

JBoss Enterprise Application Platform 6

Development Guide

For Use with JBoss Enterprise Application Platform 6

Edition 2



Sande Gilda

Eamon Logue

Darrin Mison

David Ryan

Misty Stanley-Jones

Keerat Verma

Tom Wells

Legal Notice

Copyright © 2012 Red Hat, Inc..

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at <http://creativecommons.org/licenses/by-sa/3.0/>. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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, MetaMatrix, 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.

All other trademarks are the property of their respective owners.

Abstract

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

Table of Contents

Preface

1. Document Conventions
 - 1.1. Typographic Conventions
 - 1.2. Pull-quote Conventions
 - 1.3. Notes and Warnings
2. Getting Help and Giving Feedback
 - 2.1. Do You Need Help?
 - 2.2. Give us Feedback

1. Get Started Developing Applications

- 1.1. Introduction
 - 1.1.1. Introducing JBoss Enterprise Application Platform 6
- 1.2. Prerequisites
 - 1.2.1. Become Familiar with Java Enterprise Edition 6
 - 1.2.2. About Modules and the New Modular Class Loading System used in JBoss Enterprise Application Platform 6
- 1.3. Install JBoss Enterprise Application Platform 6
 - 1.3.1. Download and Install JBoss Enterprise Application Platform 6
 - 1.3.2. Take a Quick Tour of JBoss Enterprise Application Platform 6
- 1.4. Set Up the Development Environment
 - 1.4.1. Download and Install JBoss Developer Studio
- 1.5. Run Your First Application
 - 1.5.1. Replace the Default Welcome Web Application
 - 1.5.2. Download the Quickstart Code Examples
 - 1.5.3. Run the Quickstarts
 - 1.5.4. Review the Quickstart Tutorials

2. Maven Guide

- 2.1. Learn about Maven
 - 2.1.1. About the Maven Repository
 - 2.1.2. About the Maven POM File
 - 2.1.3. Minimum Requirements of a Maven POM File
 - 2.1.4. About the Maven Settings File
- 2.2. Install Maven and the JBoss Maven Repository
 - 2.2.1. Download and Install Maven
 - 2.2.2. Install the JBoss Enterprise Application Platform 6 Maven Repository
 - 2.2.3. Install the JBoss Enterprise Application Platform 6 Maven Repository Locally
 - 2.2.4. Install the JBoss Enterprise Application Platform 6 Maven Repository for Use with Apache httpd
 - 2.2.5. Install the JBoss Enterprise Application Platform 6 Maven Repository Using Nexus Maven Repository Manager
 - 2.2.6. About Maven Repository Managers
- 2.3. Configure the Maven Repository
 - 2.3.1. Configure the JBoss Enterprise Application Platform Maven Repository
 - 2.3.2. Configure the JBoss Enterprise Application Platform Maven Repository Using the Maven Settings
 - 2.3.3. Configure the JBoss Enterprise Application Platform Maven Repository Using the Project POM

3. Class Loading and Modules

- 3.1. Introduction
 - 3.1.1. Overview of Class Loading and Modules

- [3.1.2. Class Loading](#)
 - [3.1.3. Modules](#)
 - [3.1.4. Module Dependencies](#)
 - [3.1.5. Class Loading in Deployments](#)
 - [3.1.6. Class Loading Precedence](#)
 - [3.1.7. Dynamic Module Naming](#)
 - [3.1.8. jboss-deployment-structure.xml](#)
 - [3.2. Add an Explicit Module Dependency to a Deployment](#)
 - [3.3. Generate MANIFEST.MF entries using Maven](#)
 - [3.4. Prevent a Module Being Implicitly Loaded](#)
 - [3.5. Exclude a Subsystem from a Deployment](#)
 - [3.6. Class Loading and Subdeployments](#)
 - [3.6.1. Modules and Class Loading in Enterprise Archives](#)
 - [3.6.2. Subdeployment Class Loader Isolation](#)
 - [3.6.3. Disable Subdeployment Class Loader Isolation Within a EAR](#)
 - [3.7. Reference](#)
 - [3.7.1. Implicit Module Dependencies](#)
 - [3.7.2. Included Modules](#)
 - [3.7.3. JBoss Deployment Structure Deployment Descriptor Reference](#)
- [4. Logging for Developers](#)
 - [4.1. Introduction](#)
 - [4.1.1. About Logging](#)
 - [4.1.2. Application Logging Frameworks Supported By JBoss LogManager](#)
 - [4.1.3. About Log Levels](#)
 - [4.1.4. Supported Log Levels](#)
 - [4.1.5. Default Log File Locations](#)
 - [4.2. Logging with the JBoss Logging Framework](#)
 - [4.2.1. About JBoss Logging](#)
 - [4.2.2. Features of JBoss Logging](#)
 - [4.2.3. Add Logging to an Application with JBoss Logging](#)
- [5. Internationalization and Localization](#)
 - [5.1. Introduction](#)
 - [5.1.1. About Internationalization](#)
 - [5.1.2. About Localization](#)
 - [5.2. JBoss Logging Tools](#)
 - [5.2.1. Overview](#)
 - [5.2.2. Creating Internationalized Loggers, Messages and Exceptions](#)
 - [5.2.3. Localizing Internationalized Loggers, Messages and Exceptions](#)
 - [5.2.4. Customizing Internationalized Log Messages](#)
 - [5.2.5. Customizing Internationalized Exceptions](#)
 - [5.2.6. Reference](#)
- [6. Enterprise JavaBeans](#)
 - [6.1. Introduction](#)
 - [6.1.1. Overview of Enterprise JavaBeans](#)
 - [6.1.2. EJB 3.1 Feature Set](#)
 - [6.1.3. EJB 3.1 Lite](#)
 - [6.1.4. EJB 3.1 Lite Features](#)
 - [6.1.5. Enterprise Beans](#)
 - [6.1.6. Overview of Writing Enterprise Beans](#)
 - [6.1.7. Session Bean Business Interfaces](#)
 - [6.2. Creating Enterprise Bean Projects](#)
 - [6.2.1. Create an EJB Archive Project Using JBoss Developer Studio](#)
 - [6.2.2. Create an EJB Archive Project in Maven](#)

6.2.3. Create an EAR Project containing an EJB Project

6.2.4. Add a Deployment Descriptor to an EJB Project

6.3. Session Beans

6.3.1. Session Beans

6.3.2. Stateless Session Beans

6.3.3. Stateful Session Beans

6.3.4. Singleton Session Beans

6.3.5. Add Session Beans to a Project in JBoss Developer Studio

6.4. Message-Driven Beans

6.4.1. Message-Driven Beans

6.4.2. Resource Adapters

6.4.3. Create a JMS-based Message-Driven Bean in JBoss Developer Studio

6.5. Invoking Session Beans

6.5.1. Invoke a Session Bean Remotely using JNDI

6.6. Clustered Enterprise JavaBeans

6.6.1. About Clustered Enterprise JavaBeans (EJBs)

6.7. Reference

6.7.1. EJB JNDI Naming Reference

6.7.2. EJB Reference Resolution

6.7.3. Project dependencies for Remote EJB Clients

6.7.4. jboss-ejb3.xml Deployment Descriptor Reference

7. Clustering in Web Applications

7.1. Session Replication

7.1.1. About HTTP Session Replication

7.1.2. About the Web Session Cache

7.1.3. Configure the Web Session Cache

7.1.4. Enable Session Replication in Your Application

7.2. HttpSession Passivation and Activation

7.2.1. About HTTP Session Passivation and Activation

7.2.2. Configure HttpSession Passivation in Your Application

7.3. Cookie Domain

7.3.1. About the Cookie Domain

7.3.2. Configure the Cookie Domain

7.4. Implement an HA Singleton

8. CDI

8.1. Overview of CDI

8.1.1. Overview of CDI

8.1.2. About Contexts and Dependency Injection (CDI)

8.1.3. Benefits of CDI

8.1.4. About Type-safe Dependency Injection

8.1.5. Relationship Between Weld, Seam 2, and JavaServer Faces

8.2. Use CDI

8.2.1. First Steps

8.2.2. Use CDI to Develop an Application

8.2.3. Ambiguous or Unsatisfied Dependencies

8.2.4. Managed Beans

8.2.5. Contexts, Scopes, and Dependencies

8.2.6. Bean Lifecycle

8.2.7. Named Beans and Alternative Beans

8.2.8. Stereotypes

8.2.9. Observer Methods

- 8.2.10. Interceptors
- 8.2.11. About Decorators
- 8.2.12. About Portable Extensions
- 8.2.13. Bean Proxies

9. Java Transaction API (JTA)

9.1. Overview

- 9.1.1. Overview of Java Transactions API (JTA)

9.2. Transaction Concepts

- 9.2.1. About Transactions
- 9.2.2. About ACID Properties for Transactions
- 9.2.3. About the Transaction Coordinator or Transaction Manager
- 9.2.4. About Transaction Participants
- 9.2.5. About Java Transactions API (JTA)
- 9.2.6. About Java Transaction Service (JTS)
- 9.2.7. About XA Datasources and XA Transactions
- 9.2.8. About XA Recovery
- 9.2.9. About the 2-Phase Commit Protocol
- 9.2.10. About Transaction Timeouts
- 9.2.11. About Distributed Transactions
- 9.2.12. About the ORB Portability API
- 9.2.13. About Nested Transactions
- 9.2.14. About Garbage Collection

9.3. Transaction Optimizations

- 9.3.1. Overview of Transaction Optimizations
- 9.3.2. About the LRCO Optimization for Single-phase Commit (1PC)
- 9.3.3. About the Presumed-Abort Optimization
- 9.3.4. About the Read-Only Optimization

9.4. Transaction Outcomes

- 9.4.1. About Transaction Outcomes
- 9.4.2. About Transaction Commit
- 9.4.3. About Transaction Roll-Back
- 9.4.4. About Heuristic Outcomes
- 9.4.5. JBoss Transactions Errors and Exceptions

9.5. Overview of JTA Transactions

- 9.5.1. About Java Transactions API (JTA)
- 9.5.2. Lifecycle of a JTA Transaction

9.6. Transaction Subsystem Configuration

- 9.6.1. Transactions Configuration Overview
- 9.6.2. Transactional Datasource Configuration
- 9.6.3. Transaction Logging

9.7. Use JTA Transactions

- 9.7.1. Transactions JTA Task Overview
- 9.7.2. Control Transactions
- 9.7.3. Begin a Transaction
- 9.7.4. Nest Transactions
- 9.7.5. Commit a Transaction
- 9.7.6. Roll Back a Transaction
- 9.7.7. Handle a Heuristic Outcome in a Transaction
- 9.7.8. Transaction Timeouts
- 9.7.9. JTA Transaction Error Handling

9.8. ORB Configuration

- 9.8.1. About Common Object Request Broker Architecture (CORBA)
- 9.8.2. Configure the ORB for JTS Transactions

9.9. Transaction References

- 9.9.1. JBoss Transactions Errors and Exceptions
- 9.9.2. JTA Clustering Limitations
- 9.9.3. JTA Transaction Example
- 9.9.4. API Documentation for JBoss Transactions JTA

10. Hibernate

10.1. About Hibernate Core

10.2. Java Persistence API (JPA)

- 10.2.1. About JPA
- 10.2.2. Hibernate EntityManager
- 10.2.3. Getting Started
- 10.2.4. Configuration
- 10.2.5. Second-Level Caches

10.3. Hibernate Annotations

- 10.3.1. Hibernate Annotations

10.4. Hibernate Query Language

- 10.4.1. About Hibernate Query Language
- 10.4.2. HQL Statements
- 10.4.3. About the INSERT Statement
- 10.4.4. About the FROM Clause
- 10.4.5. About the WITH Clause
- 10.4.6. About Collection Member References
- 10.4.7. About Qualified Path Expressions
- 10.4.8. About Scalar Functions
- 10.4.9. HQL Standardized Functions
- 10.4.10. About the Concatenation Operation
- 10.4.11. About Dynamic Instantiation
- 10.4.12. About HQL Predicates
- 10.4.13. About Relational Comparisons
- 10.4.14. About the IN Predicate
- 10.4.15. About HQL Ordering

10.5. Hibernate Services

- 10.5.1. About Hibernate Services
- 10.5.2. About Service Contracts
- 10.5.3. Types of Service Dependencies
- 10.5.4. The ServiceRegistry
- 10.5.5. Custom Services
- 10.5.6. The Bootstrap Registry
- 10.5.7. The SessionFactory Registry
- 10.5.8. Integrators

10.6. Bean Validation

- 10.6.1. About Bean Validation
- 10.6.2. Hibernate Validator
- 10.6.3. Validation Constraints
- 10.6.4. Configuration

10.7. Envers

- 10.7.1. About Hibernate Envers
- 10.7.2. About Auditing Persistent Classes
- 10.7.3. Auditing Strategies
- 10.7.4. Getting Started with Entity Auditing
- 10.7.5. Configuration
- 10.7.6. Queries

11. JAX-RS Web Services

11.1. About JAX-RS

- 11.2. About RESTEasy
- 11.3. About RESTful Web Services
- 11.4. RESTEasy Defined Annotations
- 11.5. RESTEasy Configuration
 - 11.5.1. RESTEasy Configuration Parameters
- 11.6. JAX-RS Web Service Security
 - 11.6.1. Enable Role-Based Security for a RESTEasy JAX-RS Web Service
 - 11.6.2. Secure a JAX-RS Web Service using Annotations
- 11.7. RESTEasy Logging
 - 11.7.1. About JAX-RS Web Service Logging
 - 11.7.2. Configure a Log Category in the Management Console
 - 11.7.3. Logging Categories Defined in RESTEasy
- 11.8. Exception Handling
 - 11.8.1. Create an Exception Mapper
 - 11.8.2. RESTEasy Internally Thrown Exceptions
- 11.9. RESTEasy Interceptors
 - 11.9.1. Intercept JAX-RS Invocations
 - 11.9.2. Bind an Interceptor to a JAX-RS Method
 - 11.9.3. Register an Interceptor
 - 11.9.4. Interceptor Precedence Families
- 11.10. String Based Annotations
 - 11.10.1. Convert String Based @*Param Annotations to Objects
- 11.11. Configure File Extensions
 - 11.11.1. Map File Extensions to Media Types in the web.xml File
 - 11.11.2. Map File Extensions to Languages in the web.xml File
 - 11.11.3. RESTEasy Supported Media Types
- 11.12. RESTEasy JavaScript API
 - 11.12.1. About the RESTEasy JavaScript API
 - 11.12.2. Enable the RESTEasy JavaScript API Servlet
 - 11.12.3. RESTEasy Javascript API Parameters
 - 11.12.4. Build AJAX Queries with the JavaScript API
 - 11.12.5. REST.Request Class Members
- 11.13. RESTEasy Asynchronous Job Service
 - 11.13.1. About the RESTEasy Asynchronous Job Service
 - 11.13.2. Enable the Asynchronous Job Service
 - 11.13.3. Configure Asynchronous Jobs for RESTEasy
 - 11.13.4. Asynchronous Job Service Configuration Parameters
- 11.14. RESTEasy JAXB
 - 11.14.1. Create a JAXB Decorator
- 11.15. RESTEasy Atom Support
 - 11.15.1. About the Atom API and Provider

12. JAX-WS Web Services

- 12.1. About JAX-WS Web Services
- 12.2. Configure the webservicess Subsystem
- 12.3. JAX-WS Web Service Endpoints
 - 12.3.1. About JAX-WS Web Service Endpoints
 - 12.3.2. Write and Deploy a JAX-WS Web Service Endpoint
- 12.4. JAX-WS Web service Clients
 - 12.4.1. Consume and Access a JAX-WS Web Service
 - 12.4.2. Develop a JAX-WS Client Application

- 12.5. JAX-WS Development Reference
 - 12.5.1. Enable Web Services Addressing (WS-Addressing)
 - 12.5.2. JAX-WS Common API Reference
- 13. Identity Within Applications
 - 13.1. Foundational Concepts
 - 13.1.1. About Encryption
 - 13.1.2. About Security Domains
 - 13.1.3. About SSL Encryption
 - 13.1.4. About Declarative Security
 - 13.2. Role-Based Security in Applications
 - 13.2.1. About Application Security
 - 13.2.2. About Authentication
 - 13.2.3. About Authorization
 - 13.2.4. About Security Auditing
 - 13.2.5. About Security Mapping
 - 13.2.6. About the Security Extension Architecture
 - 13.2.7. Java Authentication and Authorization Service (JAAS)
 - 13.2.8. About Java Authentication and Authorization Service (JAAS)
 - 13.2.9. Use a Security Domain in Your Application
 - 13.2.10. Use Role-Based Security In Servlets
 - 13.2.11. Use A Third-Party Authentication System In Your Application
 - 13.3. Security Realms
 - 13.3.1. About Security Realms
 - 13.3.2. Add a New Security Realm
 - 13.3.3. Add a User to a Security Realm
 - 13.4. EJB Application Security
 - 13.4.1. Security Identity
 - 13.4.2. EJB Method Permissions
 - 13.4.3. EJB Security Annotations
 - 13.4.4. Remote Access to EJBs
 - 13.5. JAX-RS Application Security
 - 13.5.1. Enable Role-Based Security for a RESTEasy JAX-RS Web Service
 - 13.5.2. Secure a JAX-RS Web Service using Annotations
 - 13.6. Secure Remote Password Protocol
 - 13.6.1. About Secure Remote Password Protocol (SRP)
 - 13.6.2. Configure Secure Remote Password (SRP) Protocol
 - 13.7. Password Vaults for Sensitive Strings
 - 13.7.1. About Securing Sensitive Strings in Clear-Text Files
 - 13.7.2. Create a Java Keystore to Store Sensitive Strings
 - 13.7.3. Mask the Keystore Password and Initialize the Password Vault
 - 13.7.4. Configure the JBoss Enterprise Application Platform to Use the Password Vault
 - 13.7.5. Store and Retrieve Encrypted Sensitive Strings in the Java Keystore
 - 13.7.6. Store and Resolve Sensitive Strings In Your Applications
 - 13.8. Java Authorization Contract for Containers (JACC)
 - 13.8.1. About Java Authorization Contract for Containers (JACC)
 - 13.8.2. Configure Java Authorization Contract for Containers (JACC) Security
 - 13.9. Java Authentication SPI for Containers (JASPI)
 - 13.9.1. About Java Authentication SPI for Containers (JASPI) Security
 - 13.9.2. Configure Java Authentication SPI for Containers (JASPI) Security
- 14. Single Sign On (SSO)
 - 14.1. About Single Sign On (SSO) for Web Applications

- [14.2. About Clustered Single Sign On \(SSO\) for Web Applications](#)
 - [14.3. Choose the Right SSO Implementation](#)
 - [14.4. Use Single Sign On \(SSO\) In A Web Application](#)
 - [14.5. About Kerberos](#)
 - [14.6. About SPNEGO](#)
 - [14.7. About Microsoft Active Directory](#)
 - [14.8. Configure Kerberos or Microsoft Active Directory Desktop SSO for Web Applications](#)
- [15. Development Security References](#)
 - [15.1. jboss-web.xml Configuration Reference](#)
 - [15.2. EJB Security Parameter Reference](#)
- [16. Supplemental References](#)
 - [16.1. Types of Java Archives](#)
- [A. Revision History](#)

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the [Liberation Fonts](#) set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

Mono-spaced Bold

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keys and key combinations. For example:

To see the contents of the file **my_next_bestselling_novel** in your current working directory, enter the **cat my_next_bestselling_novel** command at the shell prompt and press **Enter** to execute the command.

The above includes a file name, a shell command and a key, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from an individual key by the plus sign that connects each part of a key combination. For example:

Press **Enter** to execute the command.

Press **Ctrl+Alt+F2** to switch to a virtual terminal.

The first example highlights a particular key to press. The second example highlights a key combination: a set of three keys pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **mono-spaced bold**. For example:

File-related classes include **filesystem** for file systems, **file** for files, and **dir** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a **gedit** file, choose **Applications** → **Accessories** → **Character Map** from the main menu bar. Next, choose **Search** → **Find...** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu

bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

Mono-spaced Bold Italic* or *Proportional Bold Italic

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type **ssh *username@domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh john@example.com**.

The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount /home**.

To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — *username*, *domain.name*, *file-system*, *package*, *version* and *release*. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **mono-spaced roman** and presented thus:

```
books      Desktop  documentation  drafts  mss    photos  stuff  svn
books_tests Desktop1  downloads      images  notes  scripts svgs
```

Source-code listings are also set in **mono-spaced roman** but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
    public static void main(String args[])
        throws Exception
    {
        InitialContext iniCtx = new InitialContext();
        Object          ref    = iniCtx.lookup("EchoBean");
        EchoHome         home   = (EchoHome) ref;
        Echo              echo   = home.create();

        System.out.println("Created Echo");

        System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
    }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.



Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.



Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.



Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. Getting Help and Giving Feedback

2.1. Do You Need Help?

If you experience difficulty with a procedure described in this documentation, visit the Red Hat Customer Portal at <http://access.redhat.com>. Through the customer portal, you can:

- search or browse through a knowledgebase of technical support articles about Red Hat products.
- submit a support case to Red Hat Global Support Services (GSS).
- access other product documentation.

Red Hat also hosts a large number of electronic mailing lists for discussion of Red Hat software and technology. You can find a list of publicly available mailing lists at <https://www.redhat.com/mailman/listinfo>. Click on the name of any mailing list to subscribe to that list or to access the list archives.

2.2. Give us Feedback

If you find a typographical error, or know how this guide can be improved, we would love to hear from you. Submit a report in Bugzilla against the product **JBoss Enterprise Application Platform 6** and the component **doc-Development_Guide**. The following link will take you to a pre-filled bug report for this product: <https://bugzilla.redhat.com/>.

Fill out the following template in Bugzilla's **Description** field. Be as specific as possible when describing the issue; this will help ensure that we can fix it quickly.

Document URL:

Section Number and Name:

Describe the issue:

Suggestions for improvement:

Additional information:

Be sure to give us your name so that you can receive full credit for reporting the issue.

Chapter 1. Get Started Developing Applications

1.1. Introduction

1.1.1. Introducing JBoss Enterprise Application Platform 6

JBoss Enterprise Application Platform 6 is a middleware platform built on open standards, and compliant with Java EE. It integrates JBoss Application Server 7 with high-availability clustering, powerful messaging, distributed caching, and other technologies to create a stable, scalable, and fast platform. In addition, it also includes APIs and development frameworks you can use to develop secure, powerful, and scalable Java EE applications quickly.

[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 Enterprise Application Platform 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) 2.0 (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. JBoss Enterprise Edition 6 supports the Full Profile.

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 Enterprise Application Platform 6

1.2.2.1. Modules

A Module is a logical grouping of classes used for class loading and dependency management. JBoss Enterprise Application Platform 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. All modules provide the same features.

Static Modules

Static Modules are predefined in the **EAP_HOME/modules/** directory of the application server. Each sub-directory represents one module and contains one or more JAR files and a configuration file (**module.xml**). 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.

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.

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](#)

1.2.2.2. Overview of Class Loading and Modules

JBoss Enterprise Application Platform 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 favour 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.

The following material covers what developers need to know to successfully build and deploy applications on JBoss Enterprise Application Platform 6.

[Report a bug](#)

1.3. Install JBoss Enterprise Application Platform 6

1.3.1. Download and Install JBoss Enterprise Application Platform 6

1.3.1.1. Download JBoss Enterprise Application Platform 6

1. Log into the Customer Service Portal at <https://access.redhat.com>.
2. From the menu, select **Downloads** → **JBoss Enterprise Middleware** → **Downloads**.
3. Select **Application Platform** from the **Product** drop-down box.
4. Locate the appropriate Application Platform version and click the **Download** link.
5. Download any other available packages you need, such as the Quickstarts, Maven Repository, HTTP Connectors, or Native binaries.

Result

JBoss Enterprise Application Platform 6 and any supplemental files that you selected are downloaded to your computer.

[Report a bug](#)

1.3.1.2. Install JBoss Enterprise Application Platform 6 Using the ZIP Download

Summary

The Zip file installation method is appropriate for all support operating systems.

Prerequisites

Before you can install JBoss Enterprise Application Platform 6, you need to download the Zip archive from the Red Hat Customer Service Portal.

Procedure 1.1. Task

1. **Move the Zip archive to the desired location.**

Move the Zip file to the server and directory where you want to install JBoss Enterprise Application Platform 6. The directory should be accessible by the user who will start and stop the server.

2. **Use an appropriate application to extract the Zip archive.**

In Linux, the command to extract a Zip archive is typically called **unzip**. In a Microsoft Windows environment, right-click the file and select **Extract All**.

Result

The directory created by extracting the Zip archive is the top-level directory for JBoss Enterprise Application Platform 6. This is typically referred to as **EAP_HOME**. If you ever decide to move your installation, you can move this directory to another directory or another server.

[Report a bug](#)

1.3.2. Take a Quick Tour of JBoss Enterprise Application Platform 6

1.3.2.1. Installation Structure and Details

JBoss Enterprise Application Platform 6 includes a simplified directory structure, compared to previous versions. Following is a listing of the directory structure, and a description of what the directory contains.

Table 1.1. Top-level directories and files

Name	Purpose
appclient/	Contains configuration details for the application client container.
bin/	Contains start-up scripts for JBoss Enterprise Application Platform 6 on Red Hat Enterprise Linux and Microsoft Windows.
bundles/	Contains OSGi bundles which pertain to JBoss Enterprise Application Platform 6 internal functionality.
docs/	License files, schemas, and examples.
domain/	Configuration files, deployment content, and writable areas used when JBoss Enterprise Application Platform 6 runs as a managed domain.
modules/	Modules which are dynamically loaded by JBoss Enterprise Application Platform 6 when services request them.
standalone/	Configuration files, deployment content, and writable areas used when JBoss Enterprise Application Platform 6 runs as a standalone server.
welcome-content/	Contains content used by the Welcome web application which is available on port 8080 of a default installation.
jboss-modules.jar	The bootstrapping mechanism which loads modules.

Table 1.2. Directories within the domain/ directory

Name	Purpose
configuration/	Configuration files for the managed domain. These files are modified by the Management Console and Management CLI, and are not meant to be edited directly.
data/	Information about deployed services. Services are deployed using the Management Console and Management CLI, rather than by a deployment scanner. Therefore, do not place files in this directory manually.
log/	Contains the run-time log files for the host and process controllers which run on the local instance.
servers/	Contains the equivalent data/ , log/ , and tmp/ directories for each server instance in a domain, which contain similar data to the same directories within the top-level domain/ directory.
tmp/	Contains temporary data such as files pertaining to the shared-key mechanism used by the Management CLI to authenticate local users to the managed domain.

Table 1.3. Directories within the standalone/ directory

Name	Purpose
configuration/	Configuration files for the standalone server. These files are modified by the Management Console and Management CLI, and are not meant to be edited directly.
deployments/	Information about deployed services. The standalone server does include a deployment scanner, so you can place archives in this directory to be deployed. However, the recommended approach is to manage deployments using the Management Console or Management CLI.
lib/	External libraries which pertain to a standalone server mode. Empty by default.
tmp/	Contains temporary data such as files pertaining to the shared-key mechanism used by the Management CLI to authenticate local users to the server.

[Report a bug](#)

1.3.2.2. About Standalone Servers

A standalone server is one of two operating modes for the JBoss Enterprise Application Platform. The other is a managed domain. A standalone server is analogous to the only running mode of previous versions of the JBoss Enterprise Application Platform.

A JBoss Enterprise Application Platform instance running as a standalone server is a single instance only, but can optionally run in a clustered configuration.

[Report a bug](#)

1.3.2.3. About Managed Domains

A *managed domain* is one of two operating modes for a JBoss Enterprise Application Platform instance. The other is a *standalone server*.

A domain consists of one *domain controller*, one or more *host controller(s)*, and zero or more servers per host. Servers are members of server groups. The domain controller manages the configuration and applications deployed onto server groups. Each server in a server group shares the same configuration and deployments.

The domain controller is also a host controller. The other host controllers are configured to delegate domain management tasks to it. It is possible for the domain controller, a single host controller, and multiple servers to run within the same instance of the JBoss Enterprise Application Platform, on the same physical system. Host controllers are tied to specific physical (or virtual) hosts. You can run multiple host controllers on the same hardware if you use different configurations, so that the ports and other resources do not conflict.

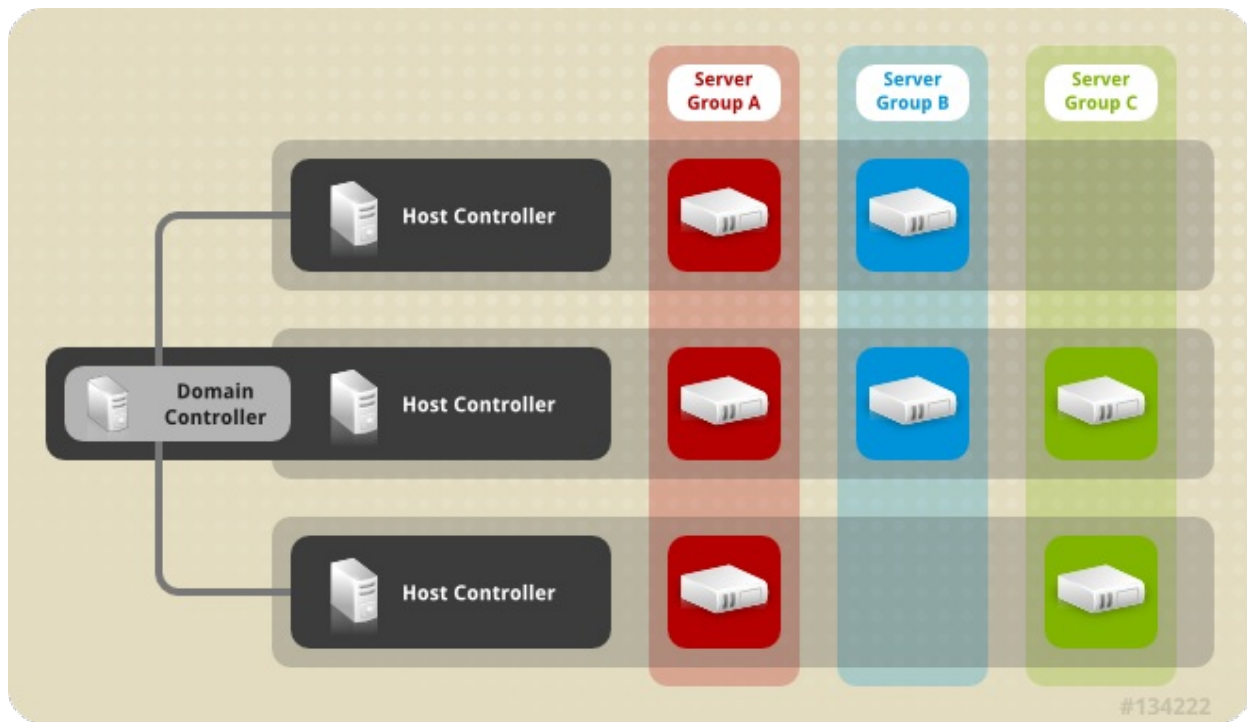


Figure 1.1. Graphical Representation of a Managed Domain

[Report a bug](#)

1.3.2.4. New and Changed Features in JBoss Enterprise Application Platform 6

- ▶ JBoss Enterprise Application Platform 6 is a certified implementation of the Java Enterprise Edition 6 Full Profile and Web Profile specifications.
- ▶ A Managed Domain provides centralized management of multiple server instances and physical hosts, while a Standalone Server allows for a single server instance.
- ▶ Configurations, deployments, socket bindings, modules, extensions, and system properties can all be managed per server group.
- ▶ The Management Console and Management CLI are brand new interfaces for managing your domain or standalone JBoss Enterprise Application Platform 6 instance. There is no longer any need to edit XML configuration files by hand. The Management CLI even offers batch mode, so that you can script and automate management tasks.
- ▶ Application security, including security domains, are managed centrally for simplified configuration.
- ▶ The directory layout of JBoss Enterprise Application Platform 6 has been simplified. The **modules/** directory now contains the application server modules, instead of using common and server-specific **lib/** directories. The **domain/** and **standalone/** directories contain the artifacts and configuration files for domain and standalone deployments.
- ▶ The classloading mechanism has been made completely modular, so that modules are loaded and unloaded on demand. This provides performance and security benefits, as well as very fast start-up and restart times.
- ▶ Datasource management is streamlined. Database drivers can be deployed just like other services. In addition, datasources are created and managed directly in the Management Console or Management CLI.
- ▶ JBoss Enterprise Application Platform 6 starts and stops very quickly, which is especially beneficial to developers. It uses fewer resources and is extremely efficient in its use of system resources.

[Report a bug](#)

1.3.2.5. New Terminology in JBoss Enterprise Application Platform 6

JBoss Enterprise Application Platform 6 introduces significant changes from previous versions, as well as many completely new features and concepts. This topic groups and summarizes new terminology, with links to more information about the specific terms.

Administration and Configuration

Managed Domain

A managed domain is one of two modes the JBoss Enterprise Application Platform can use. In a managed domain, multiple physical or virtual servers (host controllers) are managed by a central domain controller. Individual instances of the container (servers) are organized into server groups. Each server in a server group shares the same configuration and deployments. Server groups can include servers from multiple physical or virtual hosts. A server group's configuration is referred to as a profile.

A managed domain is a configuration paradigm, and has nothing to do with clustering or high availability.

Domain Controller

This term applies to managed domains. A process which runs on managed domain instance which manages and propagates configurations across physical hosts, server groups, and servers. By default, the domain controller process runs on the local host, but you can configure a physical host to connect to a different domain controller.

domain.xml

The **`EAP_HOME/domain/configuration/domain.xml`** file is the central configuration for a JBoss Enterprise Application Platform 6 managed domain. It includes all configuration details that are not host-specific. It is recommended to use the Management CLI or web-based Management Console to manage the configuration details, because the configuration file itself is overwritten on a regular basis to maintain a persistent configuration.

Host Controller

This term applies to managed domains. A process which runs on each separate instance of JBoss Enterprise Application Platform. Each physical host can be configured to use different network interfaces, JVM properties, and other settings which apply to the physical host separate from its membership in a domain.

Standalone Server

A standalone server is one of two modes in which you can run JBoss Enterprise Application Platform 6. It is similar to the way previous versions of JBoss Enterprise Application Platform worked. Each server has its own locally-managed configuration, and deployments are managed locally.

standalone.xml

The **`EAP_HOME/standalone/configuration/standalone.xml`** file is the central configuration for a JBoss Enterprise Application Platform 6 standalone server. It includes all configuration details that are not host-specific. It is recommended to use the Management CLI or web-based Management Console to manage the configuration details, because the configuration file itself is overwritten on a regular basis to maintain a persistent configuration.

Some additional example standalone configurations are included, with filenames that follow the pattern **`standalone-PROFILE.xml`**. They enable different subsystems for a standalone server.

Server Group

This terminology applies to managed domains. A server group is a group of virtual servers, which can be on multiple physical hosts within a domain. Each server in the server group shares the same configuration and deployments. The server group is not a cluster, for purposes of high availability or session replication. Also, a server group does not exist on any physical host, but is a virtual grouping. A server group's configuration is controlled by a profile.

Server

A server can mean different things in the context of a managed domain or standalone server.

In a managed domain, a server is a member of a server group. It is a virtual instance of the Platform which runs on a physical host controller. Servers which run on different physical hosts can be members of the same server group. Most server configuration is managed by the server group, but some properties are able to be set on the physical host where it resides.

In a standalone server, the server refers to the entire instance. In essence, a standalone server only has a single configuration profile, whereas a managed domain can have many profiles, and each profile may be applied to one or more server groups.

Socket Binding Group

This terminology applies to managed domains. A socket binding group is a group of socket bindings which can be applied to a server group's configuration. The socket bindings your server group needs are dictated by its deployments. Socket binding groups allow you to customize the available socket bindings in a granular way.

Socket Binding

A socket binding is a mapping between a physical network port, a network protocol such as TCP, UDP, or ICMP, and a logical name. Socket bindings abstract port assignments so that they can be referred to by logical names.

Port Offset

If you need to run multiple servers on the same physical host, network port conflicts can occur if they each use similar ports. You can assign a port offset to a socket binding group. The port offset is an integer added to each port number. For instance, a port offset of 100 applied to port 8080 would result in port 8180 being assigned. You can cancel the offset value on a port-by-port basis.

Module

Modules are logical groupings of classes used for class loading and dependency management. Modules can be static or dynamic. Modules are only loaded when a deployment requests them. This is referred to as modular or dynamic classloading.

Static modules exist before JBoss Enterprise Application Platform starts. Each API which ships with JBoss Enterprise Application Platform is a static module. Even though static modules are pre-defined, they are only loaded on demand.

Dynamic modules are created and loaded on demand, when a deployment needs them. The name of a dynamic module is derived from the name of the deployed archive.

Management Console

The Management Console is a web-based administration and management interface for JBoss Enterprise Application Platform. It is available on port 9990 by default. It allows you to configure

the Platform in a graphical way, whether you use a managed domain or standalone server. It is a deployed Web Application which uses the Management API to read and write configuration details to the central configuration file.

Management CLI

The Management CLI is a command-line based management interface, which can be run in a Red Hat Enterprise Linux or Microsoft Windows terminal. The Management CLI allows for scripting and batch operations, as well as the ability to save and revert to different versions of your configuration. It can manage local or remote instances of the Platform. It uses the Management API to read and write configuration details to the central configuration file.

Management API

The Management API is a REST-like API which allows you to manage JBoss Enterprise Application Platform in a flexible way. It uses a CRUD (Create, Read, Update, Delete) paradigm, and its output is in JSON format. The web-based Management Console and the Management CLI each use the Management API for their core functions.

Profiles

The meaning of profiles changes slightly, depending on whether you use a managed domain or standalone server.

In a managed domain, a profile is a group of configuration options which apply to a server group. A profile, a socket binding group, and a set of deployments combine together to configure a server group. You can configure profiles in the Management Console or Management CLI.

In a standalone server, a profile refers to the entire configuration of JBoss Enterprise Application Platform. Some example profiles are provided, so that you start with the configuration which is closest to the one you need. You then customize it further, so that it matches your requirements.

A Java EE 6 Profile refers to a feature of the Java EE 6 API. Refer to the definition of **Java EE 6 Profile** in this document for more information.

Java EE 6 Profile

The Java EE 6 API defines the concept of a profile, which is a group of APIs which are provided as part of a Java EE implementation. Two profiles are defined in the Java EE 6 specification: **Full Profile** and **Web Profile**. Additional profiles can be created and provided as desired by the developers of a specific implementation. JBoss Enterprise Application Platform 6 is fully compliant with both the Full Profile and Web Profile specifications.

Modular Class Loading

Modular Class Loading refers to the way that JBoss Enterprise Application Platform loads modules (groups of classes). Modules are only loaded into memory when a deployment requests them. They are unloaded as soon as no deployment needs them. Modular class loading has positive performance and security implications. Modular class loading mitigates the need to manually slim your configuration.

Subsystem

A subsystem is a configurable component of JBoss Enterprise Application Platform. A subsystem configuration applies to a module, or group of classes. Each API provided with JBoss Enterprise Application Platform, such as JPA, JCA, Security, and **mod_cluster**, is

configurable at the subsystem level. Each subsystem's configuration follows a specified Document Type Definition (DTD), which is provided in the **docs/** directory of your installation. You can create your own subsystems to extend JBoss Enterprise Application Platform 6.

Configuration File History

JBoss Enterprise Application Platform 6 is designed to be configured using the Management Console, Management CLI, or Management API, rather than editing XML by hand. Configuration changes are persisted to the filesystem automatically. In tandem with these developments, the ability has been added for you to save, load, and roll back or forward to different versions of your configuration.

Filesystem Paths

JBoss Enterprise Application Platform 6 allows logical names for filesystem paths to be specified in your configuration. This allows specific host configurations to resolve to universal logical names, and aids in configuration portability.

Password Vault

JBoss Enterprise Application Platform 6 includes a mechanism for masking passwords and other sensitive strings in an encrypted password vault. You can use the built-in vault mechanism or create your own implementation for your own applications.

JBoss LogManager

JBoss LogManager is the JBoss Enterprise Application Platform 6 subsystem responsible for capturing and dealing with log messages sent by applications and the other JBoss Enterprise Application Platform 6 subsystems. JBoss LogManager supports several popular application logging frameworks.

Resource Adapter

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.

Development

Contexts and Dependency Injection (CDI)

Contexts and Dependency Injection (CDI) is a specification designed to enable EJB components to be used as managed beans, unifying the two component models and streamlining the development process. JBoss Enterprise Application Platform implements CDI via Weld, which is the reference implementation of CDI. CDI is explained in JCP JSR-299.

EJB 3.1

Enterprise JavaBeans (EJBs) are container-managed objects that encapsulate the different business logic layers of your application. The EJB 3.1 specification defines Session Beans and Message-Driven Beans.

Portable JNDI Namespaces

EJB 3.1 introduced a standardized global JNDI namespace and a series of related namespaces that map to the various scopes of a Java EE application. The three JNDI namespaces used for

portable JNDI lookups are **java:global**, **java:module**, and **java:app**.

JBoss Enterprise Application Platform 6 has taken advantage of these new standards, and enforces properly formatted JNDI names. Unqualified relative or absolute names now need to be fully qualified, relative to one of the top-level namespaces **comp**, **module**, **app**, **global**, or **jboss**. JNDI names which do not conform to these guidelines result in an **invalid name** error.

Maven Repository

Apache Maven is a distributed build automation tool used in Java application development to create, manage, and build software deliverables in a standard way. Maven uses configuration files called POM files to identify, locate, and download build dependencies from local or remote repositories.

JBoss Enterprise Application Platform 6 includes a Maven repository which you can download from the Red Hat Customer Service Portal (CSP). It includes many of the requirements that Java EE developers typically use to build their applications. The Maven repository is provided as a convenience, and meets the requirements of organizations which need to provide the dependencies locally, rather than downloading them from remote sources.

Second Level Cache (2LC)

A Second Level Cache (2LC) is a data store which holds persistent information relating to application state. Examples of 2LC consumers are session state replication, Single Sign On (SSO), and Java Persistence API (JPA). JBoss Enterprise Application Platform 6 uses Infinispan to manage its 2LC.

JBoss Logging

JBoss Logging is the application logging framework included in JBoss Enterprise Application Platform. It provides mechanisms for adding logging to your applications, as well as localizing and internationalizing your log messages. It is the default logging framework, but you can choose to use third-party logging frameworks, such as log4j, in addition to or instead of JBoss Logging.

Entity Auditing of Persistent Classes

JBoss Enterprise Application Platform 6 includes Hibernate Envers, which allows you to audit and retain versions of persistent classes representing datasource information.

jboss-ejb3.xml Deployment Descriptor

The **jboss-ejb3.xml** deployment descriptor replaces the previous **jboss.xml** deployment descriptor. It is used for overriding and adding to the features provided by the Java Enterprise Edition (EE) defined **ejb3-jar.xml** deployment descriptor. The new file is incompatible with **jboss.xml**, and the **jboss.xml** is now ignored in deployments.

jboss-client.jar Client Application JAR

The **JBOSS_HOME/bin/client/jboss-client.jar** bundles the libraries needed by client applications to connect remotely to JBoss Enterprise Application Platform 6. This JAR replaces the **jbossall-client.jar** that was located in the **JBOSS_HOME/client/** directory in previous versions of JBoss Enterprise Application Platform.

jboss-deployment-structure.xml Deployment Descriptor

The `jboss-deployment-structure.xml` deployment descriptor is a new descriptor which controls classloading in a granular way. It is located in the **META-INF/** or **WEB-INF/** directory of the deployment. It includes the ability to prevent automatic dependencies from being added, to add additional dependencies which would not otherwise be added, to define additional modules, to change the classloading behavior of an EAR, and to add additional resource roots to a module.

[Report a bug](#)

1.4. Set Up the Development Environment

1.4.1. Download and Install JBoss Developer Studio

1.4.1.1. Setup the JBoss Developer Studio

1. [Section 1.4.1.2, “Download JBoss Developer Studio 5”](#)
2. [Section 1.4.1.3, “Install JBoss Developer Studio 5”](#)
3. [Section 1.4.1.4, “Start JBoss Developer Studio”](#)

[Report a bug](#)

1.4.1.2. Download JBoss Developer Studio 5

1. Go to <https://access.redhat.com/>.
2. Select **Downloads** → **JBoss Enterprise Middleware** → **Downloads**.
3. Select **JBoss Developer Studio** from the dropbox.
4. Select the appropriate version and click **Download**.

[Report a bug](#)

1.4.1.3. Install JBoss Developer Studio 5

Prerequisites:

[Section 1.4.1.2, “Download JBoss Developer Studio 5”](#)

Procedure 1.2. Task:

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 JBoss Developer Studio, and click **Next**.
12. Click **Done**.

[Report a bug](#)

1.4.1.4. Start JBoss Developer Studio

Prerequisites:

[Section 1.4.1.3, “Install JBoss Developer Studio 5”](#)

Procedure 1.3. Task:

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

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

[Report a bug](#)

1.4.1.5. Add the JBoss Enterprise Application Platform 6 Server to JBoss Developer Studio

These instructions assume this is your first introduction to JBoss Developer Studio and you have not yet added any JBoss Enterprise Application Platform 6 servers.

Procedure 1.4. 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 the **new server wizard** link or right click within the blank Server panel and select **New** → **Server**.

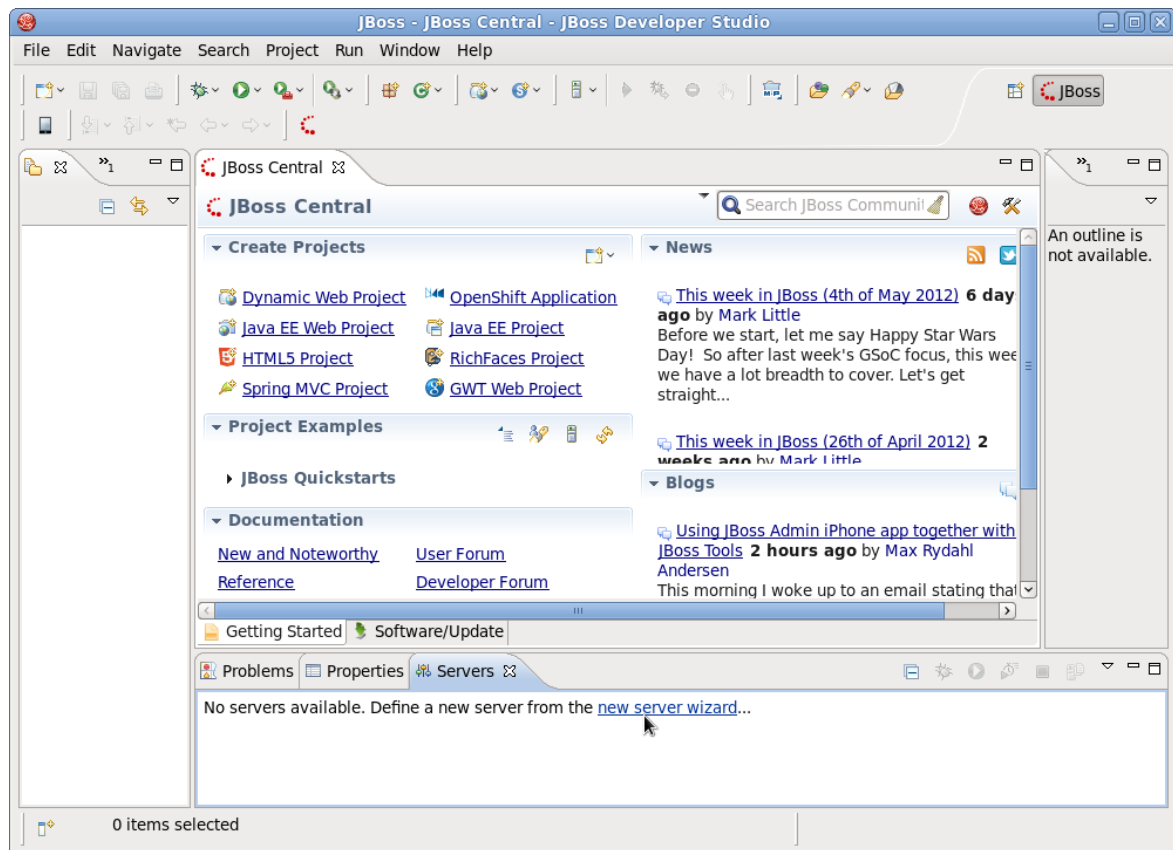


Figure 1.2. Add a new server - No servers available

3. Expand **JBoss Enterprise Middleware** and choose **JBoss Enterprise Application Platform 6.x**. Then click **Next**.

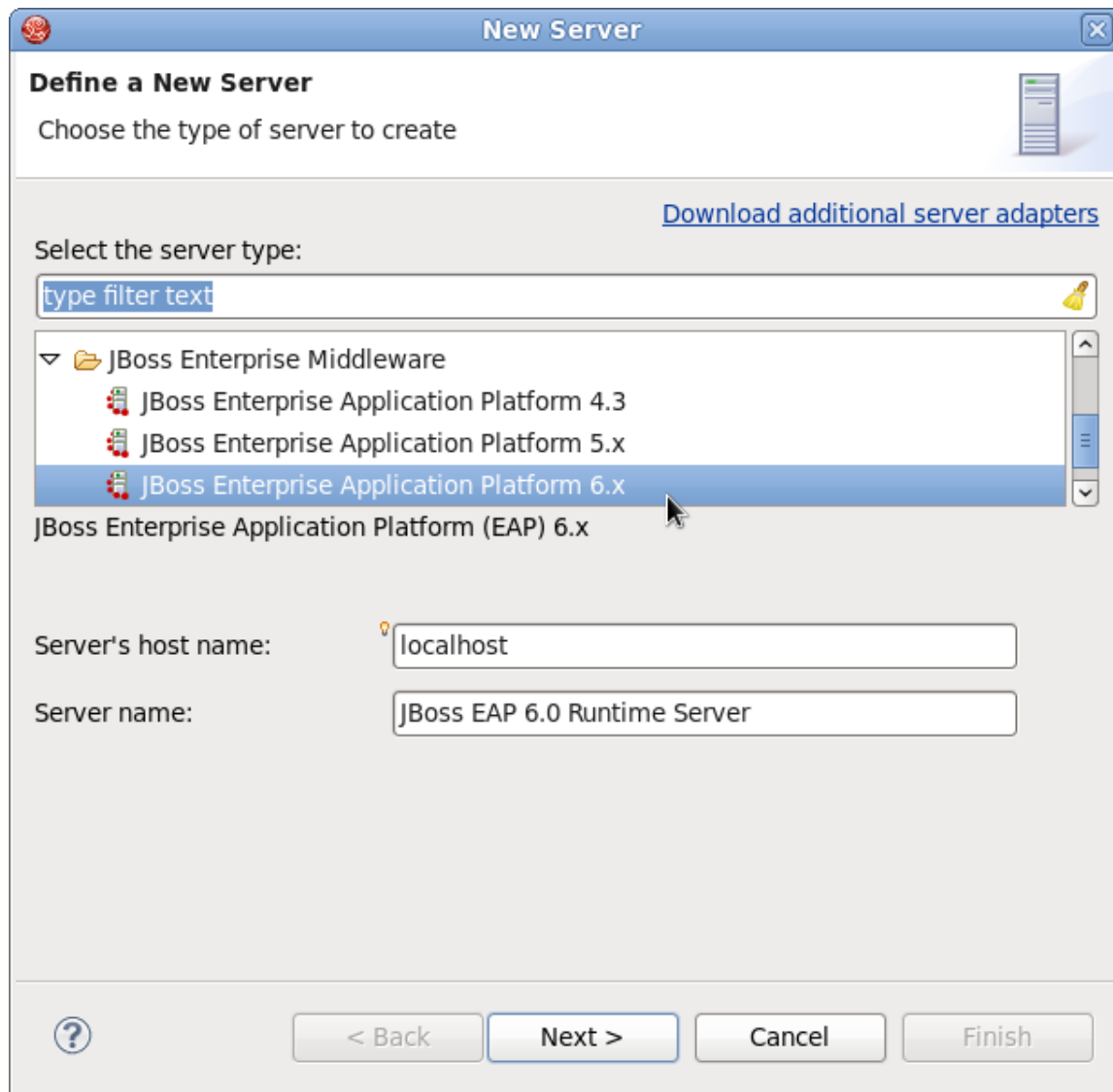


Figure 1.3. Choose server type

4. Click **Browse** and navigate to your JBoss Enterprise Application Platform 6 install location. Then click **Next**.

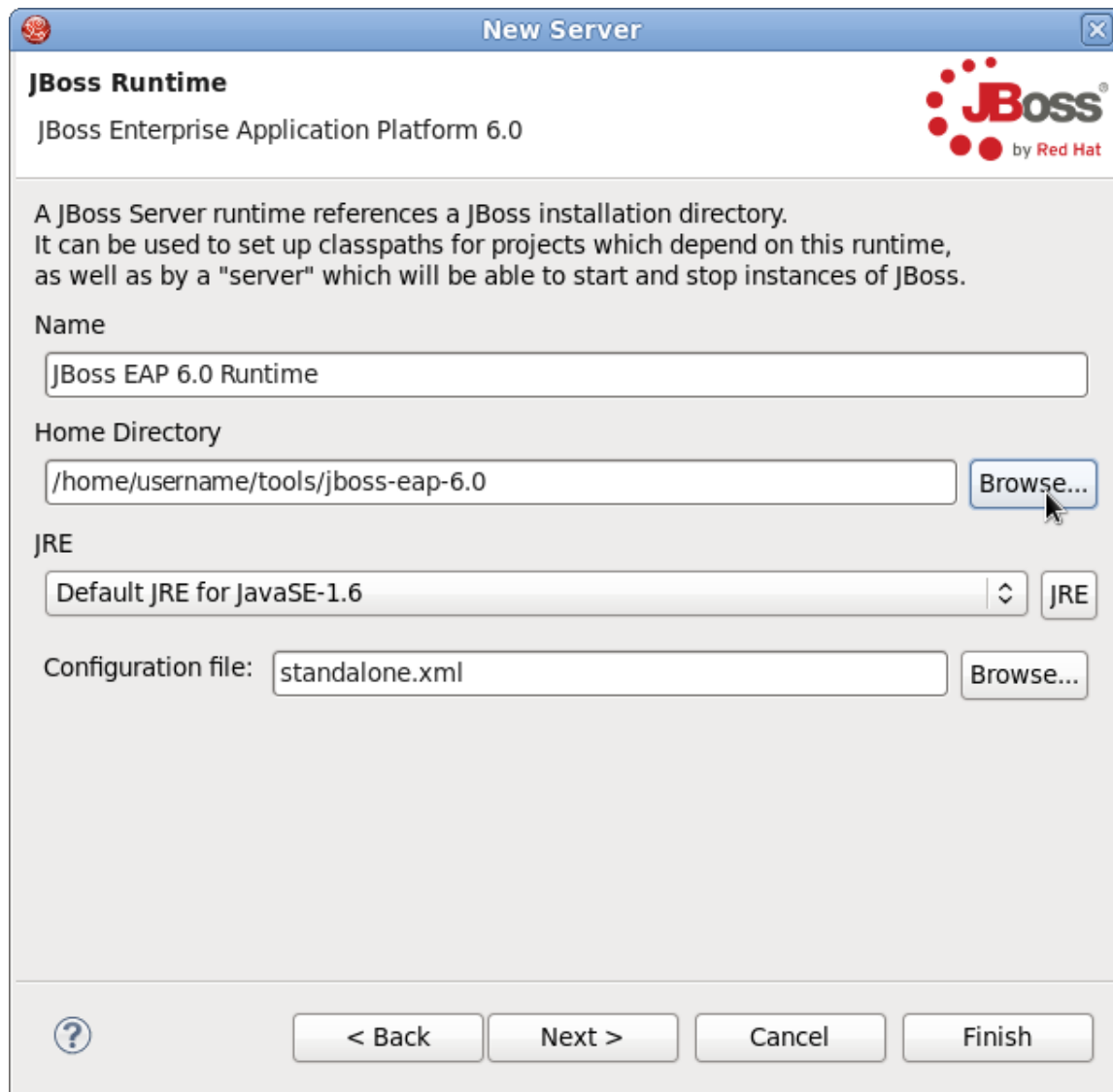



Figure 1.4. Browse to server install

5. On this screen you define the server behavior. You can start the server manually or let JBoss Developer Studio manage it for you. You can also define a remote server for deployment and determine if you want to expose the management port for that server, for example, if you need connect to it using JMX. In this example, we assume the server is local and you want JBoss Developer Studio to manage your server so you do not need to check anything. Click **Next**.



Create a new JBoss Server

JBoss Enterprise Application Platform 6.0

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

Runtime Information

If the runtime information below is incorrect, please press back, Installed Runtimes..., and then Add to create a new runtime from a different location.

Home Directory	/home/username/tools/jboss-eap-6.0
Execution Environment	Java Platform, Standard Edition 6.0
JRE	Default JRE for JavaSE-1.6

Server Behaviour

- ☐ Server is externally managed. Assume server is started.
- ☐ Listen on all interfaces to allow remote web connections
- ☐ Expose your management port as the server's hostname

Local

? < Back Next > Cancel Finish

Figure 1.5. Define the new JBoss server behavior

- This screen allows you to configure existing projects for the new server. Since you do not have any projects at this point, click **Finish**.

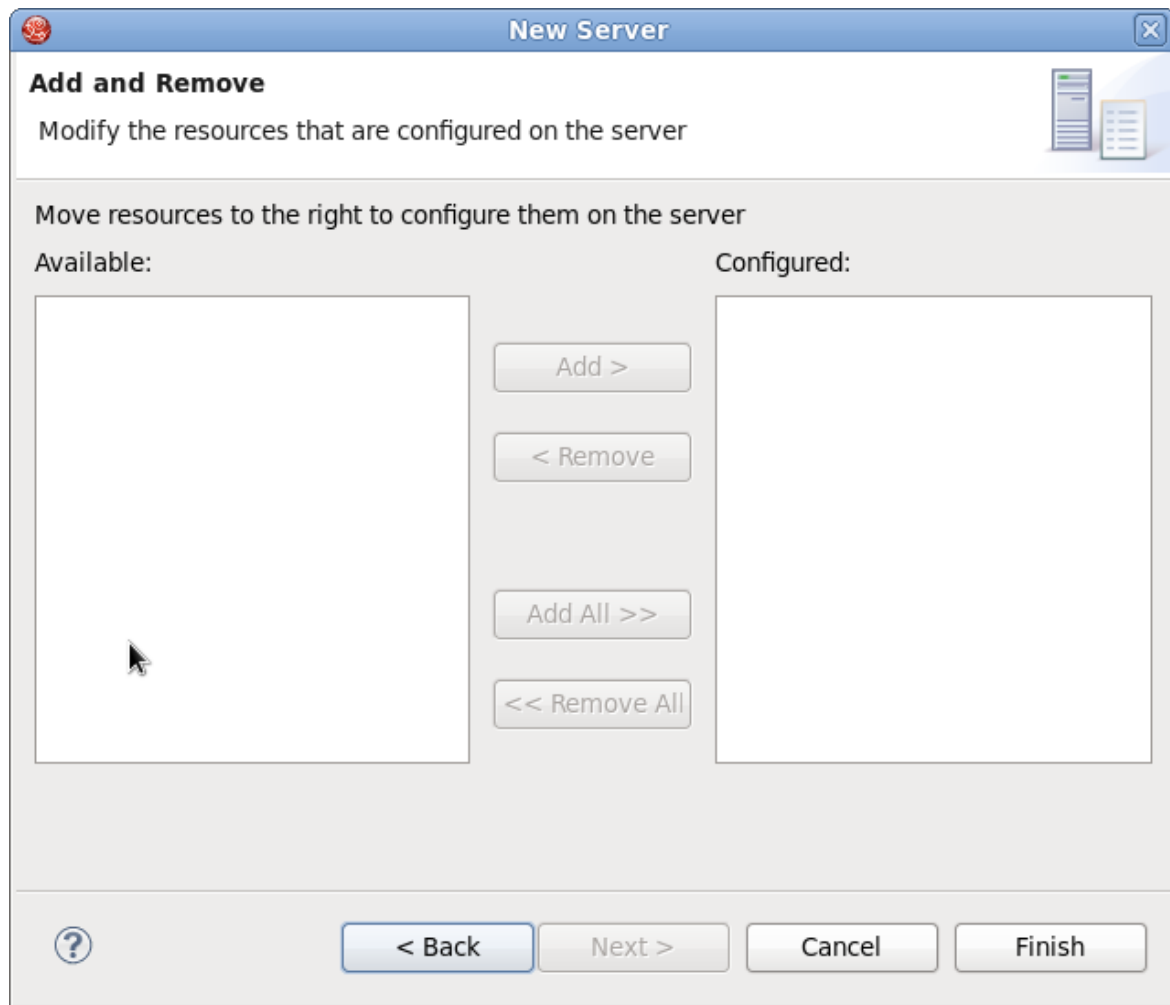


Figure 1.6. Modify resources for the new JBoss server

Result

The JBoss Enterprise Application Server 6.0 Runtime Server is listed in the **Servers** tab.

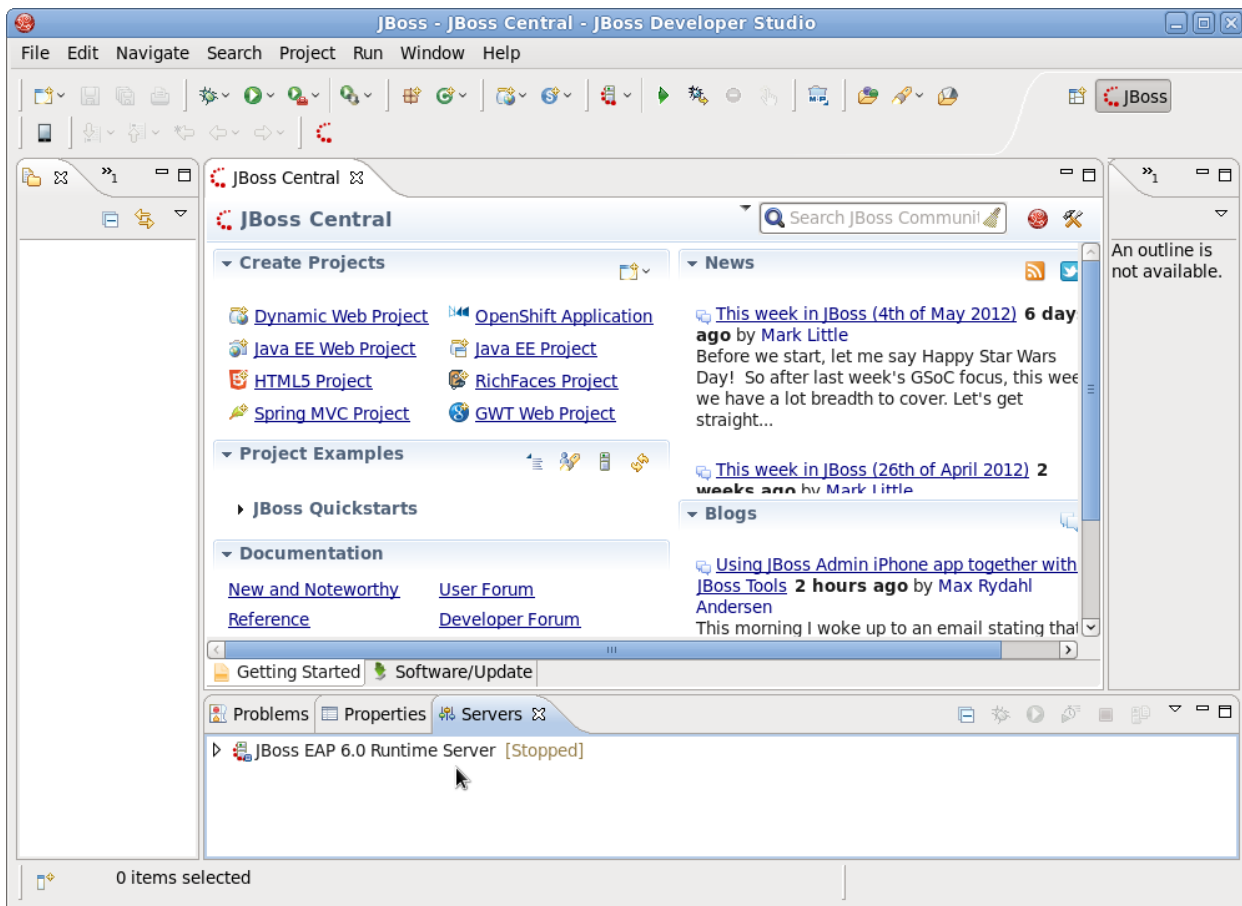


Figure 1.7. Server appears in the server list

[Report a bug](#)

1.5. Run Your First Application

1.5.1. Replace the Default Welcome Web Application

JBoss Enterprise Application Platform 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 1.5. Task

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](#)

1.5.2. Download the Quickstart Code Examples

1.5.2.1. Access the Java EE Quickstart Examples

Summary

JBoss Enterprise Application Platform 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”](#)
- [Section 2.2.3, “Install the JBoss Enterprise Application Platform 6 Maven Repository Locally”](#)
- [Section 2.3.2, “Configure the JBoss Enterprise Application Platform Maven Repository Using the Maven Settings”](#)

Procedure 1.6. Download the Quickstarts

1. Open a web browser and access this URL:
<https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?product=appplatform>.
2. Find "Application Platform 6 Quickstarts" in the list.
3. Click the **Download** button to download a **.zip** file containing the examples.
4. Unzip the archive in a directory of your choosing.

Result

The Java EE quickstart examples have been downloaded and unzipped. Refer to the **README.md** file in the **jboss-eap-6.0-quickstarts/** directory for instructions about deploying each quickstart.

[Report a bug](#)

1.5.3. Run the Quickstarts

1.5.3.1. Run the Quickstarts in JBoss Developer Studio

Procedure 1.7. Import the quickstarts into JBoss Developer Studio

The quickstarts ship with a parent POM (Project Object Model) file that contains project and configuration information for all of the quickstarts in the distribution. Using this top-level POM file, you can easily import the entire list of quickstarts into JBoss Developer Studio at one time.

1. If you have not done so, [Section 2.3.2, “Configure the JBoss Enterprise Application Platform Maven Repository Using the Maven Settings”](#).
2. Start JBoss Developer Studio.

3. From the menu, select **File** → **Import**.
4. In the selection list, choose **Maven** → **Existing Maven Projects**, then click **Next**.

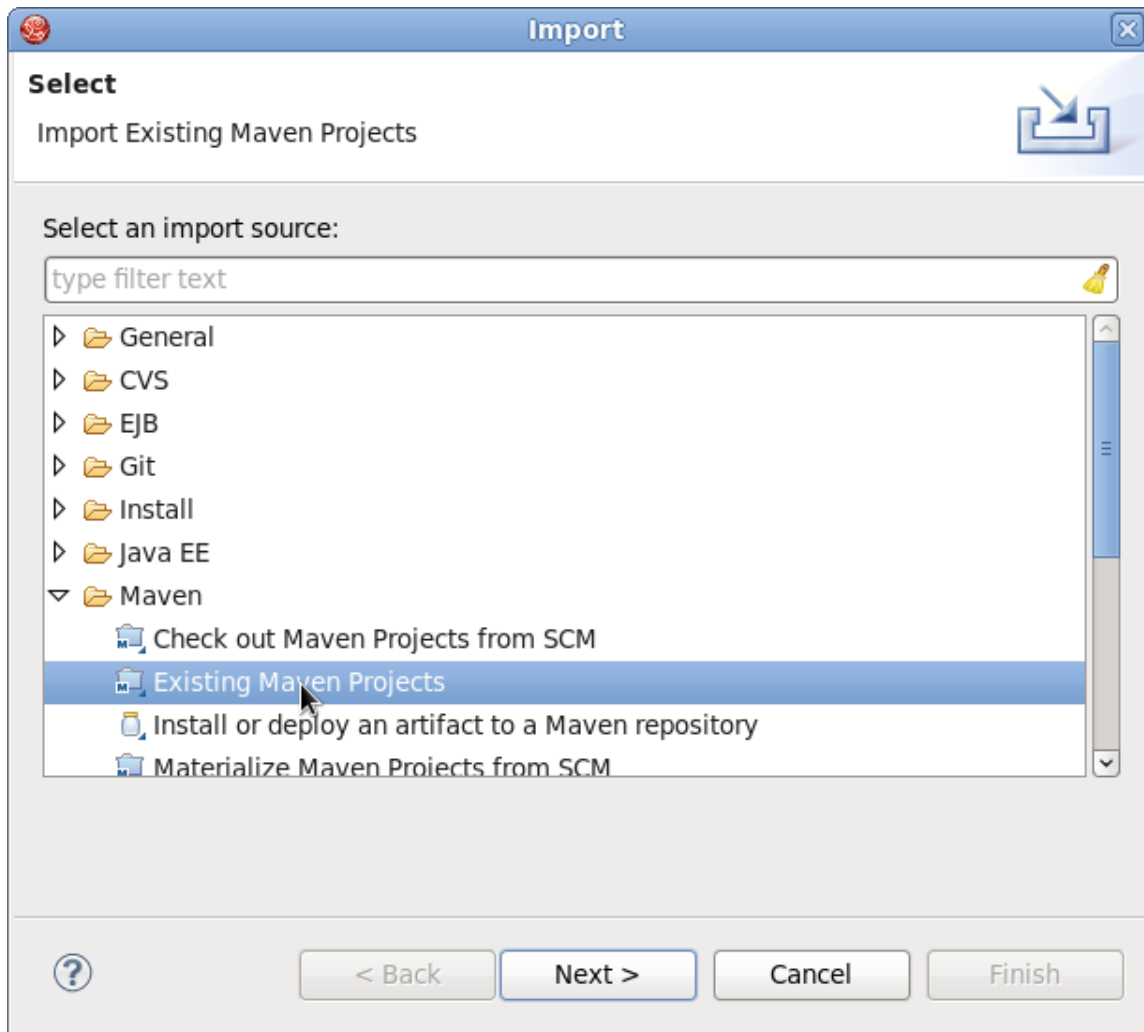


Figure 1.8. Import Existing Maven Projects

5. Browse to the root of the quickstart directory and click **OK**. The **Projects** list box will be populated with the list of the pom.xml files from all of the quickstart projects.

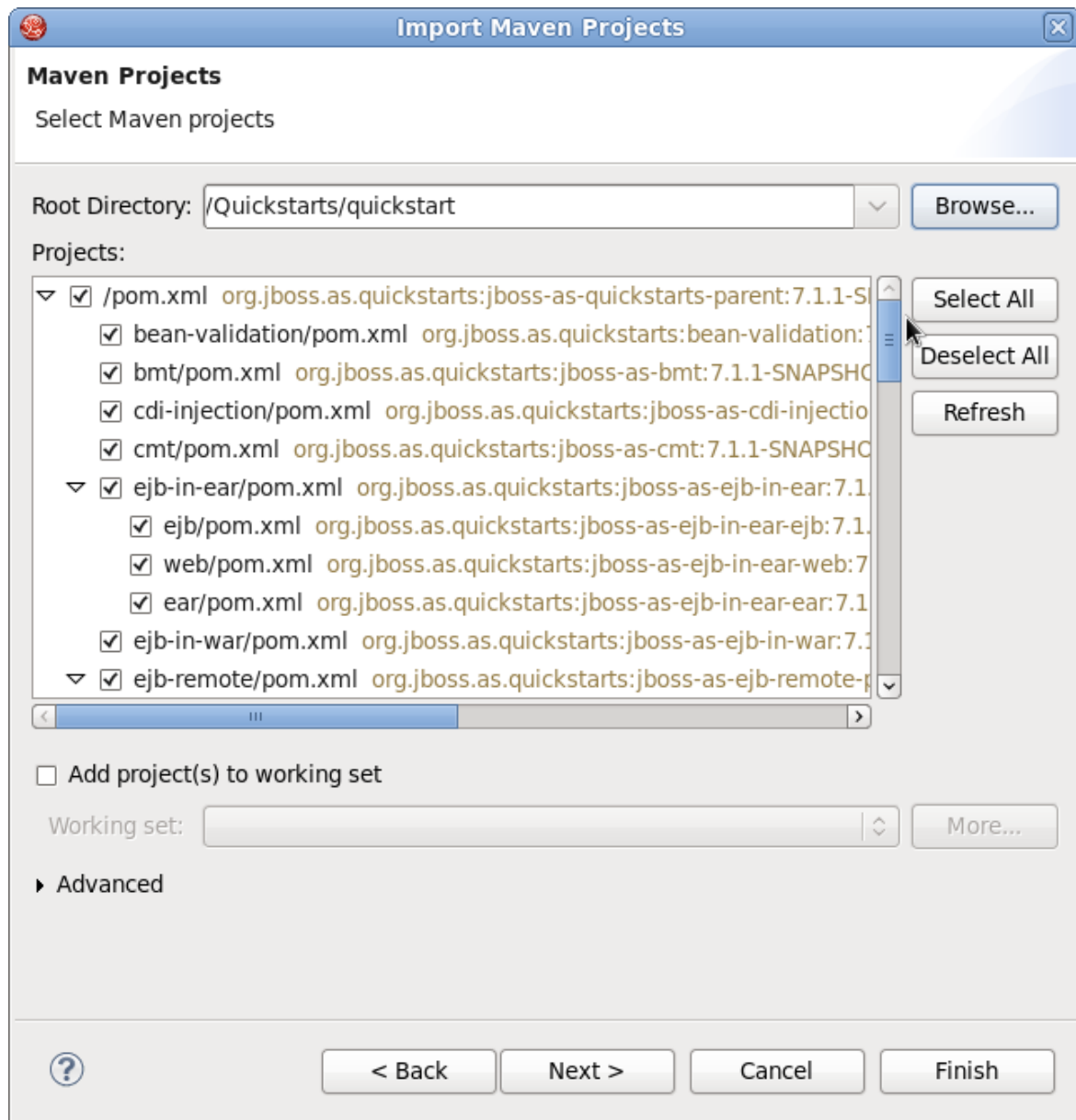


Figure 1.9. Select Maven Projects

6. Click **Next**, then click **Finish**.

Procedure 1.8. 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. Open the **Servers** tab. To add it to the panel:
 - a. Click **Window** → **Show View** → **Other...**
 - b. Select **Servers** from the **Server** folder and click **Ok**.
2. Right click on **helloworld** in the **Project Explorer** tab, and select **Run As** → **Run on Server**.
3. Select the **JBoss EAP 6.0 Runtime Server** server and click **Next**. This should deploy the **helloworld** quickstart to the JBoss server.
4. To verify that the **helloworld** quickstart was deployed successfully to the JBoss server, open a web browser and access the application at this URL: <http://localhost:8080/jboss-as-helloworld>

1.5.3.2. Run the Quickstarts Using a Command Line

Procedure 1.9. 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. **Review the README 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.

2. **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** 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

3. **Run the other quickstarts.**

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

[Report a bug](#)

1.5.4. Review the Quickstart Tutorials

1.5.4.1. Explore the helloworld Quickstart

Summary

The **helloworld** quickstart shows you how to deploy a simple Servlet to JBoss Enterprise Application Platform 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 make 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 file at the root of the **helloworld** quickstart directory. Here we show you how to use JBoss Developer Studio to run the quickstart.

Procedure 1.10. Import the helloworld quickstart into JBoss Developer Studio

If you previously imported all of the quickstarts into JBoss Developer Studio following the steps here [Section 1.5.3.1, "Run the Quickstarts in JBoss Developer Studio"](#), you can skip to the next section.

1. If you have not done so, [Section 2.3.2, "Configure the JBoss Enterprise Application Platform Maven Repository Using the Maven Settings"](#).
2. If you have not done so, [Section 1.4.1.3, "Install JBoss Developer Studio 5"](#).
3. [Section 1.4.1.4, "Start JBoss Developer Studio"](#).
4. From the menu, select **File** → **Import**.

5. In the selection list, choose **Maven** → **Existing Maven Projects**, then click **Next**.

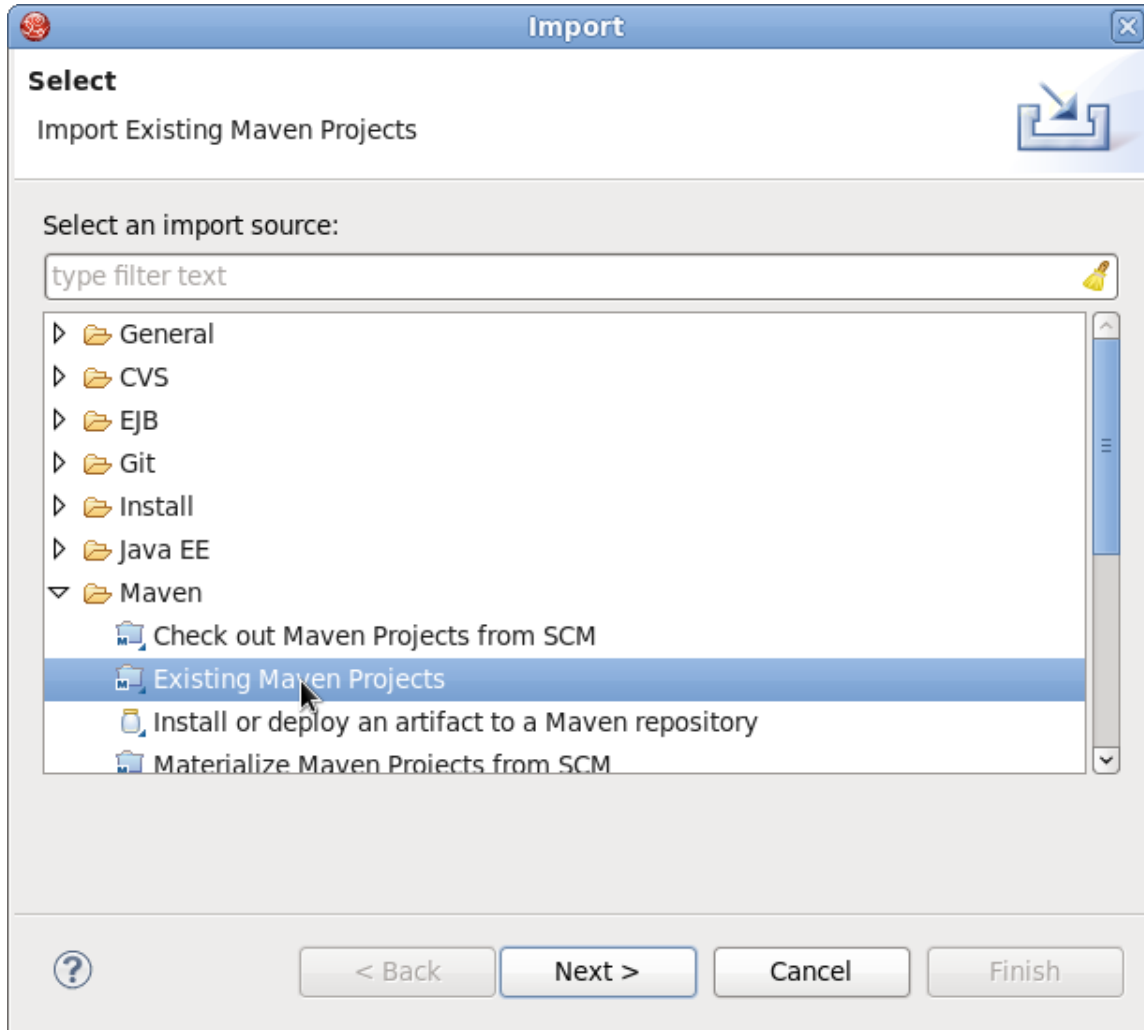


Figure 1.10. Import Existing Maven Projects

6. Browse to the `QUICKSTART_HOME/quickstart/helloworld/` directory and click **OK**. The **Projects** list box is populated with the `pom.xml` file from the `helloworld` quickstart project.

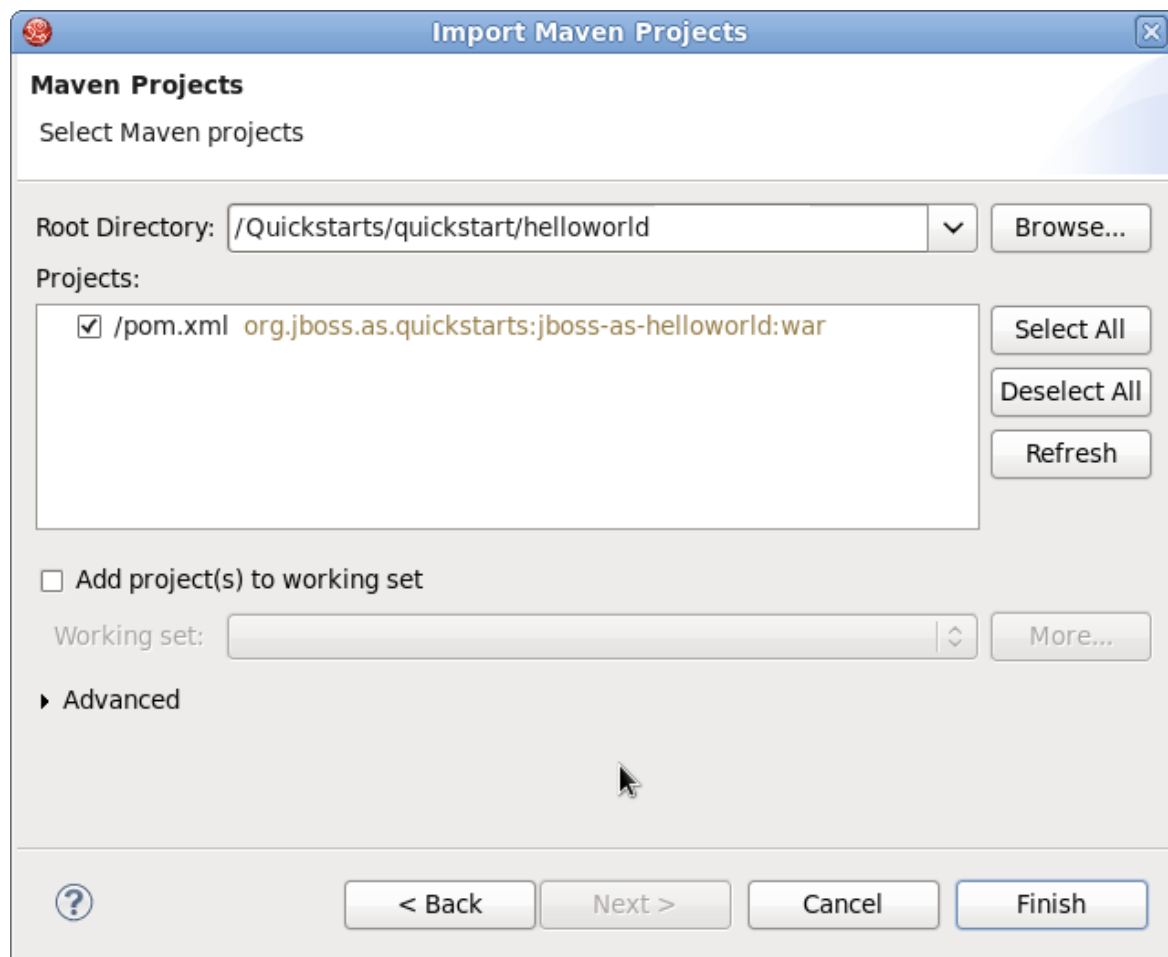


Figure 1.11. Select Maven Projects

7. Click **Finish**.

Procedure 1.11. Build and Deploy the helloworld quickstart

1. If you have not yet configured JBoss Developer Studio for JBoss Enterprise Application Platform 6, you must [Section 1.4.1.5, "Add the JBoss Enterprise Application Platform 6 Server to JBoss Developer Studio"](#).
2. Right click on **jboss-as-helloworld** in the **Project Explorer** tab, and select **Run As** → **Run on Server**.

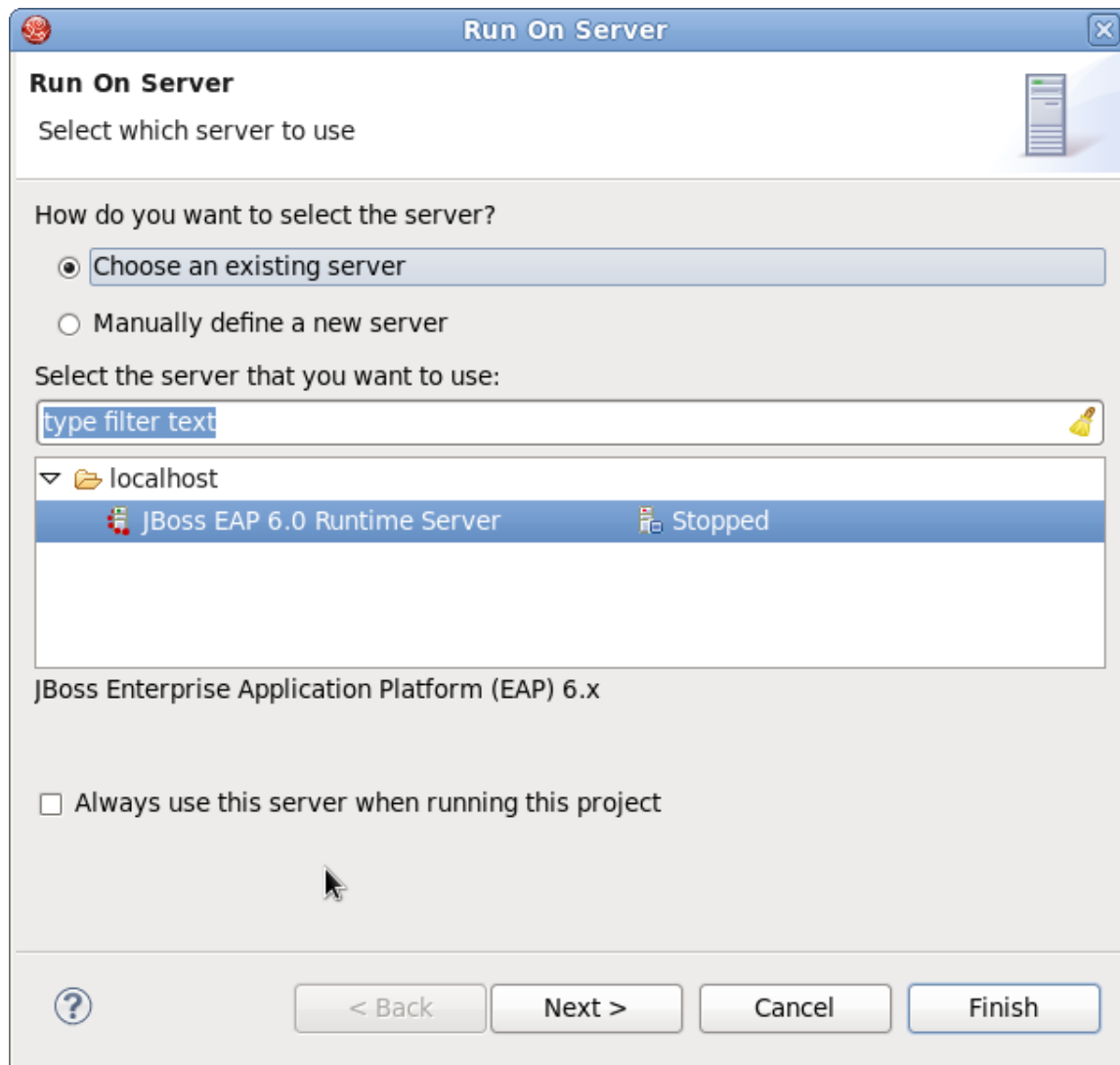


Figure 1.12. Run on Server

3. Select the **JBoss EAP 6.0 Runtime Server** server and click **Next**. This deploys the **helloworld** quickstart to the JBoss server.
4. To verify that the **helloworld** quickstart was deployed successfully to the JBoss server, open a web browser and access the application at this URL: <http://localhost:8080/jboss-as-helloworld>

Procedure 1.12. Examine the Directory Structure

The code for the **helloworld** quickstart can be found in the **QUICKSTART_HOME/helloworld** directory. The **helloworld** quickstart is comprised a Servlet and a CDI bean. It also includes an empty **beans.xml** file which tells JBoss Enterprise Application Platform 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-as-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 1.13. 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.

```

27. @WebServlet("/HelloWorld")
28. public class HelloWorldServlet extends HttpServlet {
29.
30.     static String PAGE_HEADER = "<html><head /><body>";
31.
32.     static String PAGE_FOOTER = "</body></html>";
33.
34.     @Inject
35.     HelloService helloService;
36.
37.     @Override
38.     protected void doGet(HttpServletRequest req, HttpServletResponse resp)
39.         throws ServletException, IOException {
40.         PrintWriter writer = resp.getWriter();
41.         writer.println(PAGE_HEADER);
42.         writer.println("<h1>" + helloService.createHelloMessage("World") +
43.             "</h1>");
44.         writer.println(PAGE_FOOTER);
45.         writer.close();
46.     }
47. }

```

Table 1.4. HelloWorldServlet Details

Line	Note
27	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.
30-32	Every web page needs correctly formed HTML. This quickstart uses static Strings to write the minimum header and footer output.
34-35	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.
41	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.

```

9. public class HelloService {
10.
11.     String createHelloMessage(String name) {
12.         return "Hello " + name + "!";
13.     }
14. }

```

Summary

This quickstart shows you how to create and deploy a simple application to JBoss Enterprise Application Platform 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 file at the root of the **numberguess** quickstart directory. Here we show you how to use JBoss Developer Studio to run the quickstart.

Procedure 1.14. Import the numberguess quickstart into JBoss Developer Studio

If you previously imported all of the quickstarts into JBoss Developer Studio following the steps in the following procedure, [Section 1.5.3.1, “Run the Quickstarts in JBoss Developer Studio”](#), you can skip to the next section.

1. If you have not done so, perform the following procedures: [Section 1.4.1.3, “Install JBoss Developer Studio 5”](#)
2. [Section 1.4.1.4, “Start JBoss Developer Studio”](#)
3. From the menu, select **File** → **Import**.
4. In the selection list, choose **Maven** → **Existing Maven Projects**, then click **Next**.

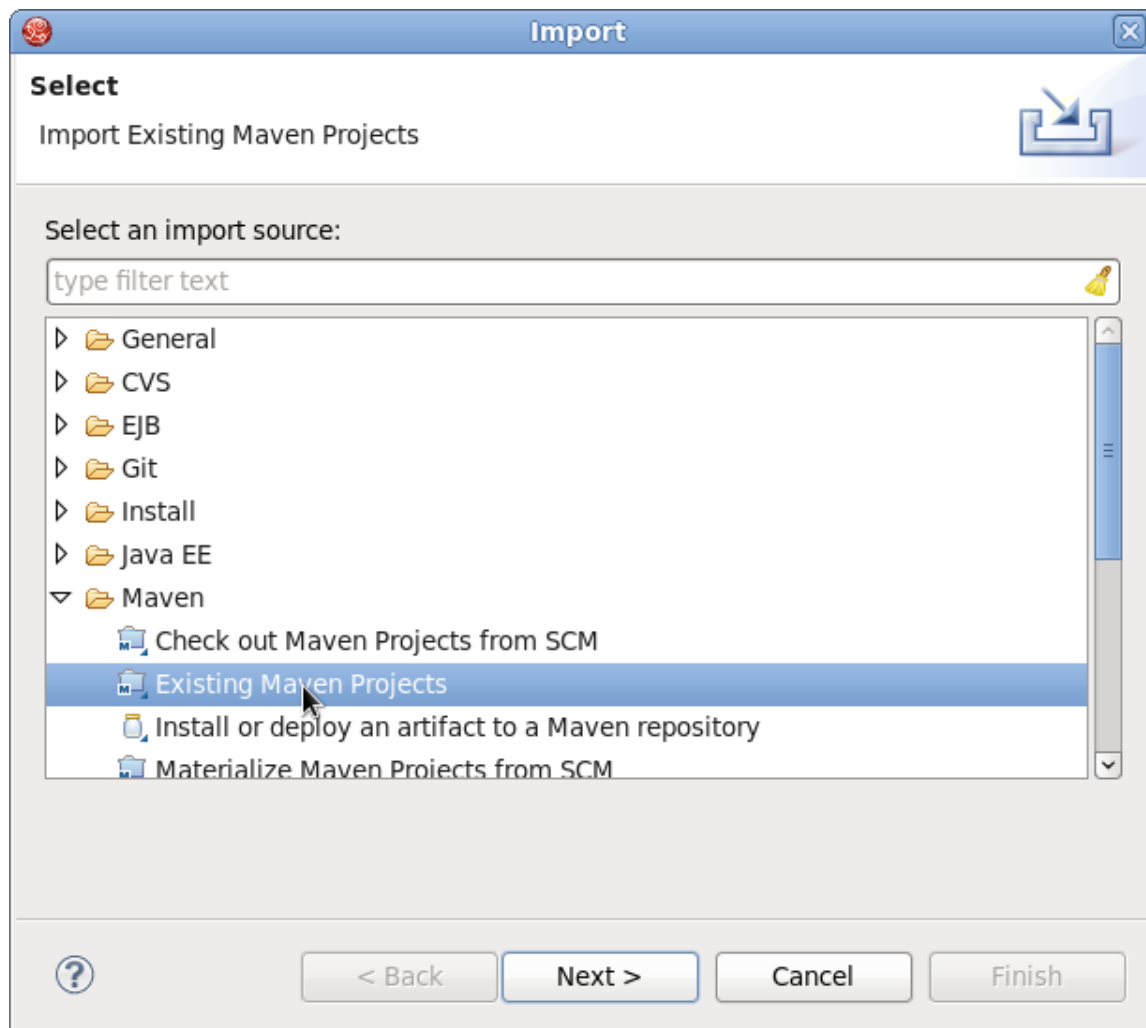


Figure 1.13. Import Existing Maven Projects

5. Browse to the `QUICKSTART_HOME/quickstart/numberguess/` directory and click **OK**. The **Projects** list box is populated with the `pom.xml` file from the `numberguess` quickstart project.
6. Click **Finish**.

Procedure 1.15. Build and Deploy the numberguess quickstart

1. If you have not yet configured JBoss Developer Studio for JBoss Enterprise Application Platform 6, you must do the following: [Section 1.4.1.5, “Add the JBoss Enterprise Application Platform 6 Server to JBoss Developer Studio”](#).
2. Right click on `jboss-as-numberguess` in the **Project Explorer** tab, and select **Run As** → **Run on Server**.
3. Select the **JBoss EAP 6.0 Runtime Server** server and click **Next**. This deploys the `numberguess` quickstart to the JBoss server.
4. To verify that the `numberguess` quickstart was deployed successfully to the JBoss server, open a web browser and access the application at this URL: <http://localhost:8080/jboss-as-numberguess>

Procedure 1.16. 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 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 Enterprise Application Platform 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:

```
03. <faces-config version="2.0"
04.     xmlns="http://java.sun.com/xml/ns/javaee"
05.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
06.     xsi:schemaLocation="
07.         http://java.sun.com/xml/ns/javaee>
08.         http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">
09.
10. </faces-config>
```

2. Examine the beans.xml file

There's also an empty `beans.xml` file, which tells JBoss Enterprise Application Platform 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 1.17. 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.

```

03. <html xmlns="http://www.w3.org/1999/xhtml"
04.     xmlns:ui="http://java.sun.com/jsf/facelets"
05.     xmlns:h="http://java.sun.com/jsf/html"
06.     xmlns:f="http://java.sun.com/jsf/core">
07.
08. <head>
09. <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
10. <title>Numberguess</title>
11. </head>
12.
13. <body>
14.     <div id="content">
15.         <h1>Guess a number...</h1>
16.         <h:form id="numberGuess">
17.
18.             <!-- Feedback for the user on their guess -->
19.             <div style="color: red">
20.                 <h:messages id="messages" globalOnly="false" />
21.                 <h:outputText id="Higher" value="Higher!"
22.                     rendered="#{game.number gt game.guess and game.guess ne 0}"
23.                 />
24.                 <h:outputText id="Lower" value="Lower!"
25.                     rendered="#{game.number lt game.guess and game.guess ne 0}"
26.                 />
27.             </div>
28.
29.             <!-- Instructions for the user -->
30.             <div>
31.                 I'm thinking of a number between <span
32.                     id="numberGuess:smallest">#{game.smallest}</span> and <span
33.                     id="numberGuess:biggest">#{game.biggest}</span>. You have
34.                     #{game.remainingGuesses} guesses remaining.
35.                 </div>
36.
37.             <!-- Input box for the users guess, plus a button to submit, and
38.                 reset -->
39.             <!-- These are bound using EL to our CDI beans -->
40.             <div>
41.                 Your guess:
42.                 <h:inputText id="inputGuess" value="#{game.guess}"
43.                     required="true" size="3"
44.                     disabled="#{game.number eq game.guess}"
45.                     validator="#{game.validateNumberRange}" />
46.                 <h:commandButton id="guessButton" value="Guess"
47.                     action="#{game.check}"
48.                     disabled="#{game.number eq game.guess}" />
49.             </div>
50.             <div>
51.                 <h:commandButton id="restartButton" value="Reset"
52.                     action="#{game.reset}" immediate="true" />
53.             </div>
54.         </h:form>
55.     </div>
56.
57.     <br style="clear: both" />
58. </body>
59. </html>

```

Table 1.5. JSF Details

Line	Note
20-24	These are the messages which can be sent to the user: "Higher!" and "Lower!"
29-32	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.
38-42	This input field is bound to a bean property using a value expression.
42	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.
43-45	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 1.18. 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 8.2.3.3, "Use a Qualifier to Resolve an Ambiguous Injection"](#)

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

```

21. @Target({ TYPE, METHOD, PARAMETER, FIELD })
22. @Retention(RUNTIME)
23. @Documented
24. @Qualifier
25. public @interface Random {
26.
27. }
```

2. Review the MaxNumber.java qualifier code

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

```

21. @Target({ TYPE, METHOD, PARAMETER, FIELD })
22. @Retention(RUNTIME)
23. @Documented
24. @Qualifier
25. public @interface MaxNumber {
26.
27. }
```

3. Review the Generator 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.

```

28. @ApplicationScoped
29. public class Generator implements Serializable {
30.     private static final long serialVersionUID = -7213673465118041882L;
31.
32.     private java.util.Random random = new
java.util.Random(System.currentTimeMillis());
33.
34.     private int maxNumber = 100;
35.
36.     java.util.Random getRandom() {
37.         return random;
38.     }
39.
40.     @Produces
41.     @Random
42.     int next() {
43.         // a number between 1 and 100
44.         return getRandom().nextInt(maxNumber - 1) + 1;
45.     }
46.
47.     @Produces
48.     @MaxNumber
49.     int getMaxNumber() {
50.         return maxNumber;
51.     }
52. }

```

4. Review the Game 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}**.


```
035. @Named
036. @SessionScoped
037. public class Game implements Serializable {
038.
039.     private static final long serialVersionUID = 991300443278089016L;
040.
041.     /**
042.      * The number that the user needs to guess
043.      */
044.     private int number;
045.
046.     /**
047.      * The users latest guess
048.      */
049.     private int guess;
050.
051.     /**
052.      * The smallest number guessed so far (so we can track the valid guess
053.      * range).
054.      */
055.     private int smallest;
056.
057.     /**
058.      * The largest number guessed so far
059.      */
060.     private int biggest;
061.
062.     /**
063.      * The number of guesses remaining
064.      */
065.     private int remainingGuesses;
066.
067.     /**
068.      * The maximum number we should ask them to guess
069.      */
070.     @Inject
071.     @MaxNumber
072.     private int maxNumber;
073.
074.     /**
075.      * The random number to guess
076.      */
077.     @Inject
078.     @Random
079.     Instance<Integer> randomNumber;
080.
081.     public Game() {
082.
083.     public int getNumber() {
084.         return number;
085.     }
086.
087.     public int getGuess() {
088.         return guess;
089.     }
090.
091.     public void setGuess(int guess) {
092.         this.guess = guess;
093.     }
094.
095.     public int getSmallest() {
096.         return smallest;
097.     }
098.
099.     public int getBiggest() {
100.         return biggest;
```

```

101.     }
102.
103.     public int getRemainingGuesses() {
104.         return remainingGuesses;
105.     }
106.
107.     /**
108.      * Check whether the current guess is correct, and update the
109.      * biggest/smallest guesses as needed.
110.      * Give feedback to the user if they are correct.
111.      */
112.     public void check() {
113.         if (guess > number) {
114.             biggest = guess - 1;
115.         } else if (guess < number) {
116.             smallest = guess + 1;
117.         } else if (guess == number) {
118.             FacesContext.getCurrentInstance().addMessage(null, new
119.             FacesMessage("Correct!"));
120.         }
121.         remainingGuesses--;
122.     }
123.
124.     /**
125.      * Reset the game, by putting all values back to their defaults, and
126.      * getting a new random number.
127.      * We also call this method when the user starts playing for the first
128.      * time using
129.      * {@linkplain PostConstruct @PostConstruct} to set the initial
130.      * values.
131.      */
132.     @PostConstruct
133.     public void reset() {
134.         this.smallest = 0;
135.         this.guess = 0;
136.         this.remainingGuesses = 10;
137.         this.biggest = maxNumber;
138.         this.number = randomNumber.get();
139.     }
140.
141.     /**
142.      * A JSF validation method which checks whether the guess is valid. It
143.      * might not be valid because
144.      * there are no guesses left, or because the guess is not in range.
145.      */
146.     public void validateNumberRange(FacesContext context, UIComponent
147.     toValidate, Object value) {
148.         if (remainingGuesses <= 0) {
149.             FacesMessage message = new FacesMessage("No guesses left!");
150.             context.addMessage(toValidate.getClientId(context), message);
151.             ((UIInput) toValidate).setValid(false);
152.             return;
153.         }
154.         int input = (Integer) value;
155.         if (input < smallest || input > biggest) {
156.             ((UIInput) toValidate).setValid(false);
157.
158.             FacesMessage message = new FacesMessage("Invalid guess");
159.             context.addMessage(toValidate.getClientId(context), message);
160.         }
161.     }
162. }

```


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 Enterprise Application Platform 6 includes a Maven repository that contains many of the requirements that Java EE developers typically use to build applications on JBoss Enterprise Application Platform 6. To configure your project to use this repository, see [Section 2.3.1, “Configure the JBoss Enterprise Application Platform Maven Repository”](#).

A repository can be local or remote. Remote repositories are accessed using common protocols such as **http://** for a repository on an HTTP server or **file://** for a repository a file server. A local repository is a cached download of the artifacts from a remote repository.

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 should 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 install

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 install

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:

```

<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-6.0-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-6.0-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

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 Enterprise Application Platform 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 Enterprise Application Platform 6 Maven Repository Locally”](#)

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

[Report a bug](#)

2.2.3. Install the JBoss Enterprise Application Platform 6 Maven Repository Locally

Summary

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 covers the steps to download the JBoss Enterprise Application Platform 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 the using Maven for development but is not recommended for team production environments.

Procedure 2.1. Task

1. **Download the JBoss Enterprise Application Platform 6 Maven Repository ZIP archive**

Open a web browser and access this URL:

<https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?product=appplatform>.

2. Find "Application Platform 6 Maven Repository" in the list.
3. Click the **Download** button to download a **.zip** file containing the repository.
4. Unzip the file in the same directory on the local file system into a directory of your choosing.
5. [Section 2.3.2, “Configure the JBoss Enterprise Application Platform Maven Repository Using the Maven Settings”](#).

Result

This creates a Maven repository directory called **jboss-eap-6.0.0.maven-repository**.



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 Enterprise Application Platform 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 Enterprise Application Platform 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 Enterprise Application Platform 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 "Application Platform 6 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 Enterprise Application Platform Maven Repository Using the Maven Settings"](#).

Result

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

[Report a bug](#)

2.2.5. Install the JBoss Enterprise Application Platform 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 licence 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 Enterprise Application Platform 6 Maven Repository using Sonatype Nexus Maven Repository Manager. For more complete instructions, see [Sonatype Nexus: Manage Artifacts](#).

Procedure 2.3. Download the JBoss Enterprise Application Platform 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 "Application Platform 6 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.

Procedure 2.4. Add the JBoss Enterprise Application Platform 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 Enterprise Application Platform 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. Configure the Maven Repository

2.3.1. Configure the JBoss Enterprise Application Platform Maven Repository

Overview

There are two approaches to direct Maven to use the JBoss Enterprise Application Platform 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 Enterprise Application Platform 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 [Using Mirrors for Repositories](#).

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 Enterprise Application Platform 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. See [Why Putting Repositories in your POMs is a Bad Idea](#) for a complete explanation of the possible consequences of this approach.

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

[Section 2.3.3, “Configure the JBoss Enterprise Application Platform Maven Repository Using the Project POM”](#).

[Report a bug](#)

2.3.2. Configure the JBoss Enterprise Application Platform Maven Repository Using the Maven Settings

There are two approaches to direct Maven to use the JBoss Enterprise Application Platform 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 direct Maven to use the JBoss Enterprise Application Platform Maven Repository across all projects using the Maven global or user settings. This is the recommended approach.



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, refer to the Maven chapter of the Development Guide for JBoss Enterprise Application Platform 6. The following are examples for each of the installation options:

File System

file:///path/to/repo/jboss-eap-6.0.0-maven-repository

Apache Web Server

http://intranet.acme.com/jboss-eap-6.0.0-maven-repository/

Nexus Repository Manager

https://intranet.acme.com/nexus/content/repositories/jboss-eap-6.0.0-maven-repository

You can configure Maven to use the JBoss Enterprise Application Platform Repository using either the Maven install or the user install settings. For more information about the location of the settings and how they behave, refer to the Maven chapter of the Development Guide for JBoss Enterprise Application Platform 6. .

To use the JBoss Enterprise Application Platform repository on a local user system, follow these instructions:

Procedure 2.6. Configure the Settings

1. Open the **settings.xml** for the type of configuration you have chosen.

A. Global Settings

If you are configuring the **global** settings, open the **M2_HOME/conf/settings.xml** file.

B. User Settings

If you are configuring user specific settings and you do not yet have a **USER_HOME/.m2/settings.xml** file, copy the **settings.xml** file from the **M2_HOME/conf/** directory into the **USER_HOME/.m2/** directory.

2. 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-6.0.0-maven-repository</url>
      <layout>default</layout>
      <releases>
        <enabled>true</enabled>
        <updatePolicy>never</updatePolicy>
      </releases>
      <snapshots>
        <enabled>false</enabled>
        <updatePolicy>never</updatePolicy>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>jboss-eap-repository-group</id>
      <name>JBoss EAP Maven Repository</name>
      <url>
file:///path/to/repo/jboss-eap-6.0.0-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>
```

3. If you modify the **settings.xml** file while JBoss Developer Studio is running, you must refresh the user settings. From the menu, choose **Window** → **Preferences**. In the **Preferences** Window, expand **Maven** and choose **User Settings**. Click the **Update Settings** button to

refresh the Maven user settings in JBoss Developer Studio.

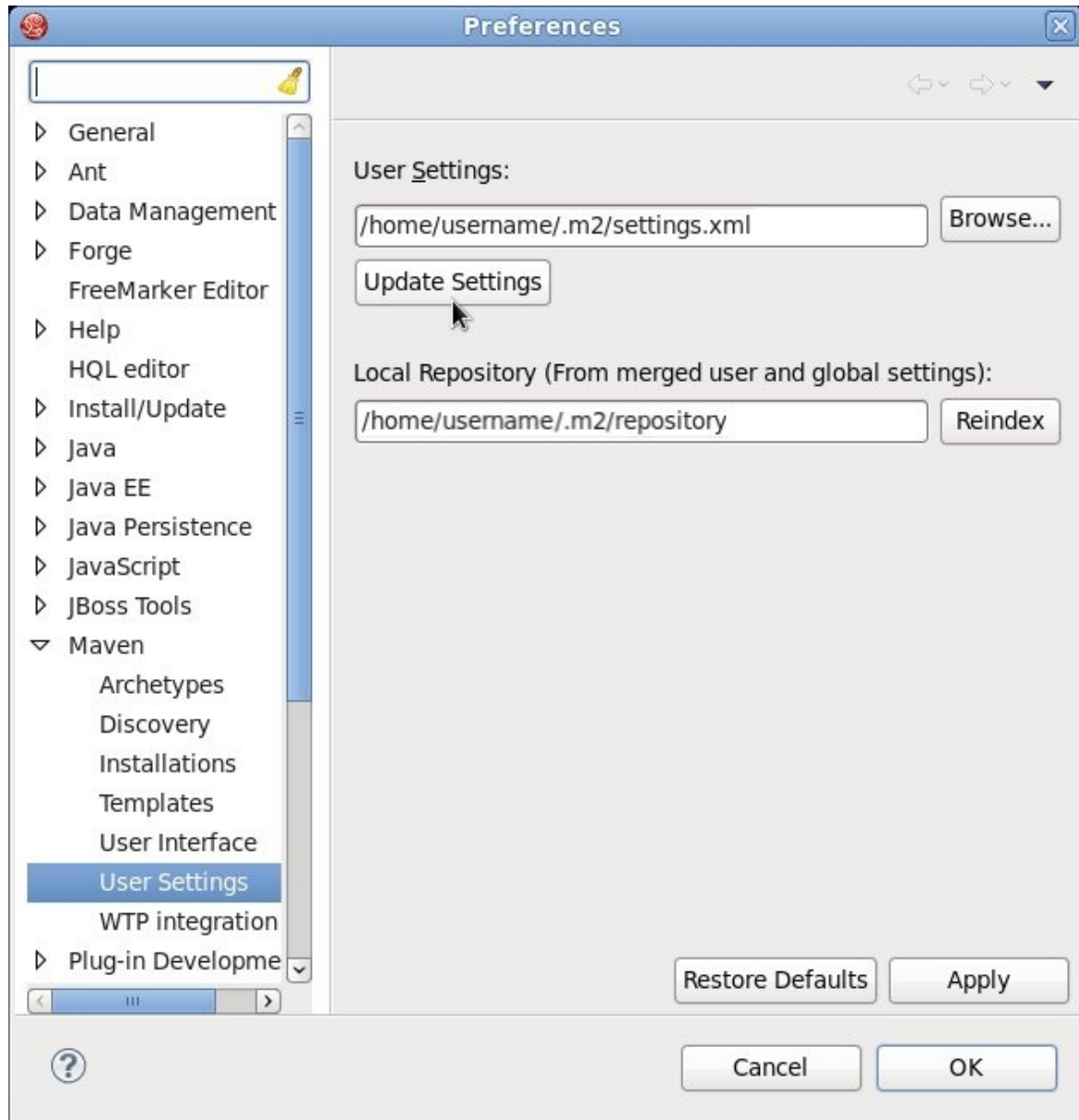


Figure 2.1. Update Maven User Settings



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.

Result

The JBoss Enterprise Application Platform repository has now been configured.

[Report a bug](#)

2.3.3. Configure the JBoss Enterprise Application Platform Maven Repository Using the Project POM

There are two approaches to direct Maven to use the JBoss Enterprise Application Platform 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 Enterprise Application Platform 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. See [Why Putting Repositories in your POMs is a Bad Idea](#) for a complete explanation of the possible consequences of this approach.



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 Enterprise Application Platform 6 Maven Repository”](#). The following are examples for each of the installation options:

File System

file:///path/to/repo/jboss-eap-6.0.0-maven-repository

Apache Web Server

http://intranet.acme.com/jboss-eap-6.0.0-maven-repository/

Nexus Repository Manager

https://intranet.acme.com/nexus/content/repositories/jboss-eap-6.0.0-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-6.0.0-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-6.0.0-maven-repository/</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </pluginRepository>
</pluginRepositories>
```

[Report a bug](#)

Chapter 3. Class Loading and Modules

3.1. Introduction

3.1.1. Overview of Class Loading and Modules

JBoss Enterprise Application Platform 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 favour 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.

The following material covers what developers need to know to successfully build and deploy applications on JBoss Enterprise Application Platform 6.

[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 Enterprise Application Platform 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. All modules provide the same features.

Static Modules

Static Modules are predefined in the **EAP_HOME/modules/** directory of the application server. Each sub-directory represents one module and contains one or more JAR files and a configuration file (**module.xml**). 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.

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.

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 Enterprise Application Platform 6.

There are two types of dependencies: explicit and implicit.

Explicit dependencies are declared in configuration by the developer. Static modules can declare dependencies in the `modules.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 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 Enterprise Application Platform 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 3.1. 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 classloading all deployments are treated as modules by JBoss Enterprise Application Platform. 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 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 Enterprise Application Platform 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 Enterprise Application Platform 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 Enterprise Application Platform 6.

Refer to [Section 3.7.1, “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 Enterprise Application Platform 6 and named according to the following conventions.

1. 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.

2. 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.accounting.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 Enterprise Application Platform 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 Enterprise Application Platform 6. Refer to [Section 3.7.1, “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. Refer to [Section 3.7.2, “Included Modules”](#) for the list of static modules included with JBoss Enterprise Application Platform. If the module is another deployment then refer to [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. Refer to [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
```

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 3.2. 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 Enterprise Application Platform 6 will add the classes from the specified modules to the class path of the application when it is deployed.

[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 3.7.2, "Included Modules"](#) for the list of static modules included with JBoss Enterprise Application Platform 6. If the module is another deployment, then refer to [Section 3.1.7, "Dynamic Module Naming"](#) to determine the module name.

Procedure 3.3. 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>
```

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 3.3. 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 3.7.1, “Implicit Module Dependencies”](#) for a list of implicit dependencies and their conditions.

Procedure 3.4. 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 3.4. 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 3.5. 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 3.5. Example jboss-deployment-structure.xml file.

```

<jboss-deployment-structure>
  <ear-subdeployments-isolated>true</ear-subdeployments-isolated>
  <deployment>
    <exclude-subsystems>
      <subsystem name="resteasy" />
    </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. Class Loading and Subdeployments

3.6.1. 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.

- Each WAR and EJB JAR subdeployment is a module.
- The contents of the **lib/** directory in the root of the EAR archive is a module. This is called the

parent module.

These modules have the same behaviour as any other module with the following additional implicit dependencies:

- WAR subdeployments have implicit dependencies on the parent module and any EJB JAR subdeployments.
- EJB JAR subdeployments have implicit dependencies on the parent module and any other EJB JAR subdeployments.



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.

The implicit dependencies described above occur because JBoss Enterprise Application Platform 6 has subdeployment class loader isolation disabled by default.

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](#)

3.6.2. Subdeployment Class Loader Isolation

Each subdeployment in an Enterprise Archive (EAR) is a dynamic module with its own class loader and cannot access the resources of other subdeployments. This is called subdeployment class loader isolation.

JBoss Enterprise Application Platform 6 has strict subdeployment class loader isolation disabled by default. It can be enabled if required.

[Report a bug](#)

3.6.3. Disable Subdeployment Class Loader Isolation Within a EAR

This task shows you how to disable 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 **false**.

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

Result:

Subdeployment class loader isolation will now be disabled for this EAR deployment. This means that the subdeployments of the EAR will have automatic dependencies on each of the non-WAR subdeployments.

[Report a bug](#)

3.7. Reference

3.7.1. 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	Modules Always added	Modules Conditional added	Conditions
Core Server	<ul style="list-style-type: none"> ▸ <code>javax.api</code> ▸ <code>sun.jdk</code> 	-	-
EE Subsystem	<ul style="list-style-type: none"> ▸ <code>javaee.api</code> 	-	-
EJB3 subsystem	-	<ul style="list-style-type: none"> ▸ <code>javaee.api</code> 	The presence of <code>ejb-jar.xml</code> in valid locations in the deployment, as specified by the Java EE 6 specification or the presence of annotation-based EJBs (e.g. <code>@Stateless</code> , <code>@Stateful</code> , <code>@MessageDriven</code> etc)
JAX-RS (Resteasy) subsystem	<ul style="list-style-type: none"> ▸ <code>javax.xml.bind.api</code> 	<ul style="list-style-type: none"> ▸ <code>org.jboss.resteasy.resteasy-atom-provider</code> ▸ <code>org.jboss.resteasy.resteasy-cdi</code> ▸ <code>org.jboss.resteasy.resteasy-jaxrs</code> ▸ <code>org.jboss.resteasy.resteasy-jaxb-provider</code> ▸ <code>org.jboss.resteasy.resteasy-jackson-provider</code> ▸ <code>org.jboss.resteasy.resteasy-jsapi</code> ▸ <code>org.jboss.resteasy.resteasy-multipart-provider</code> ▸ <code>org.jboss.resteasy.async-http-servlet-30</code> 	The presence of JAX-RS annotations in the deployment
JCA sub-system	<ul style="list-style-type: none"> ▸ <code>javax.resource.api</code> 	<ul style="list-style-type: none"> ▸ <code>javax.jms.api</code> ▸ <code>javax.validation.api</code> 	If the deployment is a resource adaptor (RAR) deployment.

		<ul style="list-style-type: none"> » <code>org.jboss.logging</code> » <code>org.jboss.ironjacamar.api</code> » <code>org.jboss.ironjacamar.impl</code> » <code>org.hibernate.validator</code> 	
JPA (Hibernate) subsystem	» <code>javax.persistence.api</code>	<ul style="list-style-type: none"> » <code>javaee.api</code> » <code>org.jboss.as.jpa</code> » <code>org.hibernate</code> » <code>org.javassist</code> 	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> in a deployment descriptor.
SAR Subsystem	-	<ul style="list-style-type: none"> » <code>org.jboss.logging</code> » <code>org.jboss.modules</code> 	The deployment is a SAR archive
Security Subsystem	» <code>org.picketbox</code>	-	-
Web Subsystem	-	<ul style="list-style-type: none"> » <code>javaee.api</code> » <code>com.sun.jsf-impl</code> » <code>org.hibernate.validator</code> » <code>org.jboss.as.web</code> » <code>org.jboss.logging</code> 	The deployment is a WAR archive. JavaServer Faces(JSF) is only added if used.
Web Services Subsystem	<ul style="list-style-type: none"> » <code>org.jboss.ws.api</code> » <code>org.jboss.ws.sp</code> 	-	-
Weld (CDI) Subsystem	-	<ul style="list-style-type: none"> » <code>javax.persistence.api</code> » <code>javaee.api</code> 	If a <code>beans.xml</code> file is detected in the deployment

```
» org.javassist
»
org.jboss.interceptor
»
org.jboss.as.weld
ld
»
org.jboss.logging
ing
»
org.jboss.weld
.core
»
org.jboss.weld
.api
»
org.jboss.weld
.spi
```

[Report a bug](#)

3.7.2. Included Modules

```
» asm.asm
» ch.qos.cal10n
» com.google.guava
» com.h2database.h2
» com.sun.jsf-impl
» com.sun.jsf-impl
» com.sun.xml.bind
» com.sun.xml.messaging.saa
j
» gnu.getopt
» javaee.api
» javax.activation.api
» javax.annotation.api
» javax.api
» javax.ejb.api
» javax.el.api
» javax.enterprise.api
» javax.enterprise.deploy
.api
» javax.faces.api
» javax.faces.api
» javax.inject.api
» javax.interceptor.api
» javax.jms.api
» javax.jws.api
» javax.mail.api
» javax.management.j2ee
.api
» javax.persistence.api
```

- `javax.resource.api`
- `javax.rmi.api`
- `javax.security.auth.message.api`
- `javax.security.jacc.api`
- `javax.servlet.api`
- `javax.servlet.jsp.api`
- `javax.servlet.jstl.api`
- `javax.transaction.api`
- `javax.validation.api`
- `javax.ws.rs.api`
- `javax.wsd14j.api`
- `javax.xml.bind.api`
- `javax.xml.jaxp-provider`
- `javax.xml.registry.api`
- `javax.xml.rpc.api`
- `javax.xml.soap.api`
- `javax.xml.stream.api`
- `javax.xml.ws.api`
- `jline`
- `net.sourceforge.cssparser`
- `net.sourceforge.htmlunit`
- `net.sourceforge.nekohtml`
- `nu.xom`
- `org.antlr`
- `org.apache.ant`
- `org.apache.commons.beanutils`
- `org.apache.commons.cli`
- `org.apache.commons.codec`
- `org.apache.commons.collections`
- `org.apache.commons.io`
- `org.apache.commons.lang`
- `org.apache.commons.logging`
- `org.apache.commons.pool`
- `org.apache.cxf`
- `org.apache.httpcomponents`
- `org.apache.james.mime4j`
- `org.apache.log4j`
- `org.apache.neethi`
- `org.apache.santuario.xmlsec`
- `org.apache.velocity`
- `org.apache.ws.scout`
- `org.apache.ws.security`
- `org.apache.ws.xmlschema`
- `org.apache.xalan`
- `org.apache.xerces`
- `org.apache.xml-resolver`
- `org.codehaus.jackson.jackson-core-asl`

- `org.codehaus.jackson.jackson-jaxrs`
- `org.codehaus.jackson.jackson-mapper-asl`
- `org.codehaus.jackson.jackson-xc`
- `org.codehaus.woodstox`
- `org.dom4j`
- `org.hibernate`
- `org.hibernate.envers`
- `org.hibernate.infinispan`
- `org.hibernate.validator`
- `org.hornetq`
- `org.hornetq.ra`
- `org.infinispan`
- `org.infinispan.cachestore.jdbc`
- `org.infinispan.cachestore.remote`
- `org.infinispan.client.hotrod`
- `org.jacorb`
- `org.javassist`
- `org.jaxen`
- `org.jboss.as.aggregate`
- `org.jboss.as.appclient`
- `org.jboss.as.cli`
- `org.jboss.as.clustering.api`
- `org.jboss.as.clustering.common`
- `org.jboss.as.clustering.ejb3.infinispan`
- `org.jboss.as.clustering.impl`
- `org.jboss.as.clustering.infinispan`
- `org.jboss.as.clustering.jgroups`
- `org.jboss.as.clustering.service`
- `org.jboss.as.clustering.singleton`
- `org.jboss.as.clustering.web.infinispan`
- `org.jboss.as.clustering.web.spi`
- `org.jboss.as.cmp`
- `org.jboss.as.connector`
- `org.jboss.as.console`
- `org.jboss.as.controller`
- `org.jboss.as.controller-client`
- `org.jboss.as.deployment-repository`
- `org.jboss.as.deployment-scanner`
- `org.jboss.as.domain-add-user`
- `org.jboss.as.domain-http-error-context`
- `org.jboss.as.domain-http-interface`
- `org.jboss.as.domain-management`
- `org.jboss.as.ee`
- `org.jboss.as.ee.deployment`
- `org.jboss.as.ejb3`
- `org.jboss.as.embedded`

- `org.jboss.as.host-controller`
- `org.jboss.as.jacorb`
- `org.jboss.as.jaxr`
- `org.jboss.as.jaxrs`
- `org.jboss.as.jdr`
- `org.jboss.as.jmx`
- `org.jboss.as.jpa`
- `org.jboss.as.jpa.hibernate`
- `org.jboss.as.jpa.hibernate`
- `org.jboss.as.jpa.hibernate.infinispan`
- `org.jboss.as.jpa.openjpa`
- `org.jboss.as.jpa.spi`
- `org.jboss.as.jpa.util`
- `org.jboss.as.jsr77`
- `org.jboss.as.logging`
- `org.jboss.as.mail`
- `org.jboss.as.management-client-content`
- `org.jboss.as.messaging`
- `org.jboss.as.modcluster`
- `org.jboss.as.naming`
- `org.jboss.as.network`
- `org.jboss.as.osgi`
- `org.jboss.as.platform-mbean`
- `org.jboss.as.pojo`
- `org.jboss.as.process-controller`
- `org.jboss.as.protocol`
- `org.jboss.as.remoting`
- `org.jboss.as.sar`
- `org.jboss.as.security`
- `org.jboss.as.server`
- `org.jboss.as.standalone`
- `org.jboss.as.threads`
- `org.jboss.as.transactions`
- `org.jboss.as.web`
- `org.jboss.as.webservices`
- `org.jboss.as.webservices.server.integration`
- `org.jboss.as.webservices.server.jaxrpc-integration`
- `org.jboss.as.weld`
- `org.jboss.as.xts`
- `org.jboss.classfilewriter`
- `org.jboss.com.sun.httpserver`
- `org.jboss.common-core`
- `org.jboss.dmr`
- `org.jboss.ejb-client`
- `org.jboss.ejb3`
- `org.jboss.iiop-client`
- `org.jboss.integration.ext-content`

- `org.jboss.interceptor`
- `org.jboss.interceptor.spi`
- `org.jboss.invocation`
- `org.jboss.ironjacamar.api`
- `org.jboss.ironjacamar.impl`
- `org.jboss.ironjacamar.jdbcadapters`
- `org.jboss.jandex`
- `org.jboss.jaxbintros`
- `org.jboss.jboss-transaction-spi`
- `org.jboss.jsfunit.core`
- `org.jboss.jts`
- `org.jboss.jts.integration`
- `org.jboss.logging`
- `org.jboss.logmanager`
- `org.jboss.logmanager.log4j`
- `org.jboss.marshalling`
- `org.jboss.marshalling.river`
- `org.jboss.metadata`
- `org.jboss.modules`
- `org.jboss.msc`
- `org.jboss.netty`
- `org.jboss.osgi.deployment`
- `org.jboss.osgi.framework`
- `org.jboss.osgi.resolver`
- `org.jboss.osgi.spi`
- `org.jboss.osgi.vfs`
- `org.jboss.remoting3`
- `org.jboss.resteasy.resteasy-atom-provider`
- `org.jboss.resteasy.resteasy-cdi`
- `org.jboss.resteasy.resteasy-jackson-provider`
- `org.jboss.resteasy.resteasy-jaxb-provider`
- `org.jboss.resteasy.resteasy-jaxrs`
- `org.jboss.resteasy.resteasy-jsapi`
- `org.jboss.resteasy.resteasy-multipart-provider`
- `org.jboss.sasl`
- `org.jboss.security.negotiation`
- `org.jboss.security.xacml`
- `org.jboss.shrinkwrap.core`
- `org.jboss.staxmapper`
- `org.jboss.stdio`
- `org.jboss.threads`
- `org.jboss.vfs`
- `org.jboss.weld.api`
- `org.jboss.weld.core`
- `org.jboss.weld.spi`
- `org.jboss.ws.api`

- `org.jboss.ws.common`
- `org.jboss.ws.cxf.jbossws-cxf-client`
- `org.jboss.ws.cxf.jbossws-cxf-factories`
- `org.jboss.ws.cxf.jbossws-cxf-server`
- `org.jboss.ws.cxf.jbossws-cxf-transport-httpserver`
- `org.jboss.ws.jaxws-client`
- `org.jboss.ws.jaxws-jboss-httpserver-httpspi`
- `org.jboss.ws.native.jbossws-native-core`
- `org.jboss.ws.native.jbossws-native-factories`
- `org.jboss.ws.native.jbossws-native-services`
- `org.jboss.ws.saaj-impl`
- `org.jboss.ws.spi`
- `org.jboss.ws.tools.common`
- `org.jboss.ws.tools.wsconsume`
- `org.jboss.ws.tools.wsprovide`
- `org.jboss.xb`
- `org.jboss.xnio`
- `org.jboss.xnio.nio`
- `org.jboss.xts`
- `org.jdom`
- `org.jgroups`
- `org.joda.time`
- `org.junit`
- `org.omg.api`
- `org.osgi.core`
- `org.picketbox`
- `org.picketlink`
- `org.python.jython.standalone`
- `org.scannotation.scannotation`
- `org.slf4j`
- `org.slf4j.ext`
- `org.slf4j.impl`
- `org.slf4j.jcl-over-slf4j`
- `org.w3c.css.sac`
- `sun.jdk`

[Report a bug](#)

3.7.3. 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 behaviour in that EAR deployment.
- Adding additional resource roots to a module in an EAR.

[Report a bug](#)

Chapter 4. Logging for Developers

4.1. Introduction

4.1.1. 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](#)

4.1.2. Application Logging Frameworks Supported By JBoss LogManager

JBoss LogManager supports the following logging frameworks:

- ▶ JBoss Logging - included with JBoss Enterprise Application Platform 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>

[Report a bug](#)

4.1.3. 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 Enterprise Application Platform 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](#)

4.1.4. Supported Log Levels

Table 4.1. 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.
FATAL	1100	Use to indicate events that could cause critical service failure and application shutdown and possibly cause JBoss Enterprise Application Platform 6 to shutdown.

[Report a bug](#)

4.1.5. 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 4.2. Default Log Files for a standalone server

Log File	Description
<i>EAP_HOME/standalone/log/boot.log</i>	The server boot log. Contains log messages related to the startup of the server.
<i>EAP_HOME/standalone/log/server.log</i>	The Server Log. Contains all log messages once the server has launched.

Table 4.3. Default Log Files for a managed domain

Log File	Description
<i>EAP_HOME/domain/log/host-controller/boot.log</i>	Host Controller boot log. Contains log messages related to the startup of the host controller.
<i>EAP_HOME/domain/log/process-controller/boot.log</i>	Process controller boot log. Contains log messages related to the startup of the process controller.
<i>EAP_HOME/domain/servers/SERVERNAME/log/boot.log</i>	Server Boot log for the named server. Contains log messages related to the startup of the specified server.
<i>EAP_HOME/domain/servers/SERVERNAME/log/server.log</i>	The server log for the named server. Contains all log messages for that server once it has launched.

[Report a bug](#)

4.2. Logging with the JBoss Logging Framework

4.2.1. About JBoss Logging

JBoss Logging is the application logging framework that is included in JBoss Enterprise Application Platform 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](#)

4.2.2. 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](#)

4.2.3. 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

You must meet the following conditions before continuing with this task:

- If you are using Maven as your build system, the project must already be configured to include the JBoss Maven Repository. Refer to [Section 2.3.2, “Configure the JBoss Enterprise Application Platform 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 JBoss Developer Studio or with Maven.
 - When building using JBoss Developer Studio this can be done selecting Project -> Properties from the JBoss Developer Studio menu, selecting Targeted Runtimes and ensuring the runtime for JBoss Enterprise Application Platform 6 is checked.
 - When building using Maven this can be done by adding the following dependency configuration to your project's **`pom.xml`** file.

```
<dependency>
  <groupId>org.jboss.logging</groupId>
  <artifactId>jboss-logging</artifactId>
  <version>3.0.0.CR1</version>
  <scope>provided</scope>
</dependency>
```

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

Once your project is setup correctly. You need to follow the following steps for each class that you want

to add logging to:

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.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 JBoss Logging Javadoc at <http://docs.jboss.org/jbosslogging/latest/>.

Example 4.1. 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, a `ERROR` level log message is recorded.

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

    public Properties openCustomProperties(String configname)
    {
        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](#)

Chapter 5. Internationalization and Localization

5.1. Introduction

5.1.1. 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](#)

5.1.2. 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](#)

5.2. JBoss Logging Tools

5.2.1. Overview

5.2.1.1. 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](#)

5.2.1.2. 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](#)

5.2.1.3. 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](#)

5.2.1.4. 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](#)

5.2.1.5. 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](#)

5.2.1.6. 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](#)

5.2.1.7. 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](#)

5.2.1.8. 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](#)

5.2.1.9. 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](#)

5.2.1.10. 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](#)

5.2.2. Creating Internationalized Loggers, Messages and Exceptions

5.2.2.1. 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 5.2.6.1, “JBoss Logging Tools Maven Configuration”](#).
2. The project must have the required maven configuration for JBoss Logging Tools.

Procedure 5.1. 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
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 now be localized.

[Report a bug](#)

5.2.2.2. 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 Enterprise Application Platform 6 repository. Refer to [Section 2.3.2, “Configure the JBoss Enterprise Application Platform Maven Repository Using the Maven Settings”](#).
2. The required Maven configuration for JBoss Logging Tools has been added. Refer to [Section 5.2.6.1, “JBoss Logging Tools Maven Configuration”](#).

Procedure 5.2. 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
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](#)

5.2.2.3. 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-qs** 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 JBoss Developer Studio or Maven, to which you want to add internationalized exceptions.

Procedure 5.3. Create and use Internationalized Exceptions

1. Add JBoss Logging Tools configuration

Add the required project configuration to support JBoss Logging Tools. Refer to [Section 5.2.6.1, “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
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](#)

5.2.3. Localizing Internationalized Loggers, Messages and Exceptions

5.2.3.1. 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-qs 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.

Procedure 5.4. 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](#)

5.2.3.2. 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 5.5. 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](#)

5.2.4. Customizing Internationalized Log Messages

5.2.4.1. 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 5.2.2.1, "Create Internationalized Log Messages"](#).
2. You need to know what the project code you will be using is. You can use a single project code, or define different ones for each interface.

Procedure 5.6. 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 have been 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](#)

5.2.4.2. 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 5.7. 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](#)

5.2.4.3. 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 5.8. 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 third parameter, and so on.
- To use explicit indexes, insert the characters **%{#}** in the message where **#** is the number of

the parameter you want 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 5.1. 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 5.2. Message parameters using explicit indexes

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

[Report a bug](#)

5.2.4.4. 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 5.9. 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.

```
@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

@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](#)

5.2.5. Customizing Internationalized Exceptions

5.2.5.1. 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 5.2.2.3, “Create Internationalized Exceptions”](#).
2. You need to know what the project code you will be using is. You can use a single project code, or define different ones for each interface.

Procedure 5.10. 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 5.3. 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](#)

5.2.5.2. 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 5.11. 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 third parameter, and so on.
- To use explicit indexes, insert the characters `%{#}` in the message where `#` is the number of the parameter you want 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 5.4. Using ordinary indexes

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

Example 5.5. Using explicit indexes

```
@Message(id=143, value = "The config file %{1} could not be opened.")
IOException configFileAccessError(File config);
```

[Report a bug](#)

5.2.5.3. 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 5.12. 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 5.6. 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](#)

5.2.6. Reference

5.2.6.1. 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 `jboss-logging-tool-qs` 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 Enterprise Application Platform 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 Enterprise Application Platform 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](#)

5.2.6.2. 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 5.7. 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](#)

5.2.6.3. 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 5.1. 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 method that is a logging method.	level (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 6. Enterprise JavaBeans

6.1. Introduction

6.1.1. 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 Enterprise Application Platform 6 has full support for applications built using the Enterprise JavaBeans 3.1 specification. The EJB Container is implemented using the JBoss EJB3 community project, <http://www.jboss.org/ejb3>.

[Report a bug](#)

6.1.2. 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](#)

6.1.3. 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](#)

6.1.4. 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](#)

6.1.5. 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](#)

6.1.6. 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 if the beans comply with the EJB 3.1 Lite specification.

[Report a bug](#)

6.1.7. Session Bean Business Interfaces

6.1.7.1. 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](#)

6.1.7.2. 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](#)

6.1.7.3. 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. This is done using the same methods as remote clients and incurs all the normal overhead of making a remote request.

[Report a bug](#)

6.1.7.4. 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](#)

6.2. Creating Enterprise Bean Projects

6.2.1. Create an EJB Archive Project Using JBoss Developer Studio

This task describes how to create an Enterprise JavaBeans (EJB) project in JBoss Developer Studio.

Task Prerequisites:



- A server and server runtime for JBoss Enterprise Application Platform 6 has been set up.

Procedure 6.1. Create an EJB Project in 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**.

EJB Project

  Name cannot be empty.

Project name:

Project location

☒ Use default location

Location:

Target runtime

EJB module version

Configuration

A good starting point for working with JBoss EAP 6.0 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




Figure 6.1. 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 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 Enterprise Application Platform 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.

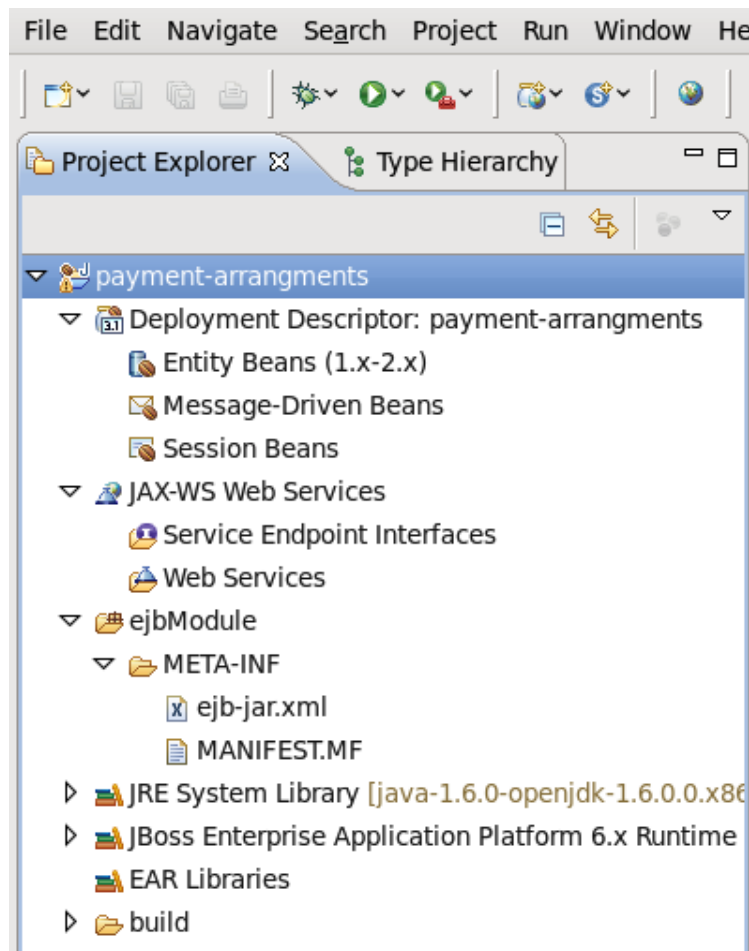


Figure 6.2. Newly created EJB Project in the Project Explorer

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 6.3. Add and Remove dialog

RESULT: You now have an EJB Project in JBoss Developer Studio that can build and deploy to the specified server.

If no enterprise beans are added to the project then 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](#)

6.2.2. 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 6.2. 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](#)

6.2.3. Create an EAR Project containing an EJB Project

This task describes how to create a new Enterprise Archive (EAR) project in JBoss Developer Studio that contains an EJB Project.

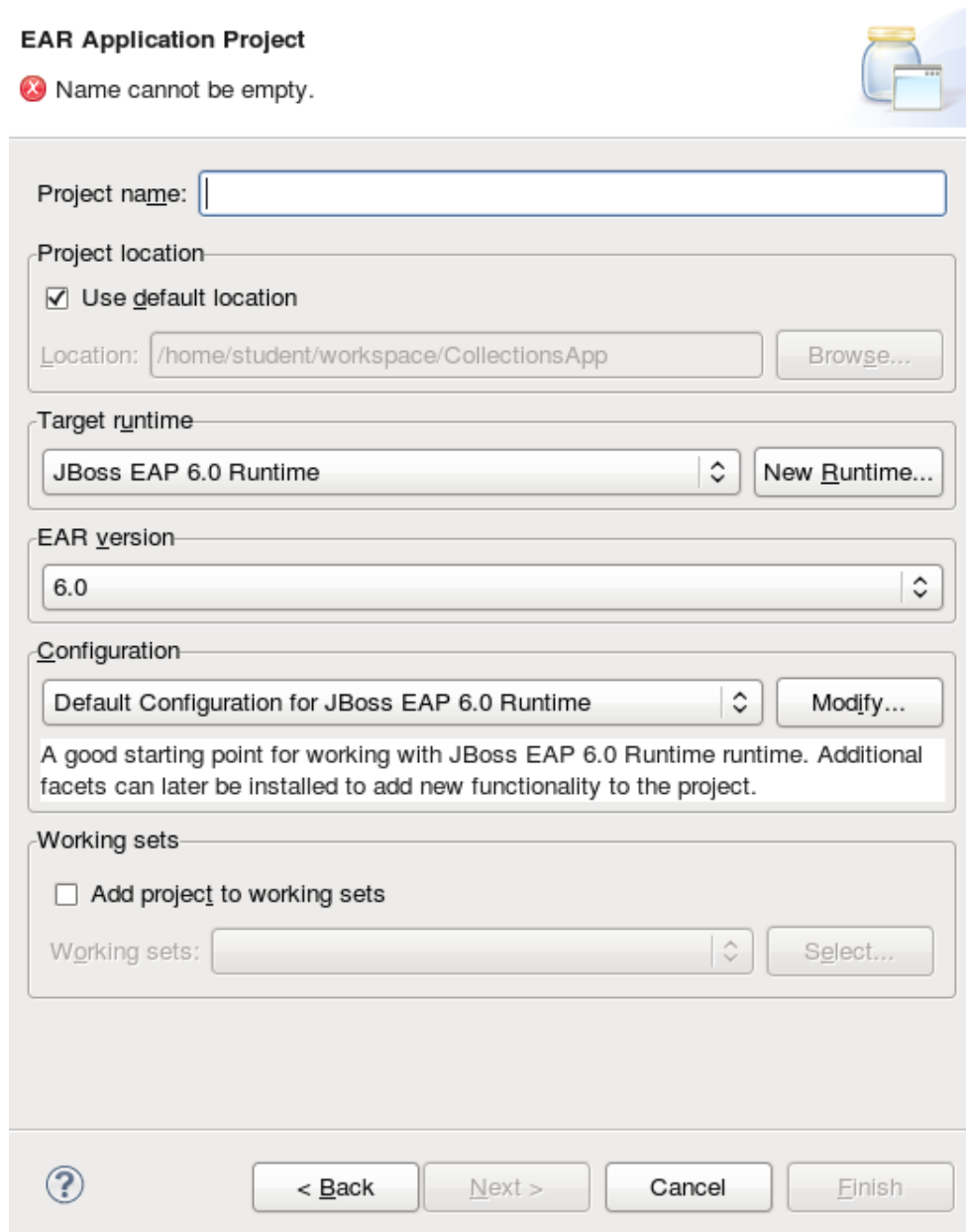
Task Prerequisites:

- A server and server runtime for JBoss Enterprise Application Platform 6 has been set up. Refer to [Section 1.4.1.5, “Add the JBoss Enterprise Application Platform 6 Server to JBoss Developer Studio”](#).


Procedure 6.3. 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**.



EAR Application Project

 Name cannot be empty.

Project name:

Project location

☒ Use default location

Location:

Target runtime

EAR version

Configuration

A good starting point for working with JBoss EAP 6.0 Runtime runtime. Additional facets can later be installed to add new functionality to the project.

Working sets

☐ Add project to working sets

Working sets:




Figure 6.4. 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 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 Enterprise Application Platform 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 6.2.1, "Create an EJB Archive Project Using 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:

Add >

< Remove

Add All >>

<< Remove All

Configured:

▼ CollectionsApp
CollectionsAppEJB

☒ If server is started, publish changes immediately



< Back

Next >

Cancel

Finish

Figure 6.5. 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](#)

6.2.4. 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.

Prerequisites:

- You have a EJB Project in JBoss Developer Studio to which you want to add an EJB deployment descriptor.

Procedure 6.4. Add an Deployment Descriptor to an EJB Project

1. Open the Project

Open the project in JBoss Developer Studio.

2. Add Deployment Descriptor

Right-click on the Deployment Descriptor folder in the project view and select **Generate Deployment Descriptor Stub**.

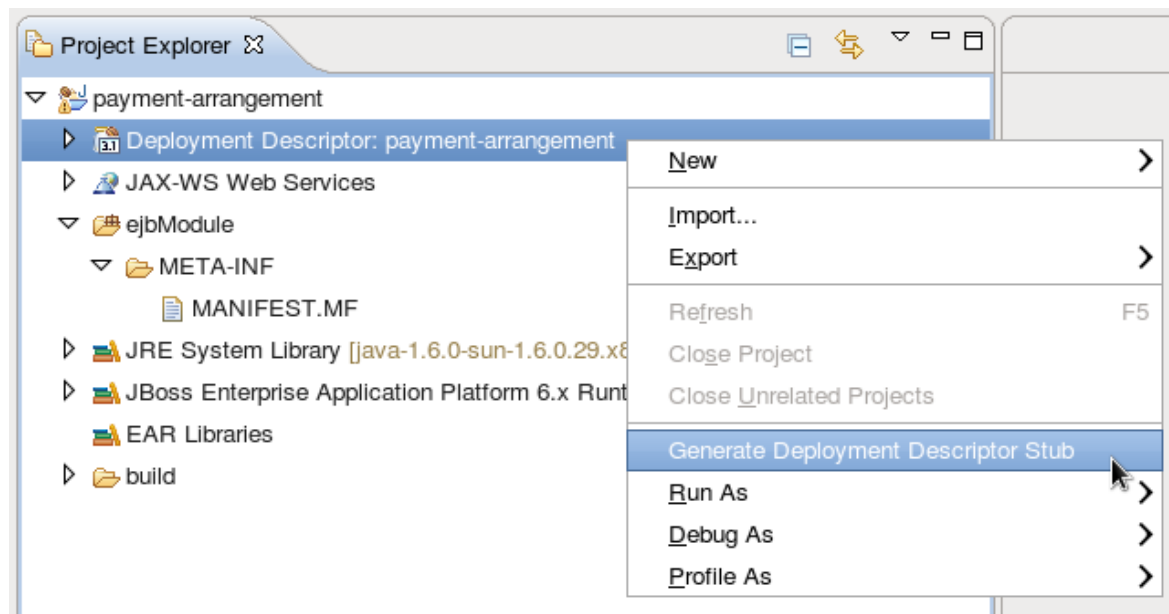


Figure 6.6. Adding a Deployment Descriptor

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](#)

6.3. Session Beans

6.3.1. 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](#)

6.3.2. 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](#)

6.3.3. 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](#)

6.3.4. 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](#)

6.3.5. Add Session Beans to a Project in JBoss Developer Studio

JBoss Developer Studio has several wizards that can be used to quickly create enterprise bean classes. The following procedure shows how to use the JBoss Developer Studio wizards to add a session bean to a project.

Prerequisites:

- You have a EJB or Dynamic Web Project in JBoss Developer Studio to which you want to add one or more session beans.

Procedure 6.5. Add Session Beans to a Project in JBoss Developer Studio

1. **Open the Project**

Open the project in 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 6.7. 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.

RESULT: A new session bean is added to the project.

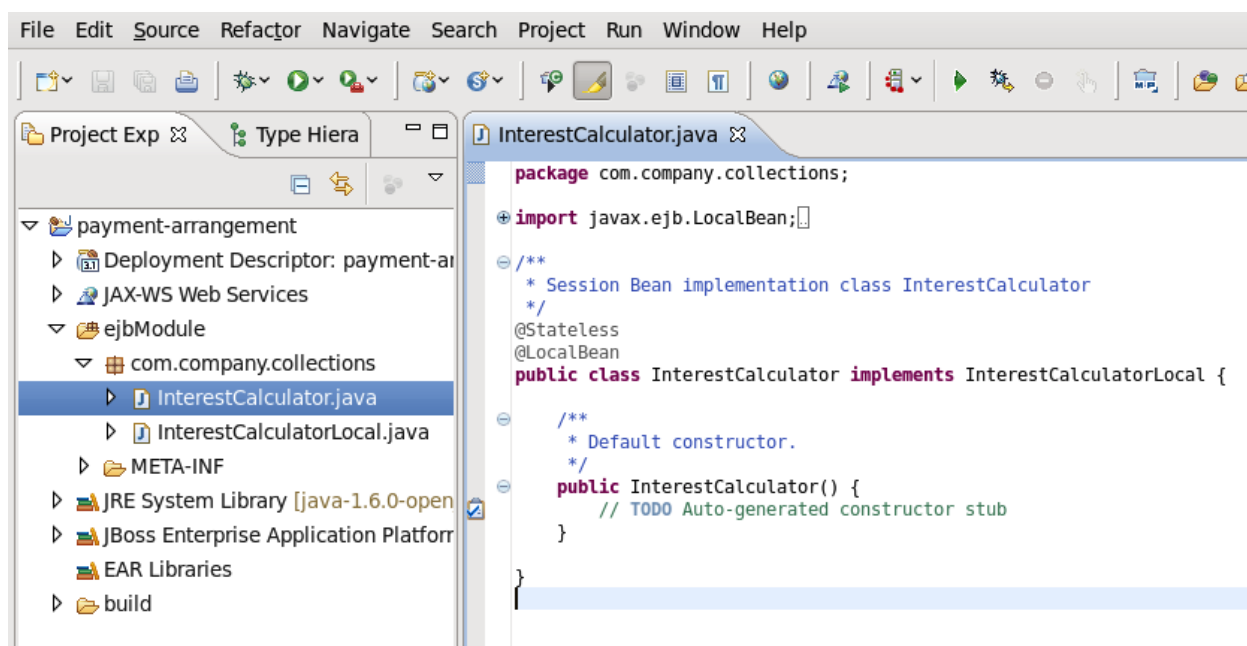


Figure 6.8. New Session Bean in JBoss Developer Studio

[Report a bug](#)

6.4. Message-Driven Beans

6.4.1. 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](#)

6.4.2. 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 Systems 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 Enterprise Application Platform 6. A RAR file may also be included in an Enterprise Archive (EAR) deployment.

[Report a bug](#)

6.4.3. Create a JMS-based Message-Driven Bean in JBoss Developer Studio

This procedure shows how to add a JMS-based Message-Driven Bean to a project in JBoss Developer Studio. This procedure creates an EJB 3.x Message-Driven Bean that uses annotations.

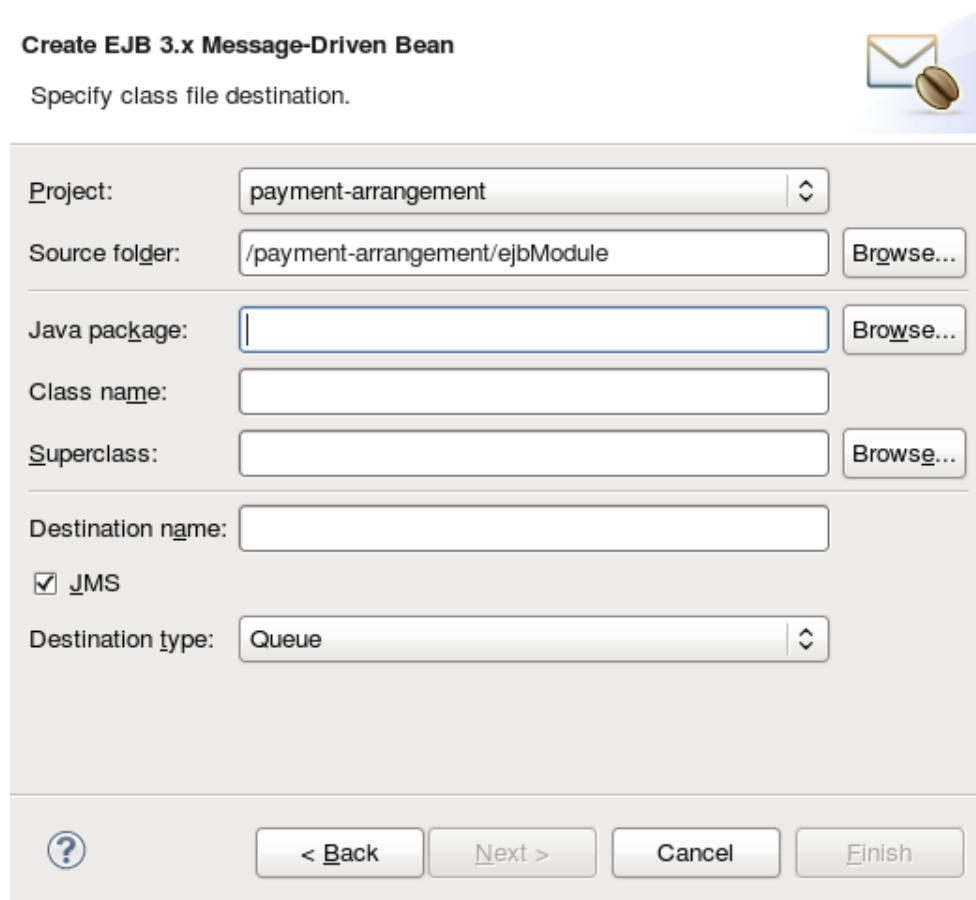
Prerequisites:

1. You must have an existing project open in 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 Enterprise Application Platform configuration to which this bean will be deployed.

Procedure 6.6. Add a JMS-based Message-Driven Bean in 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 6.9. 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 JBoss Developer Studio editor window opened with the corresponding file.

[Report a bug](#)

6.5. Invoking Session Beans

6.5.1. 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 **remote-ejb** quick start contains working Maven projects that demonstrate this functionality. The quick start 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.

Prerequisites

The following prerequisites must be satisfied before beginning:

- You must already have a Maven project created ready to use.

- ▶ Configuration for the JBoss Enterprise Application Platform 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 6.7. 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/resources/** directory of your project with the following content.

```
# Set this to true for SSL
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
remote.connections=default
# Uncomment this 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 other 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** lines. 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-as-ejb-remote-server-side</artifactId>
  <type>ejb-client</type>
  <version>7.1.0.CR1-SNAPSHOT</version>
</dependency>
```

Now that the project has been configured correctly, you can add the code to access and invoke the session beans.

Procedure 6.8. 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 `remote-ejb` 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-as-ejb-remote-app/CalculatorBean!" +
    RemoteCalculator.class.getName()
);
```

Session bean JNDI names are defined using a special syntax.

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](#)

6.6. Clustered Enterprise JavaBeans

6.6.1. 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 7.4, "Implement an HA Singleton"](#).

**Note**

EJB 2 entity beans cannot be clustered. This limitation is not expected to be changed.

[Report a bug](#)

6.7. Reference

6.7.1. 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 Enterprise Application Platform 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](#)

6.7.2. 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](#)

6.7.3. 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 Enterprise Application Platform 6 Maven repository.

Table 6.1. Maven dependencies for Remote EJB Clients

GroupID	ArtifactID	Version
org.jboss.spec	jboss-javaee-6.0	3.0.0.Final-redhat-1
org.jboss.as	jboss-as-ejb-client-bom	7.1.1.Final-redhat-1
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**.

```

<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 the **remote-ejb/client/pom.xml** for a complete example of dependency configuration for remote session bean invocation.

[Report a bug](#)

6.7.4. 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 adaptor namespace: `urn:resource-adapter-binding`

```
xmlns:r="urn:resource-adapter-binding"
```

This allows you to set the resource adaptor for an Message-Driven Bean.

```
<r:resource-adapter-binding>
  <ejb-name>*</ejb-name>
  <r:resource-adapter-name>myResourceAdaptor</r:resource-adapter-name>
</r:resource-adapter-binding>
```

The IIOP namespace: `urn:iiop`

```
xmlns:u="urn:iiop"
```

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 6.1. Example jboss-ejb3.xml file

```

<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
http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-
jar_3_1.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.ReplyingMDB</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>

```

[Report a bug](#)

Chapter 7. Clustering in Web Applications

7.1. Session Replication

7.1.1. About HTTP Session Replication

Session replication ensures that client sessions of distributable applications are not disrupted by failovers by 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](#)

7.1.2. 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.

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](#)

7.1.3. 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 7.1. 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 7.1.4, “Enable Session Replication in Your Application”](#).

[Report a bug](#)

7.1.4. Enable Session Replication in Your Application**Overview**

To take advantage of the JBoss Enterprise Application Platform's High Availability (HA) features, 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 7.2. Task**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 7.1. Minimum Configuration for a Distributable Application

```
<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. 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 7.2. Default <replication-config>Values

```
<jboss-web>

  <replication-config>
    <cache-name>custom-session-cache</cache-name>
    <replication-trigger>SET</replication-trigger>
    <replication-granularity>ATTRIBUTE</replication-granularity>
    <replication-field-batch-mode>true</replication-field-batch-mode>
    <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 7.1. Common Options for session Replication

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 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 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 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>

The following options rarely need to be changed.

Table 7.2. Less Commonly Changed Options for Session Replication

Option	Description
<code><useJK></code>	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.
<code><max-unreplicated-interval></code>	<p>The maximum interval (in seconds) to wait after a session 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. This means that the jvmRoute configuration of the container determines whether JK failover is being used. 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>
<code><snapshot-mode></code>	<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 <code><snapshot-interval></code> option is ignored.</p> <p>In INTERVAL mode, a background task runs at the interval specified by <code><snapshot-interval></code>, and replicates modified sessions.</p>
<code><snapshot-interval></code>	The interval, in milliseconds, at which modified sessions should be replicated when using INTERVAL for the value of <code><snapshot-mode></code> .
<code><session-notification-policy></code>	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](#)

7.2. HttpSession Passivation and Activation

7.2.1. 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](#)

7.2.2. 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 7.3. Example `jboss-web.xml` File

```
<jboss-web>

  <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.



REPL and DIST Replication Modes

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 **owner** parameter. Refer to [Section 7.1.2, “About the Web Session Cache”](#) and [Section 7.1.3, “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 100 sessions in memory.

[Report a bug](#)

7.3. Cookie Domain

7.3.1. 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 7.3.2, “Configure the Cookie Domain”](#).

[Report a bug](#)

7.3.2. Configure the Cookie Domain

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.

Example 7.4. Example Cookie Domain Configuration

```
<Valve className="org.jboss.web.tomcat.service.sso.ClusteredSingleSignOn"
        cookieDomain="xyz.com" />
```

[Report a bug](#)

7.4. Implement an HA Singleton

Summary

In JBoss Enterprise Application Platform 5, HA singleton archives were deployed in the **deploy-hasingleton/** directory separate from other deployments. This was done to prevent automatic deployment and to ensure the HASingletonDeployer service controlled the deployment and deployed the archive only on the master node in the cluster. There was no hot deployment feature, so redeployment required a server restart. Also, if the master node failed requiring another node to take over as master, the singleton service had to go through the entire deployment process in order to provide the service.

In JBoss Enterprise Application Platform 6 this has changed. Using a SingletonService, the target service is installed on every node in the cluster but is only started on one node at any given time. This approach simplifies the deployment requirements and minimizes the time required to relocate the singleton master service between nodes.

Procedure 7.3. 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. Create a singleton service.

```

package com.mycompany.hasingleton.service.ejb;

import java.util.concurrent.atomic.AtomicBoolean;
import java.util.logging.Logger;

import org.jboss.as.server.ServerEnvironment;
import org.jboss.msc.inject.Injector;
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;
import org.jboss.msc.value.InjectedException;

/**
 * @author <a href="mailto:wfink@redhat.com">Wolf-Dieter Fink</a>
 */
public class EnvironmentService implements Service<String> {
    private static final Logger LOGGER =
        Logger.getLogger(EnvironmentService.class.getCanonicalName());
    private static final ServiceName SINGLETON_SERVICE_NAME =
        ServiceName.JBOSS.append("quickstart", "ha", "singleton");
    /**
     * A flag whether the service is started.
     */
    private final AtomicBoolean started = new AtomicBoolean(false);

    private String nodeName;

    private final InjectedValue<ServerEnvironment> env = new
        InjectedValue<ServerEnvironment>();

    public Injector<ServerEnvironment> getEnvInjector() {
        return this.env;
    }

    /**
     * @return the name of the server node
     */
    public String getValue() throws IllegalStateException,
        IllegalArgumentException {
        if (!started.get()) {
            throw new IllegalStateException("The service '" +
                this.getClass().getName() + "' is not ready!");
        }
        return this.nodeName;
    }

    public void start(StartContext arg0) throws StartException {
        if (!started.compareAndSet(false, true)) {
            throw new StartException("The service is still started!");
        }
        LOGGER.info("Start service '" + this.getClass().getName() + "'");
        this.nodeName = this.env.getValue().getNodeName();
    }

    public void stop(StopContext arg0) {
        if (!started.compareAndSet(true, false)) {
            LOGGER.warning("The service '" + this.getClass().getName() +
                "' is not active!");
        } else {
            LOGGER.info("Stop service '" + this.getClass().getName() +
                "'");
        }
    }
}

```

- b. Create a singleton EJB to start the service as a SingletonService at server start.


```

package com.mycompany.hasingleton.service.ejb;

import java.util.Collection;
import java.util.EnumSet;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.Singleton;
import javax.ejb.Startup;

import org.jboss.as.clustering.singleton.SingletonService;
import org.jboss.as.server.CurrentServiceContainer;
import org.jboss.as.server.ServerEnvironment;
import org.jboss.as.server.ServerEnvironmentService;
import org.jboss.msc.service.AbstractServiceListener;
import org.jboss.msc.service.ServiceController;
import org.jboss.msc.service.ServiceController.Transition;
import org.jboss.msc.service.ServiceListener;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * A Singleton EJB to create the SingletonService during startup.
 *
 * @author <a href="mailto:wfink@redhat.com">Wolf-Dieter Fink</a>
 */
@Singleton
@Startup
public class StartupSingleton {
    private static final Logger LOGGER =
        LoggerFactory.getLogger(StartupSingleton.class);

    /**
     * Create the Service and wait until it is started.<br/>
     * Will log a message if the service will not start in 10sec.
     */
    @PostConstruct
    protected void startup() {
        LOGGER.info("StartupSingleton will be initialized!");

        EnvironmentService service = new EnvironmentService();
        SingletonService<String> singleton = new
        SingletonService<String>(service,
        EnvironmentService.SINGLETON_SERVICE_NAME);
        // if there is a node where the Singleton should deployed the
        election policy might set,
        // otherwise the JGroups coordinator will start it
        // singleton.setElectionPolicy(new
        PreferredSingletonElectionPolicy(new NamePreference("node2/cluster"),
        new SimpleSingletonElectionPolicy()));
        ServiceController<String> controller =
        singleton.build(CurrentServiceContainer.getServiceContainer())
            .addDependency(ServerEnvironmentService.SERVICE_NAME,
        ServerEnvironment.class, service.getEnvInjector())
            .install();

        controller.setMode(ServiceController.Mode.ACTIVE);
        try {
            wait(controller, EnumSet.of(ServiceController.State.DOWN,
        ServiceController.State.STARTING), ServiceController.State.UP);
            LOGGER.info("StartupSingleton has started the Service");
        } catch (IllegalStateException e) {
            LOGGER.warn("Singleton Service {} not started, are you sure to
        start in a cluster (HA)
        environment?", EnvironmentService.SINGLETON_SERVICE_NAME);
        }
    }
}

```

```

    }

    /**
     * Remove the service during undeploy or shutdown
     */
    @PreDestroy
    protected void destroy() {
        LOGGER.info("StartupSingleton will be removed!");
        ServiceController<?> controller =
            CurrentServiceContainer.getServiceContainer().getRequiredService(EnvironmentService.SINGLETON_SERVICE_NAME);
        controller.setMode(ServiceController.Mode.REMOVE);
        try {
            wait(controller, EnumSet.of(ServiceController.State.UP,
                ServiceController.State.STOPPING, ServiceController.State.DOWN),
                ServiceController.State.REMOVED);
        } catch (IllegalStateException e) {
            LOGGER.warn("Singleton Service {} has not be stopped correctly!", EnvironmentService.SINGLETON_SERVICE_NAME);
        }
    }

    private static <T> void wait(ServiceController<T> controller,
        Collection<ServiceController.State> expectedStates,
        ServiceController.State targetState) {
        if (controller.getState() != targetState) {
            ServiceListener<T> listener = new NotifyingServiceListener<T>();
            controller.addListener(listener);
            try {
                synchronized (controller) {
                    int maxRetry = 2;
                    while (expectedStates.contains(controller.getState()) &&
maxRetry > 0) {
                        LOGGER.info("Service controller state is {}, waiting for
transition to {}", new Object[] {controller.getState(), targetState});
                        controller.wait(5000);
                        maxRetry--;
                    }
                }
            } catch (InterruptedException e) {
                LOGGER.warn("Wait on startup is interrupted!");
                Thread.currentThread().interrupt();
            }
            controller.removeListener(listener);
            ServiceController.State state = controller.getState();
            LOGGER.info("Service controller state is now {}", state);
            if (state != targetState) {
                throw new IllegalStateException(String.format("Failed to wait
for state to transition to %s. Current state is %s", targetState, state),
                    controller.getStartException());
            }
        }
    }

    private static class NotifyingServiceListener<T> extends
        AbstractServiceListener<T> {
        @Override
        public void transition(ServiceController<? extends T> controller,
            Transition transition) {
            synchronized (controller) {
                controller.notify();
            }
        }
    }
}

```

c. Create a Stateless Session Bean to access the service from a client.

```

package com.mycompany.hasingleton.service.ejb;

import javax.ejb.Stateless;

import org.jboss.as.server.CurrentServiceContainer;
import org.jboss.msc.service.ServiceController;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * A simple SLSB to access the internal SingletonService.
 *
 * @author <a href="mailto:wfink@redhat.com">Wolf-Dieter Fink</a>
 */
@Stateless
public class ServiceAccessBean implements ServiceAccess {
    private static final Logger LOGGER =
        LoggerFactory.getLogger(ServiceAccessBean.class);

    public String getNodeNameOfService() {
        LOGGER.info("getNodeNameOfService() is called()");
        ServiceController<?> service =
            CurrentServiceContainer.getServiceContainer().getService(
                EnvironmentService.SINGLETON_SERVICE_NAME);
        LOGGER.debug("SERVICE {}", service);
        if (service != null) {
            return (String) service.getValue();
        } else {
            throw new IllegalStateException("Service '" +
                EnvironmentService.SINGLETON_SERVICE_NAME + "' not found!");
        }
    }
}

```

d. Create the business logic interface for the SingletonService.

```

package com.mycompany.hasingleton.service.ejb;

import javax.ejb.Remote;

/**
 * Business interface to access the SingletonService via this EJB
 *
 * @author <a href="mailto:wfink@redhat.com">Wolf-Dieter Fink</a>
 */
@Remote
public interface ServiceAccess {
    public abstract String getNodeNameOfService();
}

```

2. Start each JBoss Enterprise Application Platform 6 instance with clustering enabled

The method for enabling clustering depends on whether the servers are standalone or running in a managed domain.

a. Enable clustering for servers running in a managed domain

To enable clustering for servers started using the domain controller, update your **domain.xml** and designate a server group to use the **ha** profile and **ha-sockets** socket binding group. For example:

```
<server-groups>
  <server-group name="main-server-group" profile="ha">
    <jvm name="default">
      <heap size="64m" max-size="512m"/>
    </jvm>
    <socket-binding-group ref="ha-sockets"/>
  </server-group>
</server-groups>
```

Modify the **host.xml** file as follows:

```
<servers>
  <server name="server-one" group="main-server-group" auto-
start="false"/>
  <server name="server-two" group="distinct2">
    <socket-bindings port-offset="100"/>
  </server>
</servers>
```

Then start the server as follows

- A. For Linux, type: **EAP_HOME/bin/domain.sh**
- B. For Microsoft Windows, type: **EAP_HOME\bin\domain.bat**

b. Enable clustering for standalone servers

To enable clustering for standalone servers, start the server using the node name and the **standalone-ha.xml** configuration file as follows:

- A. For Linux, type: **EAP_HOME/bin/standalone.sh --server-config=standalone-ha.xml -Djboss.node.name=UNIQUE_NODE_NAME**
- B. For Microsoft Windows, type: **EAP_HOME\bin\standalone.bat --server-config=standalone-ha.xml -Djboss.node.name=UNIQUE_NODE_NAME**



Note

To avoid port conflicts when running multiple servers on one machine, configure the **standalone-ha.xml** file for each server instance to bind on a separate interface. Alternatively, you can start subsequent server instances with a port offset using an argument like the following on the command line: **-Djboss.socket.binding.port-offset=100**.

3. Deploy the application to the servers

If you use Maven to deploy your application, use the following Maven command to deploy to the server running on the default ports:

```
mvn clean install jboss-as:deploy
```

To deploy to additional servers, pass the server name and port number on the command line:

```
mvn clean package jboss-as:deploy -Ddeploy.hostname=localhost -
Ddeploy.port=10099
```

[Report a bug](#)

Chapter 8. CDI

8.1. Overview of CDI

8.1.1. Overview of CDI

- [Section 8.1.2, “About Contexts and Dependency Injection \(CDI\)”](#)
- [Section 8.1.5, “Relationship Between Weld, Seam 2, and JavaServer Faces”](#)
- [Section 8.1.3, “Benefits of CDI”](#)

[Report a bug](#)

8.1.2. 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 Enterprise Application Platform includes Weld, which is the reference implementation of JSR-299. For more information, about type-safe dependency injection, see [Section 8.1.4, “About Type-safe Dependency Injection”](#).

[Report a bug](#)

8.1.3. Benefits of CDI

- CDI simplifies and shrinks your code base by replacing big chunks of code with annotations.
- CDI 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.
- CDI simplifies packaging and deployments and reduces the amount of XML you need to add to your deployments.
- CDI provides lifecycle management via contexts. You can tie injections to requests, sessions, conversations, or custom contexts.
- CDI provides type-safe dependency injection, which is safer and easier to debug than string-based injection.
- CDI decouples interceptors from beans.
- CDI provides complex event notification.

[Report a bug](#)

8.1.4. 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 8.1.2, “About Contexts and Dependency Injection \(CDI\)”](#).

[Report a bug](#)

8.1.5. 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 Enterprise Application Platform.

[Report a bug](#)

8.2. Use CDI

8.2.1. First Steps

8.2.1.1. Enable CDI

Task Summary

Contexts and Dependency Injection (CDI) is one of the core technologies in the JBoss Enterprise Application Platform, and is enabled by default. If for some reason it is disabled and you need to enable it, follow this procedure.

Procedure 8.1. Task:

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 string `<extension module="org.jboss.as.weld"/>`. If it exists, it is located inside the `<extensions>` section.

2. **Before editing any files, stop the JBoss Enterprise Application Platform.**

The JBoss Enterprise Application Platform 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. **Restart the JBoss Enterprise Application Platform.**

Start the JBoss Enterprise Application Platform with your updated configuration.

Result:

The JBoss Enterprise Application Platform starts with the CDI subsystem enabled.

[Report a bug](#)

8.2.2. Use CDI to Develop an Application

8.2.2.1. 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. The JBoss Enterprise Application Platform includes Weld, the reference implementation of CDI. These tasks show you how to use CDI in your enterprise applications.

- [Section 8.2.1.1, “Enable CDI”](#)
- [Section 8.2.2.2, “Use CDI with Existing Code”](#)
- [Section 8.2.2.3, “Exclude Beans From the Scanning Process”](#)
- [Section 8.2.2.4, “Use an Injection to Extend an Implementation”](#)
- [Section 8.2.3.3, “Use a Qualifier to Resolve an Ambiguous Injection”](#)
- [Section 8.2.7.4, “Override an Injection with an Alternative”](#)
- [Section 8.2.7.2, “Use Named Beans”](#)
- [Section 8.2.6.1, “Manage the Lifecycle of a Bean”](#)
- [Section 8.2.6.2, “Use a Producer Method”](#)
- [Section 8.2.10.2, “Use Interceptors with CDI”](#)
- [Section 8.2.8.2, “Use Stereotypes”](#)
- [Section 8.2.9.2, “Fire and Observe Events”](#)

[Report a bug](#)

8.2.2.2. 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 a file called **beans.xml** in the **META-INF/** or **WEB-INF/** directory of your archive. The file can be empty.

Procedure 8.2. 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](#)

8.2.2.3. Exclude Beans From the Scanning Process

Task Summary

One of the features of Weld, the JBoss Enterprise Application Platform's 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 8.1. 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.

```
<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](#)

8.2.2.4. Use an Injection to Extend an Implementation

Task 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 8.2. 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](#)

8.2.3. Ambiguous or Unsatisfied Dependencies

8.2.3.1. 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 8.2.3.3, "Use a Qualifier to Resolve an Ambiguous Injection"](#).

[Report a bug](#)

8.2.3.2. 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 8.3. 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 8.4. 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](#)

8.2.3.3. Use a Qualifier to Resolve an Ambiguous Injection**Task Summary**

This task shows an ambiguous injection and removes the ambiguity with a qualifier. Read more about ambiguous injections at [Section 8.2.3.1, “About Ambiguous or Unsatisfied Dependencies”](#).

Example 8.5. 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 8.3. Task:

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](#)

8.2.4. Managed Beans

8.2.4.1. About Managed Beans

Managed beans, also called MBeans, are JavaBeans which are created using dependency injection. Each MBean represents a resource which runs in the Java Virtual Machine (JVM).

Java EE 6 expands upon this definition. A bean is implemented by a Java class, which is referred to as its bean class. A managed bean is a top-level Java class.

For more information about managed beans, refer to the JSR-255 specification at <http://jcp.org/en/jsr/detail?id=255>. For more information about CDI, refer to [Section 8.1.2, "About Contexts and Dependency Injection \(CDI\)"](#).

[Report a bug](#)

8.2.4.2. 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](#)

8.2.4.3. 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 8.6. 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 8.7. 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 8.8. 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 8.9. 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](#)

8.2.5. Contexts, Scopes, and Dependencies**8.2.5.1. 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](#)

8.2.5.2. Available Contexts

Table 8.1. 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.
Custom scopes	If the above contexts do not meet your needs, you can define custom scopes.

[Report a bug](#)

8.2.6. Bean Lifecycle

8.2.6.1. Manage the Lifecycle of a Bean

Task 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 8.2.5.1, "Contexts and Scopes"](#).

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 8.2.13.1, “About Bean Proxies”](#)
- ▶ [Section 8.2.13.2, “Use a Proxy in an Injection”](#)

[Report a bug](#)

8.2.6.2. Use a Producer Method**Task Summary:**

This task shows how to use producer methods to produce a variety of different objects which are not beans for injection.

Example 8.10. 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 8.2.3.2, “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 8.11. 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 8.12. 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](#)

8.2.7. Named Beans and Alternative Beans**8.2.7.1. 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](#)

8.2.7.2. 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](#)

8.2.7.3. About Alternative Beans

Alternatives are beans whose implementation is specific to a particular client module or deployment scenario.

Example 8.13. 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](#)

8.2.7.4. Override an Injection with an Alternative

Task 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 8.4. Task:

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 à " + 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](#)

8.2.8. Stereotypes

8.2.8.1. 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 8.2.7.1, “About Named Beans”](#).

[Report a bug](#)

8.2.8.2. Use Stereotypes

Task 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 8.2.8.1, “About Stereotypes”](#).

Example 8.14. Annotation clutter

```
@Secure
@Transactional
@RequestScoped
@Named
public class AccountManager {
    public boolean transfer(Account a, Account b) {
        ...
    }
}
```

Procedure 8.5. Task

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](#)

8.2.9. Observer Methods

8.2.9.1. About Observer Methods

Observer methods receive notifications when events occur.

CDI 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](#)

8.2.9.2. Fire and Observe Events

Example 8.15. 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 8.16. 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 8.2.3.2, “About Qualifiers”](#).

```
public class AccountManager {
    @Inject @Suspicious Event <Withdrawal> event;

    public boolean transfer(Account a, Account b) {
        ...
        event.fire(new Withdrawal(a));
    }
}
```

Example 8.17. Observe an event

To observe an event, use the **@Observes** annotation.

```
public class AccountObserver {
    void checkTran(@Observes Withdrawal w) {
        ...
    }
}
```

Example 8.18. Observe a qualified event

You can use qualifiers to observe only specific types of events. For more information about qualifiers, see [Section 8.2.3.2, “About Qualifiers”](#).

```
public class AccountObserver {  
    void checkTran(@Observes @Suspicious Withdrawal w) {  
        ...  
    }  
}
```

[Report a bug](#)

8.2.10. Interceptors

8.2.10.1. 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](#)

8.2.10.2. Use Interceptors with CDI

Example 8.19. 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 8.6. 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](#)

8.2.11. 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**.

Example 8.20. 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);

    ...

}

}
```

[Report a bug](#)

8.2.12. 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](#)

8.2.13. Bean Proxies

8.2.13.1. About Bean Proxies

A proxy is a subclass of a bean, which is generated at runtime. It is injected at bean creation time, and dependent scoped beans can be injected from it, because the lifecycles of the dependent beans are tied to proxy. Proxies are used as a substitute for dependency injection, and solve two different problems.

Problems of dependency injection, which are solved by using proxies

- Performance - Proxies are much faster than dependency injection, so you can use them in beans which need good performance.
- Thread safety - Proxies forward requests to the correct bean instance, even when multiple threads access a bean at the same time. Dependency injection does not guarantee thread safety.

Types of classes that cannot be proxied

- Primitive types or array types
- Classes that are **final** or have **final** methods
- Classes which have a non-private default constructor

[Report a bug](#)

8.2.13.2. 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 8.21. 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 8.2.13.1, “About Bean Proxies”](#).

[Report a bug](#)

Chapter 9. Java Transaction API (JTA)

9.1. Overview

9.1.1. Overview of Java Transactions API (JTA)

Introduction

These topics provide a foundational understanding of the Java Transactions API (JTA).

- ▶ [Section 9.2.5, “About Java Transactions API \(JTA\)”](#)
- ▶ [Section 9.5.2, “Lifecycle of a JTA Transaction”](#)
- ▶ [Section 9.9.3, “JTA Transaction Example”](#)

[Report a bug](#)

9.2. Transaction Concepts

9.2.1. 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](#)

9.2.2. About ACID Properties for Transactions

ACID is an acronym which stands for **Atomicity**, **Consistency**, **Isolation**, and **Durability**. 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](#)

9.2.3. About the Transaction Coordinator or Transaction Manager

The terms *Transaction Coordinator* and *Transaction Manager* are mostly interchangeable in terms of transactions with the JBoss Enterprise Application Platform. The term Transaction Coordinator is usually used in the context of distributed transactions.

In JTA transactions, The *Transaction Manager* runs within the JBoss Enterprise Application Platform 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 9.2.4, “About Transaction Participants”](#)
- ▶ [Section 9.2.2, “About ACID Properties for Transactions”](#)
- ▶ [Section 9.2.9, “About the 2-Phase Commit Protocol”](#)

[Report a bug](#)

9.2.4. 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 agrees to only commit its state if every other participant can also do so. Otherwise, they each roll back.

- ▶ [Section 9.2.1, “About Transactions”](#)
- ▶ [Section 9.2.3, “About the Transaction Coordinator or Transaction Manager”](#)

[Report a bug](#)

9.2.5. About Java Transactions API (JTA)

Java Transactions API (JTA) is a specification for using transactions in Java Enterprise Edition applications. It is defined in JSR-907.

JTA transactions are not distributed across multiple application servers, and cannot be nested.

JTA transactions are managed by the EJB container. Annotations are provided for creating and controlling transactions within your code.

[Report a bug](#)

9.2.6. About Java Transaction Service (JTS)

Java Transaction Service (JTS) is a mechanism for supporting Java Transaction API (JTA) transactions when participants of the transactions reside in multiple Java Enterprise Edition containers (application servers). Just as in local JTA transactions, each container runs a process called *Transaction Manager (TM)*. The TMs communicate with each other using a process called an *Object Request Broker (ORB)*, using a communication standard called *Common Object Request Broker Architecture (CORBA)*.

From an application standpoint, a JTS transaction behaves in the same ways as a JTA transaction. The difference is that transaction participants and datasources reside in different containers.



Note

The implementation of JTS included in JBoss Enterprise Application Platform supports *distributed JTA transactions*. The difference between distributed JTA transactions and fully-compliant JTS transactions is interoperability with external third-party ORBs. This feature is unsupported with JBoss Enterprise Application Platform 6. Supported configurations distribute transactions across multiple JBoss Enterprise Application Platform containers only.

- ▶ [Section 9.2.11, “About Distributed Transactions”](#)
- ▶ [Section 9.2.3, “About the Transaction Coordinator or Transaction Manager”](#)

[Report a bug](#)

9.2.7. 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](#)

9.2.8. 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 Enterprise Application Platform 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](#)

9.2.9. About the 2-Phase Commit Protocol

The Two-phase commit protocol (2PC) refers to the typical pattern of a database 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](#)

9.2.10. 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](#)

9.2.11. About Distributed Transactions

A *distributed transaction*, or *distributed Java Transaction API (JTA) transaction* is a transaction with participants on multiple JBoss Enterprise Application Platform servers. Distributed transactions differ from *Java Transaction Service (JTS) transactions* in that the JTS specifications mandate that transactions be able to be distributed across application servers from different vendors. The JBoss Enterprise Application Platform supports distributed JTA transactions.

[Report a bug](#)

9.2.12. 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 Enterprise Application Platform.

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 Enterprise Application Platform 6 Javadocs bundle on the Red Hat Customer Portal for specific details about the methods and properties included in the ORB Portability API.

[Report a bug](#)

9.2.13. 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 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.

[Report a bug](#)

9.2.14. About Garbage Collection

Garbage collection is a form of automatic memory management provided by the Java Virtual Machine (JVM). Periodically, the garbage collector runs, and reclaims memory which was claimed by objects which are no longer in use by applications.

An object becomes eligible for garbage collection when there are no more references to it. In effect, this means that no threads refer to it anymore.

Garbage collection happens outside of user control. The JVM decides to run garbage collection based on the amount of available heap size. The heap size is tunable for performance. Refer to the documentation of your JVM for more information.

[Report a bug](#)

9.3. Transaction Optimizations

9.3.1. Overview of Transaction Optimizations

Introduction

The Transactions subsystem of the JBoss Enterprise Application Platform includes several optimizations which you can take advantage of in your applications.

- ▶ [Section 9.3.3, “About the Presumed-Abort Optimization”](#)
- ▶ [Section 9.3.4, “About the Read-Only Optimization”](#)
- ▶ [Section 9.3.2, “About the LRCO Optimization for Single-phase Commit \(1PC\)”](#)

[Report a bug](#)

9.3.2. 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 9.2.9, “About the 2-Phase Commit Protocol”](#)

[Report a bug](#)

9.3.3. 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](#)

9.3.4. 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](#)

9.4. Transaction Outcomes

9.4.1. 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 9.4.3, “About Transaction Roll-Back”](#) for more information.

Commit

If every transaction participant can commit, the transaction coordinator directs them to do so. See [Section 9.4.2, “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 9.4.4, “About Heuristic Outcomes”](#) for more information.

[Report a bug](#)

9.4.2. 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](#)

9.4.3. 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](#)

9.4.4. 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 9.2.9, “About the 2-Phase Commit Protocol”](#)

[Report a bug](#)

9.4.5. JBoss Transactions Errors and Exceptions

For details about exceptions thrown by methods of the **UserTransaction** class, see the *UserTransaction API* specification at <http://download.oracle.com/javaee/1.3/api/javax/transaction/UserTransaction.html>.

[Report a bug](#)

9.5. Overview of JTA Transactions

9.5.1. About Java Transactions API (JTA)

Java Transactions API (JTA) is a specification for using transactions in Java Enterprise Edition applications. It is defined in JSR-907.

JTA transactions are not distributed across multiple application servers, and cannot be nested.

JTA transactions are managed by the EJB container. Annotations are provided for creating and controlling transactions within your code.

[Report a bug](#)

9.5.2. 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()** starts a new transaction. 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.

2. Your application modifies its state.

In the next step, your transaction 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. It calls **UserTransaction.commit()** or

UserTransaction.rollback(). This is when the two-phase commit protocol (2PC) happens if you have enlisted more than one resource. [Section 9.2.9, “About the 2-Phase Commit Protocol”](#)

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.

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.

- [Section 9.2.1, “About Transactions”](#)
- [Section 9.2.3, “About the Transaction Coordinator or Transaction Manager”](#)
- [Section 9.2.4, “About Transaction Participants”](#)
- [Section 9.2.9, “About the 2-Phase Commit Protocol”](#)
- [Section 9.2.7, “About XA Datasources and XA Transactions”](#)

[Report a bug](#)

9.6. Transaction Subsystem Configuration

9.6.1. Transactions Configuration Overview

Introduction

The following procedures show you how to configure the transactions subsystem of the JBoss Enterprise Application Platform.

- [Section 9.6.2.1, “Configure Your Datasource to Use JTA Transactions”](#)
- [Section 9.6.2.2, “Configure an XA Datasource”](#)
- [Section 9.7.8.2, “Configure the Transaction Manager”](#)
- [Section 9.6.3.2, “Configure Logging for the Transaction Subsystem”](#)

[Report a bug](#)

9.6.2. Transactional Datasource Configuration

9.6.2.1. Configure Your Datasource to Use JTA Transactions

Task Summary

This task shows you how to enable Java Transactions API (JTA) on your datasource.

Task Prerequisites

You must meet the following conditions before continuing with this task:

- Your database or other resource must support JTA. If in doubt, consult the documentation for your database or other resource.
- Create a datasource. Refer to [Section 9.6.2.4, “Create a Non-XA Datasource with the Management Interfaces”](#).
- Stop the JBoss Enterprise Application Platform.
- Have access to edit the configuration files directly, in a text editor.

Procedure 9.1. Task:

1. **Open the configuration file in a text editor.**

Depending on whether you run the JBoss Enterprise Application Platform in a managed domain or standalone server, your configuration file will be in a different location.

- A. **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.

- B. **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"`

4. **Save the configuration file.**

Save the configuration file and exit the text editor.

5. **Start the JBoss Enterprise Application Platform.**

Relaunch the JBoss Enterprise Application Platform 6 server.

Result:

The JBoss Enterprise Application Platform starts, and your datasource is configured to use JTA transactions.

[Report a bug](#)

9.6.2.2. Configure an XA Datasource

Task Prerequisites:

In order to add an XA Datasource, you need to log into the Management Console. See [Section 9.6.2.3, “Log in to the Management Console”](#) for more information.

1. **Add a new datasource.**

Add a new datasource to the JBoss Enterprise Application Platform. Follow the instructions in [Section 9.6.2.4, “Create a Non-XA Datasource with the Management Interfaces”](#), but click the **XA Datasource** tab at the top.

2. **Configure additional properties as appropriate.**

All datasource parameters are listed in [Section 9.6.2.5, “Datasource Parameters”](#).

Result:

Your XA Datasource is configured and ready to use.

[Report a bug](#)

9.6.2.3. Log in to the Management Console

Prerequisites

- JBoss Enterprise Application Platform 6 must be running.

Procedure 9.2. Task

1. Navigate to the Management Console start page

Navigate to the Management Console in your web browser. The default location is <http://localhost:9990/console/>, where port 9990 is predefined as the Management Console socket binding.

2. Log in to the Management Console

Enter the username and password of the account that you created previously to log into the Management Console login screen.

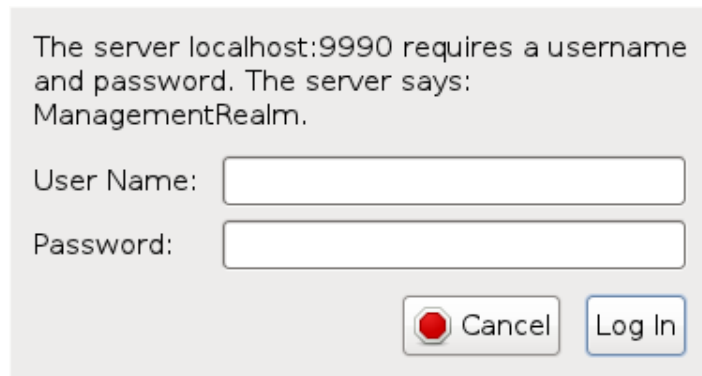
A screenshot of the Management Console login screen. It features a light gray background with a white text area at the top stating: "The server localhost:9990 requires a username and password. The server says: ManagementRealm." Below this text are two input fields: "User Name:" followed by a white text box, and "Password:" followed by a white text box. At the bottom right, there are two buttons: a "Cancel" button with a red circular icon and a "Log In" button with a blue border.

Figure 9.1. Log in screen for the Management Console

Result

Once logged in, one of the Management Console landing pages appears:

Managed domain

<http://localhost:9990/console/App.html#server-instances>

Standalone server

<http://localhost:9990/console/App.html#server-overview>

[Report a bug](#)

9.6.2.4. Create a Non-XA Datasource with the Management Interfaces

Task 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 Enterprise Application Platform 6 server must be running.

Oracle Datasources

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.

Procedure 9.3. Task

► A. Management CLI

1. Launch the CLI tool and connect to your server.
2. Run the following command to create a non-XA datasource, configuring the variables as appropriate:

```
data-source add --name=DATASOURCE_NAME --jndi-name=JNDI_NAME --driver-name=DRIVER_NAME --connection-url=CONNECTION_URL
```

3. Enable the datasource:

```
data-source enable --name=DATASOURCE_NAME
```

B. Management Console

1. Login to the Management Console.
2. **Navigate to the Datasources panel in the Management Console**
 - a. **A. Standalone Mode**
Select the **Profile** tab from the top-right of the console.
 - b. **Domain Mode**
 - a. Select the **Profiles** tab from the top-right of the console.
 - b. Select the appropriate profile from the drop-down box in the top left.
 - c. Expand the **Subsystems** menu on the left of the console.
 - b. Select **Connector** → **Datasources** from the menu on the left of the console.

The screenshot shows the JBoss Enterprise Application Platform 6.0 Management Console. The left sidebar contains a navigation tree with the following structure:

- Subsystems
 - Profile: default
 - Core
 - Connector
 - JCA
 - Datasources** (selected)
 - Resource Adapters
 - Container
 - Security
 - Web
 - OSGi
- Server Groups
 - Group Configurations
- General Configuration
 - Interfaces
 - Socket Binding
 - System Properties

The main panel displays the 'Datasources' configuration for the 'default' profile. It includes tabs for 'Datasources' and 'XA Datasources'. The 'JDBC Datasources' section shows 'JDBC datasource configurations.' and a table of 'Available Datasources'.

Name	JNDI	Enabled?
ExampleDS	java:jboss/datasources/ExampleDS	<input checked="" type="checkbox"/>

Below the table, the 'Selection' tab shows configuration details for 'ExampleDS':

- Name: ExampleDS
- JNDI: java:jboss/datasources/ExampleDS
- Is enabled?: true
- Driver: h2
- Share Prepared Statements: false
- Statement Cache Size: 0

Figure 9.2. Datasources panel

3. Create a new datasource

- a. Select the **Add** button at the top of the **Datasources** panel.
- b. Enter the new datasource attributes in the **Create Datasource** wizard and proceed with the **Next** button.
- c. Enter the JDBC driver details in the **Create Datasource** wizard and proceed with the **Next** button.
- d. Enter the connection settings in the **Create Datasource** wizard and select the **Done** button.

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](#)

9.6.2.5. Datasource Parameters

Table 9.1. Datasource parameters common to non-XA and XA datasources

Parameter	Description
jndi-name	The unique JNDI name for the datasource.
pool-name	The name of the management pool for the datasource.
enabled	Whether or not the datasource is enabled.
use-java-context	Whether to bind the datasource to global JNDI.
spy	Enable spy functionality on the JDBC layer. This logs all JDBC traffic to the datasource. The logging-category parameter must also be set to org.jboss.jdbc .
use-ccm	Enable the cached connection manager.
new-connection-sql	A SQL statement which executes when the connection is added to the connection pool.
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-delimiter	The delimiter for URLs in a connection-url for High Availability (HA) clustered databases.
url-selector-strategy-class-name	A class that implements interface org.jboss.jca.adapters.jdbc.URLSelectorStrategy .
security	Contains child elements which are security settings. Refer to Table 9.6, “Security parameters” .
validation	Contains child elements which are validation settings. Refer to Table 9.7, “Validation parameters” .
timeout	Contains child elements which are timeout settings. Refer to Table 9.8, “Timeout parameters” .
statement	Contains child elements which are statement settings. Refer to Table 9.9, “Statement parameters” .

Table 9.2. 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. Refer to Table 9.4, “Pool parameters common to non-XA and XA datasources” .

Table 9.3. XA datasource parameters

Parameter	Description
xa-datasource-property	A property to assign to implementation class XADataSource . Specified by name=value . 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 classloader module which contains the JDBC driver. The accepted format is driverName#majorVersion.minorVersion .
xa-pool	Contains child elements which are pooling settings. Refer to Table 9.4, “Pool parameters common to non-XA and XA datasources” and Table 9.5, “XA pool parameters” .
recovery	Contains child elements which are recovery settings. Refer to Table 9.10, “Recovery parameters” .

Table 9.4. 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. An empty element denotes a true value. The default is false .
use-strict-min	Whether the pool-size is strict. Defaults to false .
flush-strategy	<p>Whether the pool should be flushed in the case of an error. Valid values are:</p> <ul style="list-style-type: none"> ▸ <code>FailingConnectionOnly</code> ▸ <code>IdleConnections</code> ▸ <code>EntirePool</code> <p>The default is FailingConnectionOnly.</p>
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 should account for this behavior.

Table 9.5. XA pool parameters

Parameter	Description
is-same-rm-override	Whether the <code>javax.transaction.xa.XAResource.isSameRM(XAResource)</code> class returns true or false .
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.
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 9.6. 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 plugin to use to reauthenticate physical connections.

Table 9.7. 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. Mutually exclusive to background-validation .
background-validation	Specifies that connections are validated on a background thread, rather than being validated prior to use. Mutually exclusive to validate-on-match .
background-validation-minutes	The amount of time, in minutes, 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 .
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 should be broadcast to all instances of javax.resource.spi.ConnectionEventListener as a connectionErrorOccurred message.

Table 9.8. Timeout parameters

Parameter	Description
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. 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.
xa-resource-timeout	If non-zero, this value is passed to method XAResource.setTransactionTimeout .

Table 9.9. 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 asking for the same statement twice without closing it uses the same underlying prepared statement. The default is false .

Table 9.10. Recovery parameters

Parameter	Description
recover-credential	A username/password pair or security domain to use for recovery.
recover-plugin	An implementation of class org.jboss.jca.core.spi.recoveryRecoveryPlugin class, to be used for recovery.

[Report a bug](#)

9.6.3. Transaction Logging

9.6.3.1. 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 9.6.3.2, “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 9.11. 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</pre> <pre>tsLogger.logger.trace("BasicAction::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</pre> <pre>tsLogger.logger.trace("BasicAction::End() for action-id "+ get_uid());</pre>
Transaction Rollback	<p>When a transaction commits the following code is executed:</p> <pre>com.arjuna.ats.arjuna.coordinator.BasicAction::Abort:1575</pre> <pre>tsLogger.logger.trace("BasicAction::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::doCancellations:349</pre> <pre>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](#)

9.6.3.2. Configure Logging for the Transaction Subsystem

Task Summary

Use this procedure to control the amount of information logged about transactions, independent of other logging settings in the JBoss Enterprise Application Platform. The main procedure shows how to do this in the web-based Management Console. The Management CLI command is given afterward.

Procedure 9.4. Configure the Transaction Logger Using the Management Console

1. **Navigate to the Logging configuration area.**

In the Management Console, click the **Profiles** tab at the top left of the screen. If you use a

managed domain, choose the server profile you wish to configure, from the **Profile** selection box at the top right.

Expand the **Core** menu, and click the **Logging** label.

2. Edit the `com.arjuna` attributes.

Click the **Edit** button in the **Details** section, toward the bottom of the page. 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

- 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](#)

9.6.3.3. Browse and Manage Transactions

The command-line based 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 Enterprise Application Platform 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 9.1. 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 9.2. View All Prepared Transactions

To view all prepared transactions, first refresh the log store (see [Example 9.1, “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 recover. Refer to [Recover a transaction](#) for more details.

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 log supports recovery via the **:recover** CLI command.

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 reply 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:refresh
```



Note: Browsing JTS Transactions

For JTS transactions, if participants are on remote servers, a limited amount of information may be available to the Transaction Manager. In this case, it is recommended that you use the file-based object store, rather than the HornetQ storage mode. This is the default behavior. To use the HornetQ storage mode, you can set the value of the **use-hornetq-store** option to **true**, in the Transaction Manager configuration. Refer to [Section 9.6.2.1, “Configure Your Datasource to Use JTA Transactions”](#) for information on configuring the Transaction Manager.

View Transaction Statistics

You can view statistics about the Transaction Manager and transaction subsystem either via the web-based Management Console or the command-line Management CLI.

In the web-based Management Console, Transaction statistics are available via **Runtime** → **Subsystem Metrics** → **Transactions**. Transaction statistics are available for each server in a managed domain, as well. You can specify the server in the **Server** selection box at the top left.

The following table shows each available statistic, its description, and the CLI command to view the statistic.

Table 9.12. Transaction Subsystem Statistics

Statistic	Description	CLI Command
Total	The total number of transactions processed by the Transaction Manager on this server.	<code>/host=master/server=server-one/subsystem=transactions/:read-attribute(name=number-of-transactions,include-defaults=true)</code>
Committed	The number of committed transactions processed by the Transaction Manager on this server.	<code>/host=master/server=server-one/subsystem=transactions/:read-attribute(name=number-of-committed-transactions,include-defaults=true)</code>
Aborted	The number of aborted transactions processed by the Transaction Manager on this server.	<code>/host=master/server=server-one/subsystem=transactions/:read-attribute(name=number-of-aborted-transactions,include-defaults=true)</code>
Timed Out	The number of timed out transactions processed by the Transaction Manager on this server.	<code>/host=master/server=server-one/subsystem=transactions/:read-attribute(name=number-of-timed-out-transactions,include-defaults=true)</code>
Heuristics	Not available in the Management Console. Number of transactions in a heuristic state.	<code>/host=master/server=server-one/subsystem=transactions/:read-attribute(name=number-of-heuristics,include-defaults=true)</code>
In-Flight Transactions	Not available in the Management Console. Number of transactions which have begun but not yet terminated.	<code>/host=master/server=server-one/subsystem=transactions/:read-attribute(name=number-of-inflight-transactions,include-defaults=true)</code>
Failure Origin - Applications	The number of failed	

Failure Origin - Applications	The number of failed transactions whose failure origin was an application.	<code>/host=master/server=server-one/subsystem=transactions/:read-attribute(name=number-of-application-rollback,include-defaults=true)</code>
Failure Origin - Resources	The number of failed transactions whose failure origin was a resource.	<code>/host=master/server=server-one/subsystem=transactions/:read-attribute(name=number-of-resource-rollback,include-defaults=true)</code>

[Report a bug](#)

9.7. Use JTA Transactions

9.7.1. Transactions JTA Task Overview

Introduction

The following procedures are useful when you need to use transactions in your application.

- ▶ [Section 9.7.2, “Control Transactions”](#)
- ▶ [Section 9.7.3, “Begin a Transaction”](#)
- ▶ [Section 9.7.5, “Commit a Transaction”](#)
- ▶ [Section 9.7.6, “Roll Back a Transaction”](#)
- ▶ [Section 9.7.7, “Handle a Heuristic Outcome in a Transaction”](#)
- ▶ [Section 9.7.8.2, “Configure the Transaction Manager”](#)
- ▶ [Section 9.7.9.1, “Handle Transaction Errors”](#)

[Report a bug](#)

9.7.2. Control Transactions

Introduction

This list of procedures outlines the different ways to control transactions in your applications which use JTA or JTS APIs.

- ▶ [Section 9.7.3, “Begin a Transaction”](#)
- ▶ [Section 9.7.5, “Commit a Transaction”](#)
- ▶ [Section 9.7.6, “Roll Back a Transaction”](#)
- ▶ [Section 9.7.7, “Handle a Heuristic Outcome in a Transaction”](#)

[Report a bug](#)

9.7.3. Begin a Transaction

This procedure shows how to begin a new JTA transaction, or how to participate in a distributed transaction using the Java Transaction Service (JTS) protocol.

Distributed Transactions

A distributed transaction is one where the transaction participants are in separate applications on multiple servers. If a participant joins a transaction that already exists, rather than creating a new transaction context, the two (or more) participants which share the context are participating a distributed transaction. In order to use distributed transactions, you must configure the ORB. Refer to the *ORB Configuration* section of the *Administration and Configuration Guide* for more information on ORB configuration.

1. Get an instance of `UserTransaction`.

You can get the instance using JNDI, injection, or an EJB's `EjbContext`, if the EJB uses bean-managed transactions, by means of a

`@TransactionManagement(TransactionManagementType.BEAN)` annotation.

A. JNDI

```
new InitialContext().lookup("java:comp/UserTransaction")
```

B. Injection

```
@Resource UserTransaction userTransaction;
```

C. `EjbContext`

```
EjbContext.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 is that the container manages all of the transactions. If you have set up the ORB, the container will manage distributed transactions for you.

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 9.9.3, "JTA Transaction Example"](#).

9.7.4. Nest Transactions

Task Summary

Nested transactions are only supported when you use distributed transactions, with the JTS API. In addition, many database vendors do not support nested transactions, so check with your database vendor before you add nested transactions to your application.

The OTS specifications allow for a limited type of nested transaction, where the subtransaction commit protocol is the same as a top-level transaction's. There are two phases, a **prepare** phase and a **commit** or **abort** phase. This type of nested transaction can lead to inconsistent results, such as in a scenario in which a subtransaction coordinator discovers part of the way through committing that a resources cannot commit. The coordinator may not be able to tell the committed resources to abort, and a heuristic outcome occurs. This strict OTS nested transaction is available via the **CostTransactions::SubtransactionAwareResource** interface.

JBoss Enterprise Application Platform's implementation of JTS supports this type of nested transaction. It also supports a type of nested transaction with a multi-phase commit protocol, which avoids the problems that are possible with the strict OTS model. This type of nested transaction is available via the **ArjunaOTS::ArjunaSubtranAwareResource**. It is driven by a two-phase commit protocol whenever a nested transaction commits.

To create a nested transaction, you create a new transaction within a parent transaction. Refer to [Section 9.7.3, "Begin a Transaction"](#) for information on creating a transaction.

The effect of a nested transaction depends on upon the commit/roll back of its enclosing transactions. The effects are recovered if the enclosing transaction aborts, even if the nested transaction has committed.

[Report a bug](#)

9.7.5. Commit a Transaction

This procedure shows how to commit a JTA transaction, whether it is local or distributed, using the Java Transaction Service (JTS). To use distributed transactions, you need to configure the ORB. Refer to the *ORB Configuration* section of the *Administration and Configuration Guide* for more information on ORB configuration.

Task Prerequisites

Before you can commit a transaction, it must have been begun. Refer to [Section 9.7.3, "Begin a Transaction"](#) for information on beginning transactions.

1. **Call `UserTransaction.commit()`.**

When you call method **`UserTransaction.commit()`**, the Transaction Manager attempts to commit the transaction.

```
...
UserTransaction.commit();
catch (Exception ex) {
    ex.printStackTrace();
    System.exit(0);
}
```

2. **If you are using an EJB, you do not need to manually commit.**

If you use an EJB, you do not need to call **`commit()`**, because the container handles the transaction lifecycle.

Result:

Your datasource commits and your transaction ends, or an exception is thrown.



Note

For a full example, see [Section 9.9.3, “JTA Transaction Example”](#).

[Report a bug](#)

9.7.6. Roll Back a Transaction

This procedure shows how to roll back a JTA transaction, whether it is local or distributed, using the Java Transaction Service (JTS). To use distributed transactions, you need to configure the ORB. Refer to the *ORB Configuration* section of the *Administration and Configuration Guide* for more information on ORB configuration.

Task Prerequisites

Before you can roll back a transaction, it must have been begun. For information about beginning a transaction, see [Section 9.7.3, “Begin a Transaction”](#).

1. **Call `UserTransaction.rollback()`.**

When you call method `UserTransaction.rollback()`, the Transaction Manager attempts to roll back the transaction.

```
...
UserTransaction.rollback();
catch (Exception ex) {
    ex.printStackTrace();
    System.exit(0);
}
```

2. **If you are using an EJB, you do not need to manually roll back.**

If you use an EJB, you do not need to call `rollback()`, because the container handles the transaction lifecycle.

Result:

Your transaction is rolled back by the Transaction Manager.



Note

For a full example, see [Section 9.9.3, “JTA Transaction Example”](#).

[Report a bug](#)

9.7.7. Handle a Heuristic Outcome in a Transaction

This procedure shows how to handle a heuristic outcome in a JTA transaction, whether it is local or distributed, using the Java Transaction Service (JTS). To use distributed transactions, you need to configure the ORB. Refer to the *ORB Configuration* section of the *Administration and Configuration Guide* for more information on ORB configuration.

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 9.4.4 “About Heuristic Outcomes”](#) for more information about heuristic transaction outcomes.

Procedure 9.5. 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 the JBoss Enterprise Application Platform, or the JBoss Enterprise Application Platform 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.

**Heuristic transactions are not recovered**

The JBoss Enterprise Application Platform 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 the JBoss Enterprise Application Platform itself, contact Red Hat Global Support Services.

4. In a test environment, delete the logs and restart the JBoss Enterprise Application Platform.

In a test environment, or if you do not care about the integrity of the data, deleting the transaction logs and restarting the JBoss Enterprise Application Platform 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.

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](#)

9.7.8. Transaction Timeouts**9.7.8.1. 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](#)

9.7.8.2. 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 Enterprise Application Platform 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 **Runtime** tab from the list in the upper left side of the Management Console screen. If you use a managed domain, you have the choice of several profiles. Choose the correct one from the **Profile** selection box at the upper right of the Profiles screen. 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** header to expand them. Click the **Edit** button 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.

Table 9.13. 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.	<code>/profile=default/subsystem=transactions/:write-attribute(name=enable-tsm-status,value=false)</code>
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>
Path	The relative or absolute filesystem path where the transaction manager core stores data. By default the value is a path relative to the value of the relative-to attribute.	<code>/profile=default/subsystem=transactions/:write-attribute(name=path,value=var)</code>
Relative To	References a global path configuration in the domain model. The default value is the data directory for JBoss Enterprise Application Platform 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 path TM attribute is relative to this path. Use an empty string to disable the default behavior and force the value of the path attribute to be treated as an absolute path.	<code>/profile=default/subsystem=transactions/:write-attribute(name=relative-to,value=jboss.server.data.dir)</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>
Object Store Path Relative To	References a global path configuration in the domain model. The default value is the data directory for JBoss Enterprise Application Platform 6, which is the value of the property jboss.server.data.dir ,	<code>/profile=default/subsystem=transactions/:write-attribute(name=object-store-relative-to,value=jboss.server.data.dir)</code>

	<p>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 path TM attribute is relative to this path. Use an empty string to disable the default behavior and force the value of the path attribute to be treated as an absolute path.</p>	
Socket Binding	<p>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.</p>	/profile=default/subsystem=transactions:write-attribute(name=socket-binding,value=txn-recovery-environment)
Status Socket Binding	<p>Specifies the socket binding to use for the Transaction Status manager.</p>	/profile=default/subsystem=transactions:write-attribute(name=status-socket-binding,value=txn-status-manager)
Recovery Listener	<p>Whether or not the Transaction Recovery process should listen on a network socket. Defaults to false.</p>	/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 9.14. 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.	<code>/profile=default/subsystem=transactions/:write-attribute(name=jts,value=false)</code>
node-identifier	The node identifier for the JTS service. This should be unique per JTS service, because the Transaction Manager uses this for recovery.	<code>/profile=default/subsystem=transactions/:write-attribute(name=node-identifier,value=1)</code>
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>	<code>/profile=default/subsystem=transactions/:write-attribute(name=process-id-socket-max-ports,value=10)</code>
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.	<code>/profile=default/subsystem=transactions/:write-attribute(name=process-id-uuid,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. If you enable this, also change the log-store value to hornetq .	<code>/profile=default/subsystem=transactions/:write-attribute(name=use-hornetq-store,value=false)</code>
log-store	Set to default if you use the file-system-based object store, or hornetq if you use the HornetQ journaling storage mechanism. If you set this to	<code>/profile=default/subsystem=transactions/log-store=log-store/:write-attribute(name=type,value=default)</code>

hornetq, also set **use-hornetq-store** to **true**.

[Report a bug](#)

9.7.9. JTA Transaction Error Handling

9.7.9.1. 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.



Handle transaction errors

These guidelines do not apply to heuristic errors. If you experience heuristic errors, refer to [Section 9.7.7, “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 9.7.8.2, “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](#)

9.8. ORB Configuration

9.8.1. 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 Enterprise Application Platform 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](#)

9.8.2. Configure the ORB for JTS Transactions

In a default installation of JBoss Enterprise Application Platform, the ORB is disabled. You can enable the ORB using the command-line Management CLI.



Note

In a managed domain, the JacORB subsystem is available in **full** and **full-ha** profiles only. In a standalone server, it is available when you use the **standalone-full.xml** or **standalone-full-ha.xml** configurations.

Procedure 9.6. Configure the ORB using the Management Console

1. View the profile settings.

Select **Profiles** (managed domain) or **Profile** (standalone server) from the top right 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 at the left, if necessary. Expand the **Container** sub-menu and click **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 9.3. Enable the Security Interceptors

```
/profile=full/subsystem=jacorb/:write-attribute(name=security,value=on)
```

Example 9.4. Enable the ORB for JTS

```
/profile=full/subsystem=jacorb/:write-attribute(name=transactions,value=on)
```

[Report a bug](#)

9.9. Transaction References

9.9.1. JBoss Transactions Errors and Exceptions

For details about exceptions thrown by methods of the **UserTransaction** class, see the *UserTransaction API* specification at <http://download.oracle.com/javase/1.3/api/javax/transaction/UserTransaction.html>.

[Report a bug](#)

9.9.2. JTA Clustering Limitations

JTA transactions cannot be clustered across multiple instances of the JBoss Enterprise Application Platform. For this behavior, use JTS transactions.

To use JTS transactions, you need to configure the ORB: [Section 9.8.2, “Configure the ORB for JTS Transactions”](#).

[Report a bug](#)

9.9.3. 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 9.5. 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](#)

9.9.4. API Documentation for JBoss Transactions JTA

The API documentation for the Transaction subsystem of the JBoss Enterprise Application Platform is available at the following location:

- UserTransaction - <http://download.oracle.com/javaee/1.3/api/javax/transaction/UserTransaction.html>

If you use JBoss Development Studio to develop your applications, the APIs are included in the **Help** menu.

[Report a bug](#)

Chapter 10. Hibernate

10.1. 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 10.2.2, “Hibernate EntityManager”](#) and the [Section 10.2.1, “About JPA”](#).

[Report a bug](#)

10.2. Java Persistence API (JPA)

10.2.1. 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 the JBoss Enterprise Application Platform with a complete Java Persistence solution.

JBoss Enterprise Application Platform 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 Enterprise Application Platform 6, refer to the **bean-validation**, **greeter**, and **kitchensink** quickstarts: [Section 1.5.2.1, “Access the Java EE Quickstart Examples”](#).

[Report a bug](#)

10.2.2. Hibernate EntityManager

Hibernate EntityManager implements the programming interfaces and life-cycle rules defined by the [JPA 2.0 specification](#). It provides the JBoss Enterprise Application Platform with a complete Java Persistence solution.

For more information about Java Persistence or Hibernate, refer to the [Section 10.2.1, “About JPA”](#) and [Section 10.1, “About Hibernate Core”](#).

[Report a bug](#)

10.2.3. Getting Started

10.2.3.1. Create a JPA project in JBoss Developer Studio

Task Summary


This example covers the steps required to create a JPA project in JBoss Developer Studio.

Procedure 10.1. Task

1. In the JBoss Developer Studio window, click **File** → **New** → **JPA Project**.
2. In the project dialog, type the name of the project.

JPA Project

Configure JPA project settings.



Project name:

Project location

☒ Use default location

Location:

Target runtime

JPA version

Configuration

The default configuration provides a good starting point. Additional facets can later be installed to add new functionality to the project.

EAR membership


☐ Add project to an EAR

EAR project name:

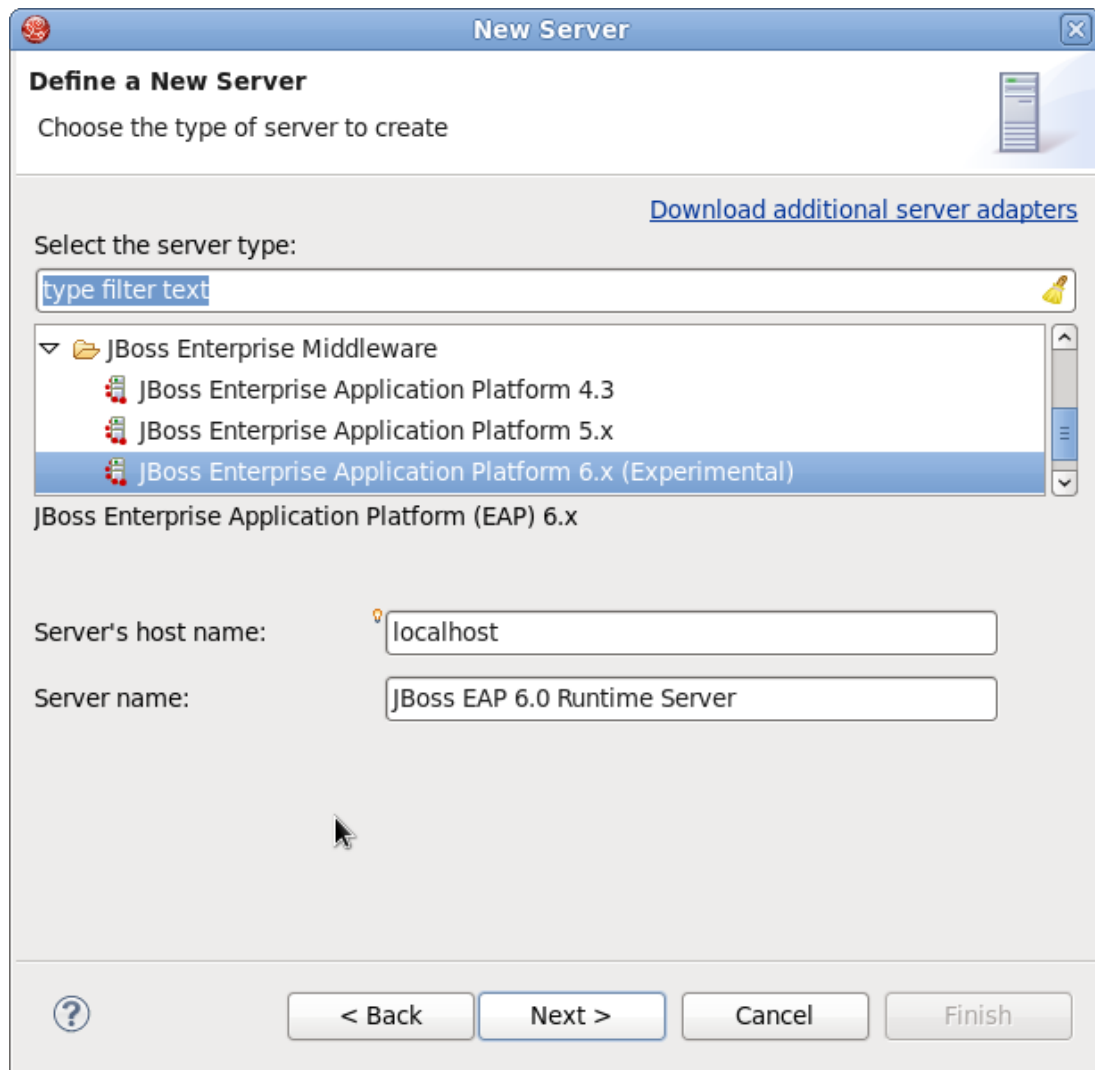
Working sets

☐ Add project to working sets

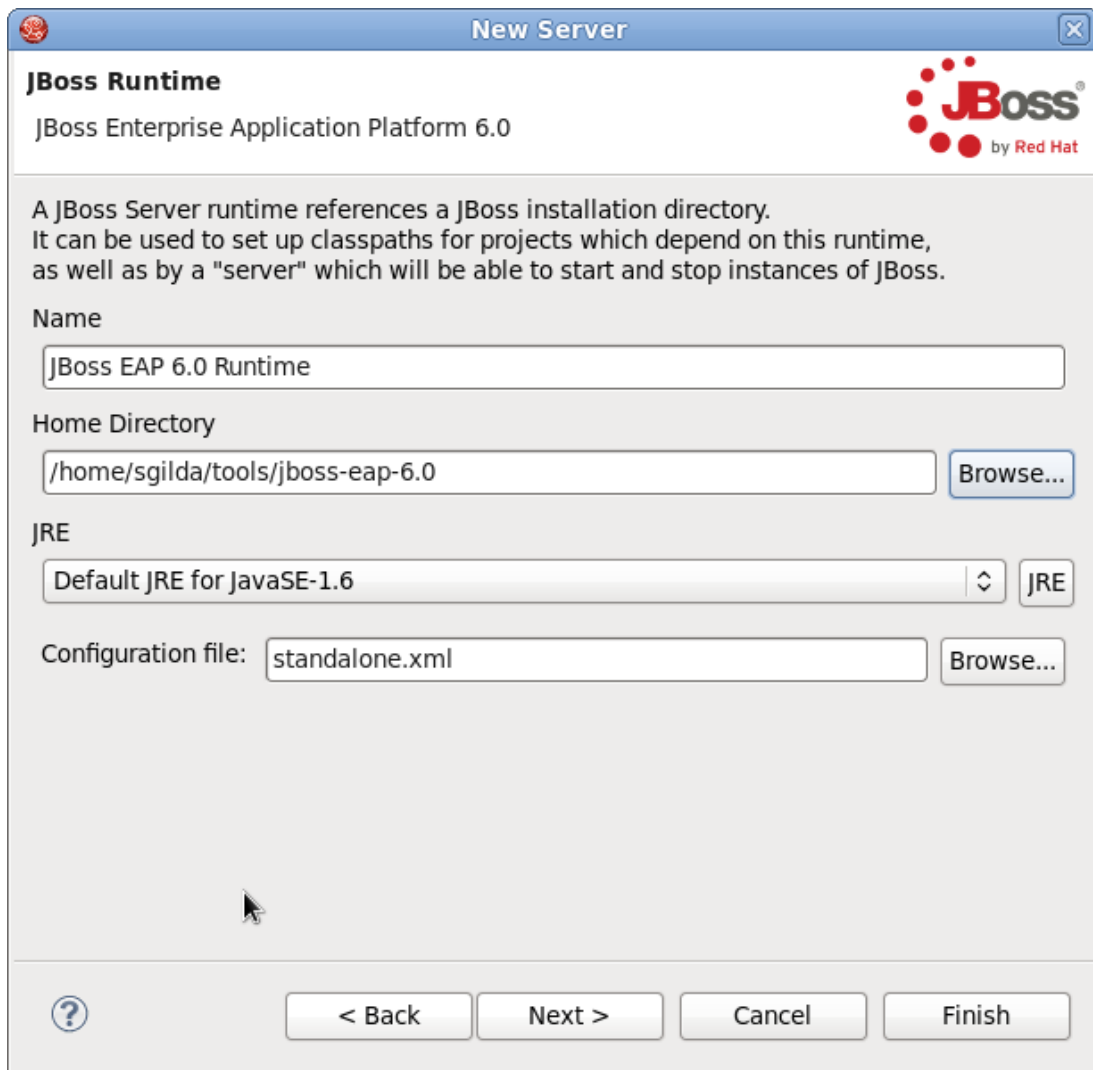
Working sets:



3. Select a Target runtime from the dropdown box.
4.
 - a. If no Target runtime is available, click **Target Runtime**.
 - b. Find the JBoss Community Folder in the list.
 - c. Select JBoss Enterprise Application Platform 6.x Runtime



- d. Click **Next**.
- e. In the Home Directory field, click **Browse** to set the JBoss EAP source folder as the Home Directory.



f. Click **Finish**.

5. Click **Next**.
6. Leave the source folders on build path window as default, and click **Next**.
7. In the Platform dropdown, ensure Hibernate (JPA 2.x) is selected.
8. Click **Finish**.
9. If prompted, choose whether you wish to open the JPA perspective window.

[Report a bug](#)

10.2.3.2. Create the Persistence Settings File in JBoss Developer Studio

Prerequisites

- [Section 1.4.1.4, “Start JBoss Developer Studio”](#)

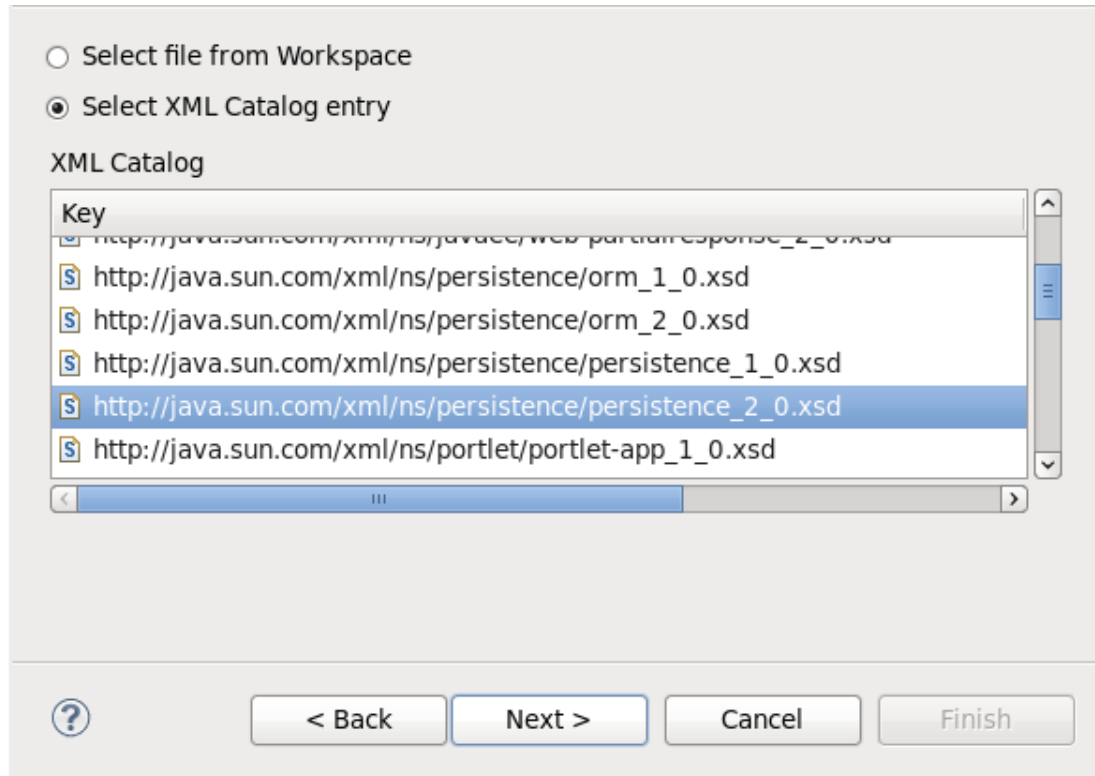
Task Summary

This topic covers the process for creating the `persistence.xml` file in a Java project using the JBoss Developer Studio.

Procedure 10.2. Task

1. Open an EJB 3.x project in the 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 10.2.3.3, "Example Persistence Settings File"](#)

[Report a bug](#)

10.2.3.3. Example Persistence Settings File

Example 10.1. 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"/>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    </properties>
  </persistence-unit>
</persistence>
```

[Report a bug](#)**10.2.3.4. Create the Hibernate Configuration File in JBoss Developer Studio****Prerequisites**

- [Section 1.4.1.4 “Start JBoss Developer Studio”](#)

Task Summary

This topic covers the process for creating the **hibernate.cfg.xml** file in a Java project using the JBoss Developer Studio.

Procedure 10.3. Task

1. Open a Java project in the 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 10.2.3.5, “Example Hibernate Configuration File”](#).

[Report a bug](#)

10.2.3.5. Example Hibernate Configuration File

Example 10.2. hibernate.cfg.xml

```
<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.provider_class">org.hibernate.cache.NoCacheProvider</property>

    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">true</property>

    <!-- Drop and re-create 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](#)

10.2.4. Configuration

10.2.4.1. Hibernate Configuration Properties

Table 10.1. Properties

Property Name	Description
hibernate.dialect	The classname of a Hibernate org.hibernate.dialect.Dialect . Allows Hibernate to generate SQL optimized for a particular relational database. 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.
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 .
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

LE and `javax.persistence.GenerationType.SEQUENCE`. Default value is `false` to keep backward compatibility.



Important

It is recommended that all new projects that use `@GeneratedValue` also set `hibernate.id.new_generator_mappings=true`. This is because the new generators are more efficient and closer to the JPA 2 specification semantic. However, they are not backward compatible with existing databases (if a sequence or a table is used for id generation).

[Report a bug](#)

10.2.4.2. Hibernate JDBC and Connection Properties

Table 10.2. 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.Connection Provider 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.
hibernate.connection.release_mode	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. Available values are auto (default) on_close after_transaction after_statement .

	This setting only affects Sessions returned from SessionFactory.openSession . For Sessions obtained through SessionFactory.getCurrentSession , the CurrentSessionContext implementation configured for use controls the connection release mode for those Sessions .
hibernate.connection.<propertyName>	Pass the JDBC property <propertyName> to DriverManager.getConnection() .
hibernate.jndi.<propertyName>	Pass the property <propertyName> to the JNDI InitialContextFactory .

[Report a bug](#)

10.2.4.3. Hibernate Cache Properties

Table 10.3. Properties

Property Name	Description
hibernate.cache.provider_class	The classname of a custom CacheProvider .
hibernate.cache.use_minimal_puts	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.
hibernate.cache.use_query_cache	Boolean. Enables the query cache. Individual queries still have to be set cacheable.
hibernate.cache.use_second_level_cache	Boolean. Used to completely disable the second level cache, which is enabled by default for classes that specify a <cache> mapping.
hibernate.cache.query_cache_factory	The classname of a custom QueryCache interface. The default value is the built-in StandardQueryCache .
hibernate.cache.region_prefix	A prefix to use for second-level cache region names.
hibernate.cache.use_structured_entries	Boolean. Forces Hibernate to store data in the second-level cache in a more human-friendly format.
hibernate.cache.default_cache_concurrency_strategy	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](#)

10.2.4.4. Hibernate Transaction Properties

Table 10.4. 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](#)

10.2.4.5. Miscellaneous Hibernate Properties

Table 10.5. Properties

Property Name	Description
<code>hibernate.current_session_context_class</code>	Supply a custom strategy for the scoping of the "current" Session . Values include <code>jta</code> <code>thread</code> <code>managed</code> <code>custom.Class</code> .
<code>hibernate.query.factory_class</code>	Chooses the HQL parser implementation: <code>org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory</code> or <code>org.hibernate.hql.internal.classic.ClassicQueryTranslatorFactory</code> .
<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, <code>hqlLiteral=SQL_LITERAL</code> , <code>hqlFunction=SQLFUNC</code> .
<code>hibernate.hbm2ddl.auto</code>	Automatically validates or exports schema DDL to the database when the SessionFactory is created. With <code>create-drop</code> , the database schema will be dropped when the SessionFactory is closed explicitly. Property value options are <code>validate</code> <code>update</code> <code>create</code> <code>create-drop</code>
<code>hibernate.hbm2ddl.import_files</code>	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 <code>/humans.sql,/dogs.sql</code> . 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 <code>hibernate.hbm2ddl.auto</code> is set to <code>create</code> or <code>create-drop</code>).
<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).
<code>hibernate.bytecode.use_reflection_optimizer</code>	Boolean. This is a system-level property, which cannot be set in the <code>hibernate.cfg.xml</code> 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](#)

10.2.4.6. 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 10.6. SQL Dialects (hibernate.dialect)

RDBMS	Dialect
DB2	<code>org.hibernate.dialect.DB2Dialect</code>
DB2 AS/400	<code>org.hibernate.dialect.DB2400Dialect</code>
DB2 OS390	<code>org.hibernate.dialect.DB2390Dialect</code>
PostgreSQL	<code>org.hibernate.dialect.PostgreSQLDialect</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>
Sybase	<code>org.hibernate.dialect.SybaseASE15Dialect</code>
Sybase Anywhere	<code>org.hibernate.dialect.SybaseAnywhereDialect</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>
SAP DB	<code>org.hibernate.dialect.SAPDBDialect</code>
Informix	<code>org.hibernate.dialect.InformixDialect</code>
HypersonicSQL	<code>org.hibernate.dialect.HSQLDialect</code>
H2 Database	<code>org.hibernate.dialect.H2Dialect</code>
Ingres	<code>org.hibernate.dialect.IngresDialect</code>
Progress	<code>org.hibernate.dialect.ProgressDialect</code>
Mckoi SQL	<code>org.hibernate.dialect.MckoiDialect</code>
Interbase	<code>org.hibernate.dialect.InterbaseDialect</code>
Pointbase	<code>org.hibernate.dialect.PointbaseDialect</code>
FrontBase	<code>org.hibernate.dialect.FrontbaseDialect</code>
Firebird	<code>org.hibernate.dialect.FirebirdDialect</code>

10.2.5. Second-Level Caches

10.2.5.1. 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 Enterprise Application Platform 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](#)

10.2.5.2. Configure a Second Level Cache for Hibernate

Task Summary

This topic covers the configuration requirements for enabling Infinispan to act as the second level cache for Hibernate.

Procedure 10.4. Task

1. Create the `hibernate.cfg.xml` file

Create the `hibernate.cfg.xml` in the deployment's classpath. For specifics, refer to [Section 10.2.3.4, "Create the Hibernate Configuration File in 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:

A. 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>
```

B. 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](#)

10.3. Hibernate Annotations

10.3.1. Hibernate Annotations

Table 10.7. 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 @manyToAny metadata.
AnyMedaDefs	Defines @Any and @ManyToAny set of metadata. Can be defined at the entity level or the package level.
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.
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	Mark an Entity or a Collection as immutable. No annotation means the element is mutable.

	<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 <code>HibernateException</code> 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.
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.
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.
WhereJoinTable	Where clause to add to the collection join table. The clause is written in SQL.

[Report a bug](#)

10.4. Hibernate Query Language

10.4.1. 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](#)

10.4.2. 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 10.8. HQL Statements

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](#)

10.4.3. 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 delegate 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 10.3. Example 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](#)

10.4.4. 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](#)

10.4.5. 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 10.4. with-clause Join Example

```
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 simply logical and does not correlate to a physical (SQL) join.

[Report a bug](#)

10.4.6. About Collection Member References

References to collection-valued associations actually refer to the *values* of that collection.

Example 10.5. Collection References Example

```
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](#)

10.4.7. 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 10.9. 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.
INDEX	According to HQL rules, this is valid for both Maps and Lists which specify a javax.persistence.OrderColumn 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 java.util.Map.Entry tuple (the combination of its key and value). ENTRY is only valid as a terminal path and only valid in the select clause.

Example 10.6. Qualified Collection References Example

```
// 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](#)

10.4.8. 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 as well as the application.

[Report a bug](#)

10.4.9. HQL Standardized Functions

The following functions are available in HQL regardless of the underlying database in use.

Table 10.10. 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. See the chapter on data types for more information.
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](#)

10.4.10. 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 10.7. Concatenation Operation Example

```
select 'Mr. ' || c.name.first || ' ' || c.name.last
from Customer c
where c.gender = Gender.MALE
```

[Report a bug](#)

10.4.11. 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 10.8. 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 10.9. 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 10.10. 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](#)

10.4.12. 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 10.11. 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 10.12. 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 a evaluation that a value is within the range of 2 other values. All the operands should have comparable types.

Example 10.13. 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'
```

[Report a bug](#)**10.4.13. About Relational Comparisons**

Comparisons involve one of the comparison operators - =, >, >=, <, <=, <>]. HQL also defines <![CDATA[!= as a comparison operator synonymous with <>. The operands should be of the same type.

Example 10.14. 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 10.15. 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.

10.4.14. 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 10.16. 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](#)

10.4.15. 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 10.17. 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](#)

10.5. Hibernate Services

10.5.1. 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](#)

10.5.2. 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](#)

10.5.3. 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](#)

10.5.4. The ServiceRegistry

10.5.4.1. 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 10.18. Use ServiceRegistryBuilder to create a ServiceRegistry

```
ServiceRegistryBuilder registryBuilder = new ServiceRegistryBuilder(  
    bootstrapServiceRegistry );  
    ServiceRegistry serviceRegistry = registryBuilder.buildServiceRegistry();
```

[Report a bug](#)

10.5.5. Custom Services

10.5.5.1. 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 2 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 10.19. Use ServiceRegistryBuilder to Replace an Existing Service with a Custom Service

```

ServiceRegistryBuilder registryBuilder = new ServiceRegistryBuilder(
bootstrapServiceRegistry );
serviceRegistryBuilder.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;
    }

    @Override
    public JdbcEnvironment getJdbcEnvironment() {
        return null;
    }
}

```

[Report a bug](#)**10.5.6. The Bootstrap Registry****10.5.6.1. 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](#)

10.5.6.2. Using BootstrapServiceRegistryBuilder

Example 10.20. 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 Hibernate should use to load application
classes
    .withApplicationClassLoader( anExplicitClassLoaderForApplicationClasses )
    // pass in a class-loader Hibernate should use to load resources
    .withResourceClassLoader( anExplicitClassLoaderForResources )
    // see BootstrapServiceRegistryBuilder for rest of available methods
    ...
    // finally, build the bootstrap registry with all the above options
    .build();
```

[Report a bug](#)

10.5.6.3. BootstrapRegistry Services

`org.hibernate.service.classloading.spi.ClassLoaderService`

Hibernate needs to interact with ClassLoaders. However, the manner in which Hibernate (or any library) should interact with ClassLoaders 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 ClassLoader, 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 others all need to integrate with Hibernate which used to require something, usually the application, to coordinate registering the pieces of each integration needed on behalf of each integrator. The intent of this service is to allow those integrators to be discovered and to have them integrate themselves with Hibernate.

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. `java.util.ServiceLoader` covers the format of this file in detail, but essentially it lists classes by FQN that implement the `org.hibernate.integrator.spi.Integrator` one per line.

[Report a bug](#)

10.5.7. The SessionFactory Registry

10.5.7.1. 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](#)

10.5.7.2. 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](#)

10.5.8. Integrators

10.5.8.1. 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](#)

10.5.8.2. 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 10.21. 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](#)

10.6. Bean Validation

10.6.1. 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 Enterprise Application Platform's implementation of Bean Validation. It is also the reference implementation of the JSR.

JBoss Enterprise Application Platform 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 1.5.2.1, "Access the Java EE Quickstart Examples"](#).

[Report a bug](#)

10.6.2. 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 10.6.1, "About Bean Validation"](#) and [Section 10.6.3.1, "About Validation Constraints"](#).

[Report a bug](#)

10.6.3. Validation Constraints

10.6.3.1. 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 10.6.3.4, "Hibernate Validator Constraints"](#)

For more information, refer to [Section 10.6.2, "Hibernate Validator"](#) and [Section 10.6.1, "About Bean Validation"](#).

[Report a bug](#)

10.6.3.2. Create a Constraint Annotation in the JBoss Developer Studio

Prerequisites

- [Section 1.4.1.4, "Start JBoss Developer Studio"](#)

Task Summary

This task covers the process of creating a constraint annotation in the JBoss Developer Studio, for use within a Java application.

Procedure 10.5. Task

1. Open a Java project in the 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 10.22. Example Data Set

```
package com.example;

public enum CaseMode {
    UPPER,
    LOWER;
}
```

3. Create the Annotation File

Create a new Java class. For more information, refer to [Section 10.6.3.3, “Create a New Java Class in the JBoss Developer Studio”](#).

4. Configure the constraint annotation and click **Save**.

Example 10.23. 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](#)

10.6.3.3. Create a New Java Class in the JBoss Developer Studio

Prerequisites

- [Section 1.4.1.4, “Start JBoss Developer Studio”](#)

Task Summary

This topic covers the process of creating a Java class for an existing Java project, using the JBoss Developer Studio.

Procedure 10.6. Task

1. Right click on the project root folder in the **Project Explorer** panel.
2. Select **New** → **Class**.
3. Configure the following elements:
 - **Package** :
 - **Name** :
4. **Optional: Add an Interface**
 - a. Click **Add . . .**
 - b. Search for the interface name
 - c. Select the correct interface
 - d. Repeat steps 2 and 3 for each required interface
 - e. Click **Add**.
5. Click **Finish** to create the file.

Result

A new Java class has been created within the project, ready for configuration.

[Report a bug](#)

10.6.3.4. Hibernate Validator Constraints

Table 10.11. 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.
@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](#)

10.6.4. Configuration

10.6.4.1. Example Validation Configuration File

Example 10.24. 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.ResourceBundleMessageInterpolator
  </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](#)

10.7. Envers

10.7.1. About Hibernate Envers

Hibernate Envers is an auditing and versioning system, providing the JBoss Enterprise Application Platform 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](#)

10.7.2. About Auditing Persistent Classes

Auditing of persistent classes is done in the JBoss Enterprise Application Platform 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](#)

10.7.3. Auditing Strategies

10.7.3.1. 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 10.7.2, “About Auditing Persistent Classes”](#). To set the

auditing strategy for the application, refer here: [Section 10.7.3.2, “Set the Auditing Strategy”](#).

[Report a bug](#)

10.7.3.2. Set the Auditing Strategy

Task Summary

There are two audit strategies supported by JBoss Enterprise Application Platform 6: the default and validity audit strategies. This task covers the steps required to define the auditing strategy for an application.

Procedure 10.7. Task

- 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 10.25. Set the Default Audit Strategy

```
<property name="org.hibernate.envers.audit_strategy"
value="org.hibernate.envers.strategy.DefaultAuditStrategy"/>
```

Example 10.26. Set the Validity Audit Strategy

```
<property name="org.hibernate.envers.audit_strategy"
value="org.hibernate.envers.strategy.ValidityAuditStrategy"/>
```

[Report a bug](#)

10.7.4. Getting Started with Entity Auditing

10.7.4.1. Add Auditing Support to a JPA Entity

Task Summary

JBoss Enterprise Application Platform 6 uses entity auditing, through [Section 10.7.1, “About Hibernate Envers”](#), to track the historical changes of a persistent class. This topic covers adding auditing support for a JPA entity.

Procedure 10.8. Add Auditing Support to a JPA Entity

1. Configure the available auditing parameters to suit the deployment: [Section 10.7.5.1, “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 10.27. 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 10.28. 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](#)

10.7.5. Configuration

10.7.5.1. Configure Envers Parameters

Task Summary

JBoss Enterprise Application Platform 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 10.9. 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 10.7.5.4, “Envers Configuration Properties”](#).

Example 10.29. Example Envers Parameters

```
<persistence-unit ...>
<provider>org.hibernate.ejb.HibernatePersistence</provider>
<class>...</class>
<properties>
  <property name="hibernate.dialect" ... />
  <!-- other hibernate properties -->

  <property name="hibernate.ejb.event.post-insert"

value="org.hibernate.ejb.event.EJB3PostInsertEventListener,org.hibernate.envers.ev
ent.AuditEventListener" />
  <property name="hibernate.ejb.event.post-update"

value="org.hibernate.ejb.event.EJB3PostUpdateEventListener,org.hibernate.envers.ev
ent.AuditEventListener" />
  <property name="hibernate.ejb.event.post-delete"

value="org.hibernate.ejb.event.EJB3PostDeleteEventListener,org.hibernate.envers.ev
ent.AuditEventListener" />
  <property name="hibernate.ejb.event.pre-collection-update"
    value="org.hibernate.envers.event.AuditEventListener" />
  <property name="hibernate.ejb.event.pre-collection-remove"
    value="org.hibernate.envers.event.AuditEventListener" />
  <property name="hibernate.ejb.event.post-collection-recreate"
    value="org.hibernate.envers.event.AuditEventListener" />

  <property name="org.hibernate.envers.versionsTableSuffix" value="_V" />
  <property name="org.hibernate.envers.revisionFieldName" value="ver_rev" />
  <!-- other envers properties -->
</properties>
</persistence-unit>
```

Result

Auditing has been configured for all JPA entities in the application.

[Report a bug](#)

10.7.5.2. Enable or Disable Auditing at Runtime

Task Summary

This task covers the configuration steps required to enable/disable entity version auditing at runtime.

Procedure 10.10. Task

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](#)

10.7.5.3. Configure Conditional Auditing

Task 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 10.11. Task

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](#)

10.7.5.4. Envers Configuration Properties

Table 10.12. Entity Data Versioning Configuration Parameters

Property Name	Default Value	Description
org.hibernate.envers.audit_table_prefix		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.
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

org.hibernate.envers.default_catalog	null (same as normal tables)	<p>schema will be the same as the schema of the normal tables.</p> <p>The default catalog name that should be used for audit tables. Can be overridden using the <code>@AuditTable(catalog="...")</code> annotation. If not present, the catalog will be the same as the catalog of the normal tables.</p>
org.hibernate.envers.audit_strategy	org.hibernate.envers.strategy.DefaultAuditStrategy	<p>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.</p>
org.hibernate.envers.audit_strategy_validity_end_rev_field_name	REVEND	<p>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.</p>
org.hibernate.envers.audit_strategy_validity_store_revend_timestamp	false	<p>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.</p>
org.hibernate.envers.audit_strategy_validity_revend_timestamp_field_name	REVEND_TIMESTAMP	<p>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.</p>

[Report a bug](#)

10.7.6. Queries

10.7.6.1. 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 10.30. 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 10.31. 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 specified if revisions in which the entity was deleted should 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 10.32. 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 10.33. 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](#)

Chapter 11. JAX-RS Web Services

11.1. 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 Enterprise Application Platform 6's implementation of JAX-RS. It also provides additional features to the specification.

JBoss Enterprise Application Platform 6 is 100% compliant with JSR 311 - JAX-RS.

To get started with JAX-RS and JBoss Enterprise Application Platform 6, refer to the **helloworld-rs**, **jax-rs-client**, and **kitchensink** quickstart: [Section 1.5.2.1, "Access the Java EE Quickstart Examples"](#).

[Report a bug](#)

11.2. 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](#)

11.3. 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 11.1, "About JAX-RS"](#) and [Section 11.2, "About RESTEasy"](#).

[Report a bug](#)

11.4. RESTEasy Defined Annotations

Table 11.1. 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.
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 IIOLImageProvider.
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.
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.

[Report a bug](#)

11.5. RESTEasy Configuration

11.5.1. RESTEasy Configuration Parameters

Table 11.2. Elements

Option Name	Default Value	Description
resteasy.servlet.mapping.prefix	No default	If the url-pattern for the Resteasy servlet-mapping is not <code>/*</code> .
resteasy.scan	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.
resteasy.scan.providers	false	Scan for <code>@Provider</code> 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 <code>@Provider</code> class names you want to register.
resteasy.use.builtin.providers	true	Whether or not to register default, built-in <code>@Provider</code> 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 <code>.xml</code> or <code>.txt</code>) 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 <code>.en</code> or <code>.fr</code>) to a language. Used when the client is unable to use a Accept-Language header to choose a language (i.e. a browser).



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.

11.6. JAX-RS Web Service Security

11.6.1. Enable Role-Based Security for a RESTEasy JAX-RS Web Service

Task 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

Do not activate role-based security if the application uses EJBs. The EJB container will provide the functionality, instead of RESTEasy.

Procedure 11.1. Task

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 11.1. 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](#)

11.6.2. 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 11.2. Task

1. Enable role-based security. For more information, refer to: [Section 11.6.1, “Enable Role-Based Security for a REST Easy 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](#)

11.7. RESTEasy Logging

11.7.1. About JAX-RS Web Service Logging

RESTEasy supports logging via `java.util.logging`, `log4j`, and `slf4j`. The framework is chosen via the following algorithm:

1. If `log4j` is in the application's classpath, `log4j` will be used.
2. If `slf4j` is in the application's classpath, `slf4j` will be used.
3. `java.util.logging` is the default if neither `log4j` or `slf4j` is in the classpath.
4. If the servlet context param `resteasy.logger.type` is set to `java.util.logging`, `log4j`, or `slf4j` will override this default behavior

To configure logging for a JAX-RS application, refer here: [Section 11.7.2, “Configure a Log Category in the Management Console”](#).

[Report a bug](#)

11.7.2. Configure a Log Category in the Management Console

Log Categories define what log messages to capture and which log handlers to send them to.



Important

Currently Log Categories can only be fully configured in the server configuration file. The final release will support full configuration using the management console and command line administration tools.

To add a new Log Handler in the server configuration file:

1. **Halt the server**
Stop the JBoss Enterprise Application Platform 6 server if it is already running.
2. **Open the configuration file in a text editor.**
Depending on whether you run the Enterprise Application Platform as a managed domain or standalone server, the default configuration file location is **`EAP_HOME/domain/configuration/domain.xml`** or **`EAP_HOME/standalone/configuration/standalone.xml`**.
3. **Find the Logging Subsystem node**
Locate the Logging Subsystem configuration node. It will look like this:

```
<subsystem xmlns="urn:jboss:domain:logging:1.1">
</subsystem>
```

The Logging Subsystem configuration will already contain many items such as Log Handlers and Categories.

4. **Add a new logger node**
Add a new **<logger>** node as a child of the Logging Subsystem node. It must have an attribute called **category** which is the class or package name which this logger will receive messages from.

```
<logger category="com.acme.accounts.receivables">

</logger>
```

Optionally the **use-parent-handlers** attribute can also be added. This attribute can be set to **true** or **false**. When set to **true** all messages received by this log category are also processed by the handlers of the root logger. If not specified, this attribute defaults to **true**.

5. Specify log level

Add a **<level>** element with an attribute called **name**. The value of **name** must be the log level that this category will apply to.

```
<logger category="com.acme.accounts.receivables">
  <level name="DEBUG"/>
</logger>
```

6. Optional: Specify log handlers

Add a **<handlers>** element containing a **<handler>** element for each log handler you wish to use to process the log messages from this category.

If no handlers are specified then the log message will not be processed any further unless **use-parent-handler** is set to **true** in which case the handlers of the root-logger are used.

```
<logger category="com.acme.accounts.receivables">
  <level name="DEBUG"/>
  <handlers>
    <handler name="ACCOUNTS-DEBUG-FILE"/>
  </handlers>
</logger>
```

7. Restart the JBoss Enterprise Application Platform 6 server.

[Report a bug](#)

11.7.3. Logging Categories Defined in RESTEasy

Table 11.3. Categories

Category	Function
org.jboss.resteasy.core	Logs all activity by the core RESTEasy implementation.
org.jboss.resteasy.plugins.providers	Logs all activity by RESTEasy entity providers.
org.jboss.resteasy.plugins.server	Logs all activity by the RESTEasy server implementation.
org.jboss.resteasy.specimpl	Logs all activity by JAX-RS implementing classes.
org.jboss.resteasy.mock	Logs all activity by the RESTEasy mock framework.

[Report a bug](#)

11.8. Exception Handling

11.8.1. Create an Exception Mapper

Summary

Exception mappers are custom, application provided components that catch thrown exceptions and write specific HTTP responses.

Example 11.2. 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
    ExceptionMapper<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 programatically through the **ResteasyProviderFactory** class.

[Report a bug](#)

11.8.2. RESTEasy Internally Thrown Exceptions

Table 11.4. 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.
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.
o.j.r.plugins.providers.jaxb.JAXB UnmarshalException	400	The JAXB providers (XML and Jettison) throw this exception on reads. They may be wrapping JAXBExceptions. This class extends ReaderException .
o.j.r.plugins.providers.jaxb.JAXB MarshalException	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.
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](#)

11.9. RESTEasy Interceptors

11.9.1. 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 11.3. 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 11.4. 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 11.5. 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 11.6. 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](#)

11.9.2. 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 11.7. 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](#)

11.9.3. Register an Interceptor

Task Summary

This topic covers how to register a RESTEasy JAX-RS interceptor in an application.

Procedure 11.3. Task

- 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.

[Report a bug](#)

11.9.4. Interceptor Precedence Families

11.9.4.1. 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.

Interceptors that are not associated with a precedence family are invoked after all others. To assign an interceptor to a precedence family, use the `@Precedence` annotation, referred to in [Section 11.4, “RESTEasy Defined Annotations”](#).

[Report a bug](#)

11.9.4.2. 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 11.8. `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 11.9. `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 11.10. `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 11.9.4.1, “About Interceptor Precedence Families”](#).

[Report a bug](#)

11.10. String Based Annotations

11.10.1. Convert String Based `@*Param` Annotations to Objects

JAX-RS `@*Param` annotations, including `@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.

RESEasy 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 11.11. 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 11.12. `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,
DateFormatter.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](#)

11.11. Configure File Extensions

11.11.1. Map File Extensions to Media Types in the `web.xml` File

Task 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 11.4. Task

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 11.13. Example Mapping

```
<context-param>
  <param-name>resteasy.media.type.mappings</param-name>
  <param-value>html : text/html, json : application/json, xml :
application/xml</param-name>
</context-param>
```

[Report a bug](#)

11.11.2. Map File Extensions to Languages in the `web.xml` File

Task 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 11.5. Task

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 11.14. 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](#)

11.11.3. RESTEasy Supported Media Types

Table 11.5. 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](#)

11.12. RESTEasy JavaScript API

11.12.1. 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 11.15. Simple JAX-RS JavaScript API Example

```
@Path("/")
public interface X{
    @GET
    public String Y();
    @PUT
    public void Z(String entity);
}
```

The interface above defines the methods Y and Z, which become properties in the JavaScript API, shown below:

```
var X = {
  Y : function(params){...},
  Z : function(params){...}
};
```

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 11.12.3, "RESTEasy Javascript API Parameters"](#).

[Report a bug](#)

11.12.2. Enable the RESTEasy JavaScript API Servlet

Task Summary

The RESTEasy JavaScript API is not enabled by default. Follow these steps to enable it using the **web.xml** file.

Procedure 11.6. Task

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](#)

11.12.3. RESTEasy Javascript API Parameters

Table 11.6. Parameter Properties

Property	Default Value	Description
\$entity		The entity to send as a PUT, POST request.
\$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](#)

11.12.4. 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 11.16. 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 11.17. 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](#)

11.12.5. REST.Request Class Members

Table 11.7. 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.
setAsync(async)	Controls whether the request should be asynchronous. Defaults to true.
addCookie(name, value)	Sets the given cookie in the current document when executing the request. This will be persistent in the browser.
addQueryParameter(name, value)	Adds a query parameter to the URI query part.
addMatrixParameter(name, value)	Adds a matrix parameter (path parameter) to the last path segment of the request URI.
addHeader(name, value)	Adds a request header.

[Report a bug](#)

11.13. RESTEasy Asynchronous Job Service

11.13.1. 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 11.13.2, "Enable the Asynchronous Job Service"](#). For examples of how the service works, refer to [Section 11.13.3, "Configure Asynchronous Jobs for RESTEasy"](#).

[Report a bug](#)

11.13.2. Enable the Asynchronous Job Service

Procedure 11.7. Task

- 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 11.13.4, "Asynchronous Job Service Configuration Parameters"](#).

[Report a bug](#)

11.13.3. 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 11.18. 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 11.19. 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 11.20. 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](#)

11.13.4. 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 11.8. 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 11.21. 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](#)

11.14. RESTEasy JAXB

11.14.1. Create a JAXB Decorator

Task 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 11.8. Task

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 11.22. 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 11.23. 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](#)

11.15. RESTEasy Atom Support

11.15.1. 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 and fastinfoset. Refer to the javadocs for more information on the API.

[Report a bug](#)

Chapter 12. JAX-WS Web Services

12.1. 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 Services 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 Enterprise Application Platform 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 Enterprise Application Platform 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 Enterprise Application Platform 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 Enterprise Application Platform 6 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](#)

12.2. Configure the webservices Subsystem

Many configuration options are available for the **webservices** subsystem, which controls the behavior of Web Services deployed into JBoss Enterprise Application Platform 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 modify it to modify the subsystem for a different profile on a managed domain.

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 needs. Modification of any of these elements requires a server restart.

Table 12.1. 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.undefiend.host , the requestor'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 Enterprise Application Platform 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 **post-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 12.1. Read an Endpoint Config

```
/profile=default/subsystem=webservices/endpoint-config=Recording-Endpoint-Config:read-resource
```

Example 12.2. 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-binding** attribute, which sets the protocols which trigger the chain to start.

Example 12.3. Read a Handler Chain

```
/profile=default/subsystem=webservices/endpoint-config=Recording-Endpoint-Config/pre-handler-chain=recording-handlers:read-resource
```

Example 12.4. 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 classloader or the classloader for module **org.jboss.as.webservices.server.integration** must be able to load the handler class.

Example 12.5. Read a Handler

```
/profile=default/subsystem=webservices/endpoint-config=Recording-Endpoint-Config/pre-handler-chain=recording-handlers/handler=RecordingHandler:read-resource
```

Example 12.6. 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")
```

Runtime Information About Web Services

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 two informations show the command for a server in a managed domain, then a standalone server.

Example 12.7. View Runtime Information about All Endpoints on A Server In A Managed Domain

This command shows information about all endpoints on the server called **server-one** hosted on physical host **master** in a managed domain.

```
/host=master/server=server-one/deployment="*/subsystem=webservices/endpoint="*":read-resource
```

Example 12.8. View Runtime Information about All Endpoints on A Server In A Standalone Server

This command shows information about all endpoints on a standalone server named **server-one** on a physical host named **master**.

```
/host=master/server=server-one/deployment="*/subsystem=webservices/endpoint="*":read-resource
```

Example 12.9. Example Endpoint Information

The following is example, 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](#)

12.3. JAX-WS Web Service Endpoints

12.3.1. 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 Enterprise Application Platform container.

You can write a WSDL descriptor by hand, or you can use JAX-WS annotations to create it automatically. This is the more normal usage pattern.

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/pfd/jsr181/index.html>.

Development Requirements

A Web Service must fulfill the requirements of the JAX-WS 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 12.10. Example POJO Endpoint

```
@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class JSEBean01
{
    @WebMethod
    public String echo(String input)
    {
        ...
    }
}
```

Example 12.11. 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 12.12. 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 Enterprise Application Platform 6 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](#)

12.3.2. 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 12.5.2, “JAX-WS Common API Reference”](#) and the API documentation bundle in Javadoc format, distributed with JBoss Enterprise Application Platform 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 12.13. 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 12.14. 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 <http://java.sun.com/webservices/jaxb/> 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 12.15. 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](#)

12.4. JAX-WS Web service Clients

12.4.1. 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 two phases:

1. Create the client artifacts.
2. Construct a service stub.
3. Access the endpoint.

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.

Example 12.16. 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 Enterprise Application Platform installation.

Example 12.17. 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
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.java
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.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.class
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.class
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 12.2. Descriptions of Artifacts Created by wsconsume.sh

File	Description
ProfileMgmt.java	Service endpoint interface.
Customer.java	Custom data type.
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 the 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 12.18. 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 at the upper left, then the **Deployments** menu item at the bottom left. Click **Webservices**, and select your deployment to view its 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](#)

12.4.2. 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 12.5.2, “JAX-WS Common API Reference”](#) and the relevant sections of the Javadocs bundle included with JBoss Enterprise Application Platform 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 12.19. 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 12.20. 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 12.21. 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 12.22. 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 12.23. @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 and convert SOAP messages into Java, and vice versa. This is called **marshalling** and **unmarshalling**.

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 12.24. 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 12.25. 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 12.26. 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 12.27. JAX-WS Timeout Configuration

```

public void testConfigureTimeout() throws Exception
{
    //Set timeout until a connection is established

    ((BindingProvider)port).getRequestContext().put("javax.xml.ws.client.connectionTim
out", "6000");

    //Set timeout until the response is received
    ((BindingProvider)
port).getRequestContext().put("javax.xml.ws.client.receiveTimeout", "1000");

    port.echo("testTimeout");
}

```

[Report a bug](#)

12.5. JAX-WS Development Reference

12.5.1. Enable Web Services Addressing (WS-Addressing)

Prerequisites

- Your application must have an existing JAX-WS service and client configuration.

Procedure 12.1. Task

1. Annotate the service endpoint

Add the **@Addressing** annotation to the application's endpoint code.

Example 12.28. @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 12.29. 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](#)

12.5.2. 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 12.30. 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 12.31. 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.

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 12.32. 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 13. Identity Within Applications

13.1. Foundational Concepts

13.1.1. 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*.

For more information on using encryption to secure the JBoss Enterprise Application Platform, refer to the following topics.

[Report a bug](#)

13.1.2. About Security Domains

Security domains are part of the JBoss Enterprise Application Platform 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.

An *authorization* is a security policy, which contains information about actions which are allowed or prohibited. In security terminology, this is often referred to as a role.

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](#)

13.1.3. 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.

[Report a bug](#)

13.1.4. 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.

The JBoss Enterprise Application Platform provides declarative security via security domains.

[Report a bug](#)

13.2. Role-Based Security in Applications

13.2.1. About Application Security

Securing your applications is a multi-faceted and important concern for every application developer. The JBoss Enterprise Application Platform provides all the tools you need to write secure applications, including the following abilities:

- ▶ [Section 13.2.2, “About Authentication”](#)
- ▶ [Section 13.2.3, “About Authorization”](#)
- ▶ [Section 13.2.4, “About Security Auditing”](#)
- ▶ [Section 13.2.5, “About Security Mapping”](#)
- ▶ [Section 13.1.4, “About Declarative Security”](#)
- ▶ [Section 13.4.2.1, “About EJB Method Permissions”](#)
- ▶ [Section 13.4.3.1, “About EJB Security Annotations”](#)

See also [Section 13.2.9, “Use a Security Domain in Your Application”](#).

[Report a bug](#)

13.2.2. 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.

The JBoss Enterprise Application Platform 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 contains 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](#)

13.2.3. 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 granted to principals.

The JBoss Enterprise Application Platform uses a modular system to configure authorization. Each security domain can 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](#)

13.2.4. 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](#)

13.2.5. 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. One example of this is using an X509 certificate for authentication, and then converting the principal from the certificate to a logical name which your application can display.

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](#)

13.2.6. About the Security Extension Architecture

The architecture of the JBoss Enterprise Application Platform's security extensions consists of three parts. These three parts connect your application to your underlying security infrastructure, whether it is LDAP, Kerberos, or another external system.

JAAS

The first part of the infrastructure is the JAAS API. JAAS is a pluggable framework which provides a layer of abstraction between your security infrastructure and your application.

The main implementation in JAAS is `org.jboss.security.plugins.JaasSecurityManager`, which implements the `AuthenticationManager` and `RealmMapping` interfaces.

`JaasSecurityManager` integrates into the EJB and web container layers, based on the `<security-`

domain> element of the corresponding component deployment descriptor.

For more information about JAAS, refer to [Section 13.2.7, “Java Authentication and Authorization Service \(JAAS\)”](#).

The **JaasSecurityManagerService** MBean

The **JaasSecurityManagerService** MBean service manages security managers. Although its name begins with Jaas, the security managers it handles need not use JAAS in their implementation. The name reflects the fact that the default security manager implementation is the **JaasSecurityManager**.

The primary role of the **JaasSecurityManagerService** is to externalize the security manager implementation. You can change the security manager implementation by providing an alternate implementation of the **AuthenticationManager** and **RealmMapping** interfaces.

The second fundamental role of the **JaasSecurityManagerService** is to provide a JNDI **javax.naming.spi.ObjectFactory** implementation to allow for simple code-free management of the binding between the JNDI name and the security manager implementation. To enable security, specify the JNDI name of the security manager implementation via the **<security-domain>** deployment descriptor element.

When you specify a JNDI name, an object-binding needs to already exist. To simplify the setup of the binding between the JNDI name and security manager, the **JaasSecurityManagerService** binds a *next naming system reference*, nominating itself as the JNDI **ObjectFactory** under the name **java:/jaas**. This permits a naming convention of the form **java:/jaas/XYZ** as the value for the **<security-domain>** element, and the security manager instance for the **XYZ** security domain is created as needed, by creating an instance of the class specified by the **SecurityManagerClassName** attribute, using a constructor that takes the name of the security domain.



The **java:/jaas** prefix is not required.

You do not need to include the **java:/jaas** prefix in your deployment descriptor. You may do so, for backward compatibility, but it is ignored.

The **JaasSecurityDomain** MBean

The **org.jboss.security.plugins.JaasSecurityDomain** is an extension of **JaasSecurityManager** which adds the notion of a **KeyStore**, a **KeyManagerFactory**, and a **TrustManagerFactory** for supporting SSL and other cryptographic use cases.

Further information

For more information, and practical examples of the security architecture in action, refer to [Section 13.2.8, “About Java Authentication and Authorization Service \(JAAS\)”](#).

[Report a bug](#)

13.2.7. 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 the JBoss Enterprise Application Platform, JAAS only provides declarative role-based security. For

more information about declarative security, refer to [Section 13.1.4, “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](#)

13.2.8. About Java Authentication and Authorization Service (JAAS)

The security architecture of JBoss Enterprise Application Platform 6 is comprised of the security configuration subsystem, application-specific security configurations which are included in several configuration files within the application, and the JAAS Security Manager, which is implemented as an MBean.

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 13.1. Application-Specific Configuration Files

File	Description
<code>ejb-jar.xml</code>	The deployment descriptor for an Enterprise JavaBean (EJB) application, located in the META-INF directory of the EJB. 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.
<code>web.xml</code>	The deployment descriptor for a Java Enterprise Edition (EE) web application. Use the web.xml to declare the security domain the application uses for authentication and authorization, as well as resource and transport constraints for the application, such as limiting which types of HTTP requests 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.
<code>jboss-ejb3.xml</code>	Contains JBoss-specific extensions to the ejb-jar.xml descriptor.
<code>jboss-web.xml</code>	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`.

The JAAS Security Manager MBean

The *Java Authentication and Authorization Service (JAAS)* is a framework for user-level security in Java applications, using pluggable authentication modules (PAM). It is integrated into the Java Runtime Environment (JRE). In JBoss Enterprise Application Platform, the container-side component is the `org.jboss.security.plugins.JaasSecurityManager` MBean. It provides the default implementations of the `AuthenticationManager` and `RealmMapping` interfaces.

The `JaasSecurityManager` MBean integrates into the EJB and web container layers based on the security domain specified in the EJB or web deployment descriptor files in the application. When an application deploys, the container associates the security domain specified in the deployment descriptor with the security manager instance of the container. The security manager enforces the configuration of the security domain as configured on the server group or standalone server.

Flow of Interaction between the Client and the Container with JAAS

The `JaasSecurityManager` uses the JAAS packages to implement the `AuthenticationManager` and `RealmMapping` interface behavior. In particular, its behavior derives from the execution of the login module instances that are configured in the security domain to which the `JaasSecurityManager` has been assigned. The login modules implement the security domain's principal authentication and role-mapping behavior. You can use the `JaasSecurityManager` across different security domains by plugging in different login module configurations for the domains.

To illustrate how the `JaasSecurityManager` uses the JAAS authentication process, the following steps outline a client invocation of method which implements method `EJBHome`. The EJB has already been deployed in the server and its `EJBHome` interface methods have been secured using `<method-permission>` elements in the `ejb-jar.xml` descriptor. It uses the `jwdomain` security domain, which is specified in the `<security-domain>` element of the `jboss-ejb3.xml` file. The image below shows the steps, which are explained afterward.

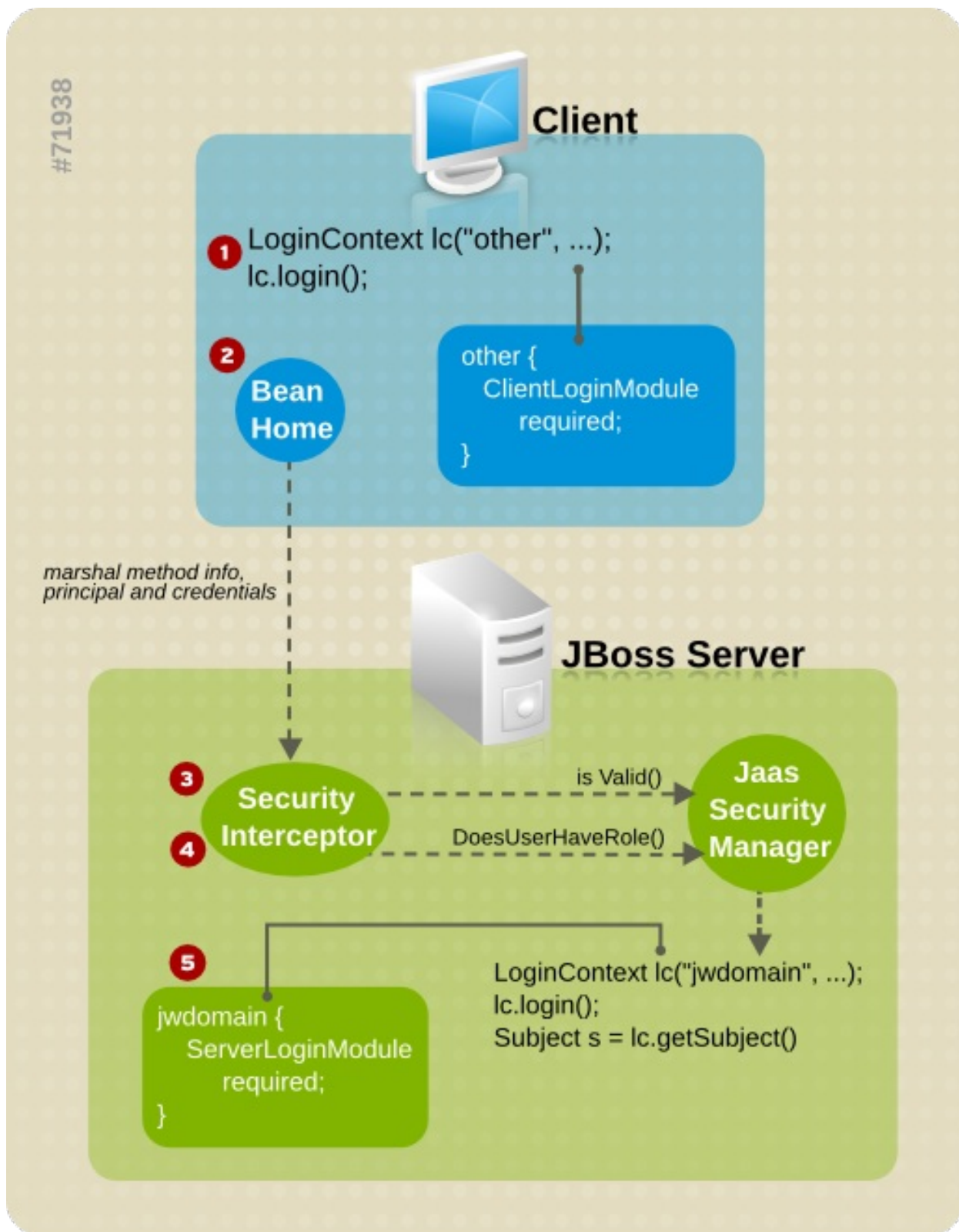


Figure 13.1. Steps of a Secured EJB Method Invocation

1. The client performs a JAAS login to establish the principal and credentials for authentication. This is labeled **Client Side Login** in the figure. This could also be performed via JNDI.

To perform a JAAS login, you create a LoginContext instance and pass in the name of the configuration to use. Here, the configuration name is **other**. This one-time login associates the login principal and credentials with all subsequent EJB method invocations. The process does not necessarily authenticate the user. The nature of the client-side login depends on the login module configuration that the client uses. In this example, the **other** client-side login configuration entry uses the **ClientLoginModule** login module. This module binds the user name and password

to the EJB invocation layer for later authentication on the server. The identity of the client is not authenticated on the client.

2. The client obtains the **EJBHome** method and invokes it on the server. The invocation includes the method arguments passed by the client, along with the user identity and credentials from the client-side JAAS login.
3. On the server, the security interceptor authenticates the user who invoked the method. This involves another JAAS login.
4. The security domain under determines the choice of login modules. The name of the security domain is passed to the **LoginContext** constructor as the login configuration entry name. The EJB security domain is **jwdomain**. If the JAAS authentication is successful, a JAAS Subject is created. A JAAS subject includes a **PrincipalSet**, which includes the following details:
 - ▶ A **java.security.Principal** instance that corresponds to the client identity from the deployment security environment.
 - ▶ A **java.security.acl.Group** called **Roles**, which contains the role names from the user's application domain. Objects of type **org.jboss.security.SimplePrincipal** objects represent the role names. These roles validate access to EJB methods according to constraints in **ejb-jar.xml** and the **EJBContext.isCallerInRole(String)** method implementation.
 - ▶ An optional **java.security.acl.Group** named **CallerPrincipal**, which contains a single **org.jboss.security.SimplePrincipal** that corresponds to the identity of the application domain's caller. The **CallerPrincipal** group member is the value returned by the **EJBContext.getCallerPrincipal()** method. This mapping allows a **Principal** in the operational security environment to map to a **Principal** known to the application. In the absence of a **CallerPrincipal** mapping, the operational principal is the same as the application domain principal.
5. The server verifies that the user calling the EJB method has the permission to do so. Performing this authorization involves the following steps:
 - ▶ Obtain the names of the roles allowed to access the EJB method from the EJB container. The role names are determined by **ejb-jar.xml** descriptor **<role-name>** elements of all **<method-permission>** elements containing the invoked method.
 - ▶ If no roles have been assigned, or the method is specified in an **exclude-list** element, access to the method is denied. Otherwise, the **doesUserHaveRole** method is invoked on the security manager by the security interceptor to check if the caller has one of the assigned role names. This method iterates through the role names and checks if the authenticated user's **Subject Roles** group contains a **SimplePrincipal** with the assigned role name. Access is allowed if any role name is a member of the **Roles** group. Access is denied if none of the role names are members.
 - ▶ If the EJB uses a custom security proxy, the method invocation is delegated to the proxy. If the security proxy denies access to the caller, it throws a **java.lang.SecurityException**. Otherwise, access to the EJB method is allowed and the method invocation passes to the next container interceptor. The **SecurityProxyInterceptor** handles this check and this interceptor is not shown.
 - ▶ For web connection requests, the web server checks the security constraints defined in **web.xml** that match the requested resource and the accessed HTTP method.
If a constraint exists for the request, the web server calls the **JaasSecurityManager** to perform the principal authentication, which in turn ensures the user roles are associated with that principal object.

[Report a bug](#)

13.2.9. Use a Security Domain in Your Application

Overview

To use a security domain in your application, first you must configure the domain in either the server's configuration file or the application's descriptor file. 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.

Procedure 13.1. Configure Your Application to Use a Security Domain

1. Define the Security Domain

You can define the security domain either in the server's configuration file or 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 Enterprise Application Platform instance is running in a managed domain, this is the **domain/configuration/domain.xml** file. If the JBoss Enterprise Application Platform 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 Enterprise Application Platform 6. The following XML example was copied from the **security** subsystem in the server's configuration file.

```
<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. Configure 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 Enterprise Application Platform 6 Quickstarts bundle, which is available from the Red Hat Customer Portal.

[Report a bug](#)

13.2.10. 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 Enterprise Application Platform container.

Procedure 13.2. 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-role>
    <role-name>eap_admin</role-name>
  </security-role>
</security-constraint>
```

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>
```

[Report a bug](#)

13.2.11. Use A Third-Party Authentication System In Your Application

You can integrate third-party security systems with JBoss Enterprise Application Platform. 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 security 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 Enterprise Application Platform 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 13.1. 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 13.2. 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 13.3. 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>
 */
```

```

    */
    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 (JMEException 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 JMEException
 */
protected String getIdentityHeaderId() throws JMEException {
    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 JMEException
 */
protected String getSessionCookieId() throws JMEException {
    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](#)

13.3. Security Realms

13.3.1. About Security Realms

A *security realm* is a series of mappings between users and passwords, and users and roles. Security realms are a mechanism for adding authentication and authorization to your EJB and Web applications. JBoss Enterprise Application Platform 6 provides two security realms by default:

- **ManagementRealm** stores user, password, and role information for the Management API, which provides the functionality for the Management CLI and web-based Management Console. It provides an authentication system for managing JBoss Enterprise Application Platform itself. You could also use the **ManagementRealm** if your application needed to authenticate with the same business rules you use for the Management API.
- **ApplicationRealm** stores user, password, and role information for Web Applications and EJBs.

Each realm is stored in two files on the filesystem:

- **REALM-users.properties** stores usernames and hashed passwords.
- **REALM-users.roles** stores user-to-role mappings.

The properties files are stored in the **domain/configuration/** and **standalone/configuration/** directories. The files are written simultaneously by the **add-user.sh** or **add-user.bat** command. When you run the command, the first decision you make is which realm to add your new user to.

[Report a bug](#)

13.3.2. 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 **MyDomain** on a domain controller or a standalone server.

```
/host=master/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.

```
/host=master/core-service=management/security-realm=MyDomainRealm/authentication=properties:add(path=myfile.properties)
```

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](#)

13.3.3. Add a User to a Security Realm

1. **Run the `add-user.sh` or `add-user.bat` command.**

Open a command-line interface (CLI). 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](#)

13.4. EJB Application Security

13.4.1. Security Identity

13.4.1.1. About EJB Security Identity

The *security identity*, which is also known as *invocation identity*, refers to the `<security-identity>` tag in the security configuration. It refers to the identity another EJB must use when it invokes methods on components.

The invocation identity can be either the current caller, or it can be a specific role. In the first case, the `<use-caller-identity>` tag is present, and in the second case, the `<run-as>` tag is used.

For information about setting the security identity of an EJB, refer to [Section 13.4.1.2, “Set the Security Identity of an EJB”](#).

[Report a bug](#)

13.4.1.2. Set the Security Identity of an EJB

Example 13.4. 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 13.5. Set the security identity of an EJB to a specific role

To set the security identity to a specific role, use the `<run-as>` or `<role>` 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>
```

**Specifying security identity in servlets**

You can also use the `<run-as>` and `<run-as-principal>` elements inside a servlet element.

See also:

- [Section 13.4.1.1, “About EJB Security Identity”](#)
- [Section 15.2, “EJB Security Parameter Reference”](#)

[Report a bug](#)

13.4.2. EJB Method Permissions**13.4.2.1. About EJB Method Permissions**

EJB provides a `<method-permission>` element declaration. This declaration sets the roles which are allowed to invoke an 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

For examples, see [Section 13.4.2.2, “Use EJB Method Permissions”](#).

[Report a bug](#)

13.4.2.2. 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 13.6. 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 13.7. 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 13.8. 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 13.9. 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 13.10. 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](#)

13.4.3. EJB Security Annotations

13.4.3.1. About EJB Security Annotations

EJBs use security annotations to pass information about security to the deployer. These include:

@DeclareRoles

Declares which roles are available.

@SecurityDomain

Specifies the security domain to use for the EJB. If the EJB is annotated for authorization with **@RolesAllowed**, authorization will only apply if the EJB is annotated with a security domain.

@RolesAllowed, @PermitAll, @DenyAll

Specifies which method permissions are allowed. For information about method permissions, refer to [Section 13.4.2.1, "About EJB Method Permissions"](#).

@RolesAllowed, @PermitAll, @DenyAll

Specifies which method permissions are allowed. For information about method permissions, refer to [Section 13.4.2.1, "About EJB Method Permissions"](#).

@RunAs

Configures the propagated security identity of a component.

For more information, refer to [Section 13.4.3.2, "Use EJB Security Annotations"](#).

[Report a bug](#)

13.4.3.2. 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 13.4.2.2, "Use EJB Method Permissions"](#).

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.

@SecurityDomain

Specifies the security domain to use for the EJB. If the EJB is annotated for authorization with **@RolesAllowed**, authorization will only apply if the EJB is annotated with a security domain.

@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.

@RunAs

Use `@RunAs` to specify a role a method will always be run as.

Example 13.11. 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
    public String GoodbyeAdmin(String msg) {
        return "See you later, " + msg;
    }
}
```

In this code, all roles can access method `welcomeEveryone`. The `GoodBye` method runs as the `tempemployee` role. Only the `admin` role can access method `GoodbyeAdmin`, and any other methods with no security annotation..

[Report a bug](#)

13.4.4. Remote Access to EJBs**13.4.4.1. 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

JBoss Remoting also provides automatic discovery via Multicast or JNDI.

It is used by many of the subsystems within JBoss Enterprise Application Platform, 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 Enterprise Application Platform.

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 applicaiton 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](#)

13.4.4.2. 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](#)

13.4.4.3. 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](#)

13.4.4.4. 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 **unsecure** 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 **unsecure** 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 4777. Refer to the documentation about socket bindings and socket binding groups for more information if you need to change this.

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 Enterprise Application Platform components which use Remoting, such as EJBs, the ORB, and the JMS provider, request secured interfaces by default.



Warning: StartTLS Security Considerations

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 13.2. 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>
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 Enterprise Application Platform. 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 13.3. Connector Attributes

Attribute	Description	CLI Command
Name	The name of the connector, used by JNDI.	<code>/profile=default/subsystem=remoting/connector=remoting-connector/:write-attribute(name=name,value=remoting-connector)</code>
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 Enterprise Application Platform installation.	<code>/profile=default/subsystem=remoting/connector=remoting-connector/:write-attribute(name=security-realm,value=ApplicationRealm)</code>

Table 13.4. 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 13.5. 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.

Attribute	Description	CLI Command
include-mechanisms	Contains a value attribute, which is a space-separated list of SASL mechanisms.	<code>/profile=default/subsystem=remoting/connector=remoting-connector/security=sasl:write-attribute(name=include-mechanisms,value=["DIGEST", "PLAIN", "GSSAPI"])</code>
qop	Contains a value attribute, which is a space-separated list of SASL Quality of protection values, in decreasing order of preference.	

strength	Contains a value attribute, which is a space-separated list of SASL cipher strength values, in decreasing order of preference.	<pre>/subsystem=remoting/connector=remoting-connector/security=sasl:write-attribute(name=strength,value=["medium"])</pre>
reuse-session	Contains a value attribute which is a boolean value. If true, attempt to reuse sessions.	<pre>/profile=default/subsystem=remoting/connector=remoting-connector/security=sasl: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/subsystem=remoting/connector=remoting-connector/security=sasl:write-attribute(name=server-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 mechanisms which are susceptible to simple plain passive attacks are allowed 	<pre>/profile=default/subsystem=remoting/connector=remoting-connector/security=sasl/sasl-policy=policy:add</pre> <pre>/profile=default/subsystem=remoting/connector=remoting-connector/security=sasl/sasl-policy=policy:write-attribute(name=forward-secrecy,value=true)</pre> <pre>/profile=default/subsystem=remoting/connector=remoting-connector/security=sasl/sasl-policy=policy:write-attribute(name=no-active,value=false)</pre> <pre>/profile=default/subsystem=remoting/connector=remoting-connector/security=sasl/sasl-policy=policy:write-attribute(name=no-dictionary,value=true)</pre>

passive attacks are allowed.

A value of **false** permits, and **true** denies.

- **pass-credentials** – whether mechanisms which pass client credentials are allowed.

```
/profile=default/subsystem=remoting/connector=remoting-connector/security=sasl/sasl-policy=policy:write-attribute(name=no-plaintext,value=false)
```

```
/profile=default/subsystem=remoting/connector=remoting-connector/security=sasl/sasl-policy=policy:write-attribute(name=pass-credentials,value=true)
```

properties

Contains one or more **<property>** elements, each with a **name** attribute and an optional **value** attribute.

```
/profile=default/subsystem=remoting/connector=remoting-connector/security=sasl/property=myprop:add(value=1)
```

```
/profile=default/subsystem=remoting/connector=remoting-connector/security=sasl/property=myprop2:add(value=2)
```


Example 13.12. Example Configurations

This example shows the default remoting subsystem that ships with JBoss Enterprise Application Platform 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="htt\
p://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](#)

13.4.4.5. 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 an EJB to a specific username and password which exists in a security realm, 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=appassword
```

- 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](#)

13.4.4.6. 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 **MyDomain** on a domain controller or a standalone server.

```
/host=master/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.

```
/host=master/core-service=management/security-
realm=MyDomainRealm/authentication=properties:add(path=myfile.properties)
```

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](#)

13.4.4.7. Add a User to a Security Realm

1. **Run the `add-user.sh` or `add-user.bat` command.**

Open a command-line interface (CLI). 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](#)

13.4.4.8. 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 firewalls that block the RMI port.

[Report a bug](#)

13.5. JAX-RS Application Security

13.5.1. Enable Role-Based Security for a RESTEasy JAX-RS Web Service

Task 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

Do not activate role-based security if the application uses EJBs. The EJB container will provide the functionality, instead of RESTEasy.

Procedure 13.3. Task

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 13.13. 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](#)

13.5.2. 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 13.4. Task

1. Enable role-based security. For more information, refer to: [Section 13.5.1, “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](#)

13.6. Secure Remote Password Protocol

13.6.1. About Secure Remote Password Protocol (SRP)

The Secure Remote Password (SRP) protocol is an implementation of a public key exchange handshake described in the Internet Standards Working Group Request For Comments 2945 (RFC2945). The RFC2945 abstract states:

This document describes a cryptographically strong network authentication mechanism known as the Secure Remote Password (SRP) protocol. This mechanism is suitable for negotiating secure connections using a user-supplied password, while eliminating the security problems traditionally associated with reusable passwords. This system also performs a secure key exchange in the process of authentication, allowing security layers (privacy and/or integrity protection) to be enabled during the session. Trusted key servers and certificate infrastructures are not required, and clients are not required to store or manage any long-term keys. SRP offers both security and deployment advantages over existing challenge-response techniques, making it an ideal drop-in replacement where secure password authentication is needed.

The complete RFC2945 specification can be obtained from <http://www.rfc-editor.org/rfc.html>. Additional information on the SRP algorithm and its history can be found at <http://srp.stanford.edu/>.

Algorithms like Diffie-Hellman and RSA are known as public key exchange algorithms. The concept of public key algorithms is that you have two keys, one public that is available to everyone, and one that is private and known only to you. When someone wants to send encrypted information to you, they encrypt the information using your public key. Only you are able to decrypt the information using your private key. Contrast this with the more traditional shared password based encryption schemes that require the sender and receiver to know the shared password. Public key algorithms eliminate the need to share passwords.

[Report a bug](#)

13.6.2. Configure Secure Remote Password (SRP) Protocol

To use Secure Remote Password (SRP) Protocol in your application, you first create an MBean which implements the **SRPVerifierStore** interface. Information about the implementation is provided in [The SRPVerifierStore Implementation](#).

Procedure 13.5. Integrate the Existing Password Store

1. Create the hashed password information store.

If your passwords are already stored in an irreversible hashed form, you need to do this on a per-user basis.

You can implement **setUserVerifier(String, VerifierInfo)** as a noOp method, or a method that throws an exception stating that the store is read-only.

2. Create the SRPVerifierStore interface.

Create a custom **SRPVerifierStore** interface implementation that can obtain the **VerifierInfo** from the store you created.

The **verifyUserChallenge(String, Object)** can be used to integrate existing hardware token based schemes like SafeWord or Radius into the SRP algorithm. This interface method is called only when the client SRPLoginModule configuration specifies the hasAuxChallenge option.

3. Create the JNDI MBean.

Create a MBean that exposes the **SRPVerifierStore** interface available to JNDI, and exposes any configurable parameters required.

The default **org.jboss.security.srp.SRPVerifierStoreService** allows you to implement this. You can also implement the MBean using a Java properties file implementation of **SRPVerifierStore**.

The SRPVerifierStore Implementation

The default implementation of the **SRPVerifierStore** interface is not recommended for production systems, because it requires all password hash information to be available as a file of serialized objects.

The **SRPVerifierStore** implementation provides access to the **SRPVerifierStore.VerifierInfo** object for a given username. The **getUserVerifier(String)** method is called by the SRPService at the start of a user SRP session to obtain the parameters needed by the SRP algorithm.

Elements of a VerifierInfo Object

username

The username or user ID used to authenticate

verifier

A one-way hash of the password the user enters as proof of identity. The **org.jboss.security.Util** class includes a **calculateVerifier** method which performs the password hashing algorithm. The output password takes the form **H(salt | H(username | ':' | password))**, where **H** is the SHA secure hash function as defined by RFC2945. The username is converted from a string to a byte[] using UTF-8 encoding.

salt

A random number used to increase the difficulty of a brute force dictionary attack on the verifier password database in the event that the database is compromised. The value should be generated from a cryptographically strong random number algorithm when the user's existing clear-text password is hashed.

The SRP algorithm primitive generator. This can be a well known fixed parameter rather than a per-user setting. The `org.jboss.security.srp.SRPConf` utility class provides several settings for `g`, including a suitable default obtained via `SRPConf.getDefaultParams().g()`.

N

The SRP algorithm safe-prime modulus. This can be a well-known fixed parameter rather than a per-user setting. The `org.jboss.security.srp.SRPConf` utility class provides several settings for `N` including a good default obtained via `SRPConf.getDefaultParams().N()`.

Example 13.14. The SRPVerifierStore Interface

```
package org.jboss.security.srp;

import java.io.IOException;
import java.io.Serializable;
import java.security.KeyException;

public interface SRPVerifierStore
{
    public static class VerifierInfo implements Serializable
    {
        public String username;

        public byte[] salt;
        public byte[] g;
        public byte[] N;
    }

    public VerifierInfo getUserVerifier(String username)
        throws KeyException, IOException;

    public void setUserVerifier(String username, VerifierInfo info)
        throws IOException;

    public void verifyUserChallenge(String username, Object auxChallenge)
        throws SecurityException;
}
```

[Report a bug](#)

13.7. Password Vaults for Sensitive Strings

13.7.1. About Securing Sensitive Strings in Clear-Text Files

Web applications and other deployments often include clear-text files, such as XML deployment descriptors, which include sensitive information such as passwords and other sensitive strings. JBoss Enterprise Application Platform includes a password vault mechanism which enables you to encrypt sensitive strings and store them in an encrypted keystore. The vault mechanism manages decrypting the strings for use with security domains, security realms, or other verification systems. This provides an extra layer of security. The mechanism relies upon tools that are included in all supported Java

Development Kit (JDK) implementations.

[Report a bug](#)

13.7.2. Create a Java Keystore to Store Sensitive Strings

Prerequisites

- The **keytool** command must be available to use. It is provided by the Java Runtime Environment (JRE). Locate the path for the file. In Red Hat Enterprise Linux, it is installed to **/usr/bin/keytool**.

Procedure 13.6. Task

1. **Create a directory to store your keystore and other encrypted information.**

Create a directory to hold your keystore and other important information. The rest of this procedure assumes that the directory is **/home/USER/vault/**.

2. **Determine the parameters to use with keytool.**

Determine the following parameters:

alias

The alias is a unique identifier for the vault or other data stored in the keystore. The alias in the example command at the end of this procedure is **vault**. Aliases are case-insensitive.

keyalg

The algorithm to use for encryption. The default is **DSA**. The example in this procedure uses **RSA**. Check 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. The default size of keys is 1024. It must be between 512 and 1024, and a multiple of 64. The example in this procedure uses **1024**.

keystore

The keystore a database which holds encrypted information and the information about how to decrypt it. If you do not specify a keystore, the default keystore to use is a file called **.keystore** in your home directory. The first time you add data to a keystore, it is created. The example in this procedure uses the **vault.keystore** keystore.

The **keystore** command has many other options. Refer to the documentation for your JRE or your operating system for more details.

3. **Determine the answers to questions the keystore command will ask.**

The **keystore** needs the following information in order to populate the keystore entry:

Keystore password

When you create a keystore, you must set a password. In order to work with the keystore in the future, you need to provide the password. Create a strong password that you will remember. The keystore is only as secure as its password and the security of the file system and operating system where it resides.

Key password (optional)

In addition to the keystore password, you can specify a password for each key it holds. In order to use such a key, the password needs to be given each time it is used. Usually, this facility is not used.

First name (given name) and last name (surname)

This, and the rest of the information in the list, helps to uniquely identify the key and place it into a hierarchy of other keys. It does not necessarily need to be a name at all, but it should be two words, and must be unique to the key. The example in this procedure uses **Accounting Administrator**. In directory terms, this becomes the *common name* of the certificate.

Organizational unit

This is a single word that identifies who uses the certificate. It may be the application or the business unit. The example in this procedure uses **AccountingServices**.

Typically, all keystores used by a group or application use the same organizational unit.

Organization

This is usually a single-word representation of your organization's name. This typically remains the same across all certificates used by an organization. This example uses **MyOrganization**.

City or municipality

Your city.

State or province

Your state or province, or the equivalent for your locality.

Country

The two-letter code for your country.

All of this information together will create a hierarchy for your keystores and certificates, ensuring that they use a consistent naming structure but are unique.

4. Run the **keytool** command, supplying the information that you gathered.

Example 13.15. Example input and output of keytool command

```
$ keytool -genkey -alias vault -keyalg RSA -keysize 1024 -keystore
/home/USER/vault/vault.keystore
Enter keystore password: vault22
Re-enter new password:vault22
What is your first and last name?
[Unknown]: Accounting Administrator
What is the name of your organizational unit?
[Unknown]: AccountingServices
What is the name of your organization?
[Unknown]: MyOrganization
What is the name of your City or Locality?
[Unknown]: Raleigh
What is the name of your State or Province?
[Unknown]: NC
What is the two-letter country code for this unit?
[Unknown]: US
Is CN=Accounting Administrator, OU=AccountingServices, O=MyOrganization,
L=Raleigh, ST=NC, C=US correct?
[no]: yes

Enter key password for <vault>
(RETURN if same as keystore password):
```

Result

A file named **vault.keystore** is created in the **/home/USER/vault/** directory. It stores a single key,

called **vault**, which will be used to store encrypted strings, such as passwords, for the JBoss Enterprise Application Platform.

[Report a bug](#)

13.7.3. Mask the Keystore Password and Initialize the Password Vault

Prerequisites

- ▶ [Section 13.7.2, “Create a Java Keystore to Store Sensitive Strings”](#)
- ▶ The **EAP_HOME/bin/vault.sh** application needs to be accessible via a command-line interface.

1. Run the **vault.sh** command.

Run **EAP_HOME/bin/vault.sh**. Start a new interactive session by typing **0**.

2. Enter the directory where encrypted files will be stored.

This directory should be reasonably secure, but the JBoss Enterprise Application Platform needs to be able to access it. If you followed [Section 13.7.2, “Create a Java Keystore to Store Sensitive Strings”](#), your keystore is in a directory called **vault/** in your home directory. This example uses the directory **/home/USER/vault/**.



Include the trailing slash on the directory name.

Do not forget to include the trailing slash on the directory name. Either use **/** or ****, depending on your operating system.

3. Enter the path to the keystore.

Enter the full path to the keystore file. This example uses **/home/USER/vault/vault.keystore**.

4. Encrypt the keystore password.

The following steps encrypt the keystore password, so that you can use it in configuration files and applications securely.

a. Enter the keystore password.

When prompted, enter the keystore password.

b. Enter a salt value.

Enter an 8-character salt value. The salt value, together with the iteration count (below), are used to create the hash value.

c. Enter the iteration count.

Enter a number for the iteration count.

d. Make a note of the masked password information.

The masked password, the salt, and the iteration count are printed to standard output. Make a note of them in a secure location. An attacker could use them to decrypt the password.

e. Enter the alias of the vault.

When prompted, enter the alias of the vault. If you followed [Section 13.7.2, “Create a Java Keystore to Store Sensitive Strings”](#) to create your vault, the alias is **vault**.

5. Exit the interactive console.

Type **exit** to exit the interactive console.

Result

Your keystore password has been masked for use in configuration files and deployments. In addition, your vault is fully configured and ready to use.

[Report a bug](#)

13.7.4. Configure the JBoss Enterprise Application Platform to Use the Password Vault

Overview

Before you can mask passwords and other sensitive attributes in configuration files, you need to make the JBoss Enterprise Application Platform aware of the password vault which stores and decrypts them. Follow this procedure to enable this functionality.

Prerequisites

- [Section 13.7.2, “Create a Java Keystore to Store Sensitive Strings”](#)
- [Section 13.7.3, “Mask the Keystore Password and Initialize the Password Vault”](#)

Procedure 13.7. Task

1. Determine the correct values for the command.

Determine the values for the following parameters, which are determined by the commands used to create the keystore itself. For information on creating a keystore, refer to the following topics: [Section 13.7.2, “Create a Java Keystore to Store Sensitive Strings”](#) and [Section 13.7.3, “Mask the Keystore Password and Initialize the Password Vault”](#).

Parameter	Description
KEYSTORE_URL	The file system path or URI of the keystore file, usually called something like vault.keystore
KEYSTORE_PASSWORD	The password used to access the keystore. This value should be masked.
KEYSTORE_ALIAS	The name of the keystore.
SALT	The salt used to encrypt and decrypt keystore values.
ITERATION_COUNT	The number of times the encryption algorithm is run.
ENC_FILE_DIR	The path to the directory from which the keystore commands are run. Typically the directory containing the password vault.
host (managed domain only)	The name of the host you are configuring

2. Use the Management CLI to enable the password vault.

Run one of the following commands, depending on whether you use a managed domain or standalone server configuraiton. Substitute the values in the command with the ones from the first step of this procedure.

A. Managed Domain

```
/host=YOUR_HOST/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")])
```

B. Standalone Server

```
/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")])
```

The following is an example of the command with hypothetical values:

```
/core-service=vault:add(vault-options=[("KEYSTORE_URL" =>
"/home/user/vault/vault.keystore"), ("KEYSTORE_PASSWORD" => "MASK-
3y28rCZ1cKR"), ("KEYSTORE_ALIAS" => "vault"), ("SALT" =>
"12438567"), ("ITERATION_COUNT" => "50"), ("ENC_FILE_DIR" =>
"/home/user/vault/"])])
```

Result

The JBoss Enterprise Application Platform is configured to decrypt masked strings using the password vault. To add strings to the vault and use them in your configuration, refer to the following topic:

[Section 13.7.5, “Store and Retrieve Encrypted Sensitive Strings in the Java Keystore”](#).

[Report a bug](#)

13.7.5. Store and Retrieve Encrypted Sensitive Strings in the Java Keystore

Overview

Including passwords and other sensitive strings in plain-text configuration files is insecure. The JBoss Enterprise Application Platform includes the ability to store and mask these sensitive strings in an encrypted keystore, and use masked values in configuration files.

Prerequisites

- ▶ [Section 13.7.2, “Create a Java Keystore to Store Sensitive Strings”](#)
- ▶ [Section 13.7.3, “Mask the Keystore Password and Initialize the Password Vault”](#)
- ▶ [Section 13.7.4, “Configure the JBoss Enterprise Application Platform to Use the Password Vault”](#)
- ▶ The `EAP_HOME/bin/vault.sh` application needs to be accessible via a command-line interface.

Procedure 13.8. Task

1. **Run the `vault.sh` command.**

Run `EAP_HOME/bin/vault.sh`. Start a new interactive session by typing `0`.

2. **Enter the directory where encrypted files will be stored.**

If you followed [Section 13.7.2, “Create a Java Keystore to Store Sensitive Strings”](#), your keystore is in a directory called `vault/` in your home directory. In most cases, it makes sense to store all of your encrypted information in the same place as the key store. This example uses the directory `/home/USER/vault/`.



Note

Do not forget to include the trailing slash on the directory name. Either use `/` or `\`, depending on your operating system.

3. **Enter the path to the keystore.**

Enter the full path to the keystore file. This example uses `/home/USER/vault/vault.keystore`.

4. **Enter the keystore password, vault name, salt, and iteration count.**

When prompted, enter the keystore password, vault name, salt, and iteration count. A handshake is performed.

5. **Select the option to store a password.**

Select option `0` to store a password or other sensitive string.

6. **Enter the value.**

When prompted, enter the value twice. If the values do not match, you are prompted to try again.

7. **Enter the vault block.**

Enter the vault block, which is a container for attributes which pertain to the same resource. An example of an attribute name would be **ds_ExampleDS**. This will form part of the reference to the encrypted string, in your datasource or other service definition.

8. Enter the attribute name.

Enter the name of the attribute you are storing. An example attribute name would be **password**.

Result

A message such as the one below shows that the attribute has been saved.

```
Attribute Value for (ds_ExampleDS, password) saved
```

9. Make note of the information about the encrypted string.

A message prints to standard output, showing the vault block, attribute name, shared key, and advice about using the string in your configuration. Make note of this information in a secure location. Example output is shown below.

```
*****
Vault Block:ds_ExampleDS
Attribute Name:password
Shared
Key:N2NhZDYzOTMtNWE0OS00ZGQ0LWE4MmEtMWN1MDMyNDdmNmI2TEl0RV9CUkVBS3ZhdWx0
Configuration should be done as follows:
VAULT::ds_ExampleDS::password::N2NhZDYzOTMtNWE0OS00ZGQ0LWE4MmEtMWN1MDMyNDdmNmI2TEl0RV9CUkVBS3ZhdWx0
*****
```

10. Use the encrypted string in your configuration.

Use the string from the previous step in your configuration, in place of a plain-text string. A datasource using the encrypted password above is shown below.

```
...
<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::N2NhZDYzOTMtNWE0OS00ZGQ0LWE4MmEtMWN1MDMyNDdmNmI2TEl0RV9CUkVBS3ZhdWx0</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>
...
```

You can use an encrypted string anywhere in your domain or standalone configuration file. After you store your string in the keystore, use the following syntax to replace any clear-text string with an encrypted one.

```
${VAULT::<replaceable>VAULT_BLOCK</replaceable>::<replaceable>ATTRIBUTE_NAME</replaceable>::<replaceable>ENCRYPTED_VALUE</replaceable>
```

Here is a sample real-world value, where the vault block is **ds_ExampleDS** and the attribute is **password**.

```
<password>${VAULT::ds_ExampleDS::password::N2NhZDYzOTMtNWE0OS00ZGQ0LWE4MmEtMW
NlMDMyNDdmNmI2TElORV9CukVBS3ZhdWx0}</password>
```

[Report a bug](#)

13.7.6. Store and Resolve Sensitive Strings In Your Applications

Overview

Configuration elements of the JBoss Enterprise Application Platform support the ability to resolve encrypted strings against values stored in a Java Keystore, via the Security Vault mechanism. You can add support for this feature to your own applications.

First, add the password to the vault. Second, replace the clear-text password with the one stored in the vault. You can use this method to obscure any sensitive string in your application.

Prerequisites

Before performing this procedure, make sure that the directory for storing your vault files exists. It does not matter where you place them, as long as the user who executes JBoss Enterprise Application Platform has permission to read and write the files. This example places the **vault/** directory into the **/home/USER/vault/** directory. The vault itself is a file called **vault.keystore** inside the **vault/** directory.

Example 13.16. Adding the Password String to the Vault

Add the string to the vault using the **EAP_HOME/bin/vault.sh** command. The full series of commands and responses is included in the following screen output. Values entered by the user are emphasized. Some output is removed for formatting. In Microsoft Windows, the name of the command is **vault.bat**. Note that in Microsoft Windows, file paths use the \ character as a directory separator, rather than the / character.

```
[user@host bin]$ ./vault.sh
*****
****   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:/home/user/vault/
Enter Keystore URL:/home/user/vault/vault.keystore
Enter Keystore password: ...
Enter Keystore password again: ...
Values match
Enter 8 character salt:12345678
Enter iteration count as a number (Eg: 44):25

Enter Keystore Alias:vault
Vault is initialized and ready for use
Handshake with Vault complete
Please enter a Digit::  0: Store a password  1: Check whether password exists
2: Exit
0
Task: Store a password
Please enter attribute value: sa
Please enter attribute value again: sa
Values match
Enter Vault Block:DS
Enter Attribute Name:thePass
Attribute Value for (DS, thePass) saved

Please make note of the following:
*****
Vault Block:DS
Attribute Name:thePass
Shared Key:0WY5M2I5NzctYzdk0S00MmZhLWExZGYtNjczM2U5ZGUyOWIxTEl0RV9CUKVBS3ZhdWx0
Configuration should be done as follows:
VAULT::DS::thePass::0WY5M2I5NzctYzdk0S00MmZhLWExZGYtNjczM2U5ZGUyOWIxTEl0RV9CUKVBS
3ZhdWx0
*****

Please enter a Digit::  0: Store a password  1: Check whether password exists
2: Exit
2
```

The string that will be added to the Java code is the last value of the output, the line beginning with **VAULT**.

The following servlet uses the vaulted string instead of a clear-text password. The clear-text version is commented out so that you can see the difference.

Example 13.17. Servlet Using a Vaulted Password

```

package vaulterror.web;

import java.io.IOException;
import java.io.Writer;

import javax.annotation.Resource;
import javax.annotation.sql.DataSourceDefinition;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;

/*@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::0WY5M2I5NzctYzdkOS00MmZhLWExZGYtNjczM2U5ZGUyOWIxTElORV9CUkVB
S3ZhdWx0",
    className = "org.h2.jdbcx.JdbcDataSource",
    url = "jdbc:h2:tcp://localhost/mem:test"
)
@WebServlet(name = "MyTestServlet", urlPatterns = { "/my/" }, loadOnStartup = 1)
public class MyTestServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    @Resource(lookup = "java:jboss/datasources/LoginDS")
    private DataSource ds;

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        Writer writer = resp.getWriter();
        writer.write((ds != null) + "");
    }
}

```

Your servlet is now able to resolve the vaulted string.

[Report a bug](#)

13.8. Java Authorization Contract for Containers (JACC)

13.8.1. 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 Enterprise Application Platform implements support for JACC within the security functionality of the security subsystem.

[Report a bug](#)

13.8.2. 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 **META-INF/** or **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 13.18. Example JACC Method Permissions in an EJB

```

<ejb-jar>
  <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>
</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 13.19. Example Security Domain Declaration in an EJB

```

<security>
  <ejb-name>*</ejb-name>
  <security-domain>myDomain</s:security-domain>
  <run-as-principal>myPrincipal</s:run-as-principal>
</s:security>

```

[Report a bug](#)

13.9. Java Authentication SPI for Containers (JASPI)

13.9.1. About Java Authentication SPI for Containers (JASPI) Security

Java Application 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](#)

13.9.2. 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 13.20. 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 the JBoss Enterprise Application Platform completely before adding the configuration directly to the **EAP_HOME/domain/configuration/domain.xml** or **EAP_HOME/standalone/configuration/standalone.xml**.

[Report a bug](#)

Chapter 14. Single Sign On (SSO)

14.1. About Single Sign On (SSO) for Web Applications

Overview

Single Sign On (SSO) allows authentication to one resource to implicitly authorize access to other resources.

Clustered and Non-Clustered SSO

Non-clustered SSO limits the sharing of authorization 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 virtual 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 authorization of all other associated resources.

If the user logs out of an application, or an application invalidates the session programmatically, all persisted authorization data is removed, and the process starts over.

A session timeout does not invalidate the SSO session if other sessions are still valid.

Limitations of SSO

No propagation across third-party boundaries.

SSO can only be used between applications deployed within JBoss Enterprise Application Platform containers.

Container-managed authentication only.

You must use container-managed authentication elements such as `<login-config>` in your application's `web.xml`.

Requires cookies.

SSO is maintained via browser cookies and URL rewriting is not supported.

Realm and security-domain limitations

Unless the `requireReauthentication` parameter is set to `true`, all web applications configured for the same SSO valve must share the same Realm configuration in `web.xml` and the same security domain.

You can nest the Realm element inside the Host element or the surrounding Engine element, but not inside a `context.xml` element for one of the involved web applications.

The `<security-domain>` configured in the `jboss-web.xml` must be consistent across all web applications.

All security integrations must accept the same credentials (for instance, username and password).

[Report a bug](#)

14.2. 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, to be granted authorization to multiple other applications. Clustered SSO stores the authentication and authorization 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.

A SSO configuration is called a valve. A valve is connected to a security domain, which is configured at the level of the server or server group. Each application which should share the same cached authentication information is configured to use the same valve. This configuration is done in the application's `jboss-web.xml`.

Some common SSO valves supported by the web subsystem of the JBoss Enterprise Application Platform include:

- Apache Tomcat ClusteredSingleSignOn
- Apache Tomcat IDPWebBrowserSSOValve
- SPNEGO-based SSO provided by PicketLink

Depending on the specific type of valve, you may need to do some additional configuration in your security domain, in order for your valve to work properly.

[Report a bug](#)

14.3. Choose the Right SSO Implementation

JBoss Enterprise Application Platform 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. Depending on your organization's needs, a few different SSO solutions are available. The solution you use depends on whether you use web applications, EJB applications, or web services; whether your applications run on the same server, multiple non-clustered servers, or multiple clustered servers; and whether you need to integrate into a desktop-based authentication system or you only need authentication between your applications themselves.

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 in JBoss Enterprise Application Platform.

Non-Clustered and Web Application SSO

If you need to propagate security information among applications which run within the same server group or instance, you can use non-clustered SSO. This only involves configuring the valve in your application's `jboss-web.xml` descriptor.

Clustered Web Application SSO

If you need to propagate security information among applications running in a clustered environment across multiple JBoss Enterprise Application Platform instances, you can use the clustered SSO valve. This is configured in your application's `jboss-web.xml`.

[Report a bug](#)

14.4. 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

- ▶ You need to have a configured security domain which handles authentication and authorization.
- ▶ The **infinispan** subsystem needs to be present. It is present in the **full-ha** profile for a managed domain, or by using the **standalone-full-ha.xml** configuration in a standalone server.
- ▶ The **web cache-container** and SSO cache-container must each be present. The example configuration files already contain the **web** cache-container, and some of the configurations already contain the SSO cache-container as well. Use the following commands to check for and enable the SSO cache container. Note that these commands modify the **full** profile of a managed domain. You can change the commands to use a different profile, or remove the **/profile=full** portion of the command, for a standalone server.

Example 14.1. 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 14.2. 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 14.3. Check for the SSO cache-container

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 cache-container is not present in your configuration.

Example 14.4. Add the SSO cache-container

```
/profile=ha/subsystem=infinispan/cache-container=web/replicated-
cache=sso:add(mode="SYNC", batching=true)
```

- 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 needs to 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.

Differences Between Clustered and Non-Clustered SSO Valves

Clustered SSO allows sharing of authentication between separate hosts, while non-clustered SSO does not. The clustered and non-clustered SSO valves are configured the same way, but the clustered SSO includes the **cacheConfig**, **processExpiresInterval** and **maxEmptyLife** parameters, which control the clustering replication of the persisted data.

Example 14.5. Example Clustered SSO Configuration

Because clustered and non-clustered SSO configurations are so similar, only a clustered configuration is shown. This example uses a security domain called **tomcat**.

```
<jboss-web>
  <security-domain>tomcat</security-domain>
  <valve>
    <class-name>org.jboss.web.tomcat.service.sso.ClusteredSingleSignOn</class-name>
    <param>
      <param-name>maxEmptyLife</param-name>
      <param-value>900</param-value>
    </param>
  </valve>
</jboss-web>
```

Table 14.1. SSO Configuration Options

Option	Description
cookieDomain	The host domain to be used for SSO cookies. The default is /. To allow app1.xyz.com and app2.xyz.com to share SSO cookies, you could set the cookieDomain to xyz.com .
maxEmptyLife	Clustered SSO only. The maximum number of seconds an SSO valve with no active sessions will be usable by a request, before expiring. A positive value allows proper handling of shutdown of a node if it is the only one with active sessions attached to the valve. If maxEmptyLife is set to 0 , the valve terminates at the same time as the local session copies, but backup copies of the sessions, from clustered applications, are available to other cluster nodes. Allowing the valve to live beyond the life of its managed sessions gives the user time to make another request which can then fail over to a different node, where it activates the backup copy of the session. Defaults to 1800 seconds (30 minutes).
processExpiresInterval	Clustered SSO only. The minimum number of seconds between efforts by the valve to find and invalidate SSO instances which have expired the MaxEmptyLife timeout. Defaults to 60 (1 minute).
requiresReauthentication	If true, each request uses cached credentials to reauthenticate to the security realm. If false (the default), a valid SSO cookie is sufficient for the valve to authenticate each new request.

Invalidate a Session

An application can programmatically invalidate a session by invoking method `javax.servlet.http.HttpSession.invalidate()`.

[Report a bug](#)

14.5. 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 your network. After obtaining the ticket, you request a Service Ticket (ST) from an Authentication Service (AS), which is another service running on your 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](#)

14.6. 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](#)

14.7. 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](#)

14.8. 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 Enterprise Application Platform 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 Enterprise Application Platform 6 and earlier versions:

- ▶ Security domains are configured centrally, 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 the security domain, as part of its central configuration. 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 authentication 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 14.2. Login Module Codes and Class Names

Simple Name	Class Name	Purpose
Kerberos	<code>com.sun.security.auth.module.Krb5LoginModule</code>	Kerberos login module
SPNEGO	<code>org.jboss.security.negotiation.spnego.SPNEGOLoginModule</code>	The mechanism which enables your Web applications to authenticate to your Kerberos authentication server.
AdvancedLdap	<code>org.jboss.security.negotiation.AdvancedLdapLoginModule</code>	Used with LDAP servers other than Microsoft Active Directory.
AdvancedAdLdap	<code>org.jboss.security.negotiation.AdvancedADLoginModule</code>	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 14.1. Task

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="UserRoles" 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 contains 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>java:/jaas/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
```

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](#)

Chapter 15. Development Security References

15.1. jboss-web.xml Configuration Reference

Introduction

The **jboss-web.xml** is a file within your deployment's **WEB-INF** or **META-INF** directory. It contains configuration information about features the JBoss Web container adds to the Servlet 3.0 specification. Settings specific to the Servlet 3.0 specification are placed into **web.xml** in the same directory.

The top-level element in the **jboss-web.xml** file is the **<jboss-web>** element.

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 15.1. Common Top-Level Attributes

Attribute	Description
env-entry	A mapping to an env-entry required by the web.xml .
ejb-ref	A mapping to an ejb-ref required by the web.xml .
ejb-local-ref	A mapping to an ejb-local-ref required by the web.xml .
service-ref	A mapping to a service-ref required by the web.xml .
resource-ref	A mapping to a resource-ref required by the web.xml .
resource-env-ref	A mapping to a resource-env-ref required by the web.xml .
message-destination-ref	A mapping to a message-destination-ref required by the web.xml .
persistence-context-ref	A mapping to a persistence-context-ref required by the web.xml .
persistence-unit-ref	A mapping to a persistence-unit-ref required by the web.xml .
post-construct	A mapping to a post-context required by the web.xml .
pre-destroy	A mapping to a pre-destroy required by the web.xml .
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.
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.
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 Enterprise Application Platform 6.
disable-audit	If this empty element is present, web security auditing is disabled. Otherwise, it is enabled. Web security auditing is not part of the Java EE specification. This element is new to JBoss Enterprise Application Platform 6.
disable-cross-context	If false , the application is able to call another application context. Defaults to true .

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 15.2. 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.
multi-part	The element, such as @MultiPart , which represents the multi-part information.

<listener>

Describes a listener. The following table lists the child elements of a **<listener>**.

Table 15.3. Listener Configuration Elements

Attribute	Description
class-name	Name of the class of the listener
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. It contains the same configuration elements as [<listener>](#).

[Report a bug](#)

15.2. EJB Security Parameter Reference

Table 15.4. EJB security parameter elements

Element	Description
<security-identity>	Contains child elements pertaining to the security identity of an EJB.
<use-caller-identity />	Indicates that the EJB uses the same security identity as the caller.
<run-as>	Contains a <role-name> element.
<run-as-principal>	If present, indicates the principal assigned to outgoing calls. If not present, outgoing calls are assigned to a principal named anonymous .
<role-name>	Specifies the role the EJB should run as.
<description>	Describes the role named in <role-name> .

Example 15.1. Security identity examples

This example shows each tag described in [Table 15.4, “EJB security parameter elements”](#). They can also be used inside a `<servlet>`.

```
<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>
```

[Report a bug](#)

Chapter 16. Supplemental References

16.1. Types of Java Archives

The JBoss Enterprise Application Platform 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 16.1.

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.	<p>META-INF/ directory, which contains one or more XML deployment descriptor files.</p> <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 Enterprise Application Platform.	META-INF/ directory containing jboss-service.xml or jboss-beans.xml file.

[Report a bug](#)

Revision History

Revision 0.1-3	Thu Dec 20 2012	Tom Wells
Updated documentation with section title corrections.		
Revision 0.1-2	Thu Dec 20 2012	Tom Wells
Update to fix error in section title.		
Revision 0.1-1	Tue Dec 18 2012	Tom Wells
JBoss Enterprise Application Platform 6.0.1 Development Guide.		
Revision 0.0-16	Tue Dec 11 2012	Tom Wells
Bugzilla fixes, new topics, and structural changes for the JBoss Enterprise Application Platform 6 Development Guide.		
Revision 0.0-15	Mon Dec 03 2012	Tom Wells
Updated build of the JBoss Enterprise Application Platform 6.0.1 Development Guide.		
Revision 0.0-14	Fri Nov 30 2012	Tom Wells
Updated build of the JBoss Enterprise Application Platform 6.0.1 Development Guide.		
Revision 0.0-13	Tue Nov 27 2012	Tom Wells
Bugzilla fixes, new topics, and structural changes for the JBoss Enterprise Application Platform 6 Development Guide.		
Revision 0.0-12	Tue Nov 13 2012	Tom Wells
Bugzilla fixes, new topics, and structural changes for the JBoss Enterprise Application Platform 6 Development Guide.		
Revision 0.0-11	Tue Oct 09 2012	Tom Wells
Release update. Bugzilla fixes, new topics, and structural changes for the JBoss Enterprise Application Platform 6 Development Guide.		
Revision 0.0-10	Wed Oct 03 2012	Tom Wells
Updated documentation. Includes bugzilla issue fixes and updates topics.		
Revision 0.0-9	Fri Sept 28 2012	Tom Wells
Added bugzilla fixes and the updated HQL section to the Development guide.		
Revision 0.0-8	Fri Sept 21 2012	Tom Wells
Bugzilla fixes and updated topics for the JBoss Enterprise Application Platform Development Guide.		
Revision 0.0-7	Fri Sept 21 2012	Tom Wells
Bugzilla fixes and updated topics for the JBoss Enterprise Application Platform Development Guide.		
Revision 0.0-6	Tue Sept 18 2012	Tom Wells
Added the Hibernate Query Language section to the Development Guide.		
Revision 0.0-5	Wed Sept 12 2012	Tom Wells
Bugzilla fixes and updated documentation for the Development guide.		
Revision 0.0-4	Mon Sept 3 2012	Tom Wells
Bug fixes for the August asynchronous update to the Development guide.		
Revision 0.0-3	Fri Aug 31 2012	Tom Wells

August update of the Development guide for JBoss Enterprise Application Platform 6.

Revision 0.0-2	Fri Aug 31 2012	Tom Wells
-----------------------	------------------------	------------------

Updated edition of the Development guide for JBoss Enterprise Application Platform 6.0.0.

Revision 0.0-1	Wed Jun 20 2012	Tom Wells
-----------------------	------------------------	------------------

First edition of the JBoss Enterprise Application Platform 6 Development Guide.