduction

The **jboss-web.xml** and **web.xml** deployment descriptors are both placed in the deployment's **WEB-INF** directory. The **jboss-web.xml** is a web application deployment descriptor for JBoss EAP which contains additional configuration options for additional features of JBoss Web. This descriptor can be used to override the settings from **web.xml** descriptor and to set JBoss EAP specific settings.

Mapping Global Resources to WAR Requirements

Many of the available settings map requirements set in the application's **web.xml** to local resources. The explanations of the **web.xml** settings can be found at

http://docs.oracle.com/cd/E13222_01/wls/docs81/webapp/web_xml.html.

For instance, if the **web.xml** requires **jdbc/MyDataSource**, the **jboss-web.xml** may map the global datasource **java:/DefaultDS** to fulfill this need. The WAR uses the global datasource to fill its need for **jdbc/MyDataSource**.

Table 21.1. Common Top-Level Attributes of jboss-web.xml

Attribute	Description
servlet	The servlet element specifies servlet specific bindings.
max-active-sessions	Determines the max number of active sessions allowed. If the number of sessions managed by the session manager exceeds this value and passivation is enabled, the excess will be passivated based on the configured passivation-min-idle-time If set to -1, means no limit.
replication-config	The replication-config element is used for configuring session replication in the jboss-web.xml file.
passivation-config	The passivation-config element is used for configuring session passivation in the jboss-web.xml file.
distinct-name	The distinct-name element specifies the EJB 3 distinct name for the web application.
data-source	A mapping to a data-source required by the web.xml .
context-root	The root context of the application. The default value is the name of the deployment without the .war suffix.

Attribute	Description
virtual-host	The name of the HTTP virtual-host the application accepts requests from. It refers to the contents of the HTTP Host header.
annotation	Describes an annotation used by the application. Refer to <annotation> for more information.
listener	Describes a listener used by the application. Refer to <listener> for more information.
session-config	This element fills the same function as the <session-config> element of the web.xml and is included for compatibility only.
valve	Describes a valve used by the application. Refer to <valve> for more information.
overlay	The name of an overlay to add to the application.
security-domain	The name of the security domain used by the application. The security domain itself is configured in the web-based management console or the management CLI.
security-role	This element fills the same function as the <security-role> element of the web.xml and is included for compatibility only.
jacc-star-role-allow	The jacc-star-role-allow element specifies whether the jacc permission generating agent in the web layer needs to generate a WebResourcePermission permission such that the jacc provider can make a decision as to bypass authorization or not.
use-jboss-authorization	If this element is present and contains the case insensitive value "true", the JBoss web authorization stack is used. If it is not present or contains any value that is not "true", then only the authorization mechanisms specified in the Java Enterprise Edition specifications are used. This element is new to JBoss EAP 6.
disable-audit	Set this boolean element to false to enable and true to disable web auditing. Web security auditing is not part of the Java EE specification. This element is new to JBoss EAP 6.

Attribute	Description
disable-cross-context	If false , the application is able to call another application context. Defaults to true .
enable-websockets	Set this element to true in jboss-web.xml to specify if websockets access should be enabled for the web application.

The following elements each have child elements.

<annotation>

Describes an annotation used by the application. The following table lists the child elements of an **<annotation>**.

Table 21.2. Annotation Configuration Elements

Attribute	Description
class-name	Name of the class of the annotation
servlet-security	The element, such as @ServletSecurity , which represents servlet security.
run-as	The element, such as @RunAs , which represents the run-as information.
multipart-config	The element, such as @MultiPart , which represents the multipart-config information.

<listener>

Describes a listener. The following table lists the child elements of a **<listener>**.

Table 21.3. Listener Configuration Elements

Attribute	Description
class-name	Name of the class of the listener

Attribute	Description
listener-type	<p>List of condition elements, which indicate what kind of listener to add to the Context of the application. Valid choices are:</p> <p>CONTAINER Adds a ContainerListener to the Context.</p> <p>LIFECYCLE Adds a LifecycleListener to the Context.</p> <p>SERVLET_INSTANCE Adds an InstanceListener to the Context.</p> <p>SERVLET_CONTAINER Adds a WrapperListener to the Context.</p> <p>SERVLET_LIFECYCLE Adds a WrapperLifecycle to the Context.</p>
module	The name of the module containing the listener class.
param	A parameter. Contains two child elements, <param-name> and <param-value> .

<valve>

Describes a valve of the application. Similar to the [<listener>](#), has class-name, module and param elements.

[Report a bug](#)

CHAPTER 22. SUPPLEMENTAL REFERENCES

22.1. TYPES OF JAVA ARCHIVES

JBoss EAP 6 recognizes several different types of archive files. Archive files are used to package deployable services and applications.

In general, archive files are Zip archives, with specific file extensions and specific directory structures. If the Zip archive is extracted before being deployed on the application server, it is referred to as an exploded archive. In that case, the directory name still contains the file extension, and the directory structure requirements still apply.

Table 22.1.

Archive Type	Extension	Purpose	Directory structure requirements
Java Archive	.jar	Contains Java class libraries.	META-INF/MANIFEST.MF file (optional), which specifies information such as which class is the main class.
Web Archive	.war	Contains Java Server Pages (JSP) files, servlets, and XML files, in addition to Java classes and libraries. The Web Archive's contents are also referred to as a Web Application.	WEB-INF/web.xml file, which contains information about the structure of the web application. Other files may also be present in WEB-INF/ .
Resource Adapter Archive	.rar	The directory structure is specified by the JCA specification.	Contains a Java Connector Architecture (JCA) resource adapter. Also called a connector.
Enterprise Archive	.ear	Used by Java Enterprise Edition (EE) to package one or more modules into a single archive, so that the modules can be deployed onto the application server simultaneously. Maven and Ant are the most common tools used to build EAR archives.	META-INF/ directory, which contains one or more XML deployment descriptor files.

Archive Type	Extension	Purpose	Directory structure requirements
			<p>Any of the following types of modules.</p> <ul style="list-style-type: none"> • A Web Archive (WAR). • One or more Java Archives (JARs) containing Plain Old Java Objects (POJOs). • One or more Enterprise JavaBean (EJB) modules, containing its own META-INF/ directory. This directory includes descriptors for the persistent classes which are deployed. • One or more Resource Archives (RARs).
Service Archive	.sar	Similar to an Enterprise Archive, but specific to the JBoss EAP.	META-INF/ directory containing jboss-service.xml or jboss-beans.xml file.

[Report a bug](#)

APPENDIX A. REVISION HISTORY

Revision 6.4.0-48	Thursday November 16 2017	Red Hat Customer Content Services
Red Hat JBoss Enterprise Application Platform 6.4.0.GA Continuous Release		