Red Hat JBoss Enterprise Application Platform 7.3

Development Guide

Instructions for developing Jakarta and Java EE applications for Red Hat JBoss Enterprise Application Platform.

Last Updated: 2022-02-01

Red Hat JBoss Enterprise Application Platform 7.3 Development Guide

Instructions for developing Jakarta and Java EE applications for Red Hat JBoss Enterprise Application Platform.

Legal Notice

Copyright © 2022 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, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

Abstract

This document provides instructions and information for quickly developing secure and scalable Jakarta EE applications. You will learn about setting up the development environment, using the Maven repository, and class loading in deployments. The document also has detailed information about: Logging Remote JNDI lookup Clustering in web applications Jakarta Contexts and Dependency Injection Jakarta EE APIs, such as Jakarta Transactions and Jakarta Persistence

Table of Contents

CHAPTER 1. GET STARTED DEVELOPING APPLICATIONS **10**
1.1. ABOUT JAKARTA EE 10
1.1.1. Jakarta EE 8 10
1.2. ABOUT JAVA EE 10
1.2.1. Overview of Java EE Profiles 10
1.2.2. Transition from Java EE to Jakarta EE 11
1.3. SETTING UP THE DEVELOPMENT ENVIRONMENT 11
1.4. CONFIGURE ANNOTATION PROCESSING IN RED HAT CODEREADY STUDIO 11
Enable Annotation Processing for an Individual Project 11
Enable Annotation Processing Globally in Red Hat CodeReady Studio 11
1.5. CONFIGURE THE DEFAULT WELCOME WEB APPLICATION 12
Change the welcome-content File Handler 12
Change the default-web-module 12
Disable the Default Welcome Web Application 12

CHAPTER 2. USING MAVEN WITH JBOSS EAP **14**
2.1. LEARN ABOUT MAVEN 14
2.1.1. About the Maven Repository 14
2.1.2. About the Maven POM File 14
Minimum Requirements of a Maven POM File 14
2.1.3. About the Maven Settings File 15
2.1.4. About Maven Repository Managers 16
Commonly used Maven repository managers 16
2.2. INSTALL MAVEN AND THE JBOSS EAP MAVEN REPOSITORY 16
2.2.1. Download and Install Maven 16
2.2.2. Download the JBoss EAP Maven Repository 17
2.2.2.1. Download the JBoss EAP Maven Repository ZIP File 17
2.2.2.2. Download the JBoss EAP Maven Repository with the Offliner Application 17
2.2.3. Install the JBoss EAP Maven Repository 18
2.2.3.1. Install the JBoss EAP Maven Repository Locally 18
2.2.3.2. Install the JBoss EAP Maven Repository for Use with Apache httpd 19
2.3. USE THE MAVEN REPOSITORY 19
2.3.1. Configure the JBoss EAP Maven Repository 19
Configure the JBoss EAP Maven Repository Using the Maven Settings 19
Configure the JBoss EAP Maven Repository Using the Project POM 21
Determine the URL of the JBoss EAP Repository 23
2.3.2. Configure Maven for Use with Red Hat CodeReady Studio 23
2.3.3. Manage Project Dependencies 25
Supported Maven Artifacts 26
Dependency Management 26
JBoss EAP Jakarta EE Specs BOM 27
JBoss EAP BOMs Available for Application Development 28
JBoss EAP Client BOMs 29

CHAPTER 3. CLASS LOADING AND MODULES **30**
3.1. INTRODUCTION 30
3.1.1. Overview of Class Loading and Modules 30
3.1.2. Class Loading in Deployments 30
3.1.3. Class Loading Precedence 30
3.1.4. jboss-deployment-structure.xml 31
3.2. ADD AN EXPLICIT MODULE DEPENDENCY TO A DEPLOYMENT 31

Prerequisites 31
Add a Dependency Configuration to MANIFEST.MF 32
Add a Dependency Configuration to the jboss-deployment-structure.xml 32
Creating a Jandex Index 34
3.3. GENERATE MANIFEST.MF ENTRIES USING MAVEN 35
Generate a MANIFEST.MF File Containing Module Dependencies 35
3.4. PREVENT A MODULE BEING IMPLICITLY LOADED 36
3.5. EXCLUDE A SUBSYSTEM FROM A DEPLOYMENT 37
3.6. USE THE CLASS LOADER PROGRAMMATICALLY IN A DEPLOYMENT 38
3.6.1. Programmatically Load Classes and Resources in a Deployment 38
3.6.2. Programmatically Iterate Resources in a Deployment 40
3.7. CLASS LOADING AND SUBDEPLOYMENTS 42
3.7.1. Modules and Class Loading in Enterprise Archives 42
3.7.2. Subdeployment Class Loader Isolation 43
3.7.3. Enable Subdeployment Class Loader Isolation Within a EAR 43
3.7.4. Configuring Session Sharing between Subdeployments in Enterprise Archives 44
3.7.4.1. Reference of Shared Session Configuration Options 44
3.8. DEPLOY TAG LIBRARY DESCRIPTORS (TLDS) IN A CUSTOM MODULE 46
Deploy TLDs in a Custom Module 47
3.9. DISPLAY MODULES BY DEPLOYMENT 48
3.10. CLASS LOADING REFERENCE 50
3.10.1. Implicit Module Dependencies 50
3.10.2. Included Modules 58

CHAPTER 4. LOGGING 59
4.1. ABOUT LOGGING 59
4.1.1. Supported Application Logging Frameworks 59
4.2. LOGGING WITH THE JBOSS LOGGING FRAMEWORK 59
4.2.1. About JBoss Logging 59
4.2.2. Add Logging to an Application with JBoss Logging 60
4.3. PER-DEPLOYMENT LOGGING 61
4.3.1. Add Per-deployment Logging to an Application 62
Configuring logging.properties 62
JBoss Log Manager Configuration Options 62
4.4. LOGGING PROFILES 65
4.4.1. Specify a Logging Profile in an Application 65
4.5. INTERNATIONALIZATION AND LOCALIZATION 66
4.5.1. Introduction 66
4.5.1.1. About Internationalization 66
4.5.1.2. About Localization 66
4.5.2. JBoss Logging Tools Internationalization and Localization 66
4.5.3. Creating Internationalized Loggers, Messages and Exceptions 68
4.5.3.1. Create Internationalized Log Messages 68
4.5.3.2. Create and Use Internationalized Messages 70
4.5.3.3. Create Internationalized Exceptions 71
4.5.4. Localizing Internationalized Loggers, Messages and Exceptions 72
4.5.4.1. Generate New Translation Properties Files with Maven 72
4.5.4.2. Translate an Internationalized Logger, Exception, or Message 73
4.5.5. Customizing Internationalized Log Messages 74
4.5.5.1. Add Message IDs and Project Codes to Log Messages 74
4.5.5.2. Specify the Log Level for a Message 75
4.5.5.3. Customize Log Messages with Parameters 76
4.5.5.4. Specify an Exception as the Cause of a Log Message 76

4.5.6. Customizing Internationalized Exceptions 77
4.5.6.1. Add Message IDs and Project Codes to Exception Messages 77
4.5.6.2. Customize Exception Messages with Parameters 79
4.5.6.3. Specify One Exception as the Cause of Another Exception 79
4.5.7. JBoss Logging Tools References 81
4.5.7.1. JBoss Logging Tools Maven Configuration 81
4.5.7.2. Translation Property File Format 82
4.5.7.3. JBoss Logging Tools Annotations Reference 83
4.5.7.4. Project Codes Used in JBoss EAP 83

CHAPTER 5. REMOTE JNDI LOOKUP **87**
5.1. REGISTERING OBJECTS TO JAVA NAMING AND DIRECTORY INTERFACE 87
5.2. CONFIGURING REMOTE JNDI 87
5.3. JNDI INVOCATION OVER HTTP 87
5.3.1. Client-side Implementation 87
5.3.2. Server-side Implementation 88

CHAPTER 6. CLUSTERING IN WEB APPLICATIONS **89**
6.1. SESSION REPLICATION 89
6.1.1. About HTTP Session Replication 89
6.1.2. Enable Session Replication in Your Application 89
Make your Application Distributable 89
Immutable Session Attributes 90
6.2. HTTP SESSION PASSIVATION AND ACTIVATION 91
6.2.1. About HTTP Session Passivation and Activation 91
6.2.2. Configure HTTP Session Passivation in Your Application 91
6.3. PUBLIC API FOR CLUSTERING SERVICES 92
6.4. HA SINGLETON SERVICE 93
HA Singleton ServiceBuilder API 93
HA Singleton Service Election Policies 93
HA Singleton Service Preferences 93
Quorum 93
HA Singleton Service Election Listener 94
Create an HA Singleton Service Application 94
6.5. HA SINGLETON DEPLOYMENTS 97
Defining or choosing a singleton deployment 97
Creating a Singleton Deployment 98
Preferences 99
Define a Quorum 100
Determine the Primary Singleton Service Provider Using the CLI 101
6.6. APACHE MOD_CLUSTER-MANAGER APPLICATION 101
6.6.1. About mod_cluster-manager Application 101
Exploring mod_cluster-manager Application 101
6.7. THE DISTRIBUTABLE-WEB SUBSYSTEM FOR DISTRIBUTABLE WEB SESSION CONFIGURATIONS 103
6.7.1. Storing Web Session Data In a Remote Red Hat Data Grid 104

CHAPTER 7. JAKARTA CONTEXTS AND DEPENDENCY INJECTION **106**
7.1. INTRODUCTION TO JAKARTA CONTEXTS AND DEPENDENCY INJECTION 106
7.1.1. About Jakarta Contexts and Dependency Injection 106
Benefits of Jakarta Contexts and Dependency Injection 106
7.1.2. Relationship Between Weld, Seam 2, and Jakarta Server Faces 106
7.2. USE CONTEXTS AND DEPENDENCY INJECTION TO DEVELOP AN APPLICATION 107
7.2.1. Default Bean Discovery Mode 107
Bean Defining Annotations 108

7.2.2. Exclude Beans From the Scanning Process 108
7.2.3. Use an Injection to Extend an Implementation 110
7.3. AMBIGUOUS OR UNSATISFIED DEPENDENCIES 110
7.3.1. Qualifiers 111
'@Any' 111
7.3.2. Use a Qualifier to Resolve an Ambiguous Injection 112
Resolve an Ambiguous Injection with a Qualifier 112
7.4. MANAGED BEANS 113
7.4.1. Types of Classes That are Beans 113
@Vetoed 113
7.4.2. Use Contexts and Dependency Injection to Inject an Object Into a Bean 114
Inject Objects into Other Objects 114
7.5. CONTEXTS AND SCOPES 115
7.6. NAMED BEANS 116
7.6.1. Use Named Beans 116
Configure Bean Names Using the @Named Annotation 116
7.7. BEAN LIFECYCLE 116
Manage Bean Lifecycles 117
7.7.1. Use a Producer Method 117
7.8. ALTERNATIVE BEANS 118
Declaring Selected Alternatives 119
7.8.1. Override an Injection with an Alternative 119
Override an Injection 119
7.9. STEREOTYPES 120
7.9.1. Use Stereotypes 120
Define and Use Stereotypes 121
7.10. OBSERVER METHODS 121
7.10.1. Fire and Observe Events 121
7.10.2. Transactional Observers 122
7.11. INTERCEPTORS 124
Enabling Interceptors 124
7.11.1. Use Interceptors with Contexts and Dependency Injection 125
Using Interceptors with Contexts and Dependency Injection 125
7.12. DECORATORS 126
7.13. PORTABLE EXTENSIONS 127
7.14. BEAN PROXIES 127
7.15. USE A PROXY IN AN INJECTION 128

CHAPTER 8. JBOSS EAP MBEAN SERVICES 129
8.1. WRITING JBOSS MBEAN SERVICES 129
8.1.1. A Standard MBean Example 129
8.2. DEPLOYING JBOSS MBEAN SERVICES 131

CHAPTER 9. JAKARTA CONCURRENCY 132
9.1. CONTEXT SERVICE 132
9.2. MANAGED THREAD FACTORY 133
9.3. MANAGED EXECUTOR SERVICE 134
9.4. MANAGED SCHEDULED EXECUTOR SERVICE 135

CHAPTER 10. UNDERTOW 137
10.1. INTRODUCTION TO UNDERTOW HANDLER 137
Request Lifecycle 137
Ending the Exchange 138
10.2. USING EXISTING UNDERTOW HANDLERS WITH A DEPLOYMENT 138

Undertow Handler Default Parameter 138
10.3. CREATING CUSTOM HANDLERS 139
Defining Custom Handlers Using the WEB-INF/jboss-web.xml File 139
Defining Custom Handlers in the WEB-INF/undertow-handlers.conf File 140
10.4. DEVELOPING A CUSTOM HTTP MECHANISM 141
Using a Custom HTTP Mechanism 142

CHAPTER 11. JAKARTA TRANSACTIONS 144
11.1. OVERVIEW 144
11.1.1. Overview of Jakarta Transactions 144
11.2. TRANSACTION CONCEPTS 144
11.2.1. About Transactions 144
11.2.2. About ACID Properties for Transactions 144
11.2.3. About the Transaction Coordinator or Transaction Manager 145
11.2.4. About Transaction Participants 145
11.2.5. About Jakarta Transactions 145
11.2.6. About JTS 146
11.2.7. About XML Transaction Service 146
11.2.7.1. Overview of Protocols Used by XTS 146
11.2.7.2. Web Services-Atomic Transaction Process 146
11.2.7.2.1. Atomic Transaction Process 146
11.2.7.2.2. WS-AT Interoperability with Microsoft .NET Clients 147
11.2.7.3. Web Services-Business Activity Process 147
11.2.7.3.1. WS-BA Process 148
11.2.7.4. Transaction Bridging Overview 148
11.2.8. About XA Resources and XA Transactions 148
11.2.9. About XA Recovery 148
11.2.10. Limitations of the XA Recovery Process 149
11.2.11. About the 2-Phase Commit Protocol 150
Phase 1: Prepare 150
Phase 2: Commit 150
11.2.12. About Transaction Timeouts 150
11.2.13. About Distributed Transactions 150
11.2.14. About the ORB Portability API 151
11.3. TRANSACTION OPTIMIZATIONS 151
11.3.1. Overview of Transaction Optimizations 151
11.3.2. About the LRCO Optimization for Single-phase Commit (1PC) 152
Single-phase Commit (1PC) 152
Last Resource Commit Optimization (LRCO) 152
11.3.2.1. Commit Markable Resource 153
Summary 153
Create Tables in Database 153
Enabling Datasource to be Connectable 154
Updating an Existing Resource to Use the New CMR Feature 155
Add a Reference to the Transactions Subsystem 155
11.3.3. About the Presumed-Abort Optimization 155
11.3.4. About the Read-Only Optimization 156
11.4. TRANSACTION OUTCOMES 156
11.4.1. About Transaction Outcomes 156
11.4.2. About Transaction Commit 156
11.4.3. About Transaction Rollback 156
11.4.4. About Heuristic Outcomes 156
Heuristic rollback 157

Heuristic commit 157
Heuristic mixed 157
Heuristic hazard 157
11.4.5. JBoss Transactions Errors and Exceptions 157
11.5. OVERVIEW OF THE TRANSACTION LIFECYCLE 157
11.5.1. Transaction Lifecycle 157
11.6. TRANSACTION SUBSYSTEM CONFIGURATION 158
11.7. TRANSACTIONS USAGE IN PRACTICE 158
11.7.1. Transactions Usage Overview 158
11.7.2. Control Transactions 159
11.7.2.1. Begin a Transaction 159
11.7.2.1.1. Nested Transactions 160
11.7.2.2. Commit a Transaction 160
11.7.2.3. Roll Back a Transaction 161
11.7.3. Handle a Heuristic Outcome in a Transaction 162
11.7.4. Jakarta Transactions Transaction Error Handling 163
11.7.4.1. Handle Transaction Errors 163
11.8. TRANSACTION REFERENCES 164
11.8.1. Transaction Example for Jakarta Transactions 164
11.8.2. Transaction API Documentation 166

CHAPTER 12. JAKARTA PERSISTENCE 167
12.1. ABOUT JAKARTA PERSISTENCE 167
12.2. CREATE A SIMPLE JPA APPLICATION 167
12.3. JAKARTA PERSISTENCE ENTITIES 171
12.4. PERSISTENCE CONTEXT 171
12.4.1. Transaction-Scoped Persistence Context 171
12.4.2. Extended Persistence Context 172
12.5. JAKARTA PERSISTENCE ENTITYMANAGER 172
12.5.1. Application-Managed EntityManager 172
12.5.2. Container-Managed EntityManager 173
12.6. WORKING WITH THE ENTITYMANAGER 173
12.6.1. Binding the EntityManager to JNDI 173
12.7. DEPLOYING THE PERSISTENCE UNIT 174
12.8. SECOND-LEVEL CACHES 175
12.8.1. About Second-level Caches 175
12.8.1.1. Default Second-level Cache Provider 176
12.8.1.1.1. Configuring a Second-level Cache in the Persistence Unit 176

CHAPTER 13. JAKARTA BEAN VALIDATION 177
13.1. ABOUT JAKARTA BEAN VALIDATION 177
Features of Hibernate Validator 6.0.x 177
13.2. VALIDATION CONSTRAINTS 177
13.2.1. About Validation Constraints 177
13.2.2. Hibernate Validator Constraints 177
13.2.3. Using a Jakarta Bean Validation Custom Constraint 180
13.2.3.1. Creating A Constraint Annotation 180
13.2.3.2. Implementing A Constraint Validator 183
13.3. JAKARTA BEAN VALIDATION CONFIGURATION 184

CHAPTER 14. CREATING JAKARTA WEBSOCKET APPLICATIONS 185
Create the Jakarta WebSocket Application 185

CHAPTER 15. JAKARTA AUTHORIZATION 189

15.1. ABOUT JAKARTA AUTHORIZATION 189
15.2. CONFIGURE JAKARTA AUTHORIZATION SECURITY 189
Enabling Jakarta Authorization Using the elytron Subsystem 190

CHAPTER 16. JAKARTA AUTHENTICATION 193
16.1. ABOUT JAKARTA AUTHENTICATION SECURITY 193
16.2. CONFIGURE JAKARTA AUTHENTICATION 193
16.3. CONFIGURE JAKARTA AUTHENTICATION SECURITY USING ELYTRON 193
Enabling Jakarta Authentication for a Web Application 193
Additional Options 194
Subsystem Configuration 195
Programmatic Configuration 196
Authentication Process 196
Integrated Mode 196
Non-Integrated Mode 197
validateRequest 198
Control Flags 198
secureResponse 199
SecurityIdentity Creation 200

CHAPTER 17. JAKARTA SECURITY 201
17.1. ABOUT JAKARTA SECURITY 201
17.2. CONFIGURE JAKARTA SECURITY USING ELYTRON 201
Enabling Jakarta Security Using the elytron Subsystem 201
Enabling Jakarta Security for Web Applications 201

CHAPTER 18. JAKARTA BATCH APPLICATION DEVELOPMENT 202
18.1. REQUIRED BATCH DEPENDENCIES 202
18.2. JOB SPECIFICATION LANGUAGE (JSL) INHERITANCE 202
Inherit Step and Flow Within the Same Job XML File 202
Inherit a Step from a Different Job XML File 203
18.3. BATCH PROPERTY INJECTIONS 204
Injecting a Number into a Batchlet Class as Various Types 206
Injecting a Number Sequence into a Batchlet Class as Various Arrays 206
Injecting a Class Property into a Batchlet Class 207
Assigning a Default Value to a Field Annotated for Property Injection 208

CHAPTER 19. CONFIGURING CLIENTS 209
19.1. CLIENT CONFIGURATION USING THE WILDFLY-CONFIG.XML FILE 209
19.1.1. Client Authentication Configuration Using the wildfly-config.xml File 210
authentication-client Elements and Attributes 210
19.1.2. EJB Client Configuration Using the wildfly-config.xml File 231
jboss-ejb-client Elements and Attributes 231
Example EJB Client Configuration in the wildfly-config.xml File 233
19.1.3. HTTP Client Configuration Using the wildfly-config.xml File 233
19.1.4. Remoting Client Configuration Using the wildfly-config.xml File 234
endpoint Elements and Attributes 234
Example Remoting Client Configuration in the wildfly-config.xml File 235
19.1.5. Default XNIO Worker Configuration Using the wildfly-config.xml File 236
worker Elements and Attributes 236
Example XNIO Worker Configuration in the wildfly-config.xml File 238

CHAPTER 20. ECLIPSE MICROPROFILE 239
20.1. USING ECLIPSE MICROPROFILE OPENTRACING TO TRACE REQUESTS 239

20.1.1. Enable or Disable Tracing for Jakarta Contexts and Dependency Injection Beans 239
20.1.2. Enable or Disable Tracing for Jakarta RESTful Web Services Endpoints 240
20.1.3. Implement a Custom Tracer 240
20.2. USING ECLIPSE MICROPROFILE HEALTH TO MONITOR SERVER HEALTH 240
20.2.1. Implement a Custom Health Check 241

APPENDIX A. REFERENCE MATERIAL **242**
A.1. PROVIDED UNDERTOW HANDLERS 242
AccessControlListHandler 242
AccessLogHandler 242
AllowedMethodsHandler 244
BlockingHandler 244
ByteRangeHandler 244
CanonicalPathHandler 245
DisableCacheHandler 245
DisallowedMethodsHandler 245
EncodingHandler 245
FileErrorPageHandler 246
HttpTraceHandler 246
IPAddressAccessControlHandler 246
JDBCLogHandler 246
LearningPushHandler 247
LocalNameResolvingHandler 248
PathSeparatorHandler 248
PeerNameResolvingHandler 248
ProxyPeerAddressHandler 248
RedirectHandler 248
RequestBufferingHandler 249
RequestDumpingHandler 249
RequestLimitingHandler 249
ResourceHandler 249
ResponseRateLimitingHandler 250
SetHeaderHandler 250
SSLHeaderHandler 250
StuckThreadDetectionHandler 251
URLDecodingHandler 251
A.2. PERSISTENCE UNIT PROPERTIES 251
A.3. POLICY PROVIDER PROPERTIES 253
A.4. JAVA EE SPECIFICATIONS RELEVANT FOR JBOSS EAP AND THE CORRESPONDING JAKARTA EE SPECIFICATIONS 253
A.5. JAKARTA EE PROFILES AND TECHNOLOGIES REFERENCE 255

CHAPTER 1. GET STARTED DEVELOPING APPLICATIONS

1.1. ABOUT JAKARTA EE

1.1.1. Jakarta EE 8

Jakarta EE 8, maintained by the Eclipse Foundation, is based on Java Enterprise Edition 8 .

The 7.3 release of JBoss EAP is a Jakarta EE 8 compatible implementation for both Web Profile and Full Platform specifications.

For information about Jakarta EE 8, see About Jakarta EE.

Additional Resources

- About Jakarta EE
- Java EE Specifications and the Corresponding Jakarta EE Specifications
- Transition from Java EE to Jakarta EE

1.2. ABOUT JAVA EE

1.2.1. Overview of Java EE Profiles

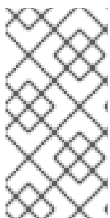
Java Enterprise Edition (Java EE) 8, as defined in JSR 366, includes support for profiles, which are subsets of APIs that represent configurations that are suited to specific classes of applications.

Java EE 8 defines specifications for the Web and the Full Platform profiles. A product can choose to implement the Full Platform, the Web Profile, or one or more custom profiles, in any combination.

- The Web Profile includes a selected subset of APIs that are designed to be useful for web application development.
- The Full Platform profile includes the APIs defined by the Java EE 8 Web Profile, plus the complete set of Java EE 8 APIs that are useful for enterprise application development.

JBoss EAP 7.3 is a certified implementation of the Java EE 8 Full Platform and the Web Profile specifications.

See Java™ EE 8 Technologies for the complete list of Java ™ EE 8 APIs.



NOTE

Java EE also includes support for JSR 375, which defines portable, plug-in interfaces for authentication and identity stores, and a new **injectable-type SecurityContext** interface that provides an access point for programmatic security. You can use the built-in implementations of these APIs, or define custom implementations.

Additional Resources

- Java EE Specifications and the Corresponding Jakarta EE Specifications

- Transition from Java EE to Jakarta EE

1.2.2. Transition from Java EE to Jakarta EE

After the Java Enterprise Edition 8 release, Oracle transferred Java EE to the Eclipse Foundation. The API code, implementation code, and Technology Compatibility Kit (TCK) code were transferred as part of a phased transfer process. A new certification process, the Jakarta EE Specification Process (JESP), was set up and a new specification license, the Eclipse Foundation Technology Compatibility Kit license, was created.

As part of this transfer process, new names were created for all Jakarta specifications corresponding to the existing Java EE specifications. All the new names start with Jakarta and are followed by a simple description of the specification. The Java EE specification names mentioned in the JBoss EAP documents and the names of the corresponding Jakarta EE specifications are listed in the Java EE Specifications Relevant for JBoss EAP and the Corresponding Jakarta EE Specifications section.

Additional Resources

*See Java EE Specifications Relevant for JBoss EAP and Corresponding Jakarta EE Specifications .

1.3. SETTING UP THE DEVELOPMENT ENVIRONMENT

1. Download and install Red Hat CodeReady Studio.
   For instructions, see Installing CodeReady Studio stand-alone using the Installer in the Red Hat CodeReady Studio *Installation Guide*.

2. Set up the JBoss EAP server in Red Hat CodeReady Studio.
   For instructions, see Downloading, Installing, and Setting Up JBoss EAP from within the IDE in the *Getting Started with CodeReady Studio Tools* guide.

1.4. CONFIGURE ANNOTATION PROCESSING IN RED HAT CODEREADY STUDIO

Annotation Processing (AP) is turned off by default in Eclipse. If your project generates implementation classes, this can result in **java.lang.ExceptionInInitializerError** exceptions, followed by ***CLASS_NAME*** **(implementation not found)** error messages when you deploy your project.

You can resolve these issues in one of the following ways. You can enable annotation processing for the individual project or you can enable annotation processing globally for all Red Hat CodeReady Studio projects.

Enable Annotation Processing for an Individual Project

To enable annotation processing for a specific project, you must add the **m2e.apt.activation** property with a value of **jdt_apt** to the project's **pom.xml** file.

```
<properties>
    <m2e.apt.activation>jdt_apt</m2e.apt.activation>
</properties>
```

You can find examples of this technique in the **pom.xml** files for the **logging-tools** and **kitchensink-ml** quickstarts that ship with JBoss EAP.

Enable Annotation Processing Globally in Red Hat CodeReady Studio

1. Select **Window** → **Preferences**.

2. Expand **Maven**, and select **Annotation Processing**.

3. Under **Select Annotation Processing Mode**, select **Automatically configure JDT APT (builds faster , but outcome may differ from Maven builds)**, then click **Apply and Close**.

1.5. CONFIGURE THE DEFAULT WELCOME WEB APPLICATION

JBoss EAP includes a default **Welcome** application, which displays at the root context on port **8080** by default.

This default **Welcome** application can be replaced with your own web application. This can be configured in one of two ways:

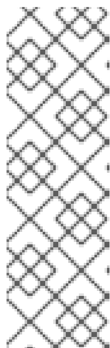
- Change the **welcome-content** file handler
- Change the **default-web-module**

You can also disable the welcome content .

Change the welcome-content File Handler

1. Modify the existing **welcome-content** file handler's path to point to the new deployment.

```
/subsystem=undertow/configuration=handler/file=welcome-content:write-attribute(name=path,value="/path/to/content")
```



NOTE

Alternatively, you could create a different file handler to be used by the server's root.

```
/subsystem=undertow/configuration=handler/file=NEW_FILE_HANDLER:add(path="/path/to/content")
/subsystem=undertow/server=default-server/host=default-host/location=\/:write-attribute(name=handler,value=NEW_FILE_HANDLER)
```

2. Reload the server for the changes to take effect.

```
reload
```

Change the default-web-module

1. Map a deployed web application to the server's root.

```
/subsystem=undertow/server=default-server/host=default-host:write-attribute(name=default-web-module,value=hello.war)
```

2. Reload the server for the changes to take effect.

```
reload
```

Disable the Default Welcome Web Application

1. Disable the welcome application by removing the **location** entry / for the **default-host**.

```
/subsystem=undertow/server=default-server/host=default-host/location=\/:remove
```

2. Reload the server for the changes to take effect.

```
reload
```

CHAPTER 2. USING MAVEN WITH JBOSS EAP

2.1. LEARN ABOUT MAVEN

2.1.1. About the Maven Repository

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

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

For more information about Maven, see Welcome to Apache Maven .

For more information about Maven repositories, see Apache Maven Project - Introduction to Repositories.

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.

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 .

Minimum Requirements of a Maven POM File
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

Example: Basic 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>
```

2.1.3. About the Maven Settings File

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

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

- **In the Maven installation:** The settings file can be found in the **$M2_HOME/conf/** directory. These settings are referred to as **global** settings. The default Maven settings file is a template that can be copied and used as a starting point for the user settings file.
- **In the user's installation:** The settings file can be found in the **${user.home}/.m2/** directory. If both the Maven and user **settings.xml** files exist, the contents are merged. Where there are overlaps, the user's **settings.xml** file takes precedence.

Example: Maven Settings File

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <profiles>
    <!-- Configure the JBoss EAP Maven repository -->
    <profile>
      <id>jboss-eap-maven-repository</id>
      <repositories>
        <repository>
          <id>jboss-eap</id>
          <url>file:///path/to/repo/jboss-eap-7.3.0.GA-maven-repository/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-7.3.0.GA-maven-repository/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.

2.1.4. 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 Best Practice - Using a Repository Manager.

Commonly used Maven repository managers

Sonatype Nexus

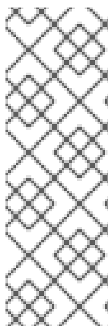
See Sonatype Nexus documentation for more information about Nexus.

Artifactory

See JFrog Artifactory documentation for more information about Artifactory.

Apache Archiva

See Apache Archiva: The Build Artifact Repository Manager for more information about Apache Archiva.



NOTE

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

2.2. INSTALL MAVEN AND THE JBOSS EAP MAVEN REPOSITORY

2.2.1. Download and Install Maven

Follow these steps to download and install Maven:

- If you are using Red Hat CodeReady Studio to build and deploy your applications, skip this procedure. Maven is distributed with Red Hat CodeReady Studio.
- If you are using the Maven command line to build and deploy your applications to JBoss EAP, you must 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.

2.2.2. Download the JBoss EAP Maven Repository

You can use either method to download the JBoss EAP Maven repository:

- Download the JBoss EAP Maven repository ZIP file .
- Download the JBoss EAP Maven repository using the Offliner application .

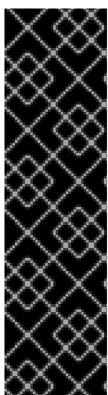
2.2.2.1. Download the JBoss EAP Maven Repository ZIP File

Follow these steps to download the JBoss EAP Maven repository.

1. Log in to the JBoss EAP download page on the Red Hat Customer Portal.
2. Select **7.3** in the **Version** drop-down menu.
3. Find the **Red Hat JBoss Enterprise Application Platform 7.3 Maven Repository** entry in the list and click **Download** to download a ZIP file containing the repository.
4. Save the ZIP file to the desired directory.
5. Extract the ZIP file.

2.2.2.2. Download the JBoss EAP Maven Repository with the Offliner Application

The Offliner application is available as an alternative option to download the Maven artifacts for developing JBoss EAP applications using the Red Hat Maven repository.



IMPORTANT

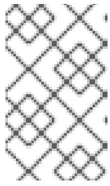
The process of downloading the JBoss EAP Maven repository using the Offliner application is provided as Technology Preview only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs), might not be functionally complete, and Red Hat does not recommend to use them for production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

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

1. Log in to the JBoss EAP download page on the Red Hat Customer Portal.
2. Select **7.3** in the **Version** drop-down menu.

3. Find the **Red Hat JBoss Enterprise Application Platform 7.3 Maven Repository Offliner Content List** entry in the list and click **Download**.

4. Save the text file to the desired directory.



NOTE

This file does not contain license information. The artifacts downloaded by the Offliner application have the same licenses as specified in the Maven repository ZIP file that is distributed with JBoss EAP.

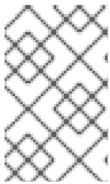
5. Download the Offliner application from the Maven Central Repository.

6. Run the Offliner application using the following command:

```
$ java -jar offliner.jar -r http://repository.redhat.com/ga/ -d DOWNLOAD_FOLDER jboss-eap-7.3.0-maven-repository-content-with-sha256-checksums.txt
```

The artifacts from the JBoss EAP Maven repository are downloaded into the ***DOWNLOAD_FOLDER*** directory.

See the Offliner documentation for more information on running the Offliner application.



NOTE

The generated JBoss EAP Maven repository will have the same content that is currently available in the JBoss EAP Maven repository ZIP file. It will not contain artifacts available in Maven Central repository.

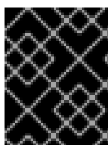
2.2.3. Install the JBoss EAP Maven Repository

You can use the JBoss EAP Maven repository available online, or download and install it locally using any one of the three listed methods:

- Install the JBoss EAP Maven repository on your local file system. For detailed instructions, see Install the JBoss EAP Maven Repository Locally .
- Install the JBoss EAP Maven repository on the Apache Web Server. For more information, see Install the JBoss EAP Maven Repository for Use with Apache httpd .
- Install the JBoss EAP Maven repository using the Nexus Maven Repository Manager. For more information, see Repository Management Using Nexus Maven Repository Manager .

2.2.3.1. Install the JBoss EAP Maven Repository Locally

Use this option to install the JBoss EAP Maven Repository to the local file system. This is easy to configure and allows you to get up and running quickly on your local machine.



IMPORTANT

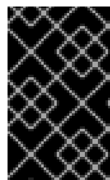
This method can help you become familiar with using Maven for development but is not recommended for team production environments.

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

To install the JBoss EAP Maven repository to the local file system:

1. Make sure you have downloaded the JBoss EAP Maven repository ZIP file to your local file system.

2. Unzip the file on the local file system of your choosing.
   This creates a new **jboss-eap-7.3.0.GA-maven-repository/** directory, which contains the Maven repository in a subdirectory named **maven-repository/**.



IMPORTANT

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

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

Installing the JBoss EAP Maven Repository for use with Apache httpd is a good option for multi-user and cross-team development environments because any developer that can access the web server can also access the Maven repository.

Before installing the JBoss EAP Maven Repository, you must first configure Apache httpd. See Apache HTTP Server Project documentation for instructions.

1. Ensure that you have the JBoss EAP Maven repository ZIP file downloaded to your local file system.

2. Unzip the file in a directory that is web accessible on the Apache server.

3. Configure Apache to allow read access and directory browsing in the created directory.
   This configuration allows a multi-user environment to access the Maven repository on Apache httpd.

2.3. USE THE MAVEN REPOSITORY

2.3.1. Configure the JBoss EAP Maven Repository

Overview

There are two approaches to direct Maven to use the JBoss EAP 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.

Configure the JBoss EAP Maven Repository Using the Maven Settings

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

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

This section describes how to configure the Maven settings. You can configure the Maven install global settings or the user's install settings.

Configure the Maven Settings File

1. Locate the Maven **settings.xml** file for your operating system. It is usually located in the **${user.home}/.m2/** directory.
   - For Linux or Mac, this is **~/.m2/**
   - For Windows, this is **\Documents and Settings\.m2** or **\Users\.m2**
2. If you do not find a **settings.xml** file, copy the **settings.xml** file from the **${user.home}/.m2/conf/** directory into the **${user.home}/.m2/** directory.
3. Copy the following XML into the **<profiles>** element of the **settings.xml** file. Determine the URL of the JBoss EAP repository and replace ***JBOSS_EAP_REPOSITORY_URL*** with it.

```
<!-- Configure the JBoss Enterprise Maven repository -->
<profile>
  <id>jboss-enterprise-maven-repository</id>
  <repositories>
    <repository>
      <id>jboss-enterprise-maven-repository</id>
      <url>JBOSS_EAP_REPOSITORY_URL</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>jboss-enterprise-maven-repository</id>
      <url>JBOSS_EAP_REPOSITORY_URL</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
</profile>
```

The following is an example configuration that accesses the online JBoss EAP Maven repository.

```
<!-- Configure the JBoss Enterprise Maven repository -->
<profile>
  <id>jboss-enterprise-maven-repository</id>
```

```
    <repositories>
      <repository>
        <id>jboss-enterprise-maven-repository</id>
        <url>https://maven.repository.redhat.com/ga/</url>
        <releases>
          <enabled>true</enabled>
        </releases>
        <snapshots>
          <enabled>false</enabled>
        </snapshots>
      </repository>
    </repositories>
    <pluginRepositories>
      <pluginRepository>
        <id>jboss-enterprise-maven-repository</id>
        <url>https://maven.repository.redhat.com/ga/</url>
        <releases>
          <enabled>true</enabled>
        </releases>
        <snapshots>
          <enabled>false</enabled>
        </snapshots>
      </pluginRepository>
    </pluginRepositories>
  </profile>
```

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

```
<activeProfile>jboss-enterprise-maven-repository</activeProfile>
```

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

   a. From the menu, choose **Window** → **Preferences**.

   b. In the **Preferences** window, expand **Maven** and choose **User Settings**.

   c. Click the **Update Settings** button to refresh the Maven user settings in Red Hat CodeReady Studio.

IMPORTANT

If your Maven repository contains outdated artifacts, you might 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*

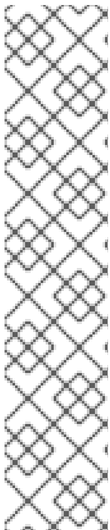
To resolve the issue, delete the cached version of your local repository to force a download of the latest Maven artifacts. The cached repository is located here: **${user.home}/.m2/repository/**

Configure the JBoss EAP Maven Repository Using the Project POM

WARNING

You should avoid this method of configuration as it overrides the global and user Maven settings for the configured project.

You must plan carefully if you decide to configure repositories using project POM file. Transitively included POMs are an issue with this type of configuration since Maven has to query the external repositories for missing artifacts and this slows the build process. It can also cause you to lose control over where your artifacts are coming from.



NOTE

The URL of the repository will depend on where the repository is located: on the file system, or web server. For information on how to install the repository, see: Install the JBoss EAP Maven Repository. The following are examples for each of the installation options:

File System

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

Apache Web Server

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

Nexus Repository Manager

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

Configuring the Project's POM File

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>JBOSS_EAP_REPOSITORY_URL</url>
    <layout>default</layout>
    <releases>
      <enabled>true</enabled>
      <updatePolicy>never</updatePolicy>
    </releases>
    <snapshots>
      <enabled>true</enabled>
      <updatePolicy>never</updatePolicy>
    </snapshots>
  </repository>
</repositories>
```

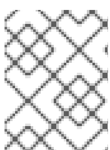
3. Add the following plug-in repository configuration. If there is already a **<pluginRepositories>** configuration in the file, then add the **<pluginRepository>** element to it.

```
<pluginRepositories>
  <pluginRepository>
    <id>jboss-eap-repository-group</id>
    <name>JBoss EAP Maven Repository</name>
    <url>JBOSS_EAP_REPOSITORY_URL</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </pluginRepository>
</pluginRepositories>
```

Determine the URL of the JBoss EAP Repository

The repository URL depends on where the repository is located. You can configure Maven to use any of the following repository locations.

- To use the online JBoss EAP Maven repository, specify the following URL: https://maven.repository.redhat.com/ga/
- To use a JBoss EAP Maven repository installed on the local file system, you must download the repository and then use the local file path for the URL. For example: *file:///path/to/repo/jboss-eap-7.3.0.GA-maven-repository/maven-repository/*
- If you install the repository on an Apache Web Server, the repository URL will be similar to the following: *http://intranet.acme.com/jboss-eap-7.3.0.GA-maven-repository/maven-repository/*
- If you install the JBoss EAP Maven repository using the Nexus Repository Manager, the URL will look something like the following: *https://intranet.acme.com/nexus/content/repositories/jboss-eap-7.3.0.GA-maven-repository/maven-repository/*



NOTE

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

2.3.2. Configure Maven for Use with Red Hat CodeReady Studio

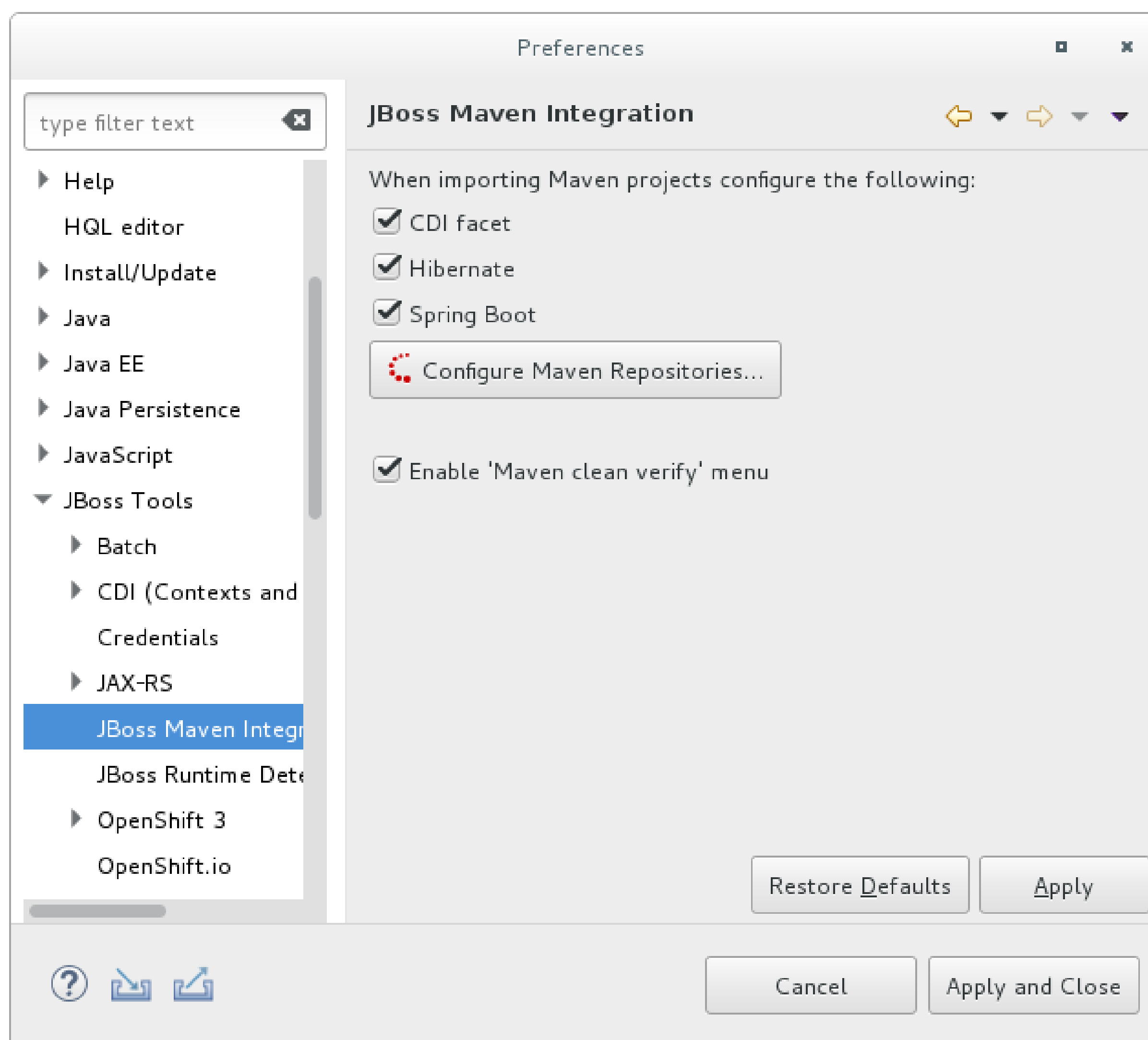
The artifacts and dependencies needed to build and deploy applications to Red Hat JBoss Enterprise Application Platform are hosted on a public repository. You must direct Maven to use this repository when you build your applications. This section covers the steps to configure Maven if you plan to build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

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

Configure Maven in Red Hat CodeReady Studio

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

Figure 2.1. JBoss Maven Integration Pane in the Preferences Window



2. Click **Configure Maven Repositories**.

3. Click **Add Repository** to configure the JBoss Enterprise Maven repository. Complete the **Add Maven Repository** dialog as follows:

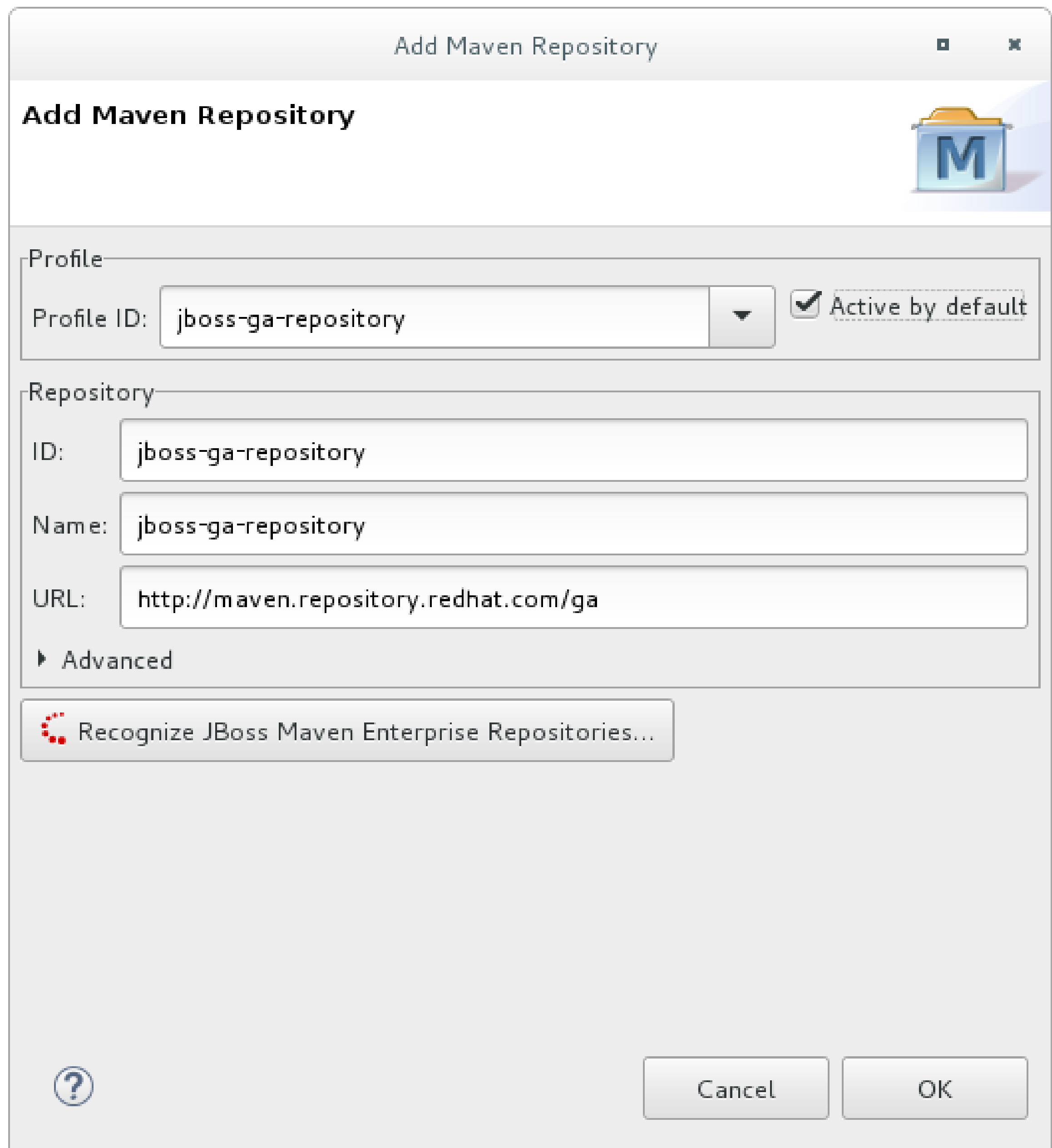
   a. Set the **Profile ID**, **Repository ID**, and **Repository Name** values to **jboss-ga-repository**.

   b. Set the **Repository URL** value to **http://maven.repository.redhat.com/ga**.

   c. Click the **Active by default** checkbox to enable the Maven repository.

   d. Click **OK**.

Figure 2.2. Add Maven Repository



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

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

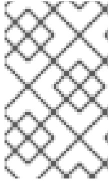
The JBoss EAP Maven repository is now configured for use with Red Hat CodeReady Studio.

2.3.3. Manage Project Dependencies

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

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

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



NOTE

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

Supported Maven Artifacts

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

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

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

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.3.1.Final-redhat-1</version>
  <scope>provided</scope>
</dependency>
```

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

Dependency Management

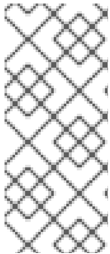
Maven includes a mechanism for managing the versions of direct and transitive dependencies throughout the build. For general information about using dependency management, see the Apache Maven Project: Introduction to the Dependency Mechanism.

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

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

```
<dependencyManagement>
  <dependencies>
    ...
    <dependency>
      <groupId>org.jboss.bom</groupId>
      <artifactId>eap-runtime-artifacts</artifactId>
      <version>7.3.0.GA</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
```

```
    ...
  </dependencies>
</dependencyManagement>
```



NOTE

In JBoss EAP 7 the name of this BOM was changed from **eap6-supported-artifacts** to **eap-runtime-artifacts**. The purpose of this change is to make it more clear that the artifacts in this POM are part of the JBoss EAP runtime, but are not necessarily part of the supported public API. Some of the JARs contain internal API and functionality, which might change between releases.

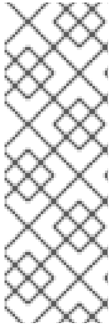
JBoss EAP Jakarta EE Specs BOM

The **jboss-jakartaee-8.0** BOM contains the Jakarta EE Specification API JARs used by JBoss EAP.

To use this BOM in a project, first add a dependency for the **jboss-jakartaee-8.0** BOM in the **dependencyManagement** section of the POM file, specifying **org.jboss.spec** for its **groupId**, and then add the dependencies for the specific APIs needed by the application. These dependencies do not require a version and use a scope of **provided** because the APIs are included in the **jboss-jakartaee-8.0** BOM.

The following example uses the **1.0.0.Alpha1** version of the **jboss-jakartaee-8.0** BOM, to add dependencies for the Servlet and Jakarta Server Pages APIs.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.spec</groupId>
      <artifactId>jboss-jakartaee-8.0</artifactId>
      <version>1.0.0.Alpha1</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.jboss.spec.javax.servlet</groupId>
    <artifactId>jboss-servlet-api_4.0_spec</artifactId>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.jboss.spec.javax.servlet.jsp</groupId>
    <artifactId>jboss-jsp-api_2.3_spec</artifactId>
    <scope>provided</scope>
  </dependency>
  ...
</dependencies>
```



NOTE

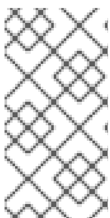
JBoss EAP packages and provides BOMs for the APIs of most of the product components. Many of these BOMs are conveniently packaged into one larger **jboss-eap-jakartaee8** BOM with a **groupId** of **org.jboss.bom**. The **jboss-jakartaee-8.0** BOM, whose **groupId** is **org.jboss.spec**, is included in this larger BOM. This means that if you are using additional JBoss EAP dependencies that are packaged in this BOM, you can just add the one **jboss-eap-jakartaee8** BOM to your project's POM file rather than separately adding the **jboss-jakartaee-8.0** and other BOM dependencies.

JBoss EAP BOMs Available for Application Development

The following table lists the Maven BOMs that are available for application development.

Table 2.1. JBoss BOMs

BOM Artifact ID	Use Case
eap-runtime-artifacts	Supported JBoss EAP runtime artifacts.
jboss-eap-jakartaee8	Supported JBoss EAP Jakarta EE 8 APIs plus additional JBoss EAP API JARs.
jboss-eap-jakartaee8-with-spring4	**jboss-eap-jakartaee8** plus recommended Spring 4 versions.
jboss-eap-jakartaee8-with-tools	**jboss-eap-jakartaee8** plus development tools such as Arquillian.



NOTE

These BOMs from JBoss EAP 6 have been consolidated into fewer BOMs to make usage simpler for most use cases. The Hibernate, logging, transactions, messaging, and other public API JARs are now included in the **jboss-eap-jakartaee8** BOM instead of a requiring a separate BOM for each case.

The following example uses the **7.3.0.GA** version of the **jboss-eap-jakartaee8** BOM.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.bom</groupId>
      <artifactId>jboss-eap-jakartaee8</artifactId>
      <version>7.3.0.GA</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
```

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

JBoss EAP Client BOMs

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

The **wildfly-ejb-client-bom**, **wildfly-jms-client-bom**, and **wildfly-jaxws-client-bom** are managed by the **jboss-eap-jakartaee8** BOM, so you do not need to manage the versions in your project dependencies.

The following is an example of how to add the **wildfly-ejb-client-bom**, **wildfly-jms-client-bom**, and **wildfly-jaxws-client-bom** dependencies to your project.

```
<dependencyManagement>
  <dependencies>
    <!-- JBoss stack of the Jakarta EE APIs and related components.  -->
    <dependency>
      <groupId>org.jboss.bom</groupId>
      <artifactId>jboss-eap-jakartaee8</artifactId>
      <version>7.3.0.GA</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
  ...
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.jboss.eap</groupId>
    <artifactId>wildfly-ejb-client-bom</artifactId>
    <type>pom</type>
  </dependency>
  <dependency>
    <groupId>org.jboss.eap</groupId>
    <artifactId>wildfly-jms-client-bom</artifactId>
    <type>pom</type>
  </dependency>
  <dependency>
    <groupId>org.jboss.eap</groupId>
    <artifactId>wildfly-jaxws-client-bom</artifactId>
    <type>pom</type>
  </dependency>
  ...
</dependencies>
```

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

CHAPTER 3. CLASS LOADING AND MODULES

3.1. INTRODUCTION

3.1.1. Overview of Class Loading and Modules

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

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

3.1.2. Class Loading in Deployments

For the purposes of class loading, JBoss EAP treats all deployments as modules. 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 the **WEB-INF/classes** directory. All classes packaged in the WAR will be loaded with the same class loader.

EAR Deployment

EAR deployments are made up of more than one module, and are defined by the following 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, for example 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 per deployment or for the entire application server.

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

3.1.3. Class Loading Precedence

The JBoss EAP 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 dependencies are automatically added by JBoss EAP, such as the Jakarta EE APIs. These dependencies have the highest class loader precedence because they contain common functionality and APIs that are supplied by JBoss EAP.
   See Implicit Module Dependencies for complete details about each implicit dependency.

2. **Explicit dependencies:** These dependencies are manually added to the application configuration using the application's **MANIFEST.MF** file or the new optional JBoss deployment descriptor **jboss-deployment-structure.xml** file.
   See Add an Explicit Module Dependency to a Deployment to learn how to add explicit dependencies.

3. **Local resources:** These are class files packaged up inside the deployment itself, for example in 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.

3.1.4. jboss-deployment-structure.xml

The **jboss-deployment-structure.xml** file is an optional deployment descriptor for JBoss EAP. This deployment descriptor provides control over class loading in the deployment.

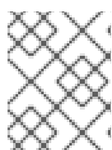
The XML schema for this deployment descriptor is located in the product install directory under ***EAP_HOME*/docs/schema/jboss-deployment-structure-1_2.xsd**.

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

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

3.2. ADD AN EXPLICIT MODULE DEPENDENCY TO A DEPLOYMENT

Explicit module dependencies can be added to applications to add the classes of those modules to the class path of the application at deployment.



NOTE

JBoss EAP automatically adds some dependencies to deployments. See Implicit Module Dependencies for details.

Prerequisites

1. A working software project that you want to add a module dependency to.

2. You must know the name of the module being added as a dependency. See Included Modules

for the list of static modules included with JBoss EAP. If the module is another deployment, then see Dynamic Module Naming in the JBoss EAP *Configuration Guide* to determine the module name.

Dependencies can be configured using two methods:

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

Add a Dependency Configuration to MANIFEST.MF

Maven projects can be configured to create the required dependency entries in the **MANIFEST.MF** file.

1. If the project does not have one, 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 a dependencies entry to the **MANIFEST.MF** file with a comma-separated list of dependency module names:

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

- To make a dependency optional, append **optional** to the module name in the dependency entry:

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

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

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

- The **annotations** flag is needed when the module dependency contains annotations that need to be processed during annotation scanning, such as when declaring EJB interceptors. Without this, an EJB interceptor declared in a module cannot be used in a deployment. There are other situations involving annotation scanning when this is needed too.

```
Dependencies: org.javassist, test.module annotations
```

- By default items in the **META-INF** of a dependency are not accessible. The **services** dependency makes items from **META-INF/services** accessible so that **services** in the modules can be loaded.

```
Dependencies: org.javassist, org.hibernate services
```

- To scan a **beans.xml** file and make its resulting beans available to the application, the **meta-inf** dependency can be used.

```
Dependencies: org.javassist, test.module meta-inf
```

Add a Dependency Configuration to the jboss-deployment-structure.xml

1. If the application does not have one, 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 a **<dependencies>** element within that.

3. 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" />
```

   - 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" />
   ```

   - 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" />
   ```

   - When the module dependency contains annotations that need to be processed during annotation scanning, the **annotations** flag is used.

   ```
   <module name="test.module" annotations="true" />
   ```

   - The **services** dependency specifies whether and how **services** found in this dependency are used. The default is **none**. Specifying a value of **import** for this attribute is equivalent to adding a filter at the end of the import filter list which includes the **META-INF/services** path from the dependency module. Setting a value of **export** for this attribute is equivalent to the same action on the export filter list.

   ```
   <module name="org.hibernate" services="import" />
   ```

   - The **META-INF** dependency specifies whether and how **META-INF** entries in this dependency are used. The default is **none**. Specifying a value of **import** for this attribute is equivalent to adding a filter at the end of the import filter list which includes the **META-INF/**** path from the dependency module. Setting a value of **export** for this attribute is equivalent to the same action on the export filter list.

   ```
   <module name="test.module" meta-inf="import" />
   ```

Example: **jboss-deployment-structure.xml** File with Two Dependencies

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

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

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

Creating a Jandex Index

The **annotations** flag requires that the module contain a Jandex index. In JBoss EAP 7.3, this is generated automatically. However, adding the index manually is still recommended for performance reasons because automatic scanning can be a long process that consumes the CPU and increases the deployment time.

To add the index manually, create a new "index JAR" to add to the module. Use the Jandex JAR to build the index, and then insert it into a new JAR file. In the current implementation, when an index is added to a JAR file inside a module, no scanning at all is executed.

Creating a Jandex index::

1. Create the index:

   ```
   java -jar modules/system/layers/base/org/jboss/jandex/main/jandex-jandex-2.0.0.Final-redhat-1.jar $JAR_FILE
   ```

2. Create a temporary working space:

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

3. Move the index file to the working directory

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

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

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

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

   b. Option 2: Add the index to an existing JAR

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

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

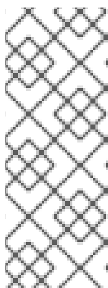
   a. Option 1: If you are adding a module dependency using **MANIFEST.MF**, add **annotations** after the module name. For example change:

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

to

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

b. Option 2: If you are adding a module dependency using **jboss-deployment-structure.xml** add **annotations="true"** on the module dependency.



NOTE

An annotation index is required when an application wants to use annotated Jakarta EE components defined in classes within the static module. In JBoss EAP 7.3, annotation indexes for static modules are automatically generated, so you do not need to create them. However, you must tell the module import to use the annotations by adding the dependencies to either the **MANIFEST.MF** or the **jboss-deployment-structure.xml** file.

3.3. GENERATE MANIFEST.MF ENTRIES USING MAVEN

Maven projects using 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, but only creates the **MANIFEST.MF** file with the details specified in the **pom.xml**.

Before generating the **MANIFEST.MF** entries using Maven, you will require:

- A working Maven project, which is using one of the JAR, EJB, or WAR plug-ins (**maven-jar-plugin**, **maven-ejb-plugin**, or **maven-war-plugin**).
- You must know the name of the project's module dependencies. Refer to Included Modules for the list of static modules included with JBoss EAP. If the module is another deployment, then refer to Dynamic Module Naming in the JBoss EAP *Configuration Guide* to determine the module name.

Generate a MANIFEST.MF File Containing Module Dependencies

1. 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. Add the list of module dependencies to the **<Dependencies>** element. Use the same format that is used when adding the dependencies to the **MANIFEST.MF** file:

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

The **optional** and **export** attributes can also be used here:

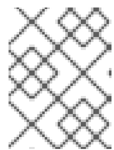
```
<Dependencies>org.javassist optional, org.apache.velocity export</Dependencies>
```

3. Build the project using the Maven assembly goal:

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

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

Example: Configured Module Dependencies in **pom.xml**



NOTE

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>
```

3.4. PREVENT A MODULE BEING IMPLICITLY LOADED

You can configure a deployable application to prevent implicit dependencies from being loaded. This can be useful when an 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

- A working software project that you want to exclude an implicit dependency from.
- You must know the name of the module to exclude. Refer to Implicit Module Dependencies for a list of implicit dependencies and their conditions.

Add dependency exclusion configuration to jboss-deployment-structure.xml

1. If the application does not have one, create a new file called **jboss-deployment-structure.xml** and add it to the project. This 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: Excluding Two Modules

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

3.5. EXCLUDE A SUBSYSTEM FROM A DEPLOYMENT

Excluding a subsystem provides the same effect as removing the subsystem, but it applies only to a single deployment. You can exclude a subsystem from a deployment by editing the **jboss-deployment-structure.xml** configuration file.

Exclude a Subsystem

1. Edit the **jboss-deployment-structure.xml** file.

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.

The subsystem's deployment unit processors will no longer run on the deployment.

Example: **jboss-deployment-structure.xml** File

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.2">
  <ear-subdeployments-isolated>true</ear-subdeployments-isolated>
  <deployment>
    <exclude-subsystems>
      <subsystem name="jaxrs" />
    </exclude-subsystems>
```

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

3.6. USE THE CLASS LOADER PROGRAMMATICALLY IN A DEPLOYMENT

3.6.1. Programmatically Load Classes and Resources in a Deployment

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

Load a Class Using the Class.forName() Method

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

- **Class.forName(String className)**:

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

- **Class.forName(String className, boolean initialize, ClassLoader loader)**:
  This signature expects three parameters: the class name, a boolean value that specifies whether to initialize the class, and the **ClassLoader** that should load the class.

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

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

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

Find All Resources with a Given Name

If you know the name and path of a resource, the best way to load it directly is to use the standard Java Development Kit (JDK) **Class** or **ClassLoader** API.

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

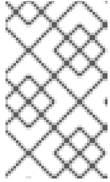
  ```
  InputStream inputStream =
  CurrentClass.class.getResourceAsStream("targetResourceName");
  ```

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

  The following example loads all instances of a resource and iterates through the results.

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

```
            } catch (Exception e) {
                // ignore
            }
        }
    }
}
```



NOTE

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

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

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

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

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

3.6.2. Programmatically Iterate Resources in a Deployment

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

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

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

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

- List resources within a deployment and within all imports.
  There are times when it is not possible to look up resources by the exact path. For example, the exact path might not be known or you might need to examine more than one file in a given path. In this case, the JBoss Modules library provides several APIs for iterating all deployment resources. You can iterate through resources in a deployment by utilizing one of two methods.
  - Iterate all resources found in a single module.
    The **ModuleClassLoader.iterateResources()** method iterates all the resources within this module class loader. This method takes two arguments: the starting directory name to search and a boolean that specifies whether it should recurse into subdirectories.

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

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

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

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

  The following example demonstrates how to find the entire set of resources in this module, including imports.

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

- Find all resources that match a pattern.
  If you need to find only specific resources within your deployment or within your deployment's full import set, you need to filter the resource iteration. The JBoss Modules filtering APIs give you several tools to accomplish this.
  - Examine the full set of dependencies.
    If you need to examine the full set of dependencies, you can use the **Module.iterateResources()** method's **PathFilter** parameter to check the name of each resource for a match.
  - Examine deployment dependencies.
    If you need to look only within the deployment, use the **ModuleClassLoader.iterateResources()** method. However, you must use additional methods to filter the resultant iterator. The **PathFilters.filtered()** method can provide a filtered view of a resource iterator this case. The **PathFilters** class includes many static methods to create and compose filters that perform various functions, including finding child paths or exact matches, or matching an Ant-style "glob" pattern.
- Additional code examples for filtering resources.
  The following examples demonstrate how to filter resources based on different criteria.

Example: Find All Files Named messages.properties in Your Deployment

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

Example: Find All Files Named **messages.properties** in Your Deployment and Imports

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

Example: Find All Files Inside Any Directory Named **my-resources** in Your Deployment

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

Example: Find All Files Named **messages** or **errors** in Your Deployment and Imports

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

Example: Find All Files in a Specific Package in Your Deployment

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

3.7. CLASS LOADING AND SUBDEPLOYMENTS

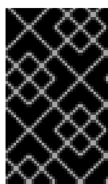
3.7.1. Modules and Class Loading in Enterprise Archives

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

The following rules determine what modules exist in an EAR:

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

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



IMPORTANT

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

Subdeployment class loader isolation can be enabled if strict compatibility is required. This can be enabled for a single EAR deployment or for all EAR deployments. The Jakarta EE 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.

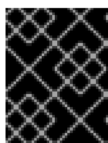
3.7.2. Subdeployment Class Loader Isolation

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

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

3.7.3. Enable Subdeployment Class Loader Isolation Within a EAR

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



IMPORTANT

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

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

```
<jboss-deployment-structure>

</jboss-deployment-structure>
```

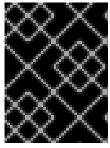
2. Add the **<ear-subdeployments-isolated>** element.
   Add the **<ear-subdeployments-isolated>** element to the **jboss-deployment-structure.xml** file if it doesn't already exist with the content of **true**.

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

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

3.7.4. Configuring Session Sharing between Subdeployments in Enterprise Archives

JBoss EAP provides the ability to configure enterprise archives (EARs) to share sessions between WAR module subdeployments contained in the EAR. This functionality is disabled by default and must be explicitly enabled in the **META-INF/jboss-all.xml** file in the EAR.



IMPORTANT

Since this feature is not a standard servlet feature, your applications might not be portable if this functionality is enabled.

To enable session sharing between WARs within an EAR, you need to declare a **shared-session-config** element in the **META-INF/jboss-all.xml** of the EAR:

Example: META-INF/jboss-all.xml

```
<jboss xmlns="urn:jboss:1.0">
  ...
  <shared-session-config xmlns="urn:jboss:shared-session-config:2.0">
  </shared-session-config>
  ...
</jboss>
```

The **shared-session-config** element is used to configure the shared session manager for all WARs within the EAR. If the **shared-session-config** element is present, all WARs within the EAR will share the same session manager. Changes made here will affect all the WARs contained within the EAR.

3.7.4.1. Reference of Shared Session Configuration Options

Example: META-INF/jboss-all.xml

```
<jboss xmlns="urn:jboss:1.0">
    <shared-session-config xmlns="urn:jboss:shared-session-config:2.0">
     <distributable/>
        <max-active-sessions>10</max-active-sessions>
        <session-config>
            <session-timeout>0</session-timeout>
            <cookie-config>
                <name>JSESSIONID</name>
                <domain>domainName</domain>
                <path>/cookiePath</path>
                <comment>cookie comment</comment>
                <http-only>true</http-only>
                <secure>true</secure>
                <max-age>-1</max-age>
            </cookie-config>
            <tracking-mode>COOKIE</tracking-mode>
        </session-config>
        <replication-config>
            <cache-name>web</cache-name>
            <replication-granularity>SESSION</replication-granularity>
        </replication-config>
    </shared-session-config>
</jboss>
```

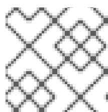
Element	Description
shared-session-config	Root element for the shared session configuration. If this is present in the **META-INF/jboss-all.xml**, then all deployed WARs contained in the EAR will share a single session manager.
distributable	Specifies that a distributable session manager should be used. Starting with the version 2.0 of the schema, a non-distributable session manager is used by default. For version 1.0, the distributable session manager continues to be the default session manager.
max-active-sessions	Number of maximum sessions allowed.
session-config	Contains the session configuration parameters for all deployed WARs contained in the EAR.
session-timeout	Defines the default session timeout interval for all sessions created in the deployed WARs contained in the EAR. The specified timeout must be expressed in a whole number of minutes. If the timeout is *0* or less, the container ensures the default behavior of sessions is to never time out. If this element is not specified, the container must set its default timeout period.
cookie-config	Contains the configuration of the session tracking cookies created by the deployed WARs contained in the EAR.
name	The name that will be assigned to any session tracking cookies created by the deployed WARs contained in the EAR. The default is **JSESSIONID**.
domain	The domain name that will be assigned to any session tracking cookies created by the deployed WARs contained in the EAR.
path	The path that will be assigned to any session tracking cookies created by the deployed WARs contained in the EAR.
comment	The comment that will be assigned to any session tracking cookies created by the deployed WARs contained in the EAR.
http-only	Specifies whether any session tracking cookies created by the deployed WARs contained in the EAR will be marked as **HttpOnly**.
secure	Specifies whether any session tracking cookies created by the deployed WARs contained in the EAR will be marked as secure even if the request that initiated the corresponding session is using plain HTTP instead of HTTPS.

Element	Description
max-age	The lifetime (in seconds) that will be assigned to any session tracking cookies created by the deployed WARs contained in the EAR. Default is **-1**.
tracking-mode	Defines the tracking modes for sessions created by the deployed WARs contained in the EAR.
replication-config	Contains the HTTP session clustering configuration.
cache-name	This option is for use in clustering only. It specifies the name of the Infinispan container and cache in which to store session data. The default value, if not explicitly set, is determined by the application server. To use a specific cache within a cache container, use the form **container.cache**, for example **web.dist**. If name is unqualified, the default cache of the specified container is used.
replication-granularity	This option is for use in clustering only. It determines the session replication granularity level. The possible values are **SESSION** and **ATTRIBUTE** with **SESSION** being the default. If **SESSION** granularity is used, all session attributes are replicated if any were modified within the scope of a request. This policy is required if an object reference is shared by multiple session attributes. However, this can be inefficient if session attributes are sufficiently large and/or are modified infrequently, since all attributes must be replicated regardless of whether they were modified or not. If **ATTRIBUTE** granularity is used, only those attributes that were modified within the scope of a request are replicated. This policy is not appropriate if an object reference is shared by multiple session attributes. This can be more efficient than **SESSION** granularity if the session attributes are sufficiently large and/or are modified infrequently.

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

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

This can be done by creating a custom JBoss EAP module that contains the TLD JARs, and declaring a dependency on that module in the applications. For more information see modules and dependencies.



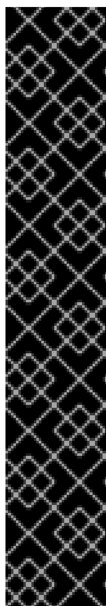
NOTE

Ensure that at least one JAR contains TLDs and that the TLDs are packed in **META-INF**.

Deploy TLDs in a Custom Module

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

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



IMPORTANT

Using the **module** management CLI command to add and remove modules is provided as Technology Preview only. This command is not appropriate for use in a managed domain or when connecting to the management CLI remotely. Modules should be added and removed manually in a production environment. For more information, see the Create a Custom Module Manually and Remove a Custom Module Manually sections of the JBoss EAP *Configuration Guide*.

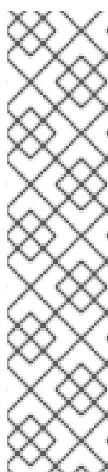
Technology Preview features are not supported with Red Hat production service level agreements (SLAs), might not be functionally complete, and Red Hat does not recommend to use them for production. These features provide early access to upcoming product features, enabling customers to test functionality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

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

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

When creating the module, you can specify multiple JAR resources by separating each one with the file system-specific separator for your system.

- For linux - **:**. Example, **--resources=<path-to-jar>:<path-to-another-jar>**
- For Windows - **;**. Example, **--resources=<path-to-jar>;<path-to-another-jar>**



NOTE

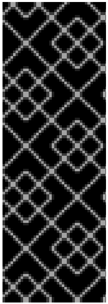
--resources

It is required unless **--module-xml** is used. It lists file system paths, usually JAR files, separated by a file system-specific path separator, for example **java.io.File.pathSeparatorChar**. The files specified will be copied to the created module's directory.

--resource-delimiter

It is an optional user-defined path separator for the resources argument. If this argument is present, the command parser will use the value here instead of the file system-specific path separator. This allows the **modules** command to be used in cross-platform scripts.

2. In your applications, declare a dependency on the new MyTagLibs custom module using one of the methods described in Add an Explicit Module Dependency to a Deployment .



IMPORTANT

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

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

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

3.9. DISPLAY MODULES BY DEPLOYMENT

Display Modules by Deployment

You can use the **list-modules** management operation to display a list of modules according to each deployment.

```
:list-modules
```

Example: Display Modules by Deployment for a Standalone Server

```
/deployment=ejb-in-ear.ear:list-modules
```

```
/deployment=ejb-in-ear.ear/subdeployment=ejb-in-ear-web.war:list-modules
```

Example: Display Modules by Deployment for a Managed Domain

```
/host=master/server=server-one/deployment=ejb-in-ear.ear:list-modules
```

```
/host=master/server=server-one/deployment=ejb-in-ear.ear/subdeployment=ejb-in-ear-web.war:list-modules
```

This operation displays the list in a compact view.

Example: Standard List Output

```
[standalone@localhost:9990 /] /deployment=sample-ear-1.0.ear:list-modules
 {
    "outcome" => "success",
    "result" => {
       "system-dependencies" => [
          {"name" => "com.fasterxml.jackson.datatype.jackson-datatype-jdk8"},
          {"name" => "com.fasterxml.jackson.datatype.jackson-datatype-jsr310"},
          {"name" => "ibm.jdk"},
          {"name" => "io.jaegertracing.jaeger"},
          {"name" => "io.opentracing.contrib.opentracing-tracerresolver"},
          ...
       ],
       "local-dependencies" => [
          {"name" => "deployment.ejb-in-ear.ear.ejb-in-ear-ejb.jar"},
```

```
            ...
        ],
        "user-dependencies" => [
            {"name" => "com.fasterxml.jackson.datatype.jackson-datatype-jdk8"},
            {"name" => "org.hibernate:4.1"},
            ...
        ]
    }
}
```

Using the **verbose=[false*|true]** attribute will result in a more detailed list.

Example: Detailed List Output

```
[standalone@localhost:9990 /] /deployment=sample-ear-1.0.ear:list-modules(verbose=true)
{
    "outcome" => "success",
    "result" => {
        "system-dependencies" => [
            {
                "name" => "com.fasterxml.jackson.datatype.jackson-datatype-jdk8",
                "optional" => true,
                "export" => false,
                "import-services" => true
            },
            {
                "name" => "com.fasterxml.jackson.datatype.jackson-datatype-jsr310",
                "optional" => true,
                "export" => false,
                "import-services" => true
            },
            ...
        ],
        "local-dependencies" => [
            {
              "name" => "deployment.ejb-in-ear.ear.ejb-in-ear-ejb.jar",
              "optional" => false,
              "export" => false,
              "import-services" => true
            },
            ...
        ],
        "user-dependencies" => [
            {
                "name" => "com.fasterxml.jackson.datatype.jackson-datatype-jdk8",
                "optional" => false,
                "export" => false,
                "import-services" => false
            },
            {
                "name" => "org.hibernate:4.1",
                "optional" => false,
                "export" => false,
                "import-services" => false
            },
            ...
```

The following table describes the categories of information provided in the output:

Table 3.1. Table Categories of Output forlist-modules Operation

Category	Description
system-dependencies	Added by server implicitly.
local-dependencies	Added by other parts of the deployment.
user-dependencies	Defined by user through the **MANIFEST.MF** or **deployment-structure.xml** file.

3.10. CLASS LOADING REFERENCE

3.10.1. 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 3.2. Implicit Module Dependencies

Subsystem Responsible for Adding the Dependency	Package Dependencies That Are Always Added	Package Dependencies That Are Conditionally Added	Conditions That Trigger the Addition of the Dependency
Application Client	• org.omg.api • org.jboss.xnio		
Batch	• javax.batch.api • org.jberet.jberet-core • org.wildfly.jberet		
Jakarta Bean Validation	• org.hibernate.validator • javax.validation.api		

Subsystem Responsible for Adding the Dependency	Package Dependencies That Are Always Added	Package Dependencies That Are Conditionally Added	Conditions That Trigger the Addition of the Dependency
Core Server	• javax.api • sun.jdk • org.jboss.vfs • ibm.jdk		
DriverDependenciesProcessor		• javax.transaction.api	

Subsystem Responsible for Adding the Dependency	Package Dependencies That Are Always Added	Package Dependencies That Are Conditionally Added	Conditions That Trigger the Addition of the Dependency
EE	• org.jboss.invocation (except org.jboss.invocation.proxy.classloading) • org.jboss.as.ee (except org.jboss.as.ee.component.serialization, org.jboss.as.ee.concurrent, org.jboss.as.ee.concurrent.handle) • org.wildfly.naming • javax.annotation.api • javax.enterprise.concurrent.api • javax.interceptor.api • javax.json.api • javax.resource.api • javax.rmi.api • javax.xml.bind.api • javax.api • org.glassfish.javax.el • org.glassfish.javax.enterprise.concurrent		

Subsystem Responsible for Adding the Dependency	Package Dependencies That Are Always Added	Package Dependencies That Are Conditionally Added	Conditions That Trigger the Addition of the Dependency
EJB 3	• javax.ejb.api • javax.xml.rpc.api • org.jboss.ejb-client • org.jboss.iiop-client • org.jboss.as.ejb3	• org.wildfly.iiop-openjdk	
IIOP	• org.omg.api • javax.rmi.api • javax.orb.api		

Subsystem Responsible for Adding the Dependency	Package Dependencies That Are Always Added	Package Dependencies That Are Conditionally Added	Conditions That Trigger the Addition of the Dependency
JAX-RS (RESTEasy)	• javax.xml.bind.api • javax.ws.rs.api • javax.json.api • org.jboss.resteasy.resteasy-atom-provider • org.jboss.resteasy.resteasy-crypto • org.jboss.resteasy.resteasy-validator-provider • org.jboss.resteasy.resteasy-jaxrs • org.jboss.resteasy.resteasy-jaxb-provider • org.jboss.resteasy.resteasy-jackson2-provider • org.jboss.resteasy.resteasy-jsapi • org.jboss.resteasy.resteasy-json-p-provider • org.jboss.resteasy.resteasy-multipart-provider • org.jboss.resteasy.resteasy-yaml-provider • org.codehaus.jackson.jackson-core-asl	• org.jboss.resteasy.resteasy-cdi	The presence of JAX-RS annotations in the deployment.

Subsystem Responsible for Adding the Dependency	Package Dependencies That Are Always Added	Package Dependencies That Are Conditionally Added	Conditions That Trigger the Addition of the Dependency
Jakarta Connectors	• javax.resource.api	• javax.jms.api • javax.validation.api • org.jboss.ironjacamar.api • org.jboss.ironjacamar.impl • org.hibernate.validator	The deployment of a resource adapter (RAR) archive.
Jakarta Persistence (Hibernate)	• javax.persistence.api	• org.jboss.as.jpa • org.jboss.as.jpa.spi • org.javassist	The presence of an **@PersistenceUnit** or **@PersistenceContext** annotation, or a **<persistence-unit-ref>** or **<persistence-context-ref>** element in a deployment descriptor. JBoss EAP maps persistence provider names to module names. If you name a specific provider in the **persistence.xml** file, a dependency is added for the appropriate module. If this not the desired behavior, you can exclude it using a **jboss-deployment-structure.xml** file.
Jakarta Server Faces		• javax.faces.api • com.sun.jsf-impl • org.jboss.as.jsf • org.jboss.as.jsf-injection	Added to EAR applications. Added to WAR applications only if the **web.xml** file does NOT specify a **context-param** of **org.jboss.jbossfaces.WAR_BUNDLES_JSF_IMPL** with a value of**true**.

Subsystem Responsible for Adding the Dependency	Package Dependencies That Are Always Added	Package Dependencies That Are Conditionally Added	Conditions That Trigger the Addition of the Dependency
JSR-77	• javax.management.j2ee.api		
Logging	• org.jboss.logging • org.apache.commons.logging • org.apache.log4j • org.slf4j • org.jboss.logging.jul-to-slf4j-stub		
Mail	• javax.mail.api • javax.activation.api		
Messaging	• javax.jms.api	• org.wildfly.extension.messaging-activemq	
PicketLink Federation		• org.picketlink	
Pojo	• org.jboss.as.pojo		
SAR		• org.jboss.modules • org.jboss.as.system-jmx • org.jboss.common-beans	The deployment of a SAR archive that has a **jboss-service.xml**.
Seam2		• org.jboss.vfs	.

Subsystem Responsible for Adding the Dependency	Package Dependencies That Are Always Added	Package Dependencies That Are Conditionally Added	Conditions That Trigger the Addition of the Dependency
Security	• org.picketbox • org.jboss.as.security • javax.security.jacc.api • javax.security.auth.message.api		
ServiceActivator		• org.jboss.msc	
Transactions	• javax.transaction.api	• org.jboss.xts • org.jboss.jts • org.jboss.narayana.compensations	
Undertow	• javax.servlet.jstl.api • javax.servlet.api • javax.servlet.jsp.api • javax.websocket.api	• io.undertow.core • io.undertow.servlet • io.undertow.jsp • io.undertow.websocket • io.undertow.js • org.wildfly.clustering.web.api	
Web Services	• javax.jws.api • javax.xml.soap.api • javax.xml.ws.api	• org.jboss.ws.api • org.jboss.ws.spi	If it is not application client type, then it will add the conditional dependencies.

Subsystem Responsible for Adding the Dependency	Package Dependencies That Are Always Added	Package Dependencies That Are Conditionally Added	Conditions That Trigger the Addition of the Dependency
Weld (CDI)	• javax.enterprise.api • javax.inject.api	• javax.persistence.api • org.javassist • org.jboss.as.weld • org.jboss.weld.core • org.jboss.weld.probe • org.jboss.weld.api • org.jboss.weld.spi • org.hibernate.validator.cdi	The presence of a **beans.xml** file in the deployment.

3.10.2. Included Modules

For the complete listing of the included modules and whether they are supported, see Red Hat JBoss Enterprise Application Platform 7 Included Modules on the Red Hat Customer Portal.

CHAPTER 4. LOGGING

4.1. ABOUT LOGGING

Logging is the practice of recording a series of messages from an application that provides 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 Java logging frameworks also include details such as the exact time and the origin of the message.

4.1.1. Supported Application Logging Frameworks

JBoss LogManager supports the following logging frameworks:

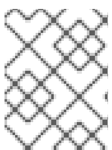
- JBoss Logging (included with JBoss EAP)
- Apache Commons Logging
- Simple Logging Facade for Java (SLF4J)
- Apache log4j
- Java SE Logging (java.util.logging)

JBoss LogManager supports the following APIs:

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

JBoss LogManager also supports the following SPIs:

- java.util.logging Handler
- Log4j Appender



NOTE

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

4.2. LOGGING WITH THE JBOSS LOGGING FRAMEWORK

4.2.1. About JBoss Logging

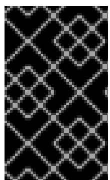
JBoss Logging is the application logging framework that is included in JBoss EAP. It provides an easy way to add logging to an application. You add code to your application that uses the framework to send log messages in a defined format. When the application is deployed to an application server, these messages can be captured by the server and displayed or written to file according to the server's configuration.

JBoss Logging provides the following features:

- An innovative, easy-to-use typed logger. A typed logger is a logger interface annotated with **org.jboss.logging.annotations.MessageLogger**. For examples, see Creating Internationalized Loggers, Messages and Exceptions.
- Full support for internationalization and localization. Translators work with message bundles in properties files while developers work with interfaces and annotations. For details, see Internationalization and Localization.
- Build-time tooling to generate typed loggers for production and runtime generation of typed loggers for development.

4.2.2. Add Logging to an Application with JBoss Logging

This procedure demonstrates how to add logging to an application using JBoss Logging.



IMPORTANT

If you use Maven to build your project, you must configure Maven to use the JBoss EAP Maven repository. For more information, see Configure the JBoss EAP Maven Repository.

1. The JBoss Logging JAR files must be in the build path for your application.
   - If you build using Red Hat CodeReady Studio, select **Properties** from the **Project** menu, then select **Targeted Runtimes** and ensure the runtime for JBoss EAP is checked.
   - If you use Maven to build your project, make sure you add the **jboss-logging** dependency to your project's **pom.xml** file for access to the JBoss Logging framework:

     ```
     <dependency>
      <groupId>org.jboss.logging</groupId>
      <artifactId>jboss-logging</artifactId>
      <version>3.3.0.Final-redhat-1</version>
      <scope>provided</scope>
     </dependency>
     ```

     The jboss-eap-jakartaee8 BOM manages the version of **jboss-logging**. For more details, see Manage Project Dependencies. See the **logging** quickstart that ships with JBoss EAP for a working example of logging in an application.

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

2. For each class to which you want to add logging:
   a. Add the import statements for the JBoss Logging class namespaces that you will be using. At a minimum you will need the following import:

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

b. Create an instance of **org.jboss.logging.Logger** and initialize it by calling the static method **Logger.getLogger(Class)**. It is recommended to create this as a single instance variable for each class.

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

3. Call the **Logger** object methods in your code where you want to send log messages. The **Logger** has many different methods with different parameters for different types of messages. Use the following methods to send a log message with the corresponding log level and the **message** parameter as a string:

```
LOGGER.debug("This is a debugging message.");
LOGGER.info("This is an informational message.");
LOGGER.error("Configuration file not found.");
LOGGER.trace("This is a trace message.");
LOGGER.fatal("A fatal error occurred.");
```

For the complete list of JBoss Logging methods, see the Logging API documentation.

The following example loads customized configuration for an application from a properties file. If the specified file is not found, an **ERROR** level log message is recorded.

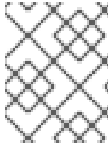
Example: Application Logging with JBoss Logging

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

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

    return props;
  }
}
```

4.3. PER-DEPLOYMENT LOGGING

Per-deployment logging allows a developer to configure the logging configuration for their application

in advance. When the application is deployed, logging begins according to the defined configuration. The log files created through this configuration contain information only about the behavior of the application.



NOTE

If the per-deployment logging configuration is not done, the configuration from **logging** subsystem is used for all the applications as well as the server.

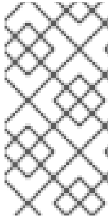
This approach has advantages and disadvantages over using system-wide logging. An advantage is that the administrator of the JBoss EAP instance does not need to configure any other logging than the server logging. A disadvantage is that the per-deployment logging configuration is read only on server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

4.3.1. Add Per-deployment Logging to an Application

To configure per-deployment logging for an application, add the **logging.properties** configuration file to your deployment. This configuration file is recommended because it can be used with any logging facade where JBoss Log Manager is the underlying log manager.

The directory into which the configuration file is added depends on the deployment method.

- For EAR deployments, copy the logging configuration file to the **META-INF/** directory.
- For WAR or JAR deployments, copy the logging configuration file to the **WEB-INF/classes/** directory.



NOTE

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

Configuring logging.properties

The **logging.properties** file is used when the server boots, until the **logging** subsystem is started. If the **logging** subsystem is not included in your configuration, then the server uses the configuration in this file as the logging configuration for the entire server.

JBoss Log Manager Configuration Options

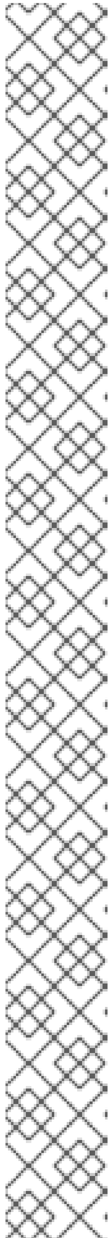
Logger options

- **loggers=<category>[,<category>,…]** - Specify a comma-separated list of logger categories to be configured. Any categories not listed here will not be configured from the following properties.
- **logger.<category>.level=<level>** - Specify the level for a category. The level can be one of the valid levels. If unspecified, the level of the nearest parent will be inherited.
- **logger.<category>.handlers=<handler>[,<handler>,…]** - Specify a comma-separated list of the handler names to be attached to this logger. The handlers must be configured in the same properties file.

- **logger.<category>.filter=<filter>** - Specify a filter for a category.
- **logger.<category>.useParentHandlers=(true|false)** - Specify whether log messages should cascade up to parent handlers. The default value is **true**.

Handler options

- **handler.<name>=<className>** - Specify the class name of the handler to instantiate. This option is mandatory.



NOTE

Table 4.1. Possible Class Names:

Name	Associated Class
Console	**org.jboss.logmanager.handlers.ConsoleHandler**
File	**org.jboss.logmanager.handlers.FileHandler**
Periodic	**org.jboss.logmanager.handlers.PeriodicRotatingFileHandler**
Size	**org.jboss.logmanager.handlers.SizeRotatingFileHandler**
Periodic Size	**org.jboss.logmanager.handlers.PeriodicSizeRotatingFileHandler**
Syslog	**org.jboss.logmanager.handlers.SyslogHandler**
Async	**org.jboss.logmanager.handlers.AsyncHandler**

The **Custom** handler can have any associated class or module. It is available in the **logging** subsystem for users to define their own log handlers.

For further information, see Log Handlers in the JBoss EAP *Configuration Guide*.

- **handler.<name>.level=<level>** - Restrict the level of this handler. If unspecified, the default value of ALL is retained.
- **handler.<name>.encoding=<encoding>** - Specify the character encoding, if it is supported by this handler type. If not specified, a handler-specific default is used.
- **handler.<name>.errorManager=<name>** - Specify the name of the error manager to use. The error manager must be configured in the same properties file. If unspecified, no error manager is configured.

- **handler.<name>.filter=<name>** - Specify a filter for a category. See the filter expressions for details on defining a filter.
- **handler.<name>.formatter=<name>** - Specify the name of the formatter to use, if it is supported by this handler type. The formatter must be configured in the same properties file. If not specified, messages will not be logged for most handler types.
- **handler.<name>.properties=<property>[,<property>,…]** - Specify a list of JavaBean-style properties to additionally configure. A rudimentary type introspection is done to ascertain the appropriate conversion for the given property.
  In case of all file handlers in JBoss Log Manager, **append** needs to be set before the **fileName**. The order in which the properties appear in **handler.<name>.properties**, is the order in which the properties will be set.
- **handler.<name>.constructorProperties=<property>[,<property>,…]** - Specify a list of properties that should be used as construction parameters. A rudimentary type introspection is done to ascertain the appropriate conversion for the given property.
- **handler.<name>.<property>=<value>** - Set the value of the named property.
- **handler.<name>.module=<name>** - Specify the name of the module the handler resides in.

For further information, see Log Handler Attributes in the JBoss EAP *Configuration Guide*.

Error manager options

- **errorManager.<name>=<className>** - Specify the class name of the error manager to instantiate. This option is mandatory.
- **errorManager.<name>.properties=<property>[,<property>,…]** - Specify a list of JavaBean-style properties to additionally configure. A rudimentary type introspection is done to ascertain the appropriate conversion for the given property.
- **errorManager.<name>.<property>=<value>** - Set the value of the named property.

Formatter options

- **formatter.<name>=<className>** - Specify the class name of the formatter to instantiate. This option is mandatory.
- **formatter.<name>.properties=<property>[,<property>,…]** - Specify a list of JavaBean-style properties to additionally configure. A rudimentary type introspection is done to ascertain the appropriate conversion for the given property.
- **formatter.<name>.constructorProperties=<property>[,<property>,…]** - Specify a list of properties that should be used as construction parameters. A rudimentary type introspection is done to ascertain the appropriate conversion for the given property.
- **formatter.<name>.<property>=<value>** - Set the value of the named property.

The following example shows the minimal configuration for **logging.properties** file that will log to the console.

Example: Minimal logging.properties Configuration

```
# Additional logger names to configure (root logger is always configured)
```

```
# loggers=

# Root logger level
logger.level=INFO

# Root logger handlers
logger.handlers=CONSOLE

# Console handler configuration
handler.CONSOLE=org.jboss.logmanager.handlers.ConsoleHandler
handler.CONSOLE.properties=autoFlush
handler.CONSOLE.autoFlush=true
handler.CONSOLE.formatter=PATTERN

# Formatter pattern configuration
formatter.PATTERN=org.jboss.logmanager.formatters.PatternFormatter
formatter.PATTERN.properties=pattern
formatter.PATTERN.pattern=%K{level}%d{HH:mm:ss,SSS} %-5p %C.%M(%L) [%c] %s%e%n
```

4.4. LOGGING PROFILES

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

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

For more information, see Configure a Logging Profile in the JBoss EAP *Configuration Guide*.

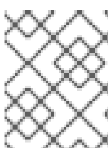
Each logging profile can have:

- A unique name. This value is required.
- Any number of log handlers.
- Any number of log categories.
- Up to one root logger.

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

4.4.1. Specify a Logging Profile in an Application

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



NOTE

You must know the name of the logging profile that has been set up on the server for this application to use.

To add a logging profile configuration to an application, edit the **MANIFEST.MF** file.

- If your application does not have a **MANIFEST.MF** file, create one with the following content to specify the logging profile name.

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

- If your application already has a **MANIFEST.MF** file, add the following line to specify the logging profile name.

```
Logging-Profile: LOGGING_PROFILE_NAME
```

> **NOTE**
>
> If you are using Maven and the **maven-war-plugin**, put your **MANIFEST.MF** file in **src/main/resources/META-INF/** and add the following configuration to your **pom.xml** file:
>
> ```
> <plugin>
>   <artifactId>maven-war-plugin</artifactId>
>   <configuration>
>     <archive>
>       <manifestFile>src/main/resources/META-INF/MANIFEST.MF</manifestFile>
>     </archive>
>   </configuration>
> </plugin>
> ```

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

For an example of how to configure a logging profile and the application using it, see Example Logging Profile Configuration in the JBoss EAP *Configuration Guide*.

4.5. INTERNATIONALIZATION AND LOCALIZATION

4.5.1. Introduction

4.5.1.1. About Internationalization

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

4.5.1.2. 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.

4.5.2. 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** annotations. Implementing the interfaces is not necessary; 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.

The JBoss Logging Tools includes support for the following features:

MessageLogger
: This interface in the **org.jboss.logging.annotations** package is used to define internationalized log messages. A message logger interface is annotated with **@MessageLogger**.

MessageBundle
: This interface 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 **@MessageBundle**.

Internationalized Log Messages
: These log messages are created by defining a method in a **MessageLogger**. The method must be annotated with the **@LogMessage** and **@Message** annotations and must 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.

Internationalized Exceptions
: An internationalized exception is an exception object returned from a method defined in a MessageBundle. These message bundles 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.

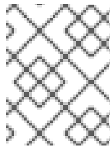
Internationalized Messages
: An internationalized message is a string returned from a method defined in a **MessageBundle**. Message bundle methods that return Java String objects can be annotated to define the default content of that string, known as the message. The default message is replaced with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

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.

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.



NOTE

For a complete list of the new log message project code prefixes, see the Project Codes used in JBoss EAP 7.3.

JBoss Logging Tools Message IDs

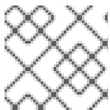
Message IDs are numbers that uniquely identify a log message when combined with a project code. Message IDs are displayed at the beginning of each log message, appended to the project code for the message. Message IDs are defined with the ID attribute of the **@Message** annotation.

The **logging-tools** quickstart that ships with JBoss EAP is a simple Maven project that provides a working example of many of the features of JBoss Logging Tools. The code examples that follow are taken from the **logging-tools** quickstart.

4.5.3. Creating Internationalized Loggers, Messages and Exceptions

4.5.3.1. Create Internationalized Log Messages

You can use JBoss Logging Tools to create internationalized log messages by creating **MessageLogger** interfaces.



NOTE

This section does not cover all optional features or the localization of the log messages.

1. If you have not yet done so, configure your Maven settings to use the JBoss EAP Maven repository.
   For more information, see Configure the JBoss EAP Maven Repository Using the Maven Settings.

2. Configure the project's **pom.xml** file to use JBoss Logging Tools.
   For details, see JBoss Logging Tools Maven Configuration .

3. Create a message logger interface by adding a Java interface to your project to contain the log message definitions.
   Name the interface to describe the log messages it will define. The log message interface has the following requirements:

   - It must be annotated with **@org.jboss.logging.annotations.MessageLogger**.
   - Optionally, it can extend **org.jboss.logging.BasicLogger**.
   - The interface must define a field that is a message logger of the same type as the interface. Do this with the **getMessageLogger()** method of **@org.jboss.logging.Logger**.

Example: Creating a Message Logger

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

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

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

4. 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.annotation.LogMessage** annotation.
   - It must be annotated with the **@org.jboss.logging.annotations.Message** annotation.
   - The default log level is **INFO**.
   - The value attribute of **@org.jboss.logging.annotations.Message** contains the default log message, which is used if no translation is available.

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

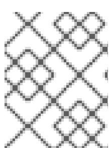
5. Invoke the methods by adding the calls to the interface methods in your code where the messages must be logged from.
   Creating implementations of the interfaces is not necessary, the annotation processor does this for you when the project is compiled.

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

   The custom loggers are subclassed from **BasicLogger**, so the logging methods of **BasicLogger** can also be used. It is not necessary to create other loggers to log non-internationalized messages.

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

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

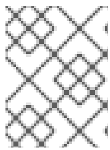


NOTE

The **logging-tools** quickstart that ships with JBoss EAP is a simple Maven project that provides a working example of how to use JBoss Logging Tools.

4.5.3.2. Create and Use Internationalized Messages

This procedure demonstrates how to create and use internationalized messages.



NOTE

This section does not cover all optional features or the process of localizing those messages.

1. If you have not yet done so, configure your Maven settings to use the JBoss EAP Maven repository. For more information, see Configure the JBoss EAP Maven Repository Using the Maven Settings.

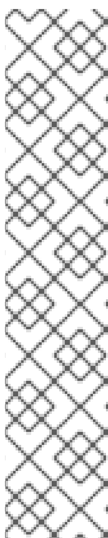
2. Configure the project's **pom.xml** file to use JBoss Logging Tools. For details, see JBoss Logging Tools Maven Configuration.

3. Create an interface for the exceptions. JBoss Logging Tools defines internationalized messages in interfaces. Name each interface descriptively for the messages that it contains. The interface has the following requirements:

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

   Example: Create a MessageBundle Interface

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

   NOTE

   Calling **Messages.getBundle(GreetingMessagesBundle.class)** is equivalent to calling **Messages.getBundle(GreetingMessagesBundle.class, Locale.getDefault())**.

   Locale.getDefault() gets the current value of the default locale for this instance of the Java Virtual Machine. The Java Virtual Machine sets the default locale during startup, based on the host environment. It is used by many locale-sensitive methods if no locale is explicitly specified. It can be changed using the **setDefault** method.

   See Set the Default Locale of the Server in the JBoss EAP *Configuration Guide* for more information.

4. 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**.

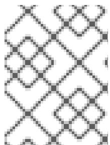
- It must be annotated with the **@org.jboss.logging.annotations.Message** annotation.
- The value attribute of **@org.jboss.logging.annotations.Message** must be set to the default message. This is the message that is used if no translation is available.

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

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

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

The project now supports internationalized message strings that can be localized.



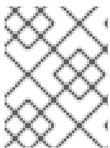
NOTE

See the **logging-tools** quickstart that ships with JBoss EAP for a complete working example.

4.5.3.3. Create Internationalized Exceptions

You can use JBoss Logging Tools to create and use internationalized exceptions.

The following instructions assume that you want to add internationalized exceptions to an existing software project that is built using either Red Hat CodeReady Studio or Maven.



NOTE

This section does not cover all optional features or the process of localization of those exceptions.

1. Configure the project's **pom.xml** file to use JBoss Logging Tools. For details, see 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 it defines. The interface has the following requirements:
   - It must be declared as **public**.
   - It must be annotated with **@MessageBundle**.
   - The interface must define a field that is a message bundle of the same type as the interface.

   Example: Create an ExceptionBundle Interface

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

3. 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 **Exception** object, or a sub-type of **Exception**.

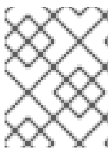
- It must return an **Exception** object, or a sub-type of **Exception**.
- It must be annotated with the **@org.jboss.logging.annotations.Message** annotation.
- The value attribute of **@org.jboss.logging.annotations.Message** must be set to the default exception message. This is the message that is 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 they are in the constructor of the exception.

```
@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 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();
}
```

The project now supports internationalized exceptions that can be localized.



NOTE

See the **logging-tools** quickstart that ships with JBoss EAP for a complete working example.

4.5.4. Localizing Internationalized Loggers, Messages and Exceptions

4.5.4.1. Generate New Translation Properties Files with Maven

Projects that are built using Maven can generate empty translation property files for each **MessageLogger** and **MessageBundle** it contains. These files can then be used as new translation property files.

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

Prerequisites

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

- The project must contain one or more interfaces that define internationalized log messages or exceptions.

Generate the Translation Properties Files

1. Add the Maven configuration by adding the **-AgenereatedTranslationFilePath** compiler argument to the Maven compiler plug-in configuration, and assign it the path where the new files will be created.
   This configuration creates the new files in the **target/generated-translation-files** directory of your Maven project.

   #### Example: Define the Translation File Path

```
<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>
```

2. Build the project using Maven:

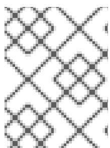
```
$ mvn compile
```

   One properties file is created for each interface annotated with **@MessageBundle** or **@MessageLogger**.

   - The new files are created in a subdirectory corresponding to the Java package in which each interface is declared.
   - Each new file is named using the following pattern where **INTERFACE_NAME** is the name of the interface used to generated the file.

```
INTERFACE_NAME.i18n_locale_COUNTRY_VARIANT.properties
```

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



NOTE

See the **logging-tools** quickstart that ships with JBoss EAP for a complete working example.

4.5.4.2. Translate an Internationalized Logger, Exception, or Message

Properties files can be used to provide translations for logging and exception messages defined in interfaces using JBoss Logging Tools.

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

Prerequisites

- You must already have a working Maven project.
- The project must already be configured for JBoss Logging Tools.
- The project must contain one or more interfaces that define internationalized log messages or exceptions.
- The project must be configured to generate template translation property files.

Translate an Internationalized Logger, Exception, or Message

1. Run the following command to create the template translation properties files:

```
$ mvn compile
```

2. 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 to indicate the language it will contain. For example: **GreeterLogger.i18n_fr_FR.properties**.

4. Edit the contents of the new translation properties file to contain the appropriate translation:

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

5. Repeat the process of copying the template and modifying it for each translation in the bundle.

The project now contains translations for one or more message or logger bundles. Building the project generates 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/**.

4.5.5. Customizing Internationalized Log Messages

4.5.5.1. Add Message IDs and Project Codes to Log Messages

This procedure demonstrates 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 to be displayed in the log. If a message does not have both a project code and a message ID, then neither is displayed.

Prerequisites

1. You must already have a project with internationalized log messages. For details, see Create Internationalized Log Messages.

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

Add Message IDs and Project Codes to Log Messages

1. Specify the project code for the interface by 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 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();
```

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

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

4.5.5.2. 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. Use the following procedure to specify a different log level.

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

2. Assign the log level for this message using the **level** attribute. 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 example 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.
```

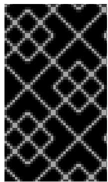
4.5.5.3. Customize Log Messages with Parameters

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.

Customize Log Messages with Parameters

1. Add parameters of any type 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 **%s** characters in the message string where you want each parameter to appear. The first instance of **%s** will insert the first parameter, the second instance will insert the second parameter, and so on.
   - To use explicit indexes, insert **%#$s** characters in the message, where # indicates the number of the parameter that you wish to appear.

Using explicit indexes allows the parameter references in the message to be in a different order than they are defined in the method. This is important for translated messages that might 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.

The following is an example of 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);
```

The following is an example of message parameters using explicit indexes:

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

4.5.5.4. 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 the **Throwable** type 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.

Specify an Exception as the Cause of a Log Message

1. Add a parameter of the type **Throwable** or its subclass to the method.

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

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

```
import org.jboss.logging.annotations.Cause

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

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

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

The following is the output of the above code example if the code throws 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)
```

4.5.6. Customizing Internationalized Exceptions

4.5.6.1. Add Message IDs and Project Codes to Exception Messages

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 for all the exception messages in an application. This allows someone to look up the meaning of an exception message written in language that they do not understand.

The following procedure demonstrates how to add message IDs and project codes to internationalized exception messages created using JBoss Logging Tools.

Prerequisites

1. You must already have a project with internationalized exceptions. For details, see Create Internationalized Exceptions.
2. You need to know the project code you will be using. You can use a single project code, or define different ones for each interface.

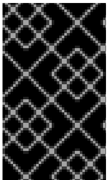
Add Message IDs and Project Codes to Exception Messages

1. 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 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: Internationalized Exception

This exception bundle interface example uses the project code of "ACCTS". It contains 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)
```

■

4.5.6.2. 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. The exact position of the parameters in the exception message is specified in the message itself using either explicit or ordinary indexing.

Customize Exception Messages with Parameters

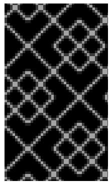
1. Add parameters of any type to the method definition. Regardless of type, the String representation of the parameter is what is displayed in the message.

2. Add parameter references to the exception message. References can use explicit or ordinary indexes.

   - To use ordinary indexes, insert **%s** characters in the message string where you want each parameter to appear. The first instance of **%s** will insert the first parameter, the second instance will insert the second parameter, and so on.

   - To use explicit indexes, insert **%#$s** characters in the message, where # indicates the number of the parameter that you wish to appear.

Using explicit indexes allows the parameter references in the message to be in a different order than they are defined in the method. This is important for translated messages that might 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: Using Ordinary Indexes

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

Example: Using Explicit Indexes

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

4.5.6.3. 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, and 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.

1. Add a parameter of the type **Throwable** or its subclass to the method.

```
@Message(id=328, value = "Error calculating: %s.")
ArithmeticException calculationError(Throwable cause, String msg);
```

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

```
import org.jboss.logging.annotations.Cause

@Message(id=328, value = "Error calculating: %s.")
ArithmeticException calculationError(@Cause Throwable cause, String msg);
```

3. Invoke the interface method to obtain an exception object. The most common use case is to throw a new exception from a **catch** block, specifying the caught exception as the cause.

```
try
{
   ...
}
catch(Exception ex)
{
   throw ExceptionBundle.EXCEPTIONS.calculationError(
                                    ex, "calculating payment due per day");
}
```

The following is an example of specifying an 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);
}
```

The following example demonstrates 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");
}
```

The following is the exception message generated from the previous example:

```
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
```

4.5.7. JBoss Logging Tools References

4.5.7.1. JBoss Logging Tools Maven Configuration

The following procedure configures a Maven project to use JBoss Logging and JBoss Logging Tools for internationalization.

1. If you have not yet done so, configure your Maven settings to use the JBoss EAP repository. For more information, see Configure the JBoss EAP Maven Repository Using the Maven Settings.
   Include the **jboss-eap-jakartaee8** BOM in the **<dependencyManagement>** section of the project's **pom.xml** file.

   ```
   <dependencyManagement>
     <dependencies>
       <!-- JBoss distributes a complete set of Jakarta EE APIs including
         a Bill of Materials (BOM). A BOM specifies the versions of a "stack" (or
         a collection) of artifacts. We use this here so that we always get the correct versions of
   artifacts.
         Here we use the jboss-javaee-7.0 stack (you can
         read this as the JBoss stack of the Jakarta EE APIs). You can actually
         use this stack with any version of JBoss EAP that implements Jakarta EE. -->
       <dependency>
         <groupId>org.jboss.bom</groupId>
          <artifactId>jboss-eap-jakartaee8</artifactId>
          <version>7.3.0.GA</version>
          <type>pom</type>
          <scope>import</scope>
       </dependency>
     <dependencies>
   <dependencyManagement>
   ```

2. Add the Maven dependencies to the project's **pom.xml** file:

   a. Add the **jboss-logging** dependency for access to JBoss Logging framework.

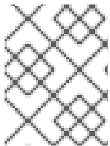
   b. If you plan to use the JBoss Logging Tools, also add the **jboss-logging-processor** dependency.
      Both of these dependencies are available in JBoss EAP BOM that was added in the previous step, so the scope element of each can be set to **provided** as shown.

      ```
      <!-- Add the JBoss Logging Tools dependencies -->
      <!-- The jboss-logging API -->
      <dependency>
         <groupId>org.jboss.logging</groupId>
         <artifactId>jboss-logging</artifactId>
      ```

```xml
    <scope>provided</scope>
  </dependency>
  <!-- Add the jboss-logging-tools processor if you are using JBoss Tools  -->
  <dependency>
    <groupId>org.jboss.logging</groupId>
    <artifactId>jboss-logging-processor</artifactId>
    <scope>provided</scope>
  </dependency>
```

3. The maven-compiler-plugin must be at least version **3.1** and configured for target and generated sources of **1.8**.

```xml
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.1</version>
  <configuration>
    <source>1.8</source>
    <target>1.8</target>
  </configuration>
</plugin>
```



NOTE

For a complete working example of a **pom.xml** file that is configured to use JBoss Logging Tools, see the **logging-tools** quickstart that ships with JBoss EAP.

4.5.7.2. Translation Property File Format

The property files used for the translation 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 **java.util.Properties** class documentation.

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: Translation Properties File

File name: **GreeterService.i18n_fr_FR_POSIX.properties**.

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

4.5.7.3. 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 4.2. 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 bundle it defines the method as being one that returns a localized String or Exception object. In a message logger it defines a method as being a localized logger.	value, id
@LogMessage	Method	Defines a method in a message logger as being 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.	-

4.5.7.4. Project Codes Used in JBoss EAP

The following table lists all the project codes used in JBoss EAP 7.3, along with the Maven modules they belong to.

Table 4.3. Project Codes Used in JBoss EAP

Maven Module	Project Code
appclient	WFLYAC

Maven Module	Project Code
batch/extension-jberet	WFLYBATCH
batch/extension	WFLYBATCH-DEPRECATED
batch/jberet	WFLYBAT
bean-validation	WFLYBV
controller-client	WFLYCC
controller	WFLYCTL
clustering/common	WFLYCLCOM
clustering/ejb/infinispan	WFLYCLEJBINF
clustering/infinispan/extension	WFLYCLINF
clustering/jgroups/extension	WFLYCLJG
clustering/server	WFLYCLSV
clustering/web/infinispan	WFLYCLWEBINF
connector	WFLYJCA
deployment-repository	WFLYDR
deployment-scanner	WFLYDS
domain-http	WFLYDMHTTP
domain-management	WFLYDM
ee	WFLYEE
ejb3	WFLYEJB
embedded	WFLYEMB
host-controller	WFLYDC
host-controller	WFLYHC
iiop-openjdk	WFLYIIOP

Maven Module	Project Code
io/subsystem	WFLYIO
jaxrs	WFLYRS
jdr	WFLYJDR
jmx	WFLYJMX
jpa/hibernate5	JIPI
jpa/spi/src/main/java/org/jipijapa/JipiLogger.java	JIPI
jpa/subsystem	WFLYJPA
jsf/subsystem	WFLYJSF
jsr77	WFLYEEMGMT
launcher	WFLYLNCHR
legacy/jacorb	WFLYORB
legacy/messaging	WFLYMSG
legacy/web	WFLYWEB
logging	WFLYLOG
mail	WFLYMAIL
management-client-content	WFLYCNT
messaging-activemq	WFLYMSGAMQ
mod_cluster/extension	WFLYMODCLS
naming	WFLYNAM
network	WFLYNET
patching	WFLYPAT
picketlink	WFLYPL
platform-mbean	WFLYPMB

Maven Module	Project Code
pojo	WFLYPOJO
process-controller	WFLYPC
protocol	WFLYPRT
remoting	WFLYRMT
request-controller	WFLYREQCON
rts	WFLYRTS
sar	WFLYSAR
security-manager	WFLYSM
security	WFLYSEC
server	WFLYSRV
system-jmx	WFLYSYSJMX
threads	WFLYTHR
transactions	WFLYTX
undertow	WFLYUT
webservices/server-integration	WFLYWS
weld	WFLYWELD
xts	WFLYXTS

CHAPTER 5. REMOTE JNDI LOOKUP

5.1. REGISTERING OBJECTS TO JAVA NAMING AND DIRECTORY INTERFACE

The Java Naming and Directory Interface is a Java API for a directory service that allows Java software clients to discover and look up objects using a name.

If an object registered to Java Naming and Directory Interface needs to be looked up by remote Java Naming and Directory Interface clients, for example clients that run in a separate JVM, then it must be registered under the **java:jboss/exported** context.

For example, if a Jakarta Messaging queue in the **messaging-activemq** subsystem must be exposed for remote Java Naming and Directory Interface clients, then it must be registered to Java Naming and Directory Interface using **java:jboss/exported/jms/queue/myTestQueue**. The remote Java Naming and Directory Interface client can then look it up by the name **jms/queue/myTestQueue**.

Example: Configuration of the Queue in standalone-full(-ha).xml

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <jms-queue name="myTestQueue" entries="java:jboss/exported/jms/queue/myTestQueue"/>
    ...
  </server>
</subsystem>
```

5.2. CONFIGURING REMOTE JNDI

A remote JNDI client can connect and look up objects by name from JNDI. To use a remote JNDI client to look up objects, it must have the **jboss-client.jar** in its class path. The **jboss-client.jar** is available at ***EAP_HOME*/bin/client/jboss-client.jar**.

The following example shows how to look up the **myTestQueue** queue from JNDI in a remote JNDI client:

Example: Configuration for an MDB Resource Adapter

```
Properties properties = new Properties();
properties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.wildfly.naming.client.WildFlyInitialContextFactory");
properties.put(Context.PROVIDER_URL, "remote+http://HOST_NAME:8080");
context = new InitialContext(properties);
Queue myTestQueue = (Queue) context.lookup("jms/queue/myTestQueue");
```

5.3. JNDI INVOCATION OVER HTTP

JNDI invocation over HTTP includes two distinct parts: the client-side and the server-side implementations.

5.3.1. Client-side Implementation

The client-side implementation is similar to the remote naming implementation, but based on HTTP using the Undertow HTTP client.

Connection management is implicit rather than direct, using a caching approach similar to the one used in the existing remote naming implementation. Connection pools are cached based on connection parameters. If they are not used in the specified timeout period, they are discarded.

In order to configure a remote JNDI client application to use HTTP transport, you must add the following dependency on the HTTP transport implementation:

```
<dependency>
    <groupId>org.wildfly.wildfly-http-client</groupId>
    <artifactId>wildfly-http-naming-client</artifactId>
</dependency>
```

To perform the HTTP invocation, you must use the **http** URL scheme and include the context name of the HTTP invoker, **wildfly-services**. For example, if you are using **remote+http://localhost:8080** as the target URL, in order to use HTTP transport, you must update this to **http://localhost:8080/wildfly-services**.

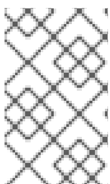
5.3.2. Server-side Implementation

The server-side implementation is similar to the existing remote naming implementation but with an HTTP transport.

In order to configure the server, you must enable the **http-invoker** on each of the virtual hosts that you wish to use in the **undertow** subsystem. This is enabled by default in the standard configurations. If it is disabled, you can re-enable it using the following management CLI command:

```
/subsystem=undertow/server=default-server/host=default-host/setting=http-invoker:add(http-authentication-factory=myfactory, path="/wildfly-services")
```

The **http-invoker** attribute takes two parameters: a **path** that defaults to **/wildfly-services** and an **http-authentication-factory** that must be a reference to an Elytron **http-authentication-factory**.



NOTE

Any deployment that aims to use the **http-authentication-factory** must use Elytron security with the same security domain corresponding to the specified HTTP authentication factory.

CHAPTER 6. CLUSTERING IN WEB APPLICATIONS

6.1. SESSION REPLICATION

6.1.1. About HTTP Session Replication

Session replication ensures that client sessions of distributable applications are not disrupted by failovers of nodes in a cluster. Each node in the cluster shares information about ongoing sessions, and can take over sessions if a node disappears.

Session replication is the mechanism by which mod_cluster, mod_jk, mod_proxy, ISAPI, and NSAPI clusters provide high availability.

6.1.2. Enable Session Replication in Your Application

To take advantage of JBoss EAP High Availability (HA) features and enable clustering of your web application, you must configure your application to be distributable. If your application is not marked as distributable, its sessions will never be distributed.

Make your Application Distributable

1. Add the **<distributable/>** element inside the **<web-app>** tag of your application's **web.xml** descriptor file:

   Example: Minimum Configuration for a Distributable Application

```
<?xml version="1.0"?>
<web-app  xmlns="http://java.sun.com/xml/ns/j2ee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                      http://java.sun.com/xml/ns/j2ee/web-app_3_0.xsd"
          version="3.0">

      <distributable/>

</web-app>
```

2. Next, if desired, modify the default replication behavior. If you want to change any of the values affecting session replication, you can override them inside a **<replication-config>** element inside **<jboss-web>** in an application's **WEB-INF/jboss-web.xml** file. For a given element, only include it if you want to override the defaults.

   Example: <replication-config> Values

```
<jboss-web xmlns="http://www.jboss.com/xml/ns/javaee"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-web_10_0.xsd">
   <replication-config>
      <replication-granularity>SESSION</replication-granularity>
    </replication-config>
</jboss-web>
```

The **<replication-granularity>** parameter determines the granularity of data that is replicated. It defaults to **SESSION**, but can be set to **ATTRIBUTE** to increase performance on sessions where most attributes remain unchanged.

Valid values for **<replication-granularity>** can be :

- **SESSION**: The default value. The entire session object is replicated if any attribute is dirty. This policy is required if an object reference is shared by multiple session attributes. The shared object references are maintained on remote nodes since the entire session is serialized in one unit.
- **ATTRIBUTE**: This is only for dirty attributes in the session and for some session data, such as the last-accessed timestamp.

Immutable Session Attributes

For JBoss EAP 7, session replication is triggered when the session is mutated or when any mutable attribute of the session is accessed. Session attributes are assumed to be mutable unless one of the following is true:

- The value is a known immutable value:
  - **null**
  - **java.util.Collections.EMPTY_LIST**, **EMPTY_MAP**, **EMPTY_SET**
- The value type is or implements a known immutable type:
  - **java.lang.Boolean**, **Character**, **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**
  - **java.lang.Class**, **Enum**, **StackTraceElement**, **String**
  - **java.io.File**, **java.nio.file.Path**
  - **java.math.BigDecimal**, **BigInteger**, **MathContext**
  - **java.net.Inet4Address**, **Inet6Address**, **InetSocketAddress**, **URI**, **URL**
  - **java.security.Permission**
  - **java.util.Currency**, **Locale**, **TimeZone**, **UUID**
  - **java.time.Clock**, **Duration**, **Instant**, **LocalDate**, **LocalDateTime**, **LocalTime**, **MonthDay**, **Period**, **Year**, **YearMonth**, **ZoneId**, **ZoneOffset**, **ZonedDateTime**
  - **java.time.chrono.ChronoLocalDate**, **Chronology**, **Era**
  - **java.time.format.DateTimeFormatter**, **DecimalStyle**
  - **java.time.temporal.TemporalField**, **TemporalUnit**, **ValueRange**, **WeekFields**
  - **java.time.zone.ZoneOffsetTransition**, **ZoneOffsetTransitionRule**, **ZoneRules**
- The value type is annotated with:
  - **@org.wildfly.clustering.web.annotation.Immutable**
  - **@net.jcip.annotations.Immutable**

6.2. HTTP SESSION PASSIVATION AND ACTIVATION

6.2.1. 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 different times in an HTTP session's lifetime:

- When the container requests the creation of a new session, if the number of currently active sessions exceeds a configurable limit, the server attempts to passivate some sessions to make room for the new one.

- 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 might be passivated.

A session is passivated if the number of active sessions exceeds a configurable maximum.

Sessions are always passivated using a Least Recently Used (LRU) algorithm.

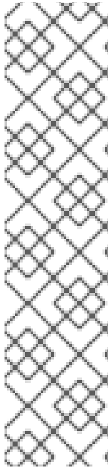
6.2.2. Configure HTTP Session Passivation in Your Application

HTTP session passivation is configured in your application's **WEB-INF/jboss-web.xml** and **META-INF/jboss-web.xml** file.

Example: jboss-web.xml File

```
<jboss-web xmlns="http://www.jboss.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-web_10_0.xsd">

   <max-active-sessions>20</max-active-sessions>
</jboss-web>
```

The **<max-active-sessions>** element dictates the maximum number of active sessions allowed, and is used to enable session passivation. If session creation would cause the number of active sessions to exceed **<max-active-sessions>**, then the oldest session known to the session manager will passivate to make room for the new session.



NOTE

The total number of sessions in memory includes sessions replicated from other cluster nodes that are not being accessed on this node. Take this into account when setting **<max-active-sessions>**. The number of sessions replicated from other nodes also depends on whether **REPL** or **DIST** cache mode is enabled. In **REPL** cache mode, each session is replicated to each node. In **DIST** cache mode, each session is replicated only to the number of nodes specified by the **owners** parameter. See Configure the Cache Mode in the JBoss EAP *Configuration Guide* for information on configuring session cache modes. For example, consider an eight node cluster, where each node handles requests from 100 users. With **REPL** cache mode, each node would store 800 sessions in memory. With **DIST** cache mode enabled, and the default **owners** setting of **2**, each node stores 200 sessions in memory.

6.3. PUBLIC API FOR CLUSTERING SERVICES

JBoss EAP 7 introduced a refined public clustering API for use by applications. The new services are designed to be lightweight, easily injectable, with no external dependencies.

org.wildfly.clustering.group.Group

The group service provides a mechanism to view the cluster topology for a JGroups channel, and to be notified when the topology changes.

```
@Resource(lookup = "java:jboss/clustering/group/channel-name")
private Group channelGroup;
```

org.wildfly.clustering.dispatcher.CommandDispatcher

The **CommandDispatcherFactory** service provides a mechanism to create a dispatcher for executing commands on nodes in the cluster. The resulting **CommandDispatcher** is a command-pattern analog to the reflection-based **GroupRpcDispatcher** from previous JBoss EAP releases.

```
@Resource(lookup = "java:jboss/clustering/dispatcher/channel-name")
private CommandDispatcherFactory factory;

public void foo() {
    String context = "Hello world!";
    // Exclude node1 and node3 from the executeOnCluster
    try (CommandDispatcher<String> dispatcher = this.factory.createCommandDispatcher(context))
{
        dispatcher.executeOnGroup(new StdOutCommand(), node1, node3);

    }
}

public static class StdOutCommand implements Command<Void, String> {
    @Override
    public Void execute(String context) {
        System.out.println(context);
        return null;
    }
}
```

6.4. HA SINGLETON SERVICE

A clustered singleton service, also known as a high-availability (HA) singleton, is a service deployed on multiple nodes in a cluster. The service is provided on only one of the nodes. The node running the singleton service is usually called the *master* node.

When the *master* node either fails or shuts down, another master is selected from the remaining nodes and the service is restarted on the new master. Other than a brief interval when one master has stopped and another has yet to take over, the service is provided by one, and only one, node.

HA Singleton ServiceBuilder API

JBoss EAP 7 introduced a new public API for building singleton services that simplifies the process significantly.

The **SingletonServiceConfigurator** implementation installs its services so they will start asynchronously, preventing deadlocking of the Modular Service Container (MSC).

HA Singleton Service Election Policies

If there is a preference for which node should start the HA singleton, you can set the election policy in the **ServiceActivator** class.

JBoss EAP provides two election policies:

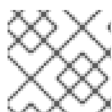
- Simple election policy
  The simple election policy selects a master node based on the relative age. The required age is configured in the position property, which is the index in the list of available nodes, where:
  - position = 0 – refers to the oldest node. This is the default.
  - position = 1 – refers to the 2nd oldest, and so on.

  Position can also be negative to indicate the youngest nodes.
  - position = -1 – refers to the youngest node.
  - position = -2 – refers to the 2nd youngest node, and so on.
- Random election policy
  The random election policy elects a random member to be the provider of a singleton service.

HA Singleton Service Preferences

An HA singleton service election policy may optionally specify one or more preferred servers. This preferred server, when available, will be the master for all singleton applications under that policy.

You can define the preferences either through the node name or through the outbound socket binding name.



NOTE

Node preferences always take precedence over the results of an election policy.

By default, JBoss EAP high availability configurations provide a simple election policy named **default** with no preferred server. You can set the preference by creating a custom policy and defining the preferred server.

Quorum

A potential issue with a singleton service arises when there is a network partition. In this situation, also known as the split-brain scenario, subsets of nodes cannot communicate with each other. Each set of servers consider all servers from the other set failed and continue to work as the surviving cluster. This might result in data consistency issues.

JBoss EAP allows you to specify a quorum in the election policy to prevent the split-brain scenario. The quorum specifies a minimum number of nodes to be present before a singleton provider election can take place.

A typical deployment scenario uses a quorum of N/2 + 1, where N is the anticipated cluster size. This value can be updated at runtime, and will immediately affect any active singleton services.

HA Singleton Service Election Listener

After electing a new primary singleton service provider, any registered **SingletonElectionListener** is triggered, notifying every member of the cluster about the new primary provider. The following example illustrates the usage of **SingletonElectionListener**:

```
public class MySingletonElectionListener implements SingletonElectionListener {
    @Override
    public void elected(List<Node> candidates, Node primary) {
        // ...
    }
}

public class MyServiceActivator implements ServiceActivator {
    @Override
    public void activate(ServiceActivatorContext context) {
        String containerName = "foo";
        SingletonElectionPolicy policy = new MySingletonElectionPolicy();
        SingletonElectionListener listener = new MySingletonElectionListener();
        int quorum = 3;
        ServiceName name = ServiceName.parse("my.service.name");
        // Use a SingletonServiceConfiguratorFactory backed by default cache of "foo" container
        Supplier<SingletonServiceConfiguratorFactory> factory = new
ActiveServiceSupplier<SingletonServiceConfiguratorFactory>(context.getServiceRegistry(),
ServiceName.parse(SingletonDefaultCacheRequirement.SINGLETON_SERVICE_CONFIGURATOR_
FACTORY.resolve(containerName)));
        ServiceBuilder<?> builder = factory.get().createSingletonServiceConfigurator(name)
            .electionListener(listener)
            .electionPolicy(policy)
            .requireQuorum(quorum)
            .build(context.getServiceTarget());
        Service service = new MyService();
        builder.setInstance(service).install();
    }
}
```

Create an HA Singleton Service Application

The following is an abbreviated example of the steps required to create and deploy an application as a cluster-wide singleton service. This example demonstrates a querying service that regularly queries a singleton service to get the name of the node on which it is running.

To see the singleton behavior, you must deploy the application to at least two servers. It is transparent whether the singleton service is running on the same node or whether the value is obtained remotely.

1. Create the **SingletonService** class. The **getValue()** method, which is called by the querying service, provides information about the node on which it is running.

```
class SingletonService implements Service {
    private Logger LOG = Logger.getLogger(this.getClass());
    private Node node;

    private Supplier<Group> groupSupplier;
    private Consumer<Node> nodeConsumer;

    SingletonService(Supplier<Group> groupSupplier, Consumer<Node> nodeConsumer) {
      this.groupSupplier = groupSupplier;
      this.nodeConsumer = nodeConsumer;
    }

    @Override
    public void start(StartContext context) {
       this.node = this.groupSupplier.get().getLocalMember();

       this.nodeConsumer.accept(this.node);

       LOG.infof("Singleton service is started on node '%s'.", this.node);
    }

    @Override
    public void stop(StopContext context) {
       LOG.infof("Singleton service is stopping on node '%s'.", this.node);

       this.node = null;
    }
}
```

2. Create the querying service. It calls the **getValue()** method of the singleton service to get the name of the node on which it is running, and then writes the result to the server log.

```
class QueryingService implements Service {
    private Logger LOG = Logger.getLogger(this.getClass());
    private ScheduledExecutorService executor;

    @Override
    public void start(StartContext context) throws {
       LOG.info("Querying service is starting.");

       executor = Executors.newSingleThreadScheduledExecutor();
       executor.scheduleAtFixedRate(() -> {

       Supplier<Node> node = new PassiveServiceSupplier<>
(context.getController().getServiceContainer(),
SingletonServiceActivator.SINGLETON_SERVICE_NAME);
         if (node.get() != null) {
            LOG.infof("Singleton service is running on this (%s) node.", node.get());
         } else {
            LOG.infof("Singleton service is not running on this node.");
         }

       }, 5, 5, TimeUnit.SECONDS);
```

```
    }

    @Override
    public void stop(StopContext context) {
        LOG.info("Querying service is stopping.");

        executor.shutdown();
    }
}
```

3. Implement the **SingletonServiceActivator** class to build and install both the singleton service and the querying service.

```
public class SingletonServiceActivator implements ServiceActivator {

    private final Logger LOG = Logger.getLogger(SingletonServiceActivator.class);

    static final ServiceName SINGLETON_SERVICE_NAME =
            ServiceName.parse("org.jboss.as.quickstarts.ha.singleton.service");
    private static final ServiceName QUERYING_SERVICE_NAME =
            ServiceName.parse("org.jboss.as.quickstarts.ha.singleton.service.querying");

    @Override
    public void activate(ServiceActivatorContext serviceActivatorContext) {
        SingletonPolicy policy = new ActiveServiceSupplier<SingletonPolicy>(
                serviceActivatorContext.getServiceRegistry(),
                ServiceName.parse(SingletonDefaultRequirement.POLICY.getName())).get();

        ServiceTarget target = serviceActivatorContext.getServiceTarget();
        ServiceBuilder<?> builder =
policy.createSingletonServiceConfigurator(SINGLETON_SERVICE_NAME).build(target);
        Consumer<Node> member = builder.provides(SINGLETON_SERVICE_NAME);
        Supplier<Group> group =
builder.requires(ServiceName.parse("org.wildfly.clustering.default-group"));
        builder.setInstance(new SingletonService(group, member)).install();

        serviceActivatorContext.getServiceTarget()
                .addService(QUERYING_SERVICE_NAME, new QueryingService())
                .setInitialMode(ServiceController.Mode.ACTIVE)
                .install();


serviceActivatorContext.getServiceTarget().addService(QUERYING_SERVICE_NAME).setInst
ance(new QueryingService()).install();

        LOG.info("Singleton and querying services activated.");
    }
}
```

4. Create a file in the **META-INF/services/** directory named **org.jboss.msc.service.ServiceActivator** that contains the name of the **ServiceActivator** class, for example, **org.jboss.as.quickstarts.ha.singleton.service.SingletonServiceActivator**.

See the **ha-singleton-service** quickstart that ships with JBoss EAP for the complete working example. This quickstart also provides a second example that demonstrates a singleton service that is installed with a backup service. The backup service is running on all nodes that are not elected to be running the

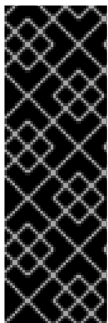
singleton service. Finally, this quickstart also demonstrates how to configure a few different election policies.

6.5. HA SINGLETON DEPLOYMENTS

You can deploy your application as a singleton deployment. When deployed to a group of clustered servers, a singleton deployment only deploys on a single node at any given time. If the node on which the deployment is active stops or fails, the deployment automatically starts on another node.

A singleton deployment can be deployed on multiple nodes in the following situations:

- A group of clustered servers on a given node cannot establish a connection due to a configuration issue or a network issue.
- A non-HA configuration is used, such as the following configuration files:
  - A **standalone.xml** configuration, which supports the Java EE 8 Web Profile, or a **standalone-full.xml** configuration, which supports the Java EE 8 Full Platform profile.
  - A **domain.xml** configuration, which consists of either default domain profiles or full-domain profiles.



IMPORTANT

Non-HA configurations do not have the singleton subsystem enabled by default. If you use this default configuration, the **singleton-deployment.xml** file is ignored to promote a successful deployment of an application.

However, using a non-HA configuration can cause errors for the **jboss-all.xml** descriptor file. To avoid these errors, add the single deployment to the **singleton-deployment.xml** descriptor. You can then deploy the application using any profile type.

The policies for controlling HA singleton behavior are managed by a new **singleton** subsystem. A deployment can either specify a specific singleton policy or use the default subsystem policy.

A deployment identifies itself as a singleton deployment by using a **META-INF/singleton-deployment.xml** deployment descriptor, which is applied to an existing deployment as a deployment overlay. Alternatively, the requisite singleton configuration can be embedded within an existing **jboss-all.xml** file.

Defining or choosing a singleton deployment

To define a deployment as a singleton deployment, include a **META-INF/singleton-deployment.xml** descriptor in your application archive.

If a Maven WAR plug-in already exists, you can migrate the plug-in to the **META-INF** directory: ****/src/main/webapp/META-INF**.

Procedure

- If an application is deployed in an EAR file, move the **singleton-deployment.xml** descriptor or the **singleton-deployment** element, which is located within the **jboss-all.xml** file, to the top-level of the **META-INF** directory.

  #### Example: Singleton deployment descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<singleton-deployment xmlns="urn:jboss:singleton-deployment:1.0"/>
```

- To add an application deployment as a WAR file or a JAR file, move the **singleton-deployment.xml** descriptor to the top-level of the **/META-INF** directory in the application archive.

  Example: Singleton deployment descriptor with a specific singleton policy

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <singleton-deployment policy="my-new-policy" xmlns="urn:jboss:singleton-deployment:1.0"/>
  ```

- Optional: To define the **singleton-deployment** in a **jboss-all.xml** file, move the **jboss-all.xml** descriptor to the top-level of the **/META-INF** directory in the application archive.

  Example: Defining singleton-deployment in jboss-all.xml

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <jboss xmlns="urn:jboss:1.0">
     <singleton-deployment xmlns="urn:jboss:singleton-deployment:1.0"/>
  </jboss>
  ```

- Optional: Use a singleton policy to define the **singleton-deployment** in the **jboss-all.xml** file. Move the **jboss-all.xml** descriptor to the top-level of the **/META-INF** directory in the application archive.

  Example: Defining singleton-deployment in jboss-all.xml with a specific singleton policy

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <jboss xmlns="urn:jboss:1.0">
     <singleton-deployment policy="my-new-policy" xmlns="urn:jboss:singleton-
  deployment:1.0"/>
  </jboss>
  ```

Creating a Singleton Deployment

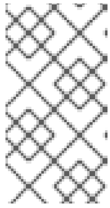
JBoss EAP provides two election policies:

- Simple election policy
  The **simple-election-policy** chooses a specific member, indicated by the **position** attribute, on which a given application will be deployed. The **position** attribute determines the index of the node to be elected from a list of candidates sorted by descending age, where **0** indicates the oldest node, **1** indicates the second oldest node, **-1** indicates the youngest node, **-2** indicates the second youngest node, and so on. If the specified position exceeds the number of candidates, a modulus operation is applied.

  Example: Create a New Singleton Policy with a simple-election-policy and Position Set to -1, Using the Management CLI

  ```
  batch
  /subsystem=singleton/singleton-policy=my-new-policy:add(cache-container=server)
  /subsystem=singleton/singleton-policy=my-new-policy/election-
  ```

```
policy=simple:add(position=-1)
run-batch
```



NOTE

To set the newly created policy **my-new-policy** as the default, run this command:

```
/subsystem=singleton:write-attribute(name=default, value=my-new-policy)
```

Example: Configure a **simple-election-policy** with Position Set to -1 Using **standalone-ha.xml**

```
<subsystem xmlns="urn:jboss:domain:singleton:1.0">
  <singleton-policies default="my-new-policy">
    <singleton-policy name="my-new-policy" cache-container="server">
      <simple-election-policy position="-1"/>
    </singleton-policy>
  </singleton-policies>
</subsystem>
```

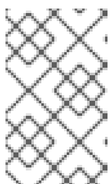
- Random election policy
  The **random-election-policy** chooses a random member on which a given application will be deployed.

Example: Creating a New Singleton Policy with a **random-election-policy**, Using the Management CLI

```
batch
/subsystem=singleton/singleton-policy=my-other-new-policy:add(cache-container=server)
/subsystem=singleton/singleton-policy=my-other-new-policy/election-policy=random:add()
run-batch
```

Example: Configure a **random-election-policy** Using **standalone-ha.xml**

```
<subsystem xmlns="urn:jboss:domain:singleton:1.0">
  <singleton-policies default="my-other-new-policy">
    <singleton-policy name="my-other-new-policy" cache-container="server">
      <random-election-policy/>
    </singleton-policy>
  </singleton-policies>
</subsystem>
```



NOTE

The **default-cache** attribute of the **cache-container** needs to be defined before trying to add the policy. Without this, if you are using a custom cache container, you might end up getting error messages.

Preferences

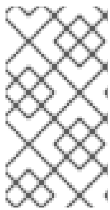
Additionally, any singleton election policy can indicate a preference for one or more members of a cluster. Preferences can be defined either by using the node name or by using the outbound socket binding name. Node preferences always take precedent over the results of an election policy.

Example: Indicate Preference in the Existing Singleton Policy Using the Management CLI

```
/subsystem=singleton/singleton-policy=foo/election-policy=simple:list-add(name=name-preferences,
value=nodeA)

/subsystem=singleton/singleton-policy=bar/election-policy=random:list-add(name=socket-binding-
preferences, value=binding1)
```

Example: Create a New Singleton Policy with a **simple-election-policy** and **name-preferences**, Using the Management CLI

```
batch
/subsystem=singleton/singleton-policy=my-new-policy:add(cache-container=server)
/subsystem=singleton/singleton-policy=my-new-policy/election-policy=simple:add(name-preferences=
[node1, node2, node3, node4])
run-batch
```



NOTE

To set the newly created policy **my-new-policy** as the default, run this command:

```
/subsystem=singleton:write-attribute(name=default, value=my-new-policy)
```

Example: Configure a **random-election-policy** with **socket-binding-preferences** Using **standalone-ha.xml**

```
<subsystem xmlns="urn:jboss:domain:singleton:1.0">
  <singleton-policies default="my-other-new-policy">
    <singleton-policy name="my-other-new-policy" cache-container="server">
      <random-election-policy>
        <socket-binding-preferences>binding1 binding2 binding3 binding4</socket-binding-
preferences>
      </random-election-policy>
    </singleton-policy>
  </singleton-policies>
</subsystem>
```

Define a Quorum

Network partitions are particularly problematic for singleton deployments, since they can trigger multiple singleton providers for the same deployment to run at the same time. To defend against this scenario, a singleton policy can define a quorum that requires a minimum number of nodes to be present before a singleton provider election can take place. A typical deployment scenario uses a quorum of N/2 + 1, where N is the anticipated cluster size. This value can be updated at runtime, and will immediately affect any singleton deployments using the respective singleton policy.

Example: Quorum Declaration in the **standalone-ha.xml** File

```
<subsystem xmlns="urn:jboss:domain:singleton:1.0">
```

```
  <singleton-policies default="default">
    <singleton-policy name="default" cache-container="server" quorum="4">
      <simple-election-policy/>
    </singleton-policy>
  </singleton-policies>
</subsystem>
```

Example: Quorum Declaration Using the Management CLI

```
/subsystem=singleton/singleton-policy=foo:write-attribute(name=quorum, value=3)
```

See the **ha-singleton-deployment** quickstart that ships with JBoss EAP for a complete working example of a service packaged in an application as a cluster-wide singleton using singleton deployments.

Determine the Primary Singleton Service Provider Using the CLI

The **singleton** subsystem exposes a runtime resource for each singleton deployment or service created from a particular singleton policy. This helps you determine the primary singleton provider using the CLI.

You can view the name of the cluster member currently acting as the singleton provider. For example:

```
/subsystem=singleton/singleton-policy=default/deployment=singleton.jar:read-
attribute(name=primary-provider)
{
    "outcome" => "success",
    "result" => "node1"
}
```

You can also view the names of the nodes on which the singleton deployment or service is installed. For example:

```
/subsystem=singleton/singleton-policy=default/deployment=singleton.jar:read-
attribute(name=providers)
{
    "outcome" => "success",
    "result" => [
        "node1",
        "node2"
    ]
}
```

6.6. APACHE MOD_CLUSTER-MANAGER APPLICATION

6.6.1. About mod_cluster-manager Application

The mod_cluster-manager application is an administration web page, which is available on Apache HTTP Server. It is used for monitoring the connected worker nodes and performing various administration tasks, such as enabling or disabling contexts, and configuring the load-balancing properties of worker nodes in a cluster.

Exploring mod_cluster-manager Application

The mod_cluster-manager application can be used for performing various administration tasks on worker nodes.

mod_cluster/1.3.1.Final 1

Auto Refresh show DUMP output show INFO output

LBGroup Group-EU-North: Enable Nodes Disable Nodes Stop Nodes

Node jboss-eap-7.0-3 (ajp://192.168.122.172:8211): 2

Enable Contexts Disable Contexts Stop Contexts 7
Balancer: qacluster,LBGroup: Group-EU-North,Flushpackets: Off,Flushwait: 10000,Ping: 10000000,Smax: 2,Ttl: 60000000,Status: OK,Elected: 10,Read: 5960,Transferred: 0,Connected: 0,Load: 73

Virtual Host 1: 4

Contexts: 5 6

/clusterbench, Status: ENABLED Request: 0 Disable Stop

Aliases:

default-host
localhost

LBGroup Group-EU-West: Enable Nodes Disable Nodes Stop Nodes

Node jboss-eap-7.0-2 (ajp://192.168.122.172:8110): 3

Enable Contexts Disable Contexts Stop Contexts
Balancer: qacluster,LBGroup: Group-EU-West,Flushpackets: Off,Flushwait: 10000,Ping: 10000000,Smax: 2,Ttl: 60000000,Status: OK,Elected: 1,Read: 593,Transferred: 0,Connected: 0,Load: 73
8 9

Virtual Host 1:

Contexts:

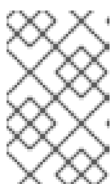
/clusterbench, Status: ENABLED Request: 0 Disable Stop

Aliases:

localhost
default-host

Figure - mod_cluster Administration Web Page

- [1] **mod_cluster/1.3.1.Final**: The version of the mod_cluster native library.
- [2] **ajp://192.168.122.204:8099**: The protocol used (either AJP, HTTP, or HTTPS), hostname or IP address of the worker node, and the port.
- [3] **jboss-eap-7.0-2**: The worker node's JVMRoute.
- [4] **Virtual Host 1**: The virtual host(s) configured on the worker node.
- [5] **Disable**: An administration option that can be used to disable the creation of new sessions on the particular context. However, the ongoing sessions do not get disabled and remain intact.
- [6] **Stop**: An administration option that can be used to stop the routing of session requests to the context. The remaining sessions will fail over to another node unless the **sticky-session-force** property is set to **true**.
- [7] **Enable Contexts Disable Contexts Stop Contexts** The operations that can be performed on the whole node. Selecting one of these options affects all the contexts of a node in all its virtual hosts.
- [8] **Load balancing group (LBGroup)**: The **load-balancing-group** property is set in the **modcluster** subsystem in JBoss EAP configuration to group all worker nodes into custom load balancing groups. Load balancing group (LBGroup) is an informational field that gives information about all set load balancing groups. If this field is not set, then all worker nodes are grouped into a single default load balancing group.

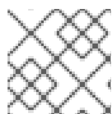


NOTE

This is only an informational field and thus cannot be used to set **load-balancing-group** property. The property has to be set in **modcluster** subsystem in JBoss EAP configuration.

- [9] **Load (value)**: The load factor on the worker node. The load factors are evaluated as below:

```
-load > 0 : A load factor with value 1 indicates that the worker node is overloaded. A load
factor of 100 denotes a free and not-loaded node.
-load = 0 : A load factor of value 0 indicates that the worker node is in standby mode. This
means that no session requests will be routed to this node until and unless the other worker
nodes are unavailable.
-load = -1 : A load factor of value -1 indicates that the worker node is in an error state.
-load = -2 : A load factor of value -2 indicates that the worker node is undergoing
CPing/CPong and is in a transition state.
```



NOTE

For JBoss EAP 7.3, it is also possible to use Undertow as load balancer.

6.7. THE DISTRIBUTABLE-WEB SUBSYSTEM FOR DISTRIBUTABLE WEB SESSION CONFIGURATIONS

The **distributable-web** subsystem facilitates flexible and distributable web session configurations. The subsystem defines a set of distributable web session management profiles. One of these profiles is designated as the default profile. It defines the default behavior of a distributable web application. For example:

```
[standalone@embedded /] /subsystem=distributable-web:read-attribute(name=default-session-
management)
{
    "outcome" => "success",
    "result" => "default"
}
```

The default session management stores web session data within an Infinispan cache as the following example illustrates:

```
[standalone@embedded /] /subsystem=distributable-web/infinispan-session-
management=default:read-resource
{
    "outcome" => "success",
    "result" => {
        "cache" => undefined,
        "cache-container" => "web",
        "granularity" => "SESSION",
        "affinity" => {"primary-owner" => undefined}
    }
}
```

The attributes used in this example and the possible values are:

- **cache**: A cache within the associated cache-container. The web application's cache is based on the configuration of this cache. If undefined, the default cache of the associated cache container is used.
- **cache-container**: A cache-container defined in the **Infinispan** subsystem into which session data is stored.
- **granularity**: Defines how the session manager maps a session into individual cache entries. The possible values are:

- **SESSION**: Stores all session attributes within a single cache entry. More expensive than the **ATTRIBUTE** granularity, but preserves any cross-attribute object references.
- **ATTRIBUTE**: Stores each session attribute within a separate cache entry. More efficient than the **SESSION** granularity, but does not preserve any cross-attribute object references.

- **affinity**: Defines the affinity that a web request must have for a server. The affinity of the associated web session determines the algorithm for generating the route to be appended onto the session ID. The possible values are:
  - **affinity=none**: Web requests do not have any affinity to any node. Use this if web session state is not maintained within the application server.
  - **affinity=local**: Web requests have an affinity to the server that last handled a request for a session. This option corresponds to the sticky session behavior.
  - **affinity=primary-owner**: Web requests have an affinity to the primary owner of a session. This is the default affinity for this distributed session manager. Behaves the same as **affinity=local** if the backing cache is not distributed or replicated.
  - **affinity=ranked**: Web requests have an affinity for the first available member in a list that include the primary and the backup owners, and for the member that last handled a session.
  - **affinity=ranked delimiter**: The delimiter used to separate the individual routes within the encoded session identifier.
  - **Affinity=ranked max routes**: The maximum number of routes to encode into the session identifier.

You must enable ranked session affinity in your load balancer to have session affinity with multiple, ordered routes. For more information, see Enabling Ranked Session Affinity in Your Load Balancer in the *Configuration Guide* for JBoss EAP.

You can override the default distributable session management behavior by referencing a session management profile by name or by providing a deployment-specific session management configuration. For more information, see Overide Default Distributable Session Management Behavior .

6.7.1. Storing Web Session Data In a Remote Red Hat Data Grid

The **distributable-web** subsystem can be configured to store web session data in a remote Red Hat Data Grid cluster using the HotRod protocol. Storing web session data in a remote cluster allows the cache layer to scale independently of the application servers.

Example configuration:

```
[standalone@embedded /]/subsystem=distributable-web/hotrod-session-
management=ExampleRemoteSessionStore:add(remote-cache-container=datagrid, cache-
configuration=__REMOTE_CACHE_CONFIG_NAME__, granularity=ATTRIBUTE)
{
    "outcome" => "success"
}
```

The attributes used in this example and the possible values are:

- **remote-cache-container**: The remote cache container defined in the **Infinispan** subsystem to store the web session data.

- **cache-configuration**: Name of the cache configuration in Red Hat Data Grid cluster. The newly created deployment-specific caches are based on this configuration.
  If a remote cache configuration matching the name is not found, a new cache configuration is created in the remote container.

- **granularity**: Defines how the session manager maps a session into individual cache entries. The possible values are:

  - **SESSION**: Stores all session attributes within a single cache entry. More expensive than the **ATTRIBUTE** granularity, but preserves any cross-attribute object references.

  - **ATTRIBUTE**: Stores each session attribute within a separate cache entry. More efficient than the **SESSION** granularity, but does not preserve any cross-attribute object references.

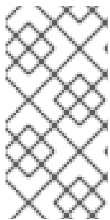
CHAPTER 7. JAKARTA CONTEXTS AND DEPENDENCY INJECTION

7.1. INTRODUCTION TO JAKARTA CONTEXTS AND DEPENDENCY INJECTION

7.1.1. About Jakarta Contexts and Dependency Injection

Jakarta Contexts and Dependency Injection 2.0 is a specification designed to enable Jakarta Enterprise Beans 3 components to be used as Jakarta Server Faces managed beans. Jakarta Contexts and Dependency Injection unifies the two component models and enables a considerable simplification to the programming model for web-based applications in Java. Details about Jakarta Contexts and Dependency Injection 2.0 can be found in Jakarta Contexts and Dependency Injection 2.0 Specification.

JBoss EAP includes Weld, which is the reference implementation of JSR 365: Contexts and Dependency Injection for Java 2.0. The Jakarta EE equivalent specification for JSR-365 is the Jakarta Contexts and Dependency Injection 2.0 Specification.



NOTE

Weld is the reference implementation of Contexts and Dependency Injection for the Java EE Platform. Contexts and Dependency Injection is a JCP standard for dependency injection and contextual lifecycle management. Further, Contexts and Dependency Injection is one of the most important and popular parts of the Java EE.

Benefits of Jakarta Contexts and Dependency Injection
The benefits of Jakarta Contexts and Dependency Injection include:

- Simplifying and shrinking your code base by replacing big chunks of code with annotations.
- Flexibility, allowing you to disable and enable injections and events, use alternative beans, and inject non-Contexts and Dependency Injection objects easily.
- Optionally, allowing you to include a **beans.xml** file in your **META-INF/** or **WEB-INF/** directory if you need to customize the configuration to differ from the default. The file can be empty.
- Simplifying packaging and deployments and reducing the amount of XML you need to add to your deployments.
- Providing lifecycle management via contexts. You can tie injections to requests, sessions, conversations, or custom contexts.
- Providing type-safe dependency injection, which is safer and easier to debug than string-based injection.
- Decoupling interceptors from beans.
- Providing complex event notification.

7.1.2. Relationship Between Weld, Seam 2, and Jakarta Server Faces

Weld is the reference implementation of Contexts and Dependency Injection for the Java EE Platform. The Jakarta equivalent of Contexts and Dependency Injection for the Java EE Platform is the Jakarta

Contexts and Dependency Injection 2.0 Specification. Weld was inspired by Seam 2 and other dependency injection frameworks, and is included in JBoss EAP.

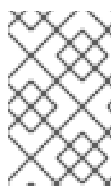
The goal of Seam 2 was to unify Enterprise JavaBeans and JavaServer Faces managed beans.

Jakarta Server Faces 2.3 specification is an API for building server-side user interfaces.

7.2. USE CONTEXTS AND DEPENDENCY INJECTION TO DEVELOP AN APPLICATION

Contexts and Dependency Injection (CDI) gives you tremendous flexibility in developing applications, reusing code, adapting your code at deployment or runtime, and unit testing.

Weld comes with a special mode for application development. When enabled, certain built-in tools, which facilitate the development of Contexts and Dependency Injection applications, are available.



NOTE

The development mode should not be used in production as it can have a negative impact on the performance of the application. Make sure to disable the development mode before deploying to production.

Enabling the Development Mode for a Web Application:

For a web application, set the servlet initialization parameter **org.jboss.weld.development** to **true**:

```
<web-app>
   <context-param>
      <param-name>org.jboss.weld.development</param-name>
      <param-value>true</param-value>
   </context-param>
</web-app>
```

Enabling Development Mode for JBoss EAP Using the Management CLI:

It is possible to enable the Weld development mode globally for all the applications deployed by setting **development-mode** attribute to **true**:

```
/subsystem=weld:write-attribute(name=development-mode,value=true)
```

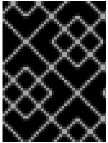
7.2.1. Default Bean Discovery Mode

The default bean discovery mode for a bean archive is **annotated**. Such a bean archive is said to be an **implicit bean archive**.

If the bean discovery mode is **annotated**, then:

- Bean classes that do not have **bean defining annotation** and are not bean classes of sessions beans are not discovered.
- Producer methods that are not on a session bean and whose bean class does not have a bean defining annotation are not discovered.

- Producer fields that are not on a session bean and whose bean class does not have a bean defining annotation are not discovered.
- Disposer methods that are not on a session bean and whose bean class does not have a bean defining annotation are not discovered.
- Observer methods that are not on a session bean and whose bean class does not have a bean defining annotation are not discovered.



IMPORTANT

All examples in the Contexts and Dependency Injection section are valid only when you have a discovery mode set to **all**.

Bean Defining Annotations
A bean class can have a **bean defining annotation**, allowing it to be placed anywhere in an application, as defined in bean archives. A bean class with a bean defining annotation is said to be an implicit bean.

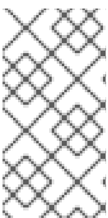
The set of bean defining annotations contains:

- **@ApplicationScoped**, **@SessionScoped**, **@ConversationScoped** and **@RequestScoped** annotations.
- All other normal scope types.
- **@Interceptor** and **@Decorator** annotations.
- All stereotype annotations, i.e. annotations annotated with **@Stereotype**.
- The **@Dependent** scope annotation.

If one of these annotations is declared on a bean class, then the bean class is said to have a bean defining annotation.

Example: Bean Defining Annotation

```
@Dependent
public class BookShop
        extends Business
        implements Shop<Book> {
    ...
}
```



NOTE

To ensure compatibility with other JSR-330 implementations and the Jakarta Contexts and Dependency Injection specification, all pseudo-scope annotations, except **@Dependent**, are not bean defining annotations. However, a stereotype annotation, including a pseudo-scope annotation, is a bean defining annotation.

7.2.2. Exclude Beans From the Scanning Process

Exclude filters are defined by **<exclude>** elements in the **beans.xml** file for the bean archive as children of the **<scan>** element. By default an exclude filter is active. The exclude filter becomes inactive, if its definition contains:

- A child element named **<if-class-available>** with a **name** attribute, and the class loader for the bean archive can not load a class for that name, or
- A child element named **<if-class-not-available>** with a **name** attribute, and the class loader for the bean archive can load a class for that name, or
- A child element named **<if-system-property>** with a **name** attribute, and there is no system property defined for that name, or
- A child element named **<if-system-property>** with a **name** attribute and a value attribute, and there is no system property defined for that name with that value.

The type is excluded from discovery, if the filter is active, and:

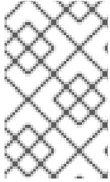
- The fully qualified name of the type being discovered matches the value of the name attribute of the exclude filter, or
- The package name of the type being discovered matches the value of the name attribute with a suffix ".*" of the exclude filter, or
- The package name of the type being discovered starts with the value of the name attribute with a suffix ".**" of the exclude filter

Example 7.1. Example: beans.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee">

  <scan>
    <exclude name="com.acme.rest.*" /> (1)

    <exclude name="com.acme.faces.**"> (2)
      <if-class-not-available name="javax.faces.context.FacesContext"/>
    </exclude>

    <exclude name="com.acme.verbose.*"> (3)
      <if-system-property name="verbosity" value="low"/>
    </exclude>

    <exclude name="com.acme.ejb.**"> (4)
      <if-class-available name="javax.enterprise.inject.Model"/>
      <if-system-property name="exclude-ejbs"/>
    </exclude>
  </scan>

</beans>
```

1. The first exclude filter will exclude all classes in **com.acme.rest** package.
2. The second exclude filter will exclude all classes in the **com.acme.faces** package, and any subpackages, but only if Jakarta Server Faces is not available.
3. The third exclude filter will exclude all classes in the **com.acme.verbose** package if the system property **verbosity** has the value **low**.

4. The fourth exclude filter will exclude all classes in the **com.acme.ejb** package, and any subpackages, if the system property **exclude-ejbs** is set with any value and if at the same time,



NOTE

It is safe to annotate Jakarta EE components with **@Vetoed** to prevent them being considered beans. An event is not fired for any type annotated with **@Vetoed**, or in a package annotated with **@Vetoed**. For more information, see **@Vetoed**.

7.2.3. Use an Injection to Extend an Implementation

You can use an injection to add or change a feature of your existing code.

The following example adds a translation ability to an existing class, and 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!".

This example injects a hypothetical **Translator** object into the **Welcome** class. The **Translator** object can be an Enterprise Java Bean 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 modifying the original **Welcome** class. The **Translator** is injected before the **buildPhrase** method is called.

Example: Inject a Translator Bean into the Welcome Class

```
public class TranslatingWelcome extends Welcome {

    @Inject Translator translator;

    public String buildPhrase(String city) {
        return translator.translate("Welcome to " + city + "!");
    }
    ...
}
```

7.3. 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 Use a Qualifier to Resolve an Ambiguous Injection .

7.3.1. Qualifiers

Qualifiers are annotations used to avoid ambiguous dependencies when the container can resolve multiple beans, which fit into an injection point. A qualifier declared at an injection point provides the set of eligible beans, which declare the same qualifier.

Qualifiers must be declared with a retention and target as shown in the example below.

Example: 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: 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) { ... }
}
```

'@Any'
Whenever a bean or injection point does not explicitly declare a qualifier, the container assumes the qualifier **@Default**. From time to time, you will need to declare an injection point without specifying a qualifier. There is a qualifier for that too. All beans have the qualifier **@Any**. Therefore, by explicitly specifying **@Any** at an injection point, you suppress the default qualifier, without otherwise restricting the beans that are eligible for injection.

This is especially useful if you want to iterate over all beans of a certain bean type.

```
import javax.enterprise.inject.Instance;
...

@Inject

void initServices(@Any Instance<Service> services) {

  for (Service service: services) {

    service.init();
```

```
    }
}
```

Every bean has the qualifier **@Any**, even if it does not explicitly declare this qualifier.

Every event also has the qualifier **@Any**, even if it was raised without explicit declaration of this qualifier.

```
@Inject @Any Event<User> anyUserEvent;
```

The **@Any** qualifier allows an injection point to refer to all beans or all events of a certain bean type.

```
@Inject @Delegate @Any Logger logger;
```

7.3.2. Use a Qualifier to Resolve an Ambiguous Injection

You can resolve an ambiguous injection using a qualifier. Read more about ambiguous injections at Ambiguous or Unsatisfied Dependencies.

The following example is ambiguous and features two implementations of **Welcome**, one which translates and one which does not. The injection needs to be specified to use the translating **Welcome**.

Example: Ambiguous Injection

```
public class Greeter {
  private Welcome welcome;

  @Inject
  void init(Welcome welcome) {
    this.welcome = welcome;
  }
  ...
}
```

Resolve an Ambiguous Injection with a Qualifier

1. To resolve the ambiguous injection, 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"));
  }
}
```

7.4. MANAGED BEANS

Jakarta EE establishes a common definition in the Jakarta Managed Beans specification. For Java EE, managed beans are defined as container-managed objects with minimal programming restrictions, otherwise known by the acronym POJO (Plain Old Java Object). They support a small set of basic services, such as resource injection, lifecycle callbacks, and interceptors. Companion specifications, such as Jakarta Enterprise Beans and Jakarta Contexts and Dependency Injection, build on this basic model.

With very few exceptions, almost every concrete Java class that has a constructor with no parameters, or a constructor designated with the annotation **@Inject**, is a bean. This includes every JavaBean and every Jakarta Enterprise Beans session bean.

7.4.1. Types of Classes That are Beans

A managed bean is a Java class. For Jakarta EE, the basic lifecycle and semantics of a managed bean are defined by the Jakarta Managed Beans 1.0 specification. You can explicitly declare a managed bean by annotating the bean class **@ManagedBean**, but in Contexts and Dependency Injection you do not need to. According to the specification, the Contexts and Dependency Injection 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 with **@Decorator**.
- It is not annotated with an EJB component-defining annotation or declared as an Enterprise Java Bean bean class in the **ejb-jar.xml** file.
- It does not implement the interface **javax.enterprise.inject.spi.Extension**.
- It has either a constructor with no parameters, or a constructor annotated with **@Inject**.
- It is not annotated with **@Vetoed** or in a package annotated with **@Vetoed**.

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**.

@Vetoed

The **@Vetoed** annotation was introduced in CDI 1.1. You can prevent a bean from injection by adding this annotation:

```
@Vetoed
public class SimpleGreeting implements Greeting {
    ...
}
```

In this code, the **SimpleGreeting** bean is not considered for injection.

All beans in a package can be prevented from injection:

```
@Vetoed
package org.sample.beans;

import javax.enterprise.inject.Vetoed;
```

This code in **package-info.java** in the **org.sample.beans** package will prevent all beans inside this package from injection.

Jakarta EE components, such as stateless Jakarta Enterprise Beans or JAX-RS resource endpoints, can be marked with **@Vetoed** to prevent them from being considered beans. Adding the **@Vetoed** annotation to all persistent entities prevents the **BeanManager** from managing an entity as a Jakarta Contexts and Dependency Injection Bean. When an entity is annotated with **@Vetoed**, no injections take place. The reasoning behind this is to prevent the **BeanManager** from performing the operations that might cause the Java Persistence provider to break.

7.4.2. Use Contexts and Dependency Injection to Inject an Object Into a Bean

Contexts and Dependency Injection is activated automatically if Contexts and Dependency Injection components are detected in an application. If you want to customize your configuration to differ from the default, you can include a **META-INF/beans.xml** file or a **WEB-INF/beans.xml** file in your deployment archive.

Inject Objects into Other Objects

1. To obtain an instance of a class, annotate the field with **@Inject** within your bean:

   ```
   public class TranslateController {
     @Inject TextTranslator textTranslator;
     ...
   ```

2. Use your injected object's methods directly. Assume that **TextTranslator** has a method **translate**:

   ```
   // in TranslateController class

   public void translate() {
     translation = textTranslator.translate(inputText);
   }
   ```

3. Use an 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:

```
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.

```
@Inject Instance<TextTranslator> textTranslatorInstance;
...
public void translate() {
  textTranslatorInstance.get().translate(inputText);
}
```

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 that already exists. For instance, a new instance would not be created if you inject a session-scoped bean during the lifetime of the session.

7.5. CONTEXTS AND SCOPES

A context, in terms of Contexts and Dependency Injection, is a storage area that holds instances of beans associated with a specific scope.

A scope is the link between a bean and a context. A scope/context combination can have a specific lifecycle. Several predefined scopes exist, and you can create your own. Examples of predefined scopes are **@RequestScoped**, **@SessionScoped**, and **@ConversationScope**.

Table 7.1. Available Scopes

Scope	Description
@Dependent	The bean is bound to the lifecycle of the bean holding the reference. The default scope for an injected bean is **@Dependent**.
@ApplicationScoped	The bean is bound to the lifecycle of the application.
@RequestScoped	The bean is bound to the lifecycle of the request.
@SessionScoped	The bean is bound to the lifecycle of the session.

Scope	Description
@ConversationScoped	The bean is 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.

7.6. NAMED BEANS

You can name a bean by using the **@Named** annotation. Naming a bean allows you to use it directly in Jakarta Server Faces and Jakarta Expression Language.

The **@Named** annotation takes an optional parameter, which is the bean name. If this parameter is omitted, the bean name defaults to the class name of the bean with its first letter converted to lowercase.

7.6.1. Use Named Beans

Configure Bean Names Using the @Named Annotation

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"));
  }
}
```

   In the example above, the default name would be **greeterBean** if no name had been specified.

2. Use the named bean in a Jakarta Server Faces view.

```
<h:form>
  <h:commandButton value="Welcome visitors" action="#{greeter.welcomeVisitors}"/>
</h:form>
```

7.7. BEAN LIFECYCLE

This task shows you how to save a bean for the life of a request.

The default scope for an injected bean is **@Dependent**. This means that the bean's lifecycle is dependent upon the lifecycle of the bean that holds the reference. Several other scopes exist, and you can define your own scopes. For more information, see Contexts and Scopes.

Manage Bean Lifecycles

1. Annotate the bean with the 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>
```

Your bean is saved in the context relating to the scope that you specify, and lasts as long as the scope applies.

7.7.1. Use a Producer Method

A *producer method* is a method that acts as a source of bean instances. When no instance exists in the specified context, the method declaration itself describes the bean, and the container invokes the method to obtain an instance of the bean. A producer method lets the application take full control of the bean instantiation process.

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

Example: Use a Producer Method

By using a producer method instead of an alternative, polymorphism after deployment is allowed.

The **@Preferred** annotation in the example is a qualifier annotation. For more information about qualifiers, see 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 Contexts and Dependency Injection injection rules. The producer method is called by the container to obtain an instance to service this injection point.

```
@Inject @Preferred PaymentStrategy paymentStrategy;
```

Example: 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.

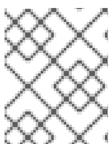
```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy() {
   ...
}
```

Example: 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.

Producer methods allow you to inject non-bean objects and change your code dynamically.

7.8. ALTERNATIVE BEANS

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

By default, **@Alternative** beans are disabled. They are enabled for a specific bean archive by editing its **beans.xml** file. However, this activation only applies to the beans in that archive. From CDI 1.1 onwards, the alternative can be enabled for the entire application using the **@Priority** annotation.

Example: Defining Alternatives

This alternative defines an implementation of the **PaymentProcessor** class using both **@Synchronous** and **@Asynchronous** alternatives:

```
@Alternative @Synchronous @Asynchronous

public class MockPaymentProcessor implements PaymentProcessor {

  public void process(Payment payment) { ... }

}
```

Example: Enabling @Alternative Using beans.xml

```
<beans
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/beans_2_0.xsd">
  <alternatives>
      <class>org.mycompany.mock.MockPaymentProcessor</class>
  </alternatives>
</beans>
```

Declaring Selected Alternatives

The **@Priority** annotation allows an alternative to be enabled for an entire application. An alternative can be given a priority for the application:

- by placing the **@Priority** annotation on the bean class of a managed bean or session bean, or
- by placing the **@Priority** annotation on the bean class that declares the producer method, field or resource.

7.8.1. Override an Injection with an Alternative

You can use alternative beans to 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.

Override an Injection

This task assumes that you already have a **TranslatingWelcome** class in your project, but you want to override it with a "mock" **TranslatingWelcome** class. This would be the case for a test deployment, where the true **Translator** bean cannot be used.

1. Define the alternative.

```
@Alternative
@Translating
```

```
public class MockTranslatingWelcome extends Welcome {
  public String buildPhrase(string city) {
    return "Bienvenue Ã  " + city + "!");
  }
}
```

2. Activate the substitute implementation by adding 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>
```

The alternative implementation is now used instead of the original one.

7.9. 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:

- A default scope.
- A set of interceptor bindings.

A stereotype can also specify either:

- All beans where the stereotypes are defaulted bean EL names.
- All beans where the stereotypes are alternatives.

A bean can declare zero, one, or multiple stereotypes. A stereotype is an **@Stereotype** annotation that packages several other annotations. Stereotype annotations can be applied to a bean class, producer method, or field.

A class that inherits a scope from a stereotype can 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 can override this name if the **@Named** annotation is specified directly on the bean. For more information about named beans, see Named Beans.

7.9.1. Use Stereotypes

Without stereotypes, annotations can become cluttered. This task shows you how to use stereotypes to reduce the clutter and streamline your code.

Example: Annotation Clutter

```
@Secure
@Transactional
```

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

Define and Use Stereotypes

1. Define the stereotype.

```
@Secure
@Transactional
@RequestScoped
@Named
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
public @interface BusinessComponent {
 ...
}
```

2. Use the stereotype.

```
@BusinessComponent
public class AccountManager {
  public boolean transfer(Account a, Account b) {
    ...
  }
}
```

7.10. OBSERVER METHODS

Observer methods receive notifications when events occur.

Contexts and Dependency Injection also provides transactional observer methods, which receive event notifications during the before completion or after completion phase of the transaction in which the event was fired.

7.10.1. Fire and Observe Events

Example: Fire an Event

The following 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: 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 Qualifiers.

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

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

Example: Observe an Event

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

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

You can use qualifiers to observe only specific types of events.

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

7.10.2. Transactional Observers

Transactional observers receive the event notifications before or after the completion phase of the transaction in which the event was raised. Transactional observers are important in a stateful object model because state is often held for longer than a single atomic transaction.

There are five kinds of transactional observers:

- **IN_PROGRESS**: By default, observers are invoked immediately.
- **AFTER_SUCCESS**: Observers are invoked after the completion phase of the transaction, but only if the transaction completes successfully.
- **AFTER_FAILURE**: Observers are invoked after the completion phase of the transaction, but only if the transaction fails to complete successfully.
- **AFTER_COMPLETION**: Observers are invoked after the completion phase of the transaction.
- **BEFORE_COMPLETION**: Observers are invoked before the completion phase of the transaction.

The following observer method refreshes a query result set cached in the application context, but only when transactions that update the Category tree are successful:

```
public void refreshCategoryTree(@Observes(during = AFTER_SUCCESS) CategoryUpdateEvent
event) { ... }
```

Assume you have cached a Jakarta Persistence query result set in the application scope as shown in the following example:

```
import javax.ejb.Singleton;
import javax.enterprise.inject.Produces;

@ApplicationScoped @Singleton

public class Catalog {
  @PersistenceContext EntityManager em;
  List<Product> products;
  @Produces @Catalog
  List<Product> getCatalog() {
    if (products==null) {
      products = em.createQuery("select p from Product p where p.deleted = false")
        .getResultList();
    }
    return products;
  }
}
```

Occasionally a **Product** is created or deleted. When this occurs, you need to refresh the **Product** catalog. But you must wait for the transaction to complete successfully before performing this refresh.

The following is an example of a bean that creates and deletes **Products** triggers events:

```
import javax.enterprise.event.Event;

@Stateless

public class ProductManager {
  @PersistenceContext EntityManager em;
  @Inject @Any Event<Product> productEvent;
  public void delete(Product product) {
    em.delete(product);
    productEvent.select(new AnnotationLiteral<Deleted>(){}).fire(product);
  }

  public void persist(Product product) {
    em.persist(product);
    productEvent.select(new AnnotationLiteral<Created>(){}).fire(product);
  }
  ...
}
```

The **Catalog** can now observe the events after successful completion of the transaction:

```
import javax.ejb.Singleton;

@ApplicationScoped @Singleton
public class Catalog {
  ...
```

```
    void addProduct(@Observes(during = AFTER_SUCCESS) @Created Product product) {
      products.add(product);
    }

    void removeProduct(@Observes(during = AFTER_SUCCESS) @Deleted Product product) {
      products.remove(product);
    }

}
```

7.11. INTERCEPTORS

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. Interceptors are defined as part of the Jakarta Enterprise Beans specification.

Contexts and Dependency Injection 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 Enterprise Java Bean timeout methods by the container.

Enabling Interceptors

By default, all interceptors are disabled. You can enable the interceptor by using the **beans.xml** descriptor of a bean archive. However, this activation only applies to the beans in that archive. From CDI 1.1 onwards the interceptor can be enabled for the whole application using the **@Priority** annotation.

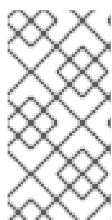
Example: Enabling Interceptors in **beans.xml**

```
<beans
   xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://xmlns.jcp.org/xml/ns/javaee
      http://xmlns.jcp.org/xml/ns/javaee/beans_2.0.xsd">
   <interceptors>
      <class>org.mycompany.myapp.TransactionInterceptor</class>
   </interceptors>
</beans>
```

Having the XML declaration solves two problems:

- It enables you to specify an ordering for the interceptors in your system, ensuring deterministic behavior.
- It lets you enable or disable interceptor classes at deployment time.

Interceptors enabled using **@Priority** are called before interceptors enabled using the **beans.xml** file.



NOTE

Having an interceptor enabled by **@Priority** and at the same time invoked by the **beans.xml** file leads to a nonportable behavior. This combination of enablement should therefore be avoided in order to maintain consistent behavior across different Contexts and Dependency Injection implementations.

7.11.1. Use Interceptors with Contexts and Dependency Injection

Contexts and Dependency Injection can simplify your interceptor code and make it easier to apply to your business code.

Without Contexts and Dependency Injection, 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.

Using Interceptors with Contexts and Dependency Injection

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 development environment.

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

4. Enable the interceptor in your deployment, by adding it to the **META-INF/beans.xml** or **WEB-INF/beans.xml** file.

```
<beans>
  <interceptors>
    <class>com.acme.SecurityInterceptor</class>
    <class>com.acme.TransactionInterceptor</class>
  </interceptors>
</beans>
```

The interceptors are applied in the order listed.

7.12. 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. A decorator is a bean, or even an abstract class, that implements the type it decorates, and is annotated with **@Decorator**. To invoke a decorator in a Contexts and Dependency Injection application, it must be specified in the **beans.xml** file.

Example: Invoke a Decorator Through beans.xml

```
<beans
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/beans_2_0.xsd">
  <decorators>
      <class>org.mycompany.myapp.LargeTransactionDecorator</class>
  </decorators>
</beans>
```

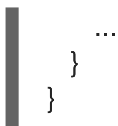
This declaration serves two main purposes:

- It enables you to specify an ordering for decorators in your system, ensuring deterministic behavior.
- It lets you enable or disable decorator classes at deployment time.

A decorator must have exactly one **@Delegate** injection point to obtain a reference to the decorated object.

Example: Decorator Class

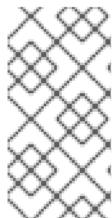
```
@Decorator
public abstract class LargeTransactionDecorator implements Account {

  @Inject @Delegate @Any Account account;
  @PersistenceContext EntityManager em;

  public void withdraw(BigDecimal amount) {
    ...
  }

  public void deposit(BigDecimal amount);
```



```
        ...
    }
}
```

From CDI 1.1 onwards, the decorator can be enabled for the whole application using **@Priority** annotation.

Decorators enabled using **@Priority** are called before decorators enabled using the **beans.xml** file. The lower priority values are called first.



NOTE

Having a decorator enabled by **@Priority** and at the same time invoked by **beans.xml**, leads to a nonportable behavior. This combination of enablement should therefore be avoided in order to maintain consistent behavior across different Contexts and Dependency Injection implementations.

7.13. PORTABLE EXTENSIONS

Contexts and Dependency Injection is intended to be a foundation for frameworks, extensions, and for integration with other technologies. Therefore, Contexts and Dependency Injection exposes a set of SPIs for the use of developers of portable extensions to Contexts and Dependency Injection.

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 Contexts and Dependency Injection programming model.

According to the Jakarta Contexts and Dependency Injection specification, a portable extension can 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 another source.

For more information, see Portable extensions in the Weld documentation.

7.14. BEAN PROXIES

Clients of an injected bean do not usually hold a direct reference to a bean instance. Unless the bean is a dependent object, scope **@Dependent**, the container must redirect all injected references to the bean using a proxy object.

A bean proxy, which can be referred to as client proxy, is responsible for ensuring the bean instance that receives a method invocation is the instance associated with the current context. The client proxy also allows beans bound to contexts, such as the session context, to be serialized to disk without recursively serializing other injected beans.

Due to Java limitations, some Java types cannot be proxied by the container. If an injection point declared with one of these types resolves to a bean with a scope other than **@Dependent**, the container aborts the deployment.

Certain Java types cannot be proxied by the container. These include:

- Classes that do not have a non-private constructor with no parameters.
- Classes that are declared **final** or have a **final** method.
- Arrays and primitive types.

7.15. USE A PROXY IN AN INJECTION

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 runtime, 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: 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);
  }
}
```

CHAPTER 8. JBOSS EAP MBEAN SERVICES

A managed bean, sometimes simply referred to as an MBean, is a type of JavaBean that is created with dependency injection. MBean services are the core building blocks of the JBoss EAP server.

8.1. WRITING JBOSS MBEAN SERVICES

Writing a custom MBean service that relies on a JBoss service requires the service interface method pattern. A JBoss MBean service interface method pattern consists of a set of life cycle operations that inform an MBean service when it can **create**, **start**, **stop**, and **destroy** itself.

You can manage the dependency state using any of the following approaches:

- If you want specific methods to be called on your MBean, declare those methods in your MBean interface. This approach allows your MBean implementation to avoid dependencies on JBoss specific classes.

- If you are not bothered about dependencies on JBoss specific classes, then you can have your MBean interface extend the **ServiceMBean** interface and **ServiceMBeanSupport** class. The **ServiceMBeanSupport** class provides implementations of the service lifecycle methods like create, start, and stop. To handle a specific event like the **start()** event, you need to override **startService()** method provided by the **ServiceMBeanSupport** class.

8.1.1. A Standard MBean Example

This section develops two example MBean services packaged together in a service archive (**.sar**).

ConfigServiceMBean interface declares specific methods like the **start**, **getTimeout**, and **stop** methods to **start**, **hold**, and **stop** the MBean correctly without using any JBoss specific classes. **ConfigService** class implements **ConfigServiceMBean** interface and consequently implements the methods used within that interface.

The **PlainThread** class extends the **ServiceMBeanSupport** class and implements the **PlainThreadMBean** interface. **PlainThread** starts a thread and uses **ConfigServiceMBean.getTimeout()** to determine how long the thread should sleep.

Example: MBean Services Class

```
package org.jboss.example.mbean.support;
public interface ConfigServiceMBean {
   int getTimeout();
   void start();
   void stop();
}
package org.jboss.example.mbean.support;
public class ConfigService implements ConfigServiceMBean {
   int timeout;
   @Override
   public int getTimeout() {
      return timeout;
   }
   @Override
   public void start() {
      //Create a random number between 3000 and 6000 milliseconds
      timeout = (int)Math.round(Math.random() * 3000) + 3000;
```

```
        System.out.println("Random timeout set to " + timeout + " seconds");
    }
    @Override
    public void stop() {
        timeout = 0;
    }
}

package org.jboss.example.mbean.support;
import org.jboss.system.ServiceMBean;
public interface PlainThreadMBean extends ServiceMBean {
    void setConfigService(ConfigServiceMBean configServiceMBean);
}

package org.jboss.example.mbean.support;
import org.jboss.system.ServiceMBeanSupport;
public class PlainThread extends ServiceMBeanSupport implements PlainThreadMBean {
    private ConfigServiceMBean configService;
    private Thread thread;
    private volatile boolean done;
    @Override
    public void setConfigService(ConfigServiceMBean configService) {
        this.configService = configService;
    }
    @Override
    protected void startService() throws Exception {
        System.out.println("Starting Plain Thread MBean");
        done = false;
        thread = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    while (!done) {
                        System.out.println("Sleeping....");
                        Thread.sleep(configService.getTimeout());
                        System.out.println("Slept!");
                    }
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });
        thread.start();
    }
    @Override
    protected void stopService() throws Exception {
        System.out.println("Stopping Plain Thread MBean");
        done = true;
    }
}
```

The **jboss-service.xml** descriptor shows how the **ConfigService** class is injected into the **PlainThread** class using the **inject** tag. The **inject** tag establishes a dependency between **PlainThreadMBean** and **ConfigServiceMBean**, and thus allows **PlainThreadMBean** to use **ConfigServiceMBean** easily.

Example: jboss-service.xml Service Descriptor

```
<server xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:jboss:service:7.0 jboss-service_7_0.xsd"
        xmlns="urn:jboss:service:7.0">
 <mbean code="org.jboss.example.mbean.support.ConfigService"
name="jboss.support:name=ConfigBean"/>
 <mbean code="org.jboss.example.mbean.support.PlainThread"
name="jboss.support:name=ThreadBean">
  <attribute name="configService">
   <inject bean="jboss.support:name=ConfigBean"/>
  </attribute>
 </mbean>
</server>
```

After writing the MBeans example, you can package the classes and the **jboss-service.xml** descriptor in the **META-INF**/ folder of a service archive (**.sar**).

8.2. DEPLOYING JBOSS MBEAN SERVICES

Example: Deploy and Test MBeans in a Managed Domain

Use the following command to deploy the example MBeans (**ServiceMBeanTest.sar**) in a managed domain:

```
deploy ~/Desktop/ServiceMBeanTest.sar --all-server-groups
```

Example: Deploy and Test MBeans on a Standalone Server

Use the following command to build and deploy the example MBeans (**ServiceMBeanTest.sar**) on a standalone server:

```
deploy ~/Desktop/ServiceMBeanTest.sar
```

Example: Undeploy the MBeans Archive

Use the following command to undeploy the MBeans example:

```
undeploy ServiceMBeanTest.sar
```

CHAPTER 9. JAKARTA CONCURRENCY

Jakarta Concurrency is an API that accommodates Java SE concurrency utilities into the Jakarta EE application environment specifications. It is defined in Jakarta Concurrency specification. JBoss EAP allows you to create, edit, and delete instances of Jakarta Concurrency, thus making these instances readily available for applications to use.

Jakarta Concurrency help to extend the invocation context by pulling in the existing context's application threads and using these in its own threads. This extending of invocation context includes class loading, JNDI, and security contexts, by default.

Types of Jakarta Concurrency include:

- Context Service
- Managed Thread Factory
- Managed Executor Service
- Managed Scheduled Executor Service

Example: Jakarta Concurrency in standalone.xml

```
<subsystem xmlns="urn:jboss:domain:ee:4.0">
            <spec-descriptor-property-replacement>false</spec-descriptor-property-replacement>
            <concurrent>
                <context-services>
                    <context-service name="default" jndi-name="java:jboss/ee/concurrency/context/default"
use-transaction-setup-provider="true"/>
                </context-services>
                <managed-thread-factories>
                    <managed-thread-factory name="default" jndi-
name="java:jboss/ee/concurrency/factory/default" context-service="default"/>
                </managed-thread-factories>
                <managed-executor-services>
                    <managed-executor-service name="default" jndi-
name="java:jboss/ee/concurrency/executor/default" context-service="default" hung-task-
threshold="60000" keepalive-time="5000"/>
                </managed-executor-services>
                <managed-scheduled-executor-services>
                    <managed-scheduled-executor-service name="default" jndi-
name="java:jboss/ee/concurrency/scheduler/default" context-service="default" hung-task-
threshold="60000" keepalive-time="3000"/>
                </managed-scheduled-executor-services>
            </concurrent>
            <default-bindings context-service="java:jboss/ee/concurrency/context/default"
datasource="java:jboss/datasources/ExampleDS" managed-executor-
service="java:jboss/ee/concurrency/executor/default" managed-scheduled-executor-
service="java:jboss/ee/concurrency/scheduler/default" managed-thread-
factory="java:jboss/ee/concurrency/factory/default"/>
</subsystem>
```

9.1. CONTEXT SERVICE

The context service (**javax.enterprise.concurrent.ContextService**) allows you to build contextual

proxies from existing objects. Contextual proxy prepares the invocation context, which is used by other Jakarta Concurrency utilities when the context is created or invoked, before transferring the invocation to the original object.

Attributes of the context service concurrency utility include:

- **name**: A unique name within all the context services.
- **jndi-name**: Defines where the context service should be placed in JNDI.
- **use-transaction-setup-provider**: Optional. Indicates if the contextual proxies built by the context service should suspend transactions in context when invoking the proxy objects. Its value defaults to **false**, but the default context service has the value **true**.

See the example above for the usage of the context service concurrency utility.

Example: Add a New Context Service

```
/subsystem=ee/context-service=newContextService:add(jndi-
name=java:jboss/ee/concurrency/contextservice/newContextService)
```

Example: Change a Context Service

```
/subsystem=ee/context-service=newContextService:write-attribute(name=jndi-name,
value=java:jboss/ee/concurrency/contextservice/changedContextService)
```

This operation requires reload.

Example: Remove a Context Service

```
/subsystem=ee/context-service=newContextService:remove()
```

This operation requires reload.

9.2. MANAGED THREAD FACTORY

The managed thread factory **(javax.enterprise.concurrent.ManagedThreadFactory)** concurrency utility allows Jakarta EE applications to create Java threads. JBoss EAP handles the managed thread factory instances, hence Jakarta EE applications cannot invoke any lifecycle related method.

Attributes of managed thread factory concurrency utility include:

- **context-service**: A unique name within all managed thread factories.
- **jndi-name**: Defines where in JNDI the managed thread factory should be placed.
- **priority**: Optional. Indicates the priority for new threads created by the factory, and defaults to **5**.

Example: Add a New Managed Thread Factory

```
/subsystem=ee/managed-thread-factory=newManagedTF:add(context-service=newContextService,
jndi-name=java:jboss/ee/concurrency/threadfactory/newManagedTF, priority=2)
```

Example: Change a Managed Thread Factory

```
/subsystem=ee/managed-thread-factory=newManagedTF:write-attribute(name=jndi-name,
value=java:jboss/ee/concurrency/threadfactory/changedManagedTF)
```

This operation requires reload. Similarly, you can change other attributes as well.

Example: Remove a Managed Thread Factory

```
/subsystem=ee/managed-thread-factory=newManagedTF:remove()
```

This operation requires reload.

9.3. MANAGED EXECUTOR SERVICE

Managed executor service **(javax.enterprise.concurrent.ManagedExecutorService)** allows Jakarta EE applications to submit tasks for asynchronous execution. JBoss EAP handles managed executor service instances, hence Jakarta EE applications cannot invoke any lifecycle related method.

Attributes of managed executor service concurrency utility include:

- **context-service**: Optional. References an existing context service by its name. If specified, then the referenced context service will capture the invocation context present when submitting a task to the executor, which will then be used when executing the task.
- **jndi-name**: Defines where the managed thread factory should be placed in JNDI.
- **max-threads**: Defines the maximum number of threads used by the executor. If undefined, the value from **core-threads** is used.
- **thread-factory**: References an existing managed thread factory by its name, to handle the creation of internal threads. If not specified, then a managed thread factory with default configuration will be created and used internally.
- **core-threads**: Defines the minimum number of threads to be used by the executor. If this attribute is undefined, the default is calculated based on the number of processors. A value of **0** is not recommended. See the **queue-length** attribute for details on how this value is used to determine the queuing strategy.
- **keepalive-time**: Defines the time, in milliseconds, that an internal thread can be idle. The attribute default value is **60000**.
- **queue-length**: Indicates the executor’s task queue capacity. A value of **0** means direct hand-off and possible rejection will occur. If this attribute is undefined or set to **Integer.MAX_VALUE**, this indicates that an unbounded queue should be used. All other values specify an exact queue size. If an unbounded queue or direct hand-off is used, a **core-threads** value greater than **0** is required.
- **hung-task-threshold**: This attribute is for future use.
- **long-running-tasks**: This attribute is for future use.
- **reject-policy**: Defines the policy to use when a task is rejected by the executor. The attribute value can be the default **ABORT**, which means an exception should be thrown, or **RETRY_ABORT**, which means the executor will try to submit it once more, before throwing an

exception

Example: Add a New Managed Executor Service

```
/subsystem=ee/managed-executor-service=newManagedExecutorService:add(jndi-name=java:jboss/ee/concurrency/executor/newManagedExecutorService, core-threads=7, thread-factory=default)
```

Example: Change a Managed Executor Service

```
/subsystem=ee/managed-executor-service=newManagedExecutorService:write-attribute(name=core-threads,value=10)
```

This operation requires reload. Similarly, you can change other attributes too.

Example: Remove a Managed Executor Service

```
/subsystem=ee/managed-executor-service=newManagedExecutorService:remove()
```

This operation requires reload.

9.4. MANAGED SCHEDULED EXECUTOR SERVICE

Managed scheduled executor service **(javax.enterprise.concurrent.ManagedScheduledExecutorService)** allows Jakarta EE applications to schedule tasks for asynchronous execution. JBoss EAP handles managed scheduled executor service instances, hence Jakarta EE applications cannot invoke any lifecycle related method.

Attributes of managed executor service concurrency utility include:

- **context-service**: References an existing context service by its name. If specified then the referenced context service will capture the invocation context present when submitting a task to the executor, which will then be used when executing the task.
- **hung-task-threshold**: This attribute is for future use.
- **keepalive-time**: Defines the time, in milliseconds, that an internal thread can be idle. The attribute default value is **60000**.
- **reject-policy**: Defines the policy to use when a task is rejected by the executor. The attribute value might be the default **ABORT**, which means an exception should be thrown, or **RETRY_ABORT**, which means the executor will try to submit it once more, before throwing an exception.
- **core-threads**: Defines the minimum number of threads to be used by the scheduled executor.
- **jndi-name**: Defines where the managed scheduled executor service should be placed in JNDI .
- **long-running-tasks**: This attribute is for future use.
- **thread-factory**: References an existing managed thread factory by its name, to handle the creation of internal threads. If not specified, then a managed thread factory with default configuration will be created and used internally.

Example: Add a New Managed Scheduled Executor Service

```
/subsystem=ee/managed-scheduled-executor-
service=newManagedScheduledExecutorService:add(jndi-
name=java:jboss/ee/concurrency/scheduledexecutor/newManagedScheduledExecutorService, core-
threads=7, context-service=default)
```

This operation requires reload.

Example: Changed a Managed Scheduled Executor Service

```
/subsystem=ee/managed-scheduled-executor-
service=newManagedScheduledExecutorService:write-attribute(name=core-threads, value=10)
```

This operation requires reload. Similarly, you can change other attributes.

Example: Remove a Managed Scheduled Executor Service

```
/subsystem=ee/managed-scheduled-executor-
service=newManagedScheduledExecutorService:remove()
```

This operation requires reload.

CHAPTER 10. UNDERTOW

10.1. INTRODUCTION TO UNDERTOW HANDLER

Undertow is a web server designed to be used for both blocking and non-blocking tasks. It replaces JBoss Web in JBoss EAP 7. Some of its main features are:

- High Performance
- Embeddable
- Servlet 4.0
- Web Sockets
- Reverse Proxy

Request Lifecycle

When a client connects to the server, Undertow creates a **io.undertow.server.HttpServerConnection**. When the client sends a request, it is parsed by the Undertow parser, and then the resulting **io.undertow.server.HttpServerExchange** is passed to the root handler. When the root handler finishes, one of four things can happen:

- The exchange is completed.
  An exchange is considered complete if both request and response channels have been fully read or written. For requests with no content, such as GET and HEAD, the request side is automatically considered fully read. The read side is considered complete when a handler has written out the full response and has closed and fully flushed the response channel. If an exchange is already complete, then no action is taken.
- The root handler returns normally without completing the exchange.
  In this case the exchange is completed by calling **HttpServerExchange.endExchange()**.
- The root handler returns with an Exception.
  In this case a response code of **500** is set and the exchange is ended using **HttpServerExchange.endExchange()**.
- The root handler can return after **HttpServerExchange.dispatch()** has been called, or after async IO has been started.
  In this case the dispatched task will be submitted to the dispatch executor, or if async IO has been started on either the request or response channels, then this will be started. In both of these cases, the exchange will not be finished. It is up to your async task to finish the exchange when it is done processing.

By far the most common use of **HttpServerExchange.dispatch()** is to move execution from an IO thread, where blocking is not allowed, into a worker thread, which does allow for blocking operations.

Example: Dispatching to a Worker Thread

```
public void handleRequest(final HttpServerExchange exchange) throws Exception {
    if (exchange.isInIoThread()) {
      exchange.dispatch(this);
      return;
```

```
    }
    //handler code
}
```

Because the exchange is not actually dispatched until the call stack returns, you can be sure that more than one thread is never active in an exchange at once. The exchange is not thread safe. However, it can be passed between multiple threads as long as both threads do not attempt to modify it at once.

Ending the Exchange

There are two ways to end an exchange, either by fully reading the request channel and calling **shutdownWrites()** on the response channel and then flushing it, or by calling **HttpServerExchange.endExchange()**. When **endExchange()** is called, Undertow will check if the content has been generated yet. If it has, then it will simply drain the request channel and close and flush the response channel. If not and there are any default response listeners registered on the exchange, then Undertow will give each of them a chance to generate a default response. This mechanism is how default error pages are generated.

For more information on configuring Undertow, see Configuring the Web Server in the JBoss EAP *Configuration Guide*.

10.2. USING EXISTING UNDERTOW HANDLERS WITH A DEPLOYMENT

Undertow provides a default set of handlers that you can use with any application deployed to JBoss EAP.

To use a handler with a deployment, you need to add a **WEB-INF/undertow-handlers.conf** file.

Example: WEB-INF/undertow-handlers.conf File

```
allowed-methods(methods='GET')
```

All handlers can also take an optional predicate to apply that handler in specific cases.

Example: WEB-INF/undertow-handlers.conf File with Optional Predicate

```
path('/my-path') -> allowed-methods(methods='GET')
```

The above example will only apply the **allowed-methods** handler to the path **/my-path**.

Undertow Handler Default Parameter

Some handlers have a default parameter, which allows you to specify the value of that parameter in the handler definition without using the name.

Example: WEB-INF/undertow-handlers.conf File Using the Default Parameter

```
path('/a') -> redirect('/b')
```

You can also update the **WEB-INF/jboss-web.xml** file to include the definition of one or more handlers, but using **WEB-INF/undertow-handlers.conf** is preferred.

Example: WEB-INF/jboss-web.xml File

```
<jboss-web>
```

```
    <http-handler>
        <class-name>io.undertow.server.handlers.AllowedMethodsHandler</class-name>
        <param>
            <param-name>methods</param-name>
            <param-value>GET</param-value>
        </param>
    </http-handler>
</jboss-web>
```

A full list of the provided Undertow handlers can be found in the Provided Undertow Handlers reference.

10.3. CREATING CUSTOM HANDLERS

There are two ways to define custom handlers:

1. Using WEB-INF/jboss-web.xml file
2. In the WEB-INF/undertow-handlers.conf

Defining Custom Handlers Using the WEB-INF/jboss-web.xml File

A custom handler can be defined in the **WEB-INF/jboss-web.xml** file.

Example: Define Custom Handler in WEB-INF/jboss-web.xml

```
<jboss-web>
    <http-handler>
        <class-name>org.jboss.example.MyHttpHandler</class-name>
    </http-handler>
</jboss-web>
```

Example: HttpHandler Class

```
package org.jboss.example;

import io.undertow.server.HttpHandler;
import io.undertow.server.HttpServerExchange;

public class MyHttpHandler implements HttpHandler {
    private HttpHandler next;

    public MyHttpHandler(HttpHandler next) {
        this.next = next;
    }

    public void handleRequest(HttpServerExchange exchange) throws Exception {
        // do something
        next.handleRequest(exchange);
    }
}
```

Parameters can also be set for the custom handler using the **WEB-INF/jboss-web.xml** file.

Example: Defining Parameters in WEB-INF/jboss-web.xml

```
<jboss-web>
  <http-handler>
    <class-name>org.jboss.example.MyHttpHandler</class-name>
    <param>
      <param-name>myParam</param-name>
      <param-value>foobar</param-value>
    </param>
  </http-handler>
</jboss-web>
```

For these parameters to work, the handler class needs to have corresponding setters.

Example: Defining Setter Methods in Handler

```
package org.jboss.example;

import io.undertow.server.HttpHandler;
import io.undertow.server.HttpServerExchange;

public class MyHttpHandler implements HttpHandler {
  private HttpHandler next;
  private String myParam;

  public MyHttpHandler(HttpHandler next) {
    this.next = next;
  }

  public void setMyParam(String myParam) {
    this.myParam = myParam;
  }

  public void handleRequest(HttpServerExchange exchange) throws Exception {
    // do something, use myParam
    next.handleRequest(exchange);
  }
}
```

Defining Custom Handlers in the WEB-INF/undertow-handlers.conf File

Instead of using the **WEB-INF/jboss-web.xml** for defining the handler, it could also be defined in the **WEB-INF/undertow-handlers.conf** file.

```
myHttpHandler(myParam='foobar')
```

For the handler defined in **WEB-INF/undertow-handlers.conf** to work, two things need to be created:

1. An implementation of **HandlerBuilder**, which defines the corresponding syntax bits for **undertow-handlers.conf** and is responsible for creating the **HttpHandler**, wrapped in a **HandlerWrapper**.

   #### Example: HandlerBuilder Class

   ```
   package org.jboss.example;

   import io.undertow.server.HandlerWrapper;
   ```

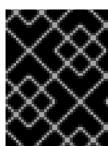
```
import io.undertow.server.HttpHandler;
import io.undertow.server.handlers.builder.HandlerBuilder;

import java.util.Collections;
import java.util.Map;
import java.util.Set;

public class MyHandlerBuilder implements HandlerBuilder {
    public String name() {
        return "myHttpHandler";
    }

    public Map<String, Class<?>> parameters() {
        return Collections.<String, Class<?>>singletonMap("myParam", String.class);
    }

    public Set<String> requiredParameters() {
        return Collections.emptySet();

    }

    public String defaultParameter() {
        return null;

    }

    public HandlerWrapper build(final Map<String, Object> config) {
        return new HandlerWrapper() {
            public HttpHandler wrap(HttpHandler handler) {
                MyHttpHandler result = new MyHttpHandler(handler);
                result.setMyParam((String) config.get("myParam"));
                return result;
            }
        };
    }
}
```

2. An entry in the file. **META-INF/services/io.undertow.server.handlers.builder.HandlerBuilder**. This file must be on the class path, for example, in **WEB-INF/classes**.

```
org.jboss.example.MyHandlerBuilder
```

10.4. DEVELOPING A CUSTOM HTTP MECHANISM

When Elytron is used to secure a web application, it is possible to implement custom HTTP authentication mechanisms that can be registered using the **elytron** subsystem. It is then also possible to override the configuration within the deployment to make use of this mechanism without requiring modifications to the deployment.



IMPORTANT

All custom HTTP mechanisms are required to implement the **HttpServerAuthenticationMechanism** interface.

In general, for an HTTP mechanism, the **evaluateRequest** method is called to handle the request passing in the **HTTPServerRequest** object. The mechanism processes the request and uses one of the following callback methods on the request to indicate the outcome:

- **authenticationComplete** - The mechanism successfully authenticated the request.
- **authenticationFailed** - Authentication was attempted but failed.
- **authenticationInProgress** - Authentication started but an additional round trip is needed.
- **badRequest** - The authentication for this mechanism failed validation of the request.
- **noAuthenticationInProgress** - The mechanism did not attempt any stage of authentication.

After creating a custom HTTP mechanism that implements the **HttpServerAuthenticationMechanism** interface, the next step is to create a factory that returns instances of this mechanism. The factory must implement the **HttpAuthenticationFactory** interface. The most important step in the factory implementation is to double check the name of the mechanism requested. It is important for the factory to return null if it cannot create the required mechanism. The mechanism factory can also take into account properties in the map passed in to decide if it can create the requested mechanism.

There are two different approaches that can be used to advertise the availability of a mechanism factory.

- The first approach is to implement a **java.security.Provider** with the **HttpAuthenticationFactory** registered as an available service once for each mechanism it supports.
- The second approach is to use a **java.util.ServiceLoader** to discover the factory instead. To achieve this, a file named **org.wildfly.security.http.HttpServerAuthenticationMechanismFactory** should be added under **META-INF/services**. The only content required in this file is the fully qualified class name of the factory implementation.

The mechanism can then be installed in the application server, as a module ready to be used:

```
module add --name=org.wildfly.security.examples.custom-http --resources=/path/to/custom-http-mechanism.jar --dependencies=org.wildfly.security.elytron,javax.api
```

Additional resources

- For more information see modules and dependencies.

Using a Custom HTTP Mechanism

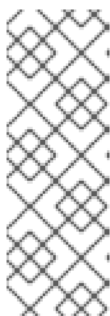
1. Add a custom module.

```
/subsystem=elytron/service-loader-http-server-mechanism-factory=custom-factory:add(module=org.wildfly.security.examples.custom-http)
```

2. Add an **http-authentication-factory** to tie the mechanism factory to a **security-domain** that will be used for the authentication.

```
/subsystem=elytron/http-authentication-factory=custom-mechanism:add(http-server-mechanism-factory=custom-factory,security-domain=ApplicationDomain,mechanism-configurations=[{mechanism-name=custom-mechanism}])
```

3. Update the **application-security-domain** resource to use the new **http-authentication-factory**.



NOTE

When an application is deployed, it by default uses the **other** security domain. Thus, you need to add a mapping to the application to map it to an Elytron HTTP authentication factory.

```
/subsystem=undertow/application-security-domain=other:add(http-authentication-factory=application-http-authentication)
```

The **application-security-domain** resource can now be updated to use the new **http-authentication-factory**.

```
/subsystem=undertow/application-security-domain=other:write-attribute(name=http-authentication-factory,value=custom-mechanism)

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

Notice that the command above overrides the deployment configuration. This means that the mechanisms from the **http-authentication-factory** will be used even if the deployment was configured to use a different mechanism. It is thus possible to override the configuration within a deployment to make use of a custom mechanism, without requiring modifications to the deployment itself.

4. Reload the server

```
reload
```

CHAPTER 11. JAKARTA TRANSACTIONS

11.1. OVERVIEW

11.1.1. Overview of Jakarta Transactions

Introduction

This section provides a foundational understanding of the Jakarta Transactions.

- About Jakarta Transactions
- Transaction Lifecycle
- Jakarta Transactions Transaction Example

11.2. TRANSACTION CONCEPTS

11.2.1. 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 rollback. In a rollback, 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).

11.2.2. 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.

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 can be related to hardware, software, network, or any other involved system.

11.2.3. About the Transaction Coordinator or Transaction Manager

The terms Transaction Coordinator and Transaction Manager ™ are mostly interchangeable in terms of transactions with JBoss EAP. The term Transaction Coordinator is usually used in the context of distributed JTS transactions.

In Jakarta Transactions transactions, the TM runs within JBoss EAP and communicates with transaction participants during the two-phase commit protocol.

The TM 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.

- About Transaction Participants
- About ACID Properties for Transactions
- About the 2-Phase Commit Protocol

11.2.4. About Transaction Participants

A transaction participant is any resource within a transaction that has the ability to commit or to roll back state. It is generally a database or a Jakarta Messaging broker, but by implementing the transaction interface, application code could also act as a transaction participant. Each participant of a transaction independently decides whether it is able to commit or roll back its state, and only if all participants can commit does the transaction as a whole succeed. Otherwise, each participant rolls back its state, and the transaction as a whole fails. The TM coordinates the commit or rollback operations and determines the outcome of the transaction.

11.2.5. About Jakarta Transactions

Jakarta Transactions is part of Jakarta EE Spec. It is defined in Jakarta Transactions 1.3 Specification.

Implementation of Jakarta Transactions is done using the TM, which is covered by project Narayana for JBoss EAP application server. The TM allows applications to assign various resources, for example, database or Jakarta Messaging brokers, through a single global transaction. The global transaction is referred as an XA transaction. Generally resources with XA capabilities are included in such transactions, but non-XA resources could also be part of global transactions. There are several optimizations which help non-XA resources to behave as XA capable resources. For more information, see LRCO Optimization for Single-phase Commit.

In this document, the term Jakarta Transactions refers to two things:

1. The Jakarta Transactions, which is defined by Jakarta EE specification.
2. It indicates how the TM processes the transactions.

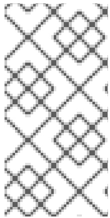
The TM works in Jakarta Transactions transactions mode, the data is shared in memory, and the transaction context is transferred by remote Jakarta Enterprise Beans calls. In the manage transactions mode, the data is shared by sending Common Object Request Broker Architecture (CORBA) messages and the transaction context is transferred by IIOP calls. Both modes support distribution of transactions over multiple JBoss EAP servers.

- About Distributed Transactions
- About XA Datasources and XA Transactions

11.2.6. About JTS

JTS is a mapping of the Object Transaction Service (OTS) to Jakarta. Jakarta EE applications use the Jakarta Transactions to manage transactions. Jakarta Transactions then interacts with a Object Transactions Service transaction implementation when the transaction manager is switched to JTS mode. JTS works over the IIOP protocol. Transaction managers that use JTS communicate with each other using a process called an Object Request Broker (ORB), using a communication standard called Common Object Request Broker Architecture (CORBA). For more information, see ORB Configuration in the JBoss EAP *Configuration Guide*.

Using the Jakarta Transactions from an application standpoint, a JTS transaction behaves in the same way as a Jakarta Transactions transaction.



NOTE

The implementation of JTS included in JBoss EAP supports distributed transactions. The difference from fully-compliant JTS transactions is interoperability with external third-party ORBs. This feature is unsupported with JBoss EAP. Supported configurations distribute transactions across multiple JBoss EAP containers only.

11.2.7. About XML Transaction Service

The XML Transaction Service (XTS) component supports the coordination of private and public web services in a business transaction. Using XTS, you can coordinate complex business transactions in a controlled and reliable manner. The XTS API supports a transactional coordination model based on the WS-Coordination, WS-Atomic Transaction, and WS-Business Activity protocols.

11.2.7.1. Overview of Protocols Used by XTS

The WS-Coordination (WS-C) specification defines a framework that allows different coordination protocols to be plugged in to coordinate work between clients, services, and participants.

The WS-Transaction (WS-T) protocol comprises the pair of transaction coordination protocols, WS-Atomic Transaction (WS-AT) and WS-Business Activity (WS-BA), which utilize the coordination framework provided by WS-C. WS-T is developed to unify existing traditional transaction processing systems, allowing them to communicate reliably with one another.

11.2.7.2. Web Services-Atomic Transaction Process

An atomic transaction (AT) is designed to support short duration interactions where ACID semantics are appropriate. Within the scope of an AT, web services typically employ bridging to access XA resources, such as databases and message queues, under the control of the WS-T. When the transaction terminates, the participant propagates the outcome decision of the AT to the XA resources, and the appropriate commit or rollback actions are taken by each participant.

11.2.7.2.1. Atomic Transaction Process

1. To initiate an AT, the client application first locates a WS-C Activation Coordinator web service that supports WS-T.
2. The client sends a WS-C **CreateCoordinationContext** message to the service, specifying http://schemas.xmlsoap.org/ws/2004/10/wsat as its coordination type.
3. The client receives an appropriate WS-T context from the activation service.

4. The response to the **CreateCoordinationContext** message, the transaction context, has its **CoordinationType** element set to the WS-AT namespace, http://schemas.xmlsoap.org/ws/2004/10/wsat. It also contains a reference to the atomic transaction coordinator endpoint, the WS-C Registration Service, where participants can be enlisted.

5. The client normally proceeds to invoke web services and complete the transaction, either committing all the changes made by the web services, or rolling them back. In order to be able to drive this completion, the client must register itself as a participant for the completion protocol, by sending a register message to the registration service whose endpoint was returned in the coordination context.

6. Once registered for completion, the client application then interacts with web services to accomplish its business-level work. With each invocation of a business web service, the client inserts the transaction context into a SOAP header block, such that each invocation is implicitly scoped by the transaction. The toolkits that support WS-AT aware web services provide facilities to correlate contexts found in SOAP header blocks with back-end operations. This ensures that modifications made by the web service are done within the scope of the same transaction as the client and subject to commit or rollback by the Transaction Coordinator.

7. Once all the necessary application work is complete, the client can terminate the transaction, with the intent of making any changes to the service state permanent. The completion participant instructs the coordinator to try to commit or roll back the transaction. When the commit or rollback operation completes, a status is returned to the participant to indicate the outcome of the transaction.

For more details, see WS-Coordination in the Naryana Project Documentation.

11.2.7.2.2. WS-AT Interoperability with Microsoft .NET Clients

The **xts** subsystem can have issues communicating with Microsoft .NET clients because of differences in the .NET implementation of the WS-AT specification. The .NET implementation of the WS-AT specification forces any call to be asynchronous.

To enable interoperability with .NET clients, an asynchronous registration option is available in the JBoss EAP **xts** subsystem. XTS asynchronous registration is disabled by default, and you should only enable it if necessary.

To enable asynchronous registration for WS-AT interoperability with .NET clients, use the following management CLI command:

```
/subsystem=xts:write-attribute(name=async-registration, value=true)
```

11.2.7.3. Web Services-Business Activity Process

Web Services-Business Activity (WS-BA) defines a protocol for web service applications to enable existing business processing and workflow systems to wrap their proprietary mechanisms and interoperate across implementations and business boundaries.

Unlike the WS-AT protocol model, where participants inform the transaction coordinator of their state only when asked, a child activity within a WS-BA can specify its outcome to the coordinator directly, without waiting for a request. A participant can choose to exit the activity or notify the coordinator of a failure at any point. This feature is useful when tasks fail because the notification can be used to modify the goals and drive processing forward, without waiting until the end of the transaction to identify failures.

11.2.7.3.1. WS-BA Process

1. Services are requested to do work.
2. Wherever these services have the ability to undo any work, they inform the WS-BA, in case the WS-BA later decides the cancel the work. If the WS-BA suffers a failure. it can instruct the service to execute its **undo** behavior.

The WS-BA protocols employ a compensation-based transaction model. When a participant in a business activity completes its work, it can choose to exit the activity. This choice does not allow any subsequent rollback. Alternatively, the participant can complete its activity, signaling to the coordinator that the work it has done can be compensated if, at some later point, another participant notifies a failure to the coordinator. In this latter case, the coordinator asks each non-exited participant to compensate for the failure, giving them the opportunity to execute whatever compensating action they consider appropriate. If all participants exit or complete without failure, the coordinator notifies each completed participant that the activity has been closed.

For more details, see WS-Coordination in the Naryana Project Documentation.

11.2.7.4. Transaction Bridging Overview

Transaction Bridging describes the process of linking the Jakarta EE and WS-T domains. The transaction bridge component, **txbridge**, provides bi-directional linkage, such that either type of transaction can encompass business logic designed for use with the other type. The technique used by the bridge is a combination of interposition and protocol mapping.

In the transaction bridge, an interposed coordinator is registered into the existing transaction and performs the additional task of protocol mapping; that is, it appears to its parent coordinator to be a resource of its native transaction type, while appearing to its children to be a coordinator of their native transaction type, even though these transaction types differ.

The transaction bridge resides in the package **org.jboss.jbossts.txbridge** and its subpackages. It consists of two distinct sets of classes, one for bridging in each direction.

For more details, see TXBridge Guide in the Naryana Project Documentation.

11.2.8. About XA Resources and XA Transactions

XA stands for eXtended Architecture, which was developed by the X/Open Group to define a transaction that uses more than one back-end data store. The XA standard describes the interface between a global 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 all four ACID properties. One of the four ACID properties is atomicity, which 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. An XA resource is a resource that can participate in an XA global transaction.

An XA transaction is a transaction that can span multiple resources. It involves a coordinating TM, with one or more databases or other transactional resources, all involved in a single global XA transaction.

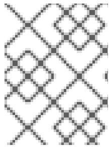
11.2.9. About XA Recovery

TM implements X/Open XA specification and supports XA transactions across multiple XA resources.

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 that are transaction participants crash or become unavailable. Within

the scope of JBoss EAP, the **transactions** subsystem provides the mechanisms for XA Recovery to any XA resources or subsystems that use them, such as XA datasources, Jakarta Messaging message queues, and Jakarta Connectors 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. The XA recovery process is driven by a periodic recovery thread which is launched by default every two minutes. The periodic recovery thread processes all unfinished transactions.



NOTE

It can take four to eight minutes to complete the recovery for an in-doubt transaction because it might require multiple runs of the recovery process.

11.2.10. Limitations of the XA Recovery Process

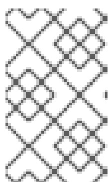
XA recovery has the following limitations:

- The transaction log might not be cleared from a successfully committed transaction.
  If the JBoss EAP server crashes after an **XAResource** commit method successfully completes and commits the transaction, but before the coordinator can update the log, you might see the following warning message in the log when you restart the server:

  ```
  ARJUNA016037: Could not find new XAResource to use for recovering non-serializable
  XAResource XAResourceRecord
  ```

  This is because upon recovery, the JBoss Transaction Manager (TM) sees the transaction participants in the log and attempts to retry the commit. Eventually the JBoss TM assumes the resources are committed and no longer retries the commit. In this situation, you can safely ignore this warning as the transaction is committed and there is no loss of data.

  To prevent the warning, set the **com.arjuna.ats.jta.xaAssumeRecoveryComplete** property value to **true** . This property is checked whenever a new **XAResource** instance cannot be located from any registered **XAResourceRecovery** instance. When set to **true**, the recovery assumes that a previous commit attempt succeeded and the instance can be removed from the log with no further recovery attempts. This property must be used with care because it is global and when used incorrectly could result in **XAResource** instances remaining in an uncommitted state.

  NOTE

  JBoss EAP 7.3 has an implemented enhancement to clear transaction logs after a successfully committed transaction and the above situation should not occur frequently.

- Rollback is not called for JTS transaction when a server crashes at the end of **XAResource.prepare()**.
  If the JBoss EAP server crashes after the completion of an **XAResource.prepare()** method call, all of the participating **XAResource** instances are locked in the prepared state and remain that way upon server restart. The transaction is not rolled back and the resources remain locked until the transaction times out or a database administrator manually rolls back the resources and clears the transaction log. For more information, see https://issues.jboss.org/browse/JBTM-2124

- Periodic recovery can occur on committed transactions.
  When the server is under excessive load, the server log might contain the following warning message, followed by a stacktrace:

  ```
  ARJUNA016027: Local XARecoveryModule.xaRecovery got XA exception
  XAException.XAER_NOTA: javax.transaction.xa.XAException
  ```

  Under heavy load, the processing time taken by a transaction can overlap with the timing of the periodic recovery process's activity. The periodic recovery process detects the transaction still in progress and attempts to initiate a rollback but in fact the transaction continues to completion. At the time the periodic recovery attempts but fails the rollback, it records the rollback failure in the server log. The underlying cause of this issue will be addressed in a future release, but in the meantime a workaround is available.

  Increase the interval between the two phases of the recovery process by setting the **com.arjuna.ats.jta.orphanSafetyInterval** property to a value higher than the default value of **10000** milliseconds. A value of **40000** milliseconds is recommended. Note that this does not solve the issue. Instead it decreases the probability that it will occur and that the warning message will be shown in the log. For more information, see https://developer.jboss.org/thread/266729

11.2.11. About the 2-Phase Commit Protocol

The two-phase commit (2PC) protocol refers to an algorithm to determine the outcome of a transaction. 2PC is driven by the Transaction Manager (TM) as a process of finishing XA transactions.

Phase 1: Prepare
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: Commit
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 resources about what to do, and they notify the coordinator when they have done it. At that point, the transaction is finished.

11.2.12. 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 an XA resource that is part of database table or message in a queue when they commit. The TM 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.

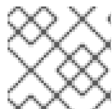
Transaction timeouts can be associated with transactions in order to control their lifecycle. If a timeout threshold passes before the transaction commits or rolls back, the timeout causes the transaction to be rolled back automatically.

You can configure default timeout values for the entire transaction subsystem, or you can disable default timeout values and specify timeouts on a per-transaction basis.

11.2.13. About Distributed Transactions

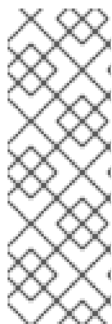
A distributed transaction is a transaction with participants on multiple JBoss EAP servers. JTS specification mandates that JTS transactions be able to be distributed across application servers from

different vendors. The Jakarta Transactions does not define that but JBoss EAP supports distributed Jakarta Transactions transactions among JBoss EAP servers.



NOTE

Transaction distribution among servers from different vendors is not supported.



NOTE

In other application server vendor documentation, you might find that the term distributed transaction means XA transaction. In the context of JBoss EAP documentation, the distributed transaction refers to transactions distributed among several JBoss EAP application servers. Transactions that consist of different resources, for example, database resources and Jakarta Messaging resources, are referred as XA transactions in this document. For more information, see About JTS and About XA Datasources and XA Transactions.

11.2.14. About the ORB Portability API

The Object Request Broker (ORB) is a process that sends and receives messages to transaction participants, coordinators, resources, and other services distributed across multiple application servers. An ORB uses a standardized Interface Description Language (IDL) to communicate and interpret messages. Common Object Request Broker Architecture (CORBA) is the IDL used by the ORB in JBoss EAP.

The main type of service that uses an ORB is a system of distributed Jakarta Transactions, using the JTS specification. Other systems, especially legacy systems, can choose to use an ORB for communication rather than other mechanisms such as remote Jakarta Enterprise Beans or Jakarta Enterprise Web Services or Jakarta RESTful 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:

- **com.arjuna.orbportability.orb**
- **com.arjuna.orbportability.oa**

See the JBoss EAP Javadocs bundle available on the Red Hat Customer Portal for specific details about the methods and properties included in the ORB Portability API.

11.3. TRANSACTION OPTIMIZATIONS

11.3.1. Overview of Transaction Optimizations

The Transaction Manager (TM) of JBoss EAP includes several optimizations that your application can take advantage of.

Optimizations serve to enhance the 2-phase commit protocol in particular cases. Generally, the TM starts a global transaction, which passes through the 2-phase commit. But when you optimize these transactions, in certain cases, the TM does not need to proceed with full 2-phased commits and thus the

process gets faster.

Different optimizations used by the TM are described in detail below.

- About the LRCO Optimization for Single-phase Commit (1PC)
- About the Presumed-Abort Optimization
- About the Read-Only Optimization

11.3.2. About the LRCO Optimization for Single-phase Commit (1PC)

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. The single phase commnit protocol is used when only one XA or non-XA resource is a part of the global transaction.

The prepare phase generally locks the resource until the second phase is processed. Single-phase commit means that the prepare phase is skipped and only the commit is processed on the resource. If not specified, the single-phase commit optimization is used automatically when the global transaction contains only one participant.

Last Resource Commit Optimization (LRCO)
In situations where non-XA datasource participate in XA transaction, an optimization known as the Last Resource Commit Optimization (LRCO) is employed. 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.

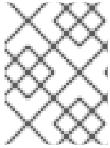
The non-XA resource is processed at the end of the prepare phase, and an attempt is made to commit it. If the commit succeeds, the transaction log is written and the remaining resources go through the commit phase. If the last resource fails to commit, the transaction is rolled back.

Where a single local TX datasource is used in a transaction, the LRCO is automatically applied to it.

Previously, adding non-XA resources to an XA transaction was achieved via the LRCO method. However, there is a window of failure in LRCO. The procedure for adding non-XA resources to an XA transaction using the LRCO method is as follows:

1. Prepare the XA transaction.
2. Commit LRCO.
3. Write the transaction log.
4. Commit the XA transaction.

If the procedure crashes between step 2 and step 3, this could lead to data inconsistency and you cannot commit the XA transaction. The data inconsistency is because the LRCO non-XA resource is committed but information about preparation of XA resource was not recorded. The recovery manager will rollback the resource after the server is up. Commit Markable Resource (CMR) eliminates this restriction and allows a non-XA resource to be reliably enlisted in an XA transaction.



NOTE

CMR is a special case of LRCO optimization that should only be used for datasources. It is not suitable for all non-XA resources.

- About the 2-Phase Commit Protocol

11.3.2.1. Commit Markable Resource

Summary

Configuring access to a resource manager using the Commit Markable Resource (CMR) interface ensures that a non-XA datasource can be reliably enlisted in an XA (2PC) transaction. It is an implementation of the LRCO algorithm, which makes non-XA resource fully recoverable.

To configure the CMR, you must:

1. Create tables in a database.
2. Enable the datasource to be connectable.
3. Add a reference to **transactions** subsystem.

Create Tables in Database

A transaction can contain only one CMR resource. You can create a table using SQL similar to the following example.

```
SELECT xid,actionuid FROM _tableName_ WHERE transactionManagerID IN (String[])
DELETE FROM _tableName_ WHERE xid IN (byte[[]])
INSERT INTO _tableName_ (xid, transactionManagerID, actionuid) VALUES (byte[],String,byte[])
```

The following are examples of the SQL syntax to create tables for various database management systems.

Example: Sybase Create Table Syntax

```
CREATE TABLE xids (xid varbinary(144), transactionManagerID varchar(64), actionuid
varbinary(28))
```

Example: Oracle Create Table Syntax

```
CREATE TABLE xids (xid RAW(144), transactionManagerID varchar(64), actionuid RAW(28))
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

Example: IBM Create Table Syntax

```
CREATE TABLE xids (xid VARCHAR(255) for bit data not null, transactionManagerID
varchar(64), actionuid VARCHAR(255) for bit data not null)
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

Example: SQL Server Create Table Syntax

```
CREATE TABLE xids (xid varbinary(144), transactionManagerID varchar(64), actionuid
varbinary(28))
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

Example: PostgreSQL Create Table Syntax

```
CREATE TABLE xids (xid bytea, transactionManagerID varchar(64), actionuid bytea)
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

Example: MariaDB Create Table Syntax

```
CREATE TABLE xids (xid BINARY(144), transactionManagerID varchar(64), actionuid BINARY(28))
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

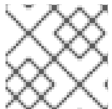
Example: MySQL Create Table Syntax

```
CREATE TABLE xids (xid VARCHAR(255), transactionManagerID varchar(64), actionuid
VARCHAR(255))
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

Enabling Datasource to be Connectable

By default, the CMR feature is disabled for datasources. To enable it, you must create or modify the datasource configuration and ensure that the **connectable** attribute is set to **true**. The following is an example of the datasources section of a server XML configuration file:

```
<datasource enabled="true" jndi-name="java:jboss/datasources/ConnectableDS" pool-
name="ConnectableDS" jta="true" use-java-context="true" connectable="true"/>
```



NOTE

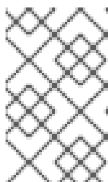
This feature is not applicable to XA datasources.

You can also enable a resource manager as a CMR, using the management CLI, as follows:

```
/subsystem=datasources/data-source=ConnectableDS:add(enabled="true", jndi-
name="java:jboss/datasources/ConnectableDS", jta="true", use-java-context="true",
connectable="true", connection-url="validConnectionURL", exception-sorter-class-
name="org.jboss.jca.adapters.jdbc.extensions.mssql.MSSQLExceptionSorter", driver-name="mssql")
```

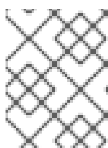
This command generates the following XML in the **datasources** section of the server configuration file.

```
<datasource jta="true" jndi-name="java:jboss/datasources/ConnectableDS" pool-
name="ConnectableDS" enabled="true" use-java-context="true" connectable="true">
  <connection-url>validConnectionURL</connection-url>
  <driver>mssql</driver>
  <validation>
    <exception-sorter class-
name="org.jboss.jca.adapters.jdbc.extensions.mssql.MSSQLExceptionSorter"/>
  </validation>
</datasource>
```



NOTE

The datasource must have a valid driver defined. The example above uses **mssql** as the **driver-name**; however the **mssql** driver does not exist. For details, see Example MySQL Datasource in the JBoss EAP *Configuration Guide*.



NOTE

Use the **exception-sorter-class-name** parameter in the datasource configuration. For details, see Example Datasource Configurations in the JBoss EAP *Configuration Guide*.

Updating an Existing Resource to Use the New CMR Feature

If you only need to update an existing datasource to use the CMR feature, then simply modify the **connectable** attribute:

```
/subsystem=datasources/data-source=ConnectableDS:write-attribute(name=connectable,value=true)
```

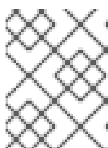
Add a Reference to the Transactions Subsystem

The **transactions** subsystem identifies the datasources that are CMR capable through an entry to the **transactions** subsystem configuration section as shown below:

```
<subsystem xmlns="urn:jboss:domain:transactions:5.0">
  ...
  <commit-markable-resources>
    <commit-markable-resource jndi-name="java:jboss/datasources/ConnectableDS">
      <xid-location name="xids" batch-size="100" immediate-cleanup="false"/>
    </commit-markable-resource>
    ...
  </commit-markable-resources>
</subsystem>
```

The same result can be achieved using the management CLI:

```
/subsystem=transactions/commit-markable-
resource=java\:jboss\/datasources\/ConnectableDS/:add(batch-size=100,immediate-
cleanup=false,name=xids)
```



NOTE

You must restart the server after adding the CMR reference under the **transactions** subsystem.

11.3.3. 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.

11.3.4. 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 affect on the transaction. This read-only participant can be omitted from the second phase of the commit protocol.

11.4. TRANSACTION OUTCOMES

11.4.1. About Transaction Outcomes

There are three possible outcomes for a transaction.

Commit

If every transaction participant can commit, the transaction coordinator directs them to do so. See About Transaction Commit for more information.

Rollback

If any transaction participant cannot commit, or if the transaction coordinator cannot direct participants to commit, the transaction is rolled back. See About Transaction Rollback 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 About Heuristic Outcomes for more information.

11.4.2. 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 a commit, information about the transaction is removed from the transaction coordinator, and the newly-written state is now the durable state.

11.4.3. About Transaction Rollback

A transaction participant rolls back by restoring its state to reflect the state before the transaction began. After a rollback, the state is the same as if the transaction had never been started.

11.4.4. About Heuristic Outcomes

A heuristic outcome, or non-atomic outcome, is a situation where the decisions of the participants in a transaction differ from that of the transaction manager. Heuristic outcomes can cause loss of integrity to the system, and usually require human intervention to resolve them. Do not write code which relies on them.

Heuristic outcomes typically occur during the second phase of the 2-phase commit (2PC) protocol. In rare cases, this outcome might occur in a 1PC. They are often caused by failures to the underlying hardware or communications subsystems of the underlying servers.

Heuristic outcomes are possible due to timeouts in various subsystems or resources even with transaction manager and full crash recovery. In any system that requires some form of distributed agreement, situations can arise where some parts of the system diverge in terms of the global outcome.

There are four different types of heuristic outcomes:

Heuristic rollback
The commit operation was not able to commit the resources but all of the participants were able to be rolled back and so an atomic outcome was still achieved.

Heuristic commit
An attempted rollback operation failed because all of the participants unilaterally committed. This can 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 might decide to commit.

Heuristic mixed
Some participants committed and others rolled back.

Heuristic hazard
The disposition of some of the updates is unknown. For those that are known, they have either all been committed or all rolled back.

- About the 2-Phase Commit Protocol

11.4.5. JBoss Transactions Errors and Exceptions

For details about exceptions thrown by methods of the **UserTransaction** class, see the **UserTransaction** API Javadoc.

11.5. OVERVIEW OF THE TRANSACTION LIFECYCLE

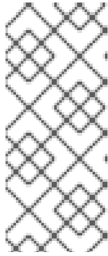
11.5.1. Transaction Lifecycle

See About Jakarta Transactions for more information on Jakarta Transactions.

When a resource asks to participate in a transaction, a chain of events is set in motion. The Transaction Manager (TM) is a process that lives within the application server and manages transactions. Transaction participants are objects which participate in a transaction. Resources are datasources, Jakarta Messaging connection factories, or other Jakarta Connectors connections.

1. The application starts a new transaction.
   To begin a transaction, the application obtains an instance of class **UserTransaction** from Java Naming and Directory Interface or, if it is a Jakarta Enterprise Beans, 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 Jakarta Transactions, so all transactions are top-level transactions.

   A Jakarta Enterprise Beans starts a transaction when the **UserTransaction.begin()** method is called. The default behavior of this transaction could be affected by use of the **TransactionAttribute** annotation or the **ejb.xml** descriptor. Any resource that is used after that point is associated with the transaction. If more than one resource is enlisted, the transaction becomes an XA transaction, and participates in the two-phase commit protocol at commit time.



NOTE

By default, transactions are driven by application containers in Jakarta Enterprise Beanss. This is called *Container Managed Transaction (CMT)*. To make the transaction user driven, change the **Transaction Management** to *Bean Managed Transaction (BMT)*. In BMT, the **UserTransaction** object is available for the user to manage the transaction.

2. The application modifies its state.
   In the next step, the application performs its work and makes changes to its state, only on enlisted resources.

3. The application decides to commit or roll back.
   When the application has finished changing its state, it decides whether to commit or roll back. It calls the appropriate method, either **UserTransaction.commit()** or **UserTransaction.rollback()**. For a CMT, this process is driven automatically, whereas for a BMT, a method commit or rollback of the **UserTransaction** has to be explicitly called.

4. The TM removes the transaction from its records.
   After the commit or rollback completes, the TM cleans up its records and removes information about the transaction from the transaction log.

Failure Recovery

If a resource, transaction participant, or the application server crashes or become unavailable, the **Transaction Manager** handles recovery when the underlying failure is resolved and the resource is available again. This process happens automatically. For more information, see XA Recovery.

11.6. TRANSACTION SUBSYSTEM CONFIGURATION

The **transactions** subsystem allows you to configure transaction manager options such as statistics, timeout values, and transaction logging. You can also manage transactions and view transaction statistics.

For more information, see Configuring Transactions in the JBoss EAP *Configuration Guide*.

11.7. TRANSACTIONS USAGE IN PRACTICE

11.7.1. Transactions Usage Overview

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

- Control Transactions
    - Begin a Transaction
    - Commit a Transaction
    - Roll Back a Transaction
- Handle a Heuristic Outcome in a Transaction
- Handle Transaction Errors
- Transaction References

11.7.2. Control Transactions

Introduction

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

- Begin a Transaction
- Commit a Transaction
- Roll Back a Transaction

11.7.2.1. Begin a Transaction

This procedure shows how to begin a new transaction. The API is the same whether you run the Transaction Manager ™ configured with Jakarta Transactions or JTS.

1. Get an instance of **UserTransaction**.
   You can get the instance using Java Naming and Directory Interface, injection, or an EJBs context if the EJB uses bean-managed transactions by means of a **@TransactionManagement(TransactionManagementType.BEAN)** annotation.

   - Get the instance using Java Naming and Directory Interface.

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

   - Get the instance using injection.

     ```
     @Resource UserTransaction userTransaction;
     ```

   - Get the instance using the EJB context.
     - In a stateless/stateful bean:

       ```
       @Resource SessionContext ctx;
       ctx.getUserTransaction();
       ```

     - In a message-driven bean:

       ```
       @Resource MessageDrivenContext ctx;
       ctx.getUserTransaction()
       ```

2. Call **UserTransaction.begin()** after you connect to your datasource.

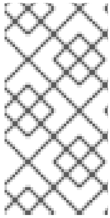
   ```
   try {
      System.out.println("\nCreating connection to database: "+url);
      stmt = conn.createStatement();  // non-tx statement
      try {
         System.out.println("Starting top-level transaction.");
         userTransaction.begin();
         stmtx = conn.createStatement(); // will be a tx-statement
         ...
      }
   }
   ```

–

Result

The transaction begins. All uses of your datasource are transactional until you commit or roll back the transaction.

For a full example, see Jakarta Transactions Transaction Example.



NOTE

One of the benefits of EJBs (either used with CMT or BMT) is that the container manages all the internals of the transactional processing, that is, you are free from taking care of transaction being part of XA transaction or transaction distribution amongst JBoss EAP containers.

11.7.2.1.1. Nested Transactions

Nested transactions allow an application to create a transaction that is embedded in an existing transaction. In this model, multiple subtransactions can be embedded recursively in a transaction. Subtransactions can be committed or rolled back without committing or rolling back the parent transaction. However, the results of a commit operation are contingent upon the commitment of all the transaction's ancestors.

For implementation specific information, see the Narayana Project Documentation.

Nested transactions are available only when used with the JTS specification. Nested transactions are not a supported feature of JBoss EAP application server. In addition, many database vendors do not support nested transactions, so consult your database vendor before you add nested transactions to your application.

11.7.2.2. Commit a Transaction

This procedure shows how to commit a transaction using the Jakarta Transactions.

Prerequisites

You must begin a transaction before you can commit it. For information on how to begin a transaction, see Begin a Transaction.

1. Call the **commit()** method on the **UserTransaction**.
   When you call the commit() method on the **UserTransaction**, the TM attempts to commit the transaction.

```
@Inject
private UserTransaction userTransaction;

public void updateTable(String key, String value) {
    EntityManager entityManager = entityManagerFactory.createEntityManager();
    try {
        userTransaction.begin();
        <!-- Perform some data manipulation using entityManager -->
        ...
        // Commit the transaction
        userTransaction.commit();
    } catch (Exception ex) {
        <!-- Log message or notify Web page -->
```

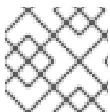
```
        ...
        try {
          userTransaction.rollback();
        } catch (SystemException se) {
          throw new RuntimeException(se);
        }
        throw new RuntimeException(ex);
    } finally {
        entityManager.close();
    }
}
```

2. If you use Container Managed Transactions (CMT), you do not need to manually commit.
   If you configure your bean to use Container Managed Transactions, the container will manage the transaction lifecycle for you based on annotations you configure in the code.

```
@PersistenceContext
private EntityManager em;

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void updateTable(String key, String value)
  <!-- Perform some data manipulation using entityManager -->
  ...
}
```

Result

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



NOTE

For a full example, see Jakarta Transactions Transaction Example.

11.7.2.3. Roll Back a Transaction

This procedure shows how to roll back a transaction using the Jakarta Transactions.

Prerequisites

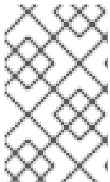
You must begin a transaction before you can roll it back. For information on how to begin a transaction, see Begin a Transaction.

1. Call the **rollback()** method on the **UserTransaction**.
   When you call the **rollback()** method on the **UserTransaction**, the TM attempts to roll back the transaction and return the data to its previous state.

```
@Inject
private UserTransaction userTransaction;

public void updateTable(String key, String value)
    EntityManager entityManager = entityManagerFactory.createEntityManager();
    try {
        userTransaction.begin():
        <!-- Perform some data manipulation using entityManager -->
          ...
```

```
        // Commit the transaction
        userTransaction.commit();
    } catch (Exception ex) {
        <!-- Log message or notify Web page -->
        ...
        try {
            userTransaction.rollback();
        } catch (SystemException se) {
            throw new RuntimeException(se);
        }
        throw new RuntimeException(e);
    } finally {
        entityManager.close();
    }
}
```

2. If you use Container Managed Transactions (CMT), you do not need to manually roll back the transaction.
   If you configure your bean to use Container Managed Transactions, the container will manage the transaction lifecycle for you based on annotations you configure in the code.

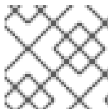


NOTE

Rollback for CMT occurs if RuntimeException is thrown. You can also explicitly call the setRollbackOnly method to gain the rollback. Or, use the @ApplicationException(rollback=true) for application exception to rollback.

Result

Your transaction is rolled back by the TM.



NOTE

For a full example, see Jakarta Transactions Transaction Example.

11.7.3. Handle a Heuristic Outcome in a Transaction

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. See About Heuristic Outcomes for more information about heuristic transaction outcomes.

This procedure shows how to handle a heuristic outcome of a transaction using the Jakarta Transactions.

1. The cause of a heuristic outcome in a transaction is that a resource manager promised it could commit or rollback, and then failed to fulfill the promise. This could be due to a problem with a third-party component, the integration layer between the third-party component and JBoss EAP, or JBoss EAP itself.
   By far, the most common two causes of heuristic errors are transient failures in the environment and coding errors dealing with resource managers.

2. Usually, 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 due to a network outage, hardware failure, database failure, power outage, or a host of other things.

If you come across a heuristic outcome in a test environment during stress testing, it implies weaknesses in your test environment.

WARNING

JBoss EAP automatically recovers transactions that were in a non-heuristic state at the time of failure, but it does not attempt to recover the heuristic transactions.

3. If you have no obvious failure in your environment, or if the heuristic outcome is easily reproducible, it is probably due to a coding error. You must contact the third-party vendors to find out if a solution is available.
   If you suspect the problem is in the transaction manager of JBoss EAP itself, you must raise a support ticket.

4. You can attempt to recover the transaction manually using the management CLI. For more information, see the Recovering a Transaction Participant section of *Managing Transactions on JBoss EAP*.

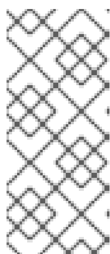
5. The process of resolving the transaction outcome manually is dependent on the exact circumstance of the failure. Perform the following steps, as applicable to your environment:

   a. Identify which resource managers were involved.

   b. Examine the state of the transaction manager and the resource managers.

   c. Manually force log cleanup and data reconciliation in one or more of the involved components.

6. In a test environment, or if you do not care about the integrity of the data, deleting the transaction logs and restarting JBoss EAP gets rid of the heuristic outcome. By default, the transaction logs are located in the ***EAP_HOME*/standalone/data/tx-object-store/** directory for a standalone server, or the ***EAP_HOME*/domain/servers/*SERVER_NAME*/data/tx-object-store/** directory in a managed domain. In the case of a managed domain, *SERVER_NAME* refers to the name of the individual server participating in a server group.



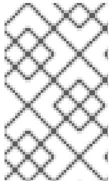
NOTE

The location of the transaction log also depends on the object store in use and the values set for the **object-store-relative-to** and **object-store-path** parameters. For file system logs, such as a standard shadow and Apache ActiveMQ Artemis logs, the default directory location is used, but when using a JDBC object store, the transaction logs are stored in a database.

11.7.4. Jakarta Transactions Transaction Error Handling

11.7.4.1. Handle Transaction Errors

Transaction errors are challenging to solve because they are often dependent on timing. Here are some common errors and ideas for troubleshooting them.



NOTE

These guidelines do not apply to heuristic errors. If you experience heuristic errors, refer to 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. See the JBoss EAP *Configuration Guide* for information on configuring the transaction manager.
If that is not feasible, you might 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 Jakarta Transactions 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 might 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 might 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.

11.8. TRANSACTION REFERENCES

11.8.1. Transaction Example for Jakarta Transactions

This example illustrates how to begin, commit, and roll back a Jakarta Transactions transaction. You need to adjust the connection and datasource parameters to suit your environment, and set up two test tables in your database.

```
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) {
        throw new RuntimeException(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) {
        throw new RuntimeException(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();
                System.out.println("\nAdding entries to table 1.");
                stmtx.executeUpdate("INSERT INTO test_table (a, b) VALUES (1,2)");
                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) {
                throw new RuntimeException(ex);

            }
        }
        catch (Exception sysEx) {
            sysEx.printStackTrace();
            System.exit(0);
        }
    }
}
```

11.8.2. Transaction API Documentation

The transaction Jakarta Transactions API documentation is available as Javadoc at the following location:

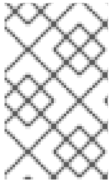
- UserTransaction - https://jakarta.ee/specifications/platform/8/apidocs/javax/transaction/UserTransaction.html

If you use Red Hat CodeReady Studio to develop your applications, the API documentation is included in the **Help** menu.

CHAPTER 12. JAKARTA PERSISTENCE

12.1. ABOUT JAKARTA PERSISTENCE

The Jakarta Persistence is a Jakarta EE specification for accessing, persisting, and managing data between Java objects or classes and a relational database. The Jakarta Persistence specification recognizes the interest and the success of the transparent object or relational mapping paradigm. It standardizes the basic APIs and the metadata needed for any object or relational persistence mechanism.



NOTE

Jakarta Persistence itself is just a specification, not a product; it cannot perform persistence or anything else by itself. Jakarta Persistence is just a set of interfaces, and requires an implementation.

12.2. CREATE A SIMPLE JPA APPLICATION

Follow the procedure below to create a simple JPA application in Red Hat CodeReady Studio.

1. Create a JPA project in Red Hat CodeReady Studio.
    a. In Red Hat CodeReady Studio, click **File → New → Project**. Find **JPA** in the list, expand it, and select **JPA Project**. You are presented with the following dialog.

Figure 12.1. New JPA Project Dialog

b. Enter a **Project name**.

c. Select a **Target runtime**. If no target runtime is available, follow these instructions to define a new server and runtime: Downloading, Installing, and Setting Up JBoss EAP from within the IDE in the *Getting Started with CodeReady Studio Tools* guide.

d. Under **JPA version**, ensure **2.1** is selected.

e. Under **Configuration**, choose **Basic JPA Configuration**.

f. Click **Finish**.

g. If prompted, choose whether you wish to associate this type of project with the JPA perspective window.

2. Create and configure a new persistence settings file.

a. Open an EJB 3.x project in Red Hat CodeReady Studio.

b. Right click the project root directory in the **Project Explorer** panel.

c. Select **New** → **Other...**.

d. Select **XML File** from the XML folder and click **Next**.

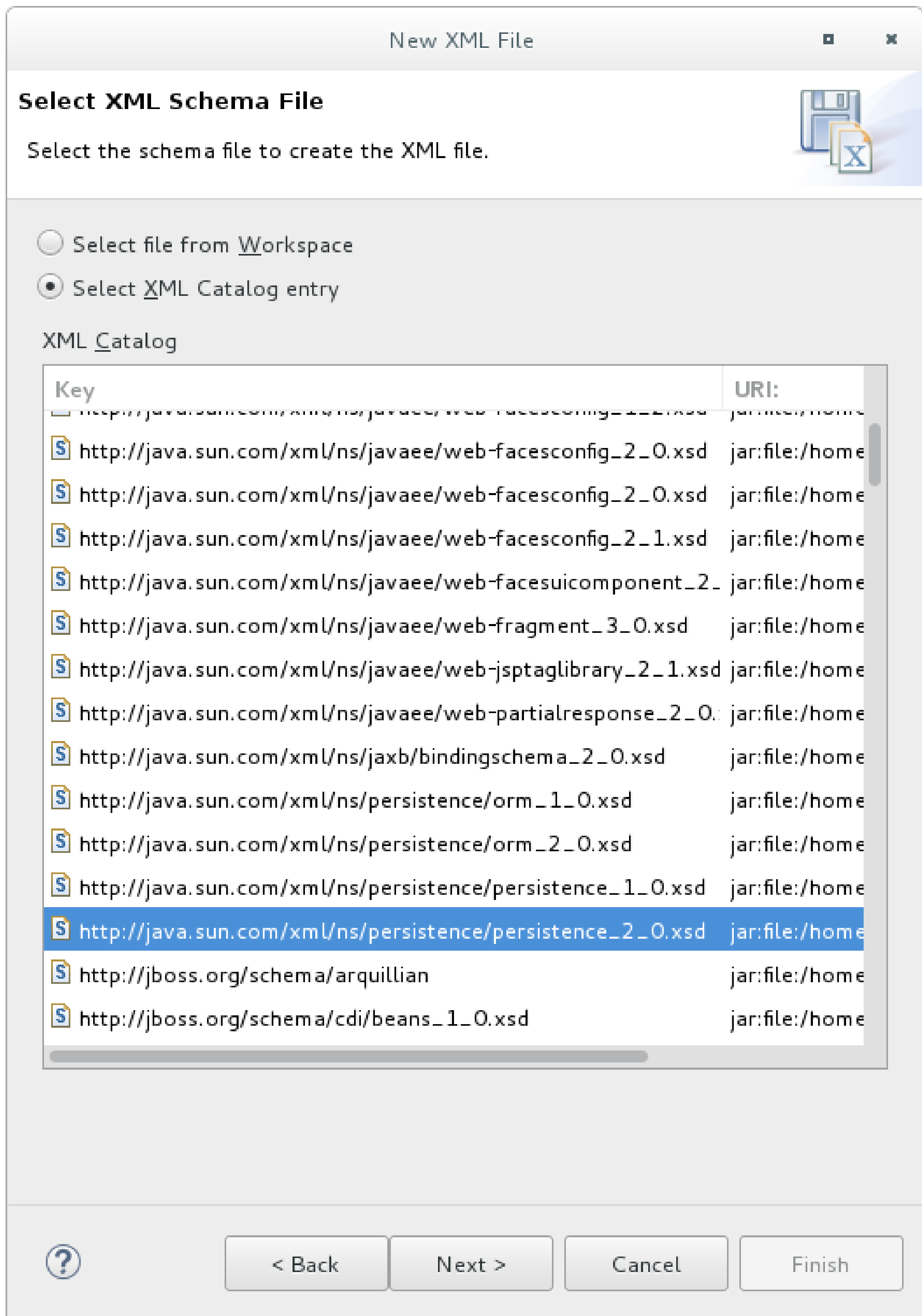
e. Select the **ejbModule/META-INF/** folder as the parent directory.

f. Name the file **persistence.xml** and click **Next**.

g. Select **Create XML file from an XML schema file** and click **Next**.

h. Select **http://java.sun.com/xml/ns/persistence/persistence_2.0.xsd** from the **Select XML Catalog entry** list and click **Next**.

Figure 12.2. Persistence XML Schema



i. Click **Finish** to create the file. The **persistence.xml** has been created in the **META-INF/** folder and is ready to be configured.

Example: Persistence Settings File

```
<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_2.xsd"
  version="2.2">
  <persistence-unit name="example" transaction-type="JTA">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</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>
```

12.3. JAKARTA PERSISTENCE ENTITIES

Once you have established the connection from your application to the database, you can start mapping the data in the database to Java objects. Java objects that are used to map against database tables are called entity objects.

Entities have relationships with other entities, which are expressed through object-relational metadata. The object-relational metadata can be specified either directly in the entity class file by using annotations, or in an XML descriptor file called **persistence.xml** included with the application.

The high-level mapping of Java objects to the database is as follows:

- Java classes map to the database tables.
- Java instances map to the database rows.
- Java fields map to the database columns.

12.4. PERSISTENCE CONTEXT

The Jakarta Persistence persistence context contains the entities managed by the persistence provider. The persistence context acts like a first level transactional cache for interacting with the datasource. It manages the entity instances and their lifecycle. Loaded entities are placed into the persistence context before being returned to the application. Entity changes are also placed into the persistence context to be saved in the database when the transaction commits.

The lifetime of a container-managed persistence context can either be scoped to a transaction, which is referred to as a transaction-scoped persistence context, or have a lifetime scope that extends beyond that of a single transaction, which is referred to as an extended persistence context. The **PersistenceContextType** property, which has the **enum** datatype, is used to define the persistence context lifetime scope for container-managed entity managers. The persistence context lifetime scope is defined when the **EntityManager** instance is created.

12.4.1. Transaction-Scoped Persistence Context

The transaction-scoped persistence context works with the active Jakarta Transactions transaction. When the transaction commits, the persistence context is flushed to the datasource; the entity objects

are detached but might still be referenced by the application code. All the entity changes that are expected to be saved to the datasource must be made during a transaction. Entities that are read outside the transaction are detached when the **EntityManager** invocation completes.

12.4.2. Extended Persistence Context

The extended persistence context spans multiple transactions and allows data modifications to be queued without an active Jakarta Transactions transaction. The container-managed extended persistence context can only be injected into a stateful session bean.

12.5. JAKARTA PERSISTENCE ENTITYMANAGER

Jakarta Persistence entity manager represents a connection to the persistence context. You can read from and write to the database defined by the persistence context using the entity manager.

Persistence context is provided through the Java annotation **@PersistenceContext** in the **javax.persistence** package. The entity manager is provided through the Java class **javax.persistence.EntityManager**. In any managed bean, the **EntityManager** instance can be injected as shown below:

Example: Entity Manager Injection

```
@Stateless
public class UserBean {
   @PersistenceContext
   EntityManager entitymanager;
   ...
}
```

12.5.1. Application-Managed EntityManager

Application-managed entity managers provide direct access to the underlying persistence provider, **org.hibernate.jpa.HibernatePersistenceProvider**. The scope of the application-managed entity manager is from when the application creates it and lasts until the application closes it. You can use the **@PersistenceUnit** annotation to inject a persistence unit into the **javax.persistence.EntityManagerFactory** interface, which returns an application-managed entity manager.

Application-managed entity managers can be used when your application needs to access a persistence context that is not propagated with the Jakarta Transactions transaction across **EntityManager** instances in a particular persistence unit. In this case, each **EntityManager** instance creates a new, isolated persistence context. The **EntityManager** instance and its associated **PersistenceContext** is created and destroyed explicitly by your application. Application-managed entity managers can also be used when you cannot inject **EntityManager** instances directly, because the **EntityManager** instances are not thread-safe. **EntityManagerFactory** instances are thread-safe.

Example: Application-Managed Entity Manager

```
@PersistenceUnit
EntityManagerFactory emf;
EntityManager em;
@Resource
UserTransaction utx;
...
```

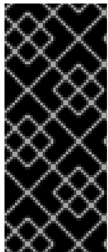
```
em = emf.createEntityManager();
try {
    utx.begin();
    em.persist(SomeEntity);
    em.merge(AnotherEntity);
    em.remove(ThirdEntity);
    utx.commit();
}
catch (Exception e) {
    utx.rollback();
}
```

12.5.2. Container-Managed EntityManager

Container-managed entity managers manage the underlying persistence provider for the application. They can use the transaction-scoped persistence contexts or the extended persistence contexts. The container-managed entity manager creates instances of the underlying persistence provider as needed. Every time a new underlying persistence provider **org.hibernate.jpa.HibernatePersistenceProvider** instance is created, a new persistence context is also created.

12.6. WORKING WITH THE ENTITYMANAGER

When you have the **persistence.xml** file located in the **/META-INF** directory, the entity manager is loaded and has an active connection to the database. The **EntityManager** property can be used to bind the entity manager to JNDI and to add, update, remove and query entities.



IMPORTANT

If you plan to use a security manager with Hibernate, be aware that Hibernate supports it only when **EntityManagerFactory** is bootstrapped by the JBoss EAP server. It is not supported when the **EntityManagerFactory** or **SessionFactory** is bootstrapped by the application. See Java Security Manager in *How to Configure Server Security* for more information about security managers.

12.6.1. Binding the EntityManager to JNDI

By default, JBoss EAP does not bind the **EntityManagerFactory** to JNDI. You can explicitly configure this in the **persistence.xml** file of your application by setting the **jboss.entity.manager.factory.jndi.name** property. The value of this property should be the JNDI name to which you want to bind the **EntityManagerFactory**.

You can also bind a container-managed transaction-scoped entity manager to JNDI by using the **jboss.entity.manager.jndi.name** property.

Example: Binding the EntityManager and the EntityManagerFactory to JNDI

```
<property name="jboss.entity.manager.jndi.name" value="java:/MyEntityManager"/>
<property name="jboss.entity.manager.factory.jndi.name" value="java:/MyEntityManagerFactory"/>
```

Example: Storing an Entity using the EntityManager

```
public User createUser(User user) {
    entityManager.persist(user);
```

```
    return user;
}
```

Example: Updating an Entity using the EntityManager

```
public void updateUser(User user) {
    entityManager.merge(user);
}
```

Example: Removing an Entity using the EntityManager

```
public void deleteUser(String user) {
    User user = findUser(username);
    if (user != null)
        entityManager.remove(user);
}
```

Example: Querying an Entity using the EntityManager

```
public User findUser(String username) {
    CriteriaBuilder builder = entityManager.getCriteriaBuilder();
    CriteriaQuery<User> criteria = builder.createQuery(User.class);
    Root<User> root = criteria.from(User.class);
    TypedQuery<User> query = entityManager
        .createQuery(criteria.select(root).where(
            builder.equal(root.<String> get("username"), username)));
    try {
        return query.getSingleResult();
    }
    catch (NoResultException e) {
        return null;
    }
}
```

12.7. DEPLOYING THE PERSISTENCE UNIT

A persistence unit is a logical grouping that includes:

- Configuration information for an entity manager factory and its entity managers.
- Classes managed by the entity managers.
- Mapping metadata specifying the mapping of the classes to the database.

The **persistence.xml** file contains persistence unit configuration, including the datasource name. The JAR file or the directory whose /**META-INF**/ directory contains the **persistence.xml** file is termed as the root of the persistence unit.

In Jakarta EE environments, the root of the persistence unit must be one of the following:

- An EJB-JAR file
- The /**WEB-INF/classes**/ directory of a WAR file

- A JAR file in the **/WEB-INF/lib/** directory of a WAR file
- A JAR file in the EAR library directory
- An application client JAR file

Example: Persistence Settings File

```
<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_2.xsd"
  version="2.2">
  <persistence-unit name="example" transaction-type="JTA">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</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>
```

12.8. SECOND-LEVEL CACHES

12.8.1. 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 runtime by keeping the data separate from the application.

JBoss EAP supports caching for the following purposes:

- Web Session Clustering
- Stateful Session Bean Clustering
- SSO Clustering
- Hibernate Second-level Cache
- Jakarta Persistence Second-level Cache

WARNING

Each cache container defines a **repl** and a **dist** cache. These caches should not be used directly by user applications.

12.8.1.1. Default Second-level Cache Provider

Infinispan is the default second-level cache provider for JBoss EAP. Infinispan is a distributed in-memory key/value data store with optional schema, available under the Apache License 2.0.

12.8.1.1.1. Configuring a Second-level Cache in the Persistence Unit

You can use the **shared-cache-mode** element of the persistence unit to configure the second-level cache.

1. See Create a Simple Jakarta Persistence Application to create the **persistence.xml** file in Red Hat CodeReady Studio.

2. Add the following to the **persistence.xml** file:

```
<persistence-unit name="...">
  (...) <!-- other configuration -->
  <shared-cache-mode>SHARED_CACHE_MODE</shared-cache-mode>
  <properties>
    <property name="hibernate.cache.use_second_level_cache" value="true" />
    <property name="hibernate.cache.use_query_cache" value="true" />
  </properties>
</persistence-unit>
```

   The ***SHARED_CACHE_MODE*** element can take the following values:

   - **ALL**: All entities should be considered cacheable.
   - **NONE**: No entities should be considered cacheable.
   - **ENABLE_SELECTIVE**: Only entities marked as cacheable should be considered cacheable.
   - **DISABLE_SELECTIVE**: All entities except the ones explicitly marked as not cacheable should be considered cacheable.
   - **UNSPECIFIED**: Behavior is not defined. Provider-specific defaults are applicable.

CHAPTER 13. JAKARTA BEAN VALIDATION

13.1. ABOUT JAKARTA BEAN VALIDATION

Jakarta Bean 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. It also offers method and constructor validation to ensure constraints on parameters and return values. The specification is documented in Jakarta Bean Validation 2.0 specification.

Hibernate Validator is the JBoss EAP implementation of Jakarta Bean Validation. It is also the reference implementation of the Jakarta Bean Validation 2.0 specification.

JBoss EAP is 100% compliant with Jakarta Bean Validation 2.0 specification. Hibernate Validator also provides additional features to the specification.

To get started with Jakarta Bean Validation, see the **bean-validation** quickstart that ships with JBoss EAP. For information about how to download and run the quickstarts, see Using the Quickstart Examples in the JBoss EAP *Getting Started Guide*.

JBoss EAP 7.3 includes Hibernate Validator 6.0.x.

Features of Hibernate Validator 6.0.x

- Jakarta Bean Validation 2.0 defines a metadata model and API for entity and method validation. The default source for the metadata is annotations, with the ability to override and extend the metadata through the use of XML.

  The API is not tied to any specific application tier or programming model. It is available for both server-side application programming and rich client Swing application development.

- In addition to bug fixes, this release of Hibernate Validator contains many performance improvements for the most common use cases.

- As of version 1.1, Jakarta Bean Validation constraints can also be applied to the parameters and return values of methods of arbitrary Java types using the Jakarta Bean Validation API.

- Hibernate Validator 6.0.x and Jakarta Bean Validation 2.0 require Java 8 or later.
  For more information, see Hibernate Validator 6.0.17.Final - JSR 380 Reference Implementation: Reference Guide.

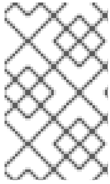
13.2. VALIDATION CONSTRAINTS

13.2.1. 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: Hibernate Validator Constraints.

13.2.2. Hibernate Validator Constraints



NOTE

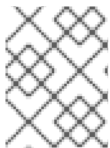
When applicable, the application-level constraints lead to creation of database-level constraints that are described in the **Hibernate Metadata Impact** column in the table below.

Java-specific Validation Constraints

The following table includes validation constraints defined in the Java specifications, which are included in the **javax.validation.constraints** package.

Annotation	Property type	Runtime checking	Hibernate Metadata impact
@AssertFalse	Boolean	Check that the method evaluates to false. Useful for constraints expressed in code rather than annotations.	None.
@AssertTrue	Boolean	Check that the method evaluates to true. Useful for constraints expressed in code rather than annotations.	None.
@Digits(integerDigits=1)	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.
@Future	Date or calendar	Check if the date is in the future.	None.
@Max(value=)	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=)	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		Check if the value is not null.	Column(s) are not null.
@Past	Date or calendar	Check if the date is in the past.	Add a check constraint on the column.

Annotation	Property type	Runtime checking	Hibernate Metadata impact
@Pattern(regexp="regexp", flag=) or @Patterns({@Pattern(...)})	String	Check if the property matches the regular expression given a match flag. See **java.util.regex.Pattern**.	None.
@Size(min=, max=)	Array, collection, map	Check if the element size is between min and max, both values included.	None.
@Valid	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.



NOTE

The parameter **@Valid** is a part of the Jakarta Bean Validation specification, even though it is located in the **javax.validation.constraints** package.

Hibernate Validator-specific Validation Constraints

The following table includes vendor-specific validation constraints, which are a part of the **org.hibernate.validator.constraints** package.

Annotation	Property type	Runtime checking	Hibernate Metadata impact
@Length(min=, max=)	String	Check if the string length matches the range.	Column length will be set to max.
@CreditCardNumber	String	Check whether the string is a well formatted credit card number, derivative of the Luhn algorithm.	None.

Annotation	Property type	Runtime checking	Hibernate Metadata impact
@EAN	String	Check whether the string is a properly formatted EAN or UPC-A code.	None.
@Email	String	Check whether the string is conform to the e-mail address specification.	None.
@NotEmpty		Check if the string is not null nor empty. Check if the connection is not null nor empty.	Columns are not null for String.
@Range(min=, max=)	Numeric or string representation of a numeric	Check if the value is between min and max, both values included.	Add a check constraint on the column.

13.2.3. Using a Jakarta Bean Validation Custom Constraint

Jakarta Bean Validation API defines a set of standard constraint annotations, such as **@NotNull**, **@Size**, and so on. However, in cases where these predefined constraints are not sufficient, you can easily create custom constraints tailored to your specific validation requirements.

Creating a Jakarta Bean Validation custom constraint requires that you create a constraint annotation and implement a constraint validator. The following abbreviated code examples are taken from the **bean-validation-custom-constraint** quickstart that ships with JBoss EAP. See that quickstart for a complete working example.

13.2.3.1. Creating A Constraint Annotation

The following example shows the **personAddress** field of entity **Person** is validated using a set of custom constraints defined in the class **AddressValidator**.

1. Create the entity **Person**.

   Example: Person Class

```
package org.jboss.as.quickstarts.bean_validation_custom_constraint;

@Entity
@Table(name = "person")
public class Person implements Serializable {

    private static final long serialVersionUID = 1L;
```

```
    @Id
    @GeneratedValue
    @Column(name = "person_id")
    private Long personId;

    @NotNull

    @Size(min = 4)
    private String firstName;

    @NotNull
    @Size(min = 4)
    private String lastName;

    // Custom Constraint @Address for bean validation
    @NotNull
    @Address
    @OneToOne(mappedBy = "person", cascade = CascadeType.ALL)
    private PersonAddress personAddress;

    public Person() {

    }

    public Person(String firstName, String lastName, PersonAddress address) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.personAddress = address;
    }

    /* getters and setters omitted for brevity*/
}
```

2. Create the constraint validator files.

Example: Address Interface

```
package org.jboss.as.quickstarts.bean_validation_custom_constraint;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;

// Linking the AddressValidator class with @Address annotation.
@Constraint(validatedBy = { AddressValidator.class })
// This constraint annotation can be used only on fields and method parameters.
@Target({ ElementType.FIELD, ElementType.PARAMETER })
@Retention(value = RetentionPolicy.RUNTIME)
@Documented
public @interface Address {

    // The message to return when the instance of MyAddress fails the validation.
```

```
    String message() default "Address Fields must not be null/empty and obey character limit
constraints";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

Example: PersonAddress Class

```
package org.jboss.as.quickstarts.bean_validation_custom_constraint;

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;

@Entity
@Table(name = "person_address")
public class PersonAddress implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @Column(name = "person_id", unique = true, nullable = false)
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Long personId;

    private String streetAddress;
    private String locality;
    private String city;
    private String state;
    private String country;
    private String pinCode;

    @OneToOne
    @PrimaryKeyJoinColumn
    private Person person;

    public PersonAddress() {

    }

    public PersonAddress(String streetAddress, String locality, String city, String state, String
country, String pinCode) {
        this.streetAddress = streetAddress;
        this.locality = locality;
        this.city = city;
        this.state = state;
        this.country = country;
```

```
        this.pinCode = pinCode;
    }

    /* getters and setters omitted for brevity*/
}
```

13.2.3.2. Implementing A Constraint Validator

Having defined the annotation, you need to create a constraint validator that is able to validate elements with an **@Address** annotation. To do so, implement the interface **ConstraintValidator** as shown below:

Example: AddressValidator Class

```
package org.jboss.as.quickstarts.bean_validation_custom_constraint;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
import org.jboss.as.quickstarts.bean_validation_custom_constraint.PersonAddress;

public class AddressValidator implements ConstraintValidator<Address, PersonAddress> {

    public void initialize(Address constraintAnnotation) {
    }

    /**
     * 1. A null address is handled by the @NotNull constraint on the @Address.
     * 2. The address should have all the data values specified.
     * 3. Pin code in the address should be of at least 6 characters.
     * 4. The country in the address should be of at least 4 characters.
     */
    public boolean isValid(PersonAddress value, ConstraintValidatorContext context) {
        if (value == null) {
            return true;
        }

        if (value.getCity() == null || value.getCountry() == null || value.getLocality() == null
            || value.getPinCode() == null || value.getState() == null || value.getStreetAddress() == null) {
            return false;
        }

        if (value.getCity().isEmpty()
            || value.getCountry().isEmpty() || value.getLocality().isEmpty()
            || value.getPinCode().isEmpty() || value.getState().isEmpty() ||
value.getStreetAddress().isEmpty()) {
            return false;
        }

        if (value.getPinCode().length() < 6) {
            return false;
        }

        if (value.getCountry().length() < 4) {
            return false;
        }
```

```
        return true;
    }
}
```

13.3. JAKARTA BEAN VALIDATION CONFIGURATION

You can configure Jakarta Bean Validation using XML descriptors in the **validation.xml** file located in the /**META-INF** directory. If this file exists in the class path, its configuration is applied when the **ValidatorFactory** gets created.

Example: Jakarta Bean Validation Configuration File

The following example shows several configuration options of the **validation.xml** file. All the settings are optional. These options can also be configured using the **javax.validation** package.

```
<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>
```

The node **default-provider** allows to choose the Jakarta Bean Validation provider. This is useful if there is more than one provider on the classpath. The **message-interpolator** and **constraint-validator-factory** properties are used to customize the used implementations for the interfaces **MessageInterpolator** and **ConstraintValidatorFactory**, which are defined in the **javax.validation** package. The **constraint-mapping** element lists additional XML files containing the actual constraint configuration.

CHAPTER 14. CREATING JAKARTA WEBSOCKET APPLICATIONS

The Jakarta WebSocket protocol provides two-way communication between web clients and servers. Communications between clients and the server are event-based, allowing for faster processing and smaller bandwidth compared with polling-based methods. Jakarta WebSocket is available for use in web applications using a JavaScript API and by client Jakarta WebSocket endpoints using the Jakarta WebSocket specification.

A connection is first established between client and server as an HTTP connection. The client then requests a Jakarta WebSocket connection using the **Upgrade** header. All communications are then full-duplex over the same TCP/IP connection, with minimal data overhead. Because each message does not include unnecessary HTTP header content, Jakarta WebSocket communications require smaller bandwidth. The result is a low latency communications path suited to applications, which require real-time responsiveness.

The JBoss EAP Jakarta WebSocket implementation provides full dependency injection support for server endpoints, however, it does not provide Contexts and Dependency Injection services for client endpoints.

A Jakarta WebSocket application requires the following components and configuration changes:

- A Java client or a Jakarta WebSocket enabled HTML client. You can verify HTML client browser support at this location: http://caniuse.com/#feat=websockets
- A Jakarta WebSocket server endpoint class.
- Project dependencies configured to declare a dependency on the Jakarta WebSocket API.

Create the Jakarta WebSocket Application

The code examples that follow are taken from the **websocket-hello** quickstart that ships with JBoss EAP. It is a simple example of a Jakarta WebSocket application that opens a connection, sends a message, and closes a connection. It does not implement any other functions or include any error handling, which would be required for a real world application.

1. Create the JavaScript HTML client.
   The following is an example of a Jakarta WebSocket client. It contains these JavaScript functions:

   - **connect()**: This function creates the Jakarta WebSocket connection passing the Jakarta WebSocket URI. The resource location matches the resource defined in the server endpoint class. This function also intercepts and handles the Jakarta WebSocket **onopen**, **onmessage**, **onerror**, and **onclose**.
   - **sendMessage()**: This function gets the name entered in the form, creates a message, and sends it using a WebSocket.send() command.
   - **disconnect()**: This function issues the WebSocket.close() command.
   - **displayMessage()**: This function sets the display message on the page to the value returned by the Jakarta WebSocket endpoint method.
   - **displayStatus()**: This function displays the Jakarta WebSocket connection status.

   Example: Application index.html Code

```
<html>
  <head>
    <title>WebSocket: Say Hello</title>
    <link rel="stylesheet" type="text/css" href="resources/css/hello.css" />
    <script type="text/javascript">
      var websocket = null;
      function connect() {
        var wsURI = 'ws://' + window.location.host + '/websocket-
hello/websocket/helloName';
        websocket = new WebSocket(wsURI);
        websocket.onopen = function() {
            displayStatus('Open');
            document.getElementById('sayHello').disabled = false;
            displayMessage('Connection is now open. Type a name and click Say Hello to
send a message.');
        };
        websocket.onmessage = function(event) {
            // log the event
            displayMessage('The response was received! ' + event.data, 'success');
        };
        websocket.onerror = function(event) {
            // log the event
            displayMessage('Error! ' + event.data, 'error');
        };
        websocket.onclose = function() {
            displayStatus('Closed');
            displayMessage('The connection was closed or timed out. Please click the Open
Connection button to reconnect.');
            document.getElementById('sayHello').disabled = true;
        };
      }
      function disconnect() {
        if (websocket !== null) {
            websocket.close();
            websocket = null;
        }
        message.setAttribute("class", "message");
        message.value = 'WebSocket closed.';
        // log the event
      }
      function sendMessage() {
        if (websocket !== null) {
            var content = document.getElementById('name').value;
            websocket.send(content);
        } else {
            displayMessage('WebSocket connection is not established. Please click the Open
Connection button.', 'error');
        }
      }
      function displayMessage(data, style) {
        var message = document.getElementById('hellomessage');
        message.setAttribute("class", style);
        message.value = data;
      }
      function displayStatus(status) {
        var currentStatus = document.getElementById('currentstatus');
```

```
        currentStatus.value = status;
      }
    </script>
  </head>
  <body>
    <div>
      <h1>Welcome to Red Hat JBoss Enterprise Application Platform!</h1>
      <div>This is a simple example of a Jakarta WebSocket implementation.</div>
      <div id="connect-container">
        <div>
          <fieldset>
            <legend>Connect or disconnect using websocket :</legend>
            <input type="button" id="connect" onclick="connect();" value="Open Connection"
/>
            <input type="button" id="disconnect" onclick="disconnect();" value="Close
Connection" />
          </fieldset>
        </div>
        <div>
            <fieldset>
              <legend>Type your name below, then click the `Say Hello` button :</legend>
              <input id="name" type="text" size="40" style="width: 40%"/>
              <input type="button" id="sayHello" onclick="sendMessage();" value="Say Hello"
disabled="disabled"/>
            </fieldset>
        </div>
        <div>Current WebSocket Connection Status: <output id="currentstatus"
class="message">Closed</output></div>
        <div>
          <output id="hellomessage" />
        </div>
      </div>
    </div>
  </body>
</html>
```

2. Create the Jakarta WebSocket server endpoint.
   You can create a Jakarta WebSocket server endpoint using either of the following methods.

   - **Programmatic Endpoint**: The endpoint extends the Endpoint class.
   - **Annotated Endpoint**: The endpoint class uses annotations to interact with the Jakarta WebSocket events. It is simpler to code than the programmatic endpoint.

   The code example below uses the annotated endpoint approach and handles the following events.

   - The **@ServerEndpoint** annotation identifies this class as a Jakarta WebSocket server endpoint and specifies the path.
   - The **@OnOpen** annotation is triggered when the Jakarta WebSocket connection is opened.
   - The **@OnMessage** annotation is triggered when a message is received.
   - The **@OnClose** annotation is triggered when the Jakarta WebSocket connection is closed.

   Example: Jakarta WebSocket Endpoint Code

```
package org.jboss.as.quickstarts.websocket_hello;

import javax.websocket.CloseReason;
import javax.websocket.OnClose;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/websocket/helloName")
public class HelloName {

    @OnMessage
    public String sayHello(String name) {
        System.out.println("Say hello to '" + name + "'");
        return ("Hello" + name);
    }

    @OnOpen
    public void helloOnOpen(Session session) {
        System.out.println("WebSocket opened: " + session.getId());
    }

    @OnClose
    public void helloOnClose(CloseReason reason) {
        System.out.println("WebSocket connection closed with CloseCode: " +
reason.getCloseCode());
    }
}
```

3. Declare the Jakarta WebSocket API dependency in your project POM file.
   If you use Maven, you add the following dependency to the project **pom.xml** file.

   Example: Maven Dependency

```
<dependency>
  <groupId>org.jboss.spec.javax.websocket</groupId>
  <artifactId>jboss-websocket-api_1.1_spec</artifactId>
  <scope>provided</scope>
</dependency>
```

The quickstarts that ship with JBoss EAP include additional Jakarta WebSocket client and endpoint code examples.

CHAPTER 15. JAKARTA AUTHORIZATION

15.1. ABOUT JAKARTA AUTHORIZATION

Jakarta Authorization is a standard which defines a contract between containers and authorization service providers, which results in the implementation of providers for use by containers. For details about the specifications, see Jakarta Authorization specification.

JBoss EAP implements support for Jakarta Authorization within the security functionality of the **security** subsystem.

15.2. CONFIGURE JAKARTA AUTHORIZATION SECURITY

You can configure Jakarta Authorization by configuring your security domain with the correct module, and then modifying your **jboss-web.xml** to include the required parameters.

Add Jakarta Authentication to the Security Domain

To add Jakarta Authorization support to the security domain, add the Jakarta Authorization 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 Jakarta Authorization support. However, it is recommended to configure the security domain from the management console or the management CLI, rather than directly modifying the XML.

Example: Security Domain with Jakarta Authentication

```
<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 Jakarta Authentication

The **jboss-web.xml** file is located in the **WEB-INF/** directory of your deployment, and contains overrides and additional JBoss-specific configuration for the web container. To use your Jakarta Authorization-enabled security domain, you need to include the **<security-domain>** element, and also set the **<use-jboss-authorization>** element to **true**. The following XML is configured to use the Jakarta Authorization security domain above.

Example: Use the Jakarta Authentication Security Domain

```
<jboss-web>
  <security-domain>jacc</security-domain>
  <use-jboss-authorization>true</use-jboss-authorization>
</jboss-web>
```

Configure an Jakarta Enterprise Beans Application to Use Jakarta Authentication

Configuring Jakarta Enterprise Beans to use a security domain and to use Jakarta Authorization differs

from web applications. For an Jakarta Enterprise Beans, 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 Jakarta Authorization roles. See the example configuration below for details. The **EJBMethodPermission** class is part of the Jakarta EE API, and is documented at Class EJBMethodPermission.

Example: Jakarta Authentication Method Permissions in an Jakarta Enterprise Beans

```
<ejb-jar>
  <assembly-descriptor>
    <method-permission>
      <description>The employee and temp-employee roles can access any method of the
EmployeeService bean </description>
      <role-name>employee</role-name>
      <role-name>temp-employee</role-name>
      <method>
        <ejb-name>EmployeeService</ejb-name>
        <method-name>*</method-name>
      </method>
    </method-permission>
  </assembly-descriptor>
</ejb-jar>
```

You can also constrain the authentication and authorization mechanisms for an Jakarta Enterprise Beans 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 that the Jakarta Enterprise Beans runs as.

Example: Security Domain Declaration in an Jakarta Enterprise Beans

```
<ejb-jar>
    <assembly-descriptor>
        <security>
        <ejb-name>*</ejb-name>
        <security-domain>myDomain</security-domain>
        <run-as-principal>myPrincipal</run-as-principal>
        </security>
    </assembly-descriptor>
</ejb-jar>
```

Enabling Jakarta Authorization Using the elytron Subsystem

Disable Jakarta Authentication in the Legacy Security Subsystem

By default, the application server uses the legacy **security** subsystem to configure the Jakarta Authorization policy provider and factory. The default configuration maps to implementations from PicketBox.

In order to use Elytron to manage Jakarta Authorization configuration, or any other policy you want to install to the application server, you must first disable Jakarta Authorization in the legacy **security** subsystem. For that, you can use the following management CLI command:

```
/subsystem=security:write-attribute(name=initialize-jacc, value=false)
```

Failure to do so can result in the following error in the server log: **MSC000004: Failure during stop of service org.wildfly.security.policy: java.lang.StackOverflowError**.

Define a Jakarta Authentication Policy Provider

The **elytron** subsystem provides a built-in policy provider based on Jakarta Authorization specification. To create the policy provider you can execute the following management CLI command:

```
/subsystem=elytron/policy=jacc:add(jacc-policy={})

reload
```

Enable Jakarta Authentication to a Web Deployment

Once a Jakarta Authorization policy provider is defined, you can enable Jakarta Authorization for web deployments by executing the following command:

```
/subsystem=undertow/application-security-domain=other:add(security-
domain=ApplicationDomain,enable-jacc=true)
```

The command above defines a default security domain for applications, if none is provided in the **jboss-web.xml** file. In case you already have a **application-security-domain** defined and just want to enable Jakarta Authorization you can execute the following command:

```
/subsystem=undertow/application-security-domain=my-security-domain:write-attribute(name=enable-
jacc,value=true)
```

Enable Jakarta Authentication to an Jakarta Enterprise Beans Deployment

Once a Jakarta Authorization policy provider is defined, you can enable Jakarta Authorization for Jakarta Enterprise Beans deployments by executing the following command:

```
/subsystem=ejb3/application-security-domain=other:add(security-domain=ApplicationDomain,enable-
jacc=true)
```

The command above defines a default security domain for Jakarta Enterprise Beans. In case you already have a **application-security-domain** defined and just want to enable Jakarta Authorization you can execute a command as follows:

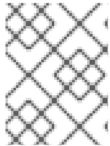
```
/subsystem=ejb3/application-security-domain=my-security-domain:write-attribute(name=enable-
jacc,value=true)
```

Creating a Custom Elytron Policy Provider

A custom policy provider is used when you need a custom **java.security.Policy**, like when you want to integrate with some external authorization service in order to check permissions. To create a custom policy provider, you will need to implement the **java.security.Policy**, create and plug in a custom module with the implementation and use the implementation from the module in the **elytron** subsystem.

```
/subsystem=elytron/policy=policy-provider-a:add(custom-policy={class-name=MyPolicyProviderA,
module=x.y.z})
```

For more information, see the Policy Provider Properties .



NOTE

In most cases, you can use the Jakarta Authorization policy provider as it is expected to be part of any Jakarta EE compliant application server.

CHAPTER 16. JAKARTA AUTHENTICATION

16.1. ABOUT JAKARTA AUTHENTICATION SECURITY

Jakarta Authentication is a pluggable interface for Java applications. For information about the specification, see the Jakarta Authentication specification.

16.2. CONFIGURE JAKARTA AUTHENTICATION

You can authenticate a Jakarta Authentication provider by adding **<authentication-jaspi>** element to your security domain. The configuration is similar to that of a standard authentication module, but login module elements are enclosed in a **<login-module-stack>** element. The structure of the configuration is:

Example: 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 the same way as a standard authentication module.

The web-based management console does not expose the configuration of JASPI authentication modules. You must stop the JBoss EAP running instance completely before adding the configuration directly to the ***EAP_HOME*/domain/configuration/domain.xml** file or the ***EAP_HOME*/standalone/configuration/standalone.xml** file.

16.3. CONFIGURE JAKARTA AUTHENTICATION SECURITY USING ELYTRON

Starting in JBoss EAP 7.3, the **elytron** subsystem provides an implementation of the **Servlet** profile from the Jakarta Authentication. This allows tighter integration with the security features provided by Elytron.

Enabling Jakarta Authentication for a Web Application

For the Jakarta Authentication integration to be enabled for a web application, the web application needs to be associated with either an Elytron **http-authentication-factory** or a **security-domain**. By doing this, the Elytron security handlers get installed for the deployment and the Elytron security framework gets activated for the deployment.

When the Elytron security framework is activated for a deployment, the globally registered **AuthConfigFactory** is queried when requests are handled. It will identify if an **AuthConfigProvider**, which should be used for that deployment, has been registered. If an **AuthConfigProvider** is found, then JASPI authentication will be used instead of the deployment's authentication configuration. If no **AuthConfigProvider** is found, then the authentication configuration for the deployment will be used instead. This could result in one of the three possibilities:

- Use of authentication mechanisms from an **http-authentication-factory**.
- Use of mechanisms specified in the **web.xml**.
- No authentication is performed if the application does not have any mechanisms defined.

Any updates made to the **AuthConfigFactory** are immediately available. This means if an **AuthConfigProvider** is registered and is a match for an existing application, it will start to be used immediately without requiring redeployment of the application.

All web applications deployed to JBoss EAP have a security domain, which will be resolved in the following order:

1. From the deployment descriptors or annotations of the deployment.
2. The value defined on the **default-security-domain** attribute on the **undertow** subsystem.
3. Default to **other**.

> **NOTE**
>
> It is assumed that this security domain is a reference to the **PicketBox** security domain, so the final step in activation is ensuring this is mapped to Elytron using an **application-security-domain** resource in the **undertow** subsystem.
>
> This mapping can do one of the following:
>
> - Reference an **elytron** security domain directly, for example:
>
> ```
> /subsystem=undertow/application-security-domain=MyAppSecurity:add(security-domain=ApplicationDomain)
> ```
>
> - Reference a **http-authentication-factory** resource to obtain instances of authentication mechanisms, for example:
>
> ```
> /subsystem=undertow/application-security-domain=MyAppSecurity:add(http-authentication-factory=application-http-authentication)
> ```

The minimal steps to enable the Jakarta Authentication integration are:

1. Leave the **default-security-domain** attribute on the **undertow** subsystem undefined so that it defaults to **other**.
2. Add an **application-security-domain** mapping from **other** to an Elytron security domain.

The security domain associated with a deployment in these steps is the security domain that will be wrapped in a **CallbackHandler** to be passed into the **ServerAuthModule** instances used for authentication.

Additional Options

Two additional attributes have been added to the **application-security-domain** resource to allow some further control of the Jakarta Authentication behavior.

Table 16.1. Attributes Added to the application-security-domain Resource

Attribute	Description
enable-jaspi	Can be set to **false** to disable Jakarta Authentication support for all deployments using this mapping.
integrated-jaspi	By default, all identities are loaded from the security domain. If set to **false**, ad-hoc identities will be created instead.

Subsystem Configuration

One way to register a configuration that will result in an **AuthConfigProvider** being returned for a deployment is to register a **jaspi-configuration** in the **elytron** subsystem.

The following command demonstrates how to add a configuration containing two **ServerAuthModule** definitions.

```
/subsystem=elytron/jaspi-configuration=simple-configuration:add(layer=HttpServlet, application-context="default-host /webctx", description="Elytron Test Configuration", server-auth-modules=[{class-name=org.wildfly.security.examples.jaspi.SimpleServerAuthModule, module=org.wildfly.security.examples.jaspi, flag=OPTIONAL, options={a=b, c=d}}, {class-name=org.wildfly.security.examples.jaspi.SecondServerAuthModule, module=org.wildfly.security.examples.jaspi}])
```

This results in the following configuration being persisted.

```
<jaspi>
    <jaspi-configuration name="simple-configuration" layer="HttpServlet" application-context="default-host /webctx" description="Elytron Test Configuration">
        <server-auth-modules>
            <server-auth-module class-name="org.wildfly.security.examples.jaspi.SimpleServerAuthModule" module="org.wildfly.security.examples.jaspi" flag="OPTIONAL">
                <options>
                    <property name="a" value="b"/>
                    <property name="c" value="d"/>
                </options>
            </server-auth-module>
            <server-auth-module class-name="org.wildfly.security.examples.jaspi.SecondServerAuthModule" module="org.wildfly.security.examples.jaspi"/>
        </server-auth-modules>
    </jaspi-configuration>
</jaspi>
```

NOTE

The **name** attribute is just a name that allows the resource to be referenced in the management model.

The **layer** and **application-context** attributes are used when registering this configuration with the **AuthConfigFactory**. Both of these attributes can be omitted allowing wildcard matching. The **description** attribute is also optional and is used to provide a description to the **AuthConfigFactory**.

Within the configuration, one or more **server-auth-module** instances can be defined with the following attributes.

- **class-name** - The fully qualified class name of the **ServerAuthModule**.
- **module** - The module to load the **ServerAuthModule** from.
- **flag** - The control flag to indicate how this module operates in relation to the other modules.
- **options** - Configuration options to be passed into the **ServerAuthModule** on initialization.

Configuration defined in this way is immediately registered with the **AuthConfigFactory**. Any existing deployments using the Elytron security framework, that matches the **layer** and **application-context**, will immediately start making use of this configuration.

Programmatic Configuration

The APIs defined within the Jakarta Authentication specification allow for applications to dynamically register custom **AuthConfigProvider** instances. However, the specification does not provide the actual implementations to be used or any standard way to create instances of the implementations. The Elytron project contains a simple utility that deployments can use to help with this.

The following code example demonstrates how to use this API to register a configuration similar to the one illustrated in the Subsystem Configuration above.

```
String registrationId = org.wildfly.security.auth.jaspi.JaspiConfigurationBuilder.builder("HttpServlet",
servletContext.getVirtualServerName() + " " + servletContext.getContextPath())
    .addAuthModuleFactory(SimpleServerAuthModule::new, Flag.OPTIONAL,
Collections.singletonMap("a", "b"))
    .addAuthModuleFactory(SecondServerAuthModule::new)
.register();
```

As an example, this code could be executed within the **init()** method of a Servlet to register the **AuthConfigProvider** specific for that deployment. In this code example, the application context has also been assembled by consulting the **ServletContext**.

The **register()** method returns the resulting registration ID that can also be used to subsequently remove this registration directly from the **AuthConfigFactory**.

As with the Subsystem Configuration, this call also has an immediate effect and will be live for all web applications using the Elytron security framework.

Authentication Process

Based on the configuration of the **application-security-domain** resource in the **undertow** subsystem, the **CallbackHandler** passed to the **ServerAuthModule** can operate in either of the following modes:

- Integrated Mode
- Non-integrated Mode

Integrated Mode

When operating in the integrated mode, although the **ServerAuthModule** instances will be handling the actual authentication, the resulting identity will be loaded from the referenced **SecurityDomain** using the **SecurityRealms** referenced by that **SecurityDomain**. In this mode, it is still possible to override the roles that will be assigned within the servlet container.

The advantage of this mode is that **ServerAuthModules** are able to take advantage of the Elytron configuration for the loading of identities, so that the identities stored in the usual locations, such as

databases and LDAP, can be loaded without the **ServerAuthModule** needing to be aware of these locations. In addition, other Elytron configuration can be applied, such as role and permission mapping. The referenced **SecurityDomain** can also be referenced in other places, such as for SASL authentication or other non JASPI applications, all backed by a common repository of identities.

Table 16.2. Operations of the CallbackHandlers method in the integrated mode.

<table>
<tr><th>Operation</th><th>Description</th></tr>
<tr><td>PasswordValidationCallback</td><td>The username and password will be used with the SecurityDomain to perform an authentication. If successful, there will be an authenticated identity.</td></tr>
<tr><td>CallerPrincipalCallback</td><td>This Callback is used to establish the authorized identity or the identity that will be available once the request reached the web application.

NOTE

If an authenticated identity has already been established via the PasswordValidationCallback, this Callback is interpreted as a run-as request. In this case, authorization checks are performed to ensure the authenticated identity is authorized to run as the identity specified in this Callback. If no authenticated identity has been established by a PasswordValidationCallback, it is assumed that the ServerAuthModule has handled the authentication step.

If a Callback is received with a null Principal and name then:
If an authenticated identity has already been established, authorization will be performed as that identity.
If no identity has been established, authorization of the anonymous identity will be performed.
Where authorization of the anonymous identity is performed, the SecurityDomain must have been configured to grant the anonymous identity the LoginPermission.</td></tr>
<tr><td>GroupPrincipalCallback</td><td>In this mode, the attribute loading, role decoding, and role mapping configured on the security domain are used to establish the identity. If this Callback is received, the groups specified are used to determine the roles that are assigned to the identity. The request will be in the servlet container and these roles are visible in the servlet container only.</td></tr>
</table>

Non-Integrated Mode

When operating in non-integrated mode, the **ServerAuthModules** are completely responsible for all authentication and identity management. The specified **Callbacks** can be used to establish an identity.

The resulting identity will be created on the **SecurityDomain** but it will be independent of any identities stored in referenced **SecurityRealms**.

The advantage of this mode is that JASPI configurations that are able to completely handle the identities can be deployed to the application server without requiring anything beyond a simple **SecurityDomain** definitions. There is no need for this **SecurityDomain** to actually contain the identities that will be used at runtime. The disadvantage of this mode is that the **ServerAuthModule** is now responsible for all identity handling, potentially making the implementation much more complex.

Table 16.3. Operations of theCallbackHandlers method in the non-integrated mode.

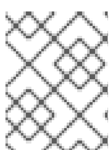
Operation	Description
PasswordValidationCallback	The **Callback** is not supported in this mode. The purpose of this mode is for the **ServerAuthModule** to operate independently of the referenced **SecurityDomain**. Requesting a password to be validated would not be suitable.
CallerPrincipalCallback	This **Callback** is used to establish the Principal for the resulting identity. Because the **ServerAuthModule** is handling all of the identity checking requirements, no checks are performed to verify if the identity exists in the security domain and no authorization checks are performed. If a **Callback** is received with a null Principal and name, then the identity will be established as the anonymous identity. Because the **ServerAuthModule** is making the decisions, no authorization check will be performed with the **SecurityDomain**.
GroupPrincipalCallback	As the identity is created in this mode without loading from the **SecurityDomain**, it will by default have no roles assigned. If this **Callback** is received, the groups will be taken and assigned to the resulting identity while the request is in the servlet container. These roles will be visible in the servlet container only.

validateRequest
During the call to **validateRequest** on the **ServerAuthContext**, the individual **ServerAuthModule** instances will be called in the order in which they are defined. A control flag can also be specified for each module. This flag defines how the response should be interpreted and if processing should continue to the next server authentication module or return immediately.

Control Flags
Whether the configuration is provided within the **elytron** subsystem or using the **JaspiConfigurationBuilder** API, it is possible to associate a control flag with each **ServerAuthModule**. If one is not specified, it defaults to **REQUIRED**. The flags have the following meanings depending on their result.

Flag	AuthStatus.SEND_SUCCESS	AuthStatus.SEND_FAILURE, AuthStatus.SEND_CONTINUE

Flag	AuthStatus.SEND_SUCCESS	AuthStatus.SEND_FAILURE, AuthStatus.SEND_CONTINUE
Required	Validation will continue to the remaining modules. Provided the requirements of the remaining modules are satisfied, the request will be allowed to proceed to authorization.	Validation will continue to the remaining modules; however, regardless of their outcomes, the validation will not be successful and control will return to the client.
Requisite	Validation will continue to the remaining modules. Provided the requirements of the remaining modules are satisfied, the request will be allowed to proceed to authorization.	The request will return immediately to the client.
Sufficient	Validation is deemed successful and complete, provided no previous **Required** or **Requisite** module has returned an **AuthStatus** other than **AuthStatus.SUCCESS**. The request will proceed to authorization of the secured resource.	Validation will continue down the list of remaining modules. This status will only affect the decision if there are no **REQUIRED** or **REQUISITE** modules.
Optional	Validation will continue to the remaining modules, provided any **Required** or **Requisite** modules have not returned **SUCCESS**. This will be sufficient for validation to be deemed successful and for the request to proceed to the authorization stage and the secured resource.	Validation will continue down the list of remaining modules. This status will only affect the decision if there are no **REQUIRED** or **REQUISITE** modules.



NOTE

For all **ServerAuthModule** instances, if they throw an **AuthException**, an error will be immediately reported to the client with no further module calls.

secureResponse

During the call to **secureResponse**, each **ServerAuthModule** is called, but this time in reverse order where a module only undertakes an action in **secureResponse**. If the module undertook an action in **validateResponse**, it is the responsibility of the module to track this.

The control flag has no effect on **secureResponse** processing. Processing ends when one of the following is true:

- All of the **ServerAuthModule** instances have been called.

- A module returns **AuthStatus.SEND_FAILURE**.
- A module throws an **AuthException**.

SecurityIdentity Creation

Once the authentication process has completed, the **org.wildfly.security.auth.server.SecurityIdentity** for the deployment's **SecurityDomain** will have been created as a result of the **Callbacks** to the **CallbackHandler**. Depending on the **Callbacks**, this will either be an identity loaded directly from the **SecurityDomain**, or it will be an ad-hoc identity described by the callbacks. This **SecurityIdentity** will be associated with the request, in the same way it is done for other authentication mechanisms.

CHAPTER 17. JAKARTA SECURITY

17.1. ABOUT JAKARTA SECURITY

Jakarta Security defines plug-in interfaces for authentication and identity stores, and a new injectable-type SecurityContext interface that provides an access point for programmatic security. For details about the specifications, see Jakarta Security Specification.

17.2. CONFIGURE JAKARTA SECURITY USING ELYTRON

Enabling Jakarta Security Using the elytron Subsystem

The **SecurityContext** interface defined in Jakarta Security uses the Jakarta Authorization policy provider to access the current authenticated identity. To enable your deployments to use the **SecurityContext** interface, you must configure the **elytron** subsystem to manage the Jakarta Authorization configuration and define a default Jakarta Authorization policy provider.

1. Disable Jakarta Authorization in the legacy **security** subsystem. Skip this step if Jakarta Authorization is already configured to be managed by Elytron.

```
/subsystem=security:write-attribute(name=initialize-jacc, value=false)
```

2. Define a Jakarta Authorization policy provider in the **etlyron** subsystem and reload the server.

```
/subsystem=elytron/policy=jacc:add(jacc-policy={})
reload
```

Enabling Jakarta Security for Web Applications

To enable Jakarta Security for a web application, the web application needs to be associated with either an Elytron **http-authentication-factory** or a **security-domain**. This installs the Elytron security handlers and activates the Elytron security framework for the deployment.

The minimal steps to enable Jakarta Security are:

1. Leave the **default-security-domain** attribute on the **undertow** subsystem undefined so that it defaults to **other**.

2. Add an **application-security-domain** mapping from **other** to an Elytron security domain:

```
/subsystem=undertow/application-security-domain=other:add(security-
domain=ApplicationDomain, integrated-jaspi=false)
```

When **integrated-jaspi** is set to **false**, ad-hoc identities are created dynamically.

Jakarta Security is built on Jakarta Authentication. For information about configuring Jakarta Authentication, see Configure Jakarta Authentication Security Using Elytron .

CHAPTER 18. JAKARTA BATCH APPLICATION DEVELOPMENT

Beginning with JBoss EAP 7, JBoss EAP supports Java batch applications as defined by JSR-352 - the equivalent Jakarta EE specification is Jakarta Batch.

The **batch-jberet** subsystem in JBoss EAP facilitates batch configuration and monitoring.

To configure your application to use batch processing on JBoss EAP, you must specify the required dependencies. Additional JBoss EAP features for batch processing include Job Specification Language (JSL) inheritance, and batch property injections.

18.1. REQUIRED BATCH DEPENDENCIES

To deploy your batch application to JBoss EAP, some additional dependencies that are required for batch processing need to be declared in your application's **pom.xml**. An example of these required dependencies is shown below. Most of the dependencies have the scope set to **provided**, as they are already included in JBoss EAP.

Example: pom.xml Batch Dependencies

```
<dependencies>
   <dependency>
      <groupId>org.jboss.spec.javax.batch</groupId>
      <artifactId>jboss-batch-api_1.0_spec</artifactId>
      <scope>provided</scope>
   </dependency>

   <dependency>
      <groupId>javax.enterprise</groupId>
      <artifactId>cdi-api</artifactId>
      <scope>provided</scope>
   </dependency>

   <dependency>
      <groupId>org.jboss.spec.javax.annotation</groupId>
      <artifactId>jboss-annotations-api_1.2_spec</artifactId>
      <scope>provided</scope>
   </dependency>

   <!-- Include your application's other dependencies. -->
   ...
</dependencies>
```

18.2. JOB SPECIFICATION LANGUAGE (JSL) INHERITANCE

A feature of the JBoss EAP **batch-jberet** subsystem is the ability to use Job Specification Language (JSL) inheritance to abstract out some common parts of your job definition. Although JSL inheritance is not included in the JSR-352 1.0 specification, the JBoss EAP **batch-jberet** subsystem implements JSL inheritance based on the JSL Inheritance v1 draft and the Jakarta Batch specifications.

Inherit Step and Flow Within the Same Job XML File

Parent elements, for example step and flow, are marked with the attribute **abstract="true"** to exclude them from direct execution. Child elements contain a **parent** attribute, which points to the parent element.

```
<job id="inheritance" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
    <!-- abstract step and flow -->
    <step id="step0" abstract="true">
        <batchlet ref="batchlet0"/>
    </step>

    <flow id="flow0" abstract="true">
        <step id="flow0.step1" parent="step0"/>
    </flow>

    <!-- concrete step and flow -->
    <step id="step1" parent="step0" next="flow1"/>

    <flow id="flow1" parent="flow0"/>
</job>
```

Inherit a Step from a Different Job XML File

Child elements, for example step and job, contain:

- A **jsl-name** attribute, which specifies the job XML file name, without the **.xml** extension, containing the parent element.
- A **parent** attribute, which points to the parent element in the job XML file specified by **jsl-name**.

Parent elements are marked with the attribute **abstract="true"** to exclude them from direct execution.

Example: chunk-child.xml

```
<job id="chunk-child" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
    <step id="chunk-child-step" parent="chunk-parent-step" jsl-name="chunk-parent">
    </step>
</job>
```

Example: chunk-parent.xml

```
<job id="chunk-parent" >
    <step id="chunk-parent-step" abstract="true">
        <chunk checkpoint-policy="item" skip-limit="5" retry-limit="5">
            <reader ref="R1"></reader>
            <processor ref="P1"></processor>
            <writer ref="W1"></writer>

            <checkpoint-algorithm ref="parent">
                <properties>
                    <property name="parent" value="parent"></property>
                </properties>
            </checkpoint-algorithm>
            <skippable-exception-classes>
                <include class="java.lang.Exception"></include>
                <exclude class="java.io.IOException"></exclude>
            </skippable-exception-classes>
            <retryable-exception-classes>
                <include class="java.lang.Exception"></include>
                <exclude class="java.io.IOException"></exclude>
            </retryable-exception-classes>
```

```
            <no-rollback-exception-classes>
                <include class="java.lang.Exception"></include>
                <exclude class="java.io.IOException"></exclude>
            </no-rollback-exception-classes>
        </chunk>
    </step>
</job>
```

18.3. BATCH PROPERTY INJECTIONS

A feature of the JBoss EAP **batch-jberet** subsystem is the ability to have properties defined in the job XML file injected into fields in the batch artifact class. Properties defined in the job XML file can be injected into fields using the **@Inject** and **@BatchProperty** annotations.

The injection field can be any of the following Java types:

- **java.lang.String**
- **java.lang.StringBuilder**
- **java.lang.StringBuffer**
- any primitive type, and its wrapper type:
  - **boolean**, **Boolean**
  - **int**, **Integer**
  - **double**, **Double**
  - **long**, **Long**
  - **char**, **Character**
  - **float**, **Float**
  - **short**, **Short**
  - **byte**, **Byte**
- **java.math.BigInteger**
- **java.math.BigDecimal**
- **java.net.URL**
- **java.net.URI**
- **java.io.File**
- **java.util.jar.JarFile**
- **java.util.Date**
- **java.lang.Class**
- **java.net.Inet4Address**

- **java.net.Inet6Address**
- **java.util.List**, **List<?>**, **List<String>**
- **java.util.Set**, **Set<?>**, **Set<String>**
- **java.util.Map**, **Map<?, ?>**, **Map<String, String>**, **Map<String, ?>**
- **java.util.logging.Logger**
- **java.util.regex.Pattern**
- **javax.management.ObjectName**

The following array types are also supported:

- **java.lang.String[]**
- any primitive type, and its wrapper type:
  - **boolean[]**, **Boolean[]**
  - **int[]**, **Integer[]**
  - **double[]**, **Double[]**
  - **long[]**, **Long[]**
  - **char[]**, **Character[]**
  - **float[]**, **Float[]**
  - **short[]**, **Short[]**
  - **byte[]**, **Byte[]**
- **java.math.BigInteger[]**
- **java.math.BigDecimal[]**
- **java.net.URL[]**
- **java.net.URI[]**
- **java.io.File[]**
- **java.util.jar.JarFile[]**
- **java.util.zip.ZipFile[]**
- **java.util.Date[]**
- **java.lang.Class[]**

Shown below are a few examples of using batch property injections:

- Injecting a Number into a Batchlet Class as Various Types

- Injecting a Number Sequence into a Batchlet Class as Various Arrays
- Injecting a Class Property into a Batchlet Class
- Assigning a Default Value to a Field Annotated for Property Injection

Injecting a Number into a Batchlet Class as Various Types

Example: Job XML File

```
<batchlet ref="myBatchlet">
    <properties>
        <property name="number" value="10"/>
    </properties>
</batchlet>
```

Example: Artifact Class

```
@Named
public class MyBatchlet extends AbstractBatchlet {
    @Inject
    @BatchProperty
    int number;  // Field name is the same as batch property name.

    @Inject
    @BatchProperty (name = "number")  // Use the name attribute to locate the batch property.
    long asLong;  // Inject it as a specific data type.

    @Inject
    @BatchProperty (name = "number")
    Double asDouble;

    @Inject
    @BatchProperty (name = "number")
    private String asString;

    @Inject
    @BatchProperty (name = "number")
    BigInteger asBigInteger;

    @Inject
    @BatchProperty (name = "number")
    BigDecimal asBigDecimal;
}
```

Injecting a Number Sequence into a Batchlet Class as Various Arrays

Example: Job XML File

```
<batchlet ref="myBatchlet">
    <properties>
        <property name="weekDays" value="1,2,3,4,5,6,7"/>
    </properties>
</batchlet>
```

Example: Artifact Class

```
@Named
public class MyBatchlet extends AbstractBatchlet {
    @Inject
    @BatchProperty
    int[] weekDays;  // Array name is the same as batch property name.

    @Inject
    @BatchProperty (name = "weekDays")  // Use the name attribute to locate the batch property.
    Integer[] asIntegers;  // Inject it as a specific array type.

    @Inject
    @BatchProperty (name = "weekDays")
    String[] asStrings;

    @Inject
    @BatchProperty (name = "weekDays")
    byte[] asBytes;

    @Inject
    @BatchProperty (name = "weekDays")
    BigInteger[] asBigIntegers;

    @Inject
    @BatchProperty (name = "weekDays")
    BigDecimal[] asBigDecimals;

    @Inject
    @BatchProperty (name = "weekDays")
    List asList;

    @Inject
    @BatchProperty (name = "weekDays")
    List<String> asListString;

    @Inject
    @BatchProperty (name = "weekDays")
    Set asSet;

    @Inject
    @BatchProperty (name = "weekDays")
    Set<String> asSetString;
}
```

Injecting a Class Property into a Batchlet Class

Example: Job XML File

```
<batchlet ref="myBatchlet">
    <properties>
        <property name="myClass" value="org.jberet.support.io.Person"/>
    </properties>
</batchlet>
```

Example: Artifact Class

```
@Named
public class MyBatchlet extends AbstractBatchlet {
    @Inject
    @BatchProperty
    private Class myClass;
}
```

Assigning a Default Value to a Field Annotated for Property Injection

You can assign a default value to a field in an artifact Java class in the case where the target batch property is not defined in the job XML file. If the target property is resolved to a valid value, it is injected into that field; otherwise, no value is injected and the default field value is used.

Example: Artifact Class

```
/**
 Comment character. If commentChar batch property is not specified in job XML file, use the default value '#'.
 */
@Inject
@BatchProperty
private char commentChar = '#';
```

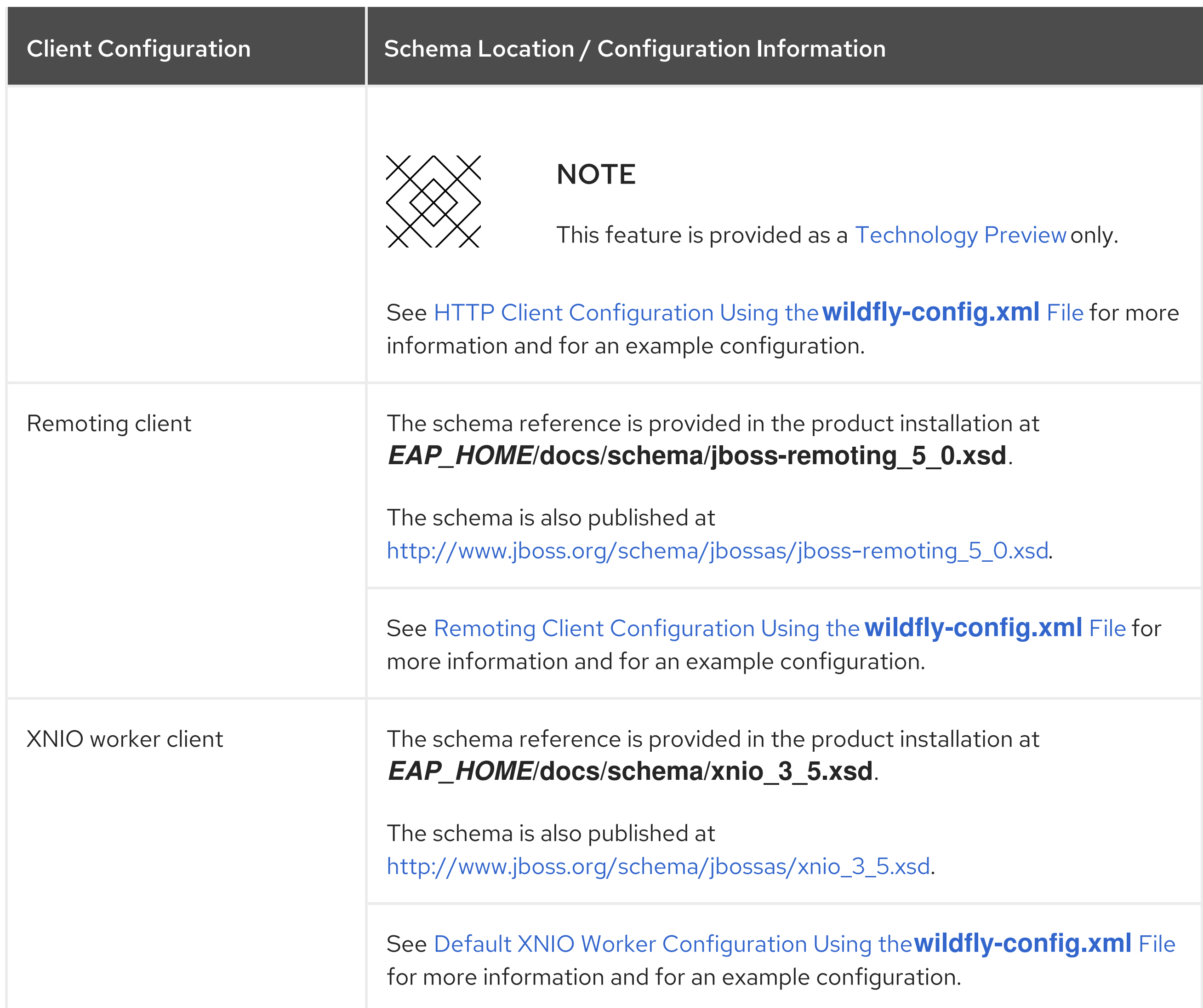
CHAPTER 19. CONFIGURING CLIENTS

19.1. CLIENT CONFIGURATION USING THE WILDFLY-CONFIG.XML FILE

Prior to release 7.1, JBoss EAP client libraries, such as EJB and naming, used different configuration strategies. JBoss EAP 7.1 introduced the **wildfly-config.xml** file with the purpose of unifying all client configurations into one single configuration file, in a similar manner to the way the server configuration is handled.

The following table describes the clients and types of configuration that can be done using the **wildfly-config.xml** file in JBoss EAP and a link to the reference schema link for each.

Client Configuration	Schema Location / Configuration Information
Authentication client	The schema reference is provided in the product installation at ***EAP_HOME*/docs/schema/elytron-client-1_2.xsd**. The schema is also published at http://www.jboss.org/schema/jbossas/elytron-client-1_2.xsd.
	See Client Authentication Configuration Using the **wildfly-config.xml** File for more information and for an example configuration. Additional information can be found in Configure Client Authentication with Elytron Client in *How to Configure Identity Management* for JBoss EAP.
EJB client	The schema reference is provided in the product installation at ***EAP_HOME*/docs/schema/wildfly-client-ejb_3_0.xsd**. The schema is also published at http://www.jboss.org/schema/jbossas/wildfly-client-ejb_3_0.xsd.
	See EJB Client Configuration Using the **wildfly-config.xml** File for more information and for an example configuration. Another simple example is located in the Migrate an EJB Client to Elytron section of the *Migration Guide* for JBoss EAP.
HTTP client	The schema reference is provided in the product installation at ***EAP_HOME*/docs/schema/wildfly-http-client_1_0.xsd**. The schema is also published at http://www.jboss.org/schema/jbossas/wildfly-http-client_1_0.xsd.



Client Configuration	Schema Location / Configuration Information
	NOTE This feature is provided as a Technology Preview only. See HTTP Client Configuration Using the **wildfly-config.xml** File for more information and for an example configuration.
Remoting client	The schema reference is provided in the product installation at ***EAP_HOME*/docs/schema/jboss-remoting_5_0.xsd**. The schema is also published at http://www.jboss.org/schema/jbossas/jboss-remoting_5_0.xsd.
	See Remoting Client Configuration Using the **wildfly-config.xml** File for more information and for an example configuration.
XNIO worker client	The schema reference is provided in the product installation at ***EAP_HOME*/docs/schema/xnio_3_5.xsd**. The schema is also published at http://www.jboss.org/schema/jbossas/xnio_3_5.xsd.
	See Default XNIO Worker Configuration Using the **wildfly-config.xml** File for more information and for an example configuration.

19.1.1. Client Authentication Configuration Using the wildfly-config.xml File

You can use the **authentication-client** element, which is in the **urn:elytron:client:1.2** namespace, to configure client authentication information using the **wildfly-config.xml** file. This section describes how to configure client authentication using this element.

authentication-client Elements and Attributes
The **authentication-client** element can optionally contain the following top level child elements, along with their child elements:

- **credential-stores**
  - **credential-store**
    - **providers**
      - **global**
      - **use-service-loader**
    - **attributes**
    - **protection-parameter-credentials**
      - **key-store-reference**

            - credential-store-reference
            - clear-password
            - key-pair
                - public-key-pem
                - private-key-pem
            - certificate
                - public-key-pem
            - bearer-token
            - oauth2-bearer-token
                - client-credentials
                - resource-owner-credentials

- key-stores
    - key-store
        - file
        - load-from
        - resource
        - key-store-clear-password
        - key-store-credential
- authentication-rules
    - rule
        - match-no-user
        - match-user
        - match-protocol
        - match-host
        - match-path
        - match-port
        - match-urn
        - match-domain-name
        - match-abstract-type
- authentication-configurations

- configuration
  - set-host-name
  - set-port-number
  - set-protocol
  - set-user-name
  - set-anonymous
  - set-mechanism-realm-name
  - rewrite-user-name-regex
  - sasl-mechanism-selector
  - set-mechanism-properties
    - property
  - credentials
    - key-store-reference
    - credential-store-reference
    - clear-password
    - key-pair
    - certificate
    - public-key-pem
    - bearer-token
    - oauth2-bearer-token
  - set-authorization-name
  - providers
    - global
    - use-service-loader
  - use-provider-sasl-factory
  - use-service-loader-sasl-factory
- net-authenticator
- ssl-context-rules
  - rule
    - match-no-user

        - match-user
        - match-protocol
        - match-host
        - match-path
        - match-port
        - match-urn
        - match-domain-name
        - match-abstract-type
- ssl-contexts
    - default-ssl-context
    - ssl-context
        - key-store-ssl-certificate
        - trust-store
        - cipher-suite
        - protocol
        - provider-name
        - certificate-revocation-list
        - providers
            - global
            - use-service-loader
- providers
    - global
    - use-service-loader

credential-stores

This optional element defines credential stores that are referenced from elsewhere in the configuration as an alternative to embedding credentials within the configuration.
It can contain any number of **credential-store** elements.

Example: credential-stores Configuration

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <credential-stores>
      <credential-store name="..." type="..." provider="..." >
        <attributes>
```

```
      <attribute name="..." value="..." />
    </attributes>
    <protection-parameter-credentials>...</protection-parameter-credentials>
   </credential-store>
  </credential-stores>
 </authentication-client>
</configuration>
```

credential-store

This element defines a credential store that is referenced from elsewhere in the configuration.
It has the following attributes.

Attribute Name	Attribute Description
name	The name of the credential store. This attribute is required.
type	The type of credential store. This attribute is optional.
provider	The name of the **java.security.Provider** to use to load the credential store. This attribute is optional.

It can contain one and only one of each of the following child elements.

- **providers**
- **attributes**
- **protection-parameter-credentials**

attributes

This element defines the configuration attributes used to initialize the credential store and can be repeated as many times as is required for the configuration.

Example: attributes Configuration

```
<attributes>
 <attribute name="..." value="..." />
</attributes>
```

protection-parameter-credentials

This element contains one or more credentials to be assembled into a protection parameter to be used when initializing the credential store.
It can contain one or more of the following child elements, which are dependent on the credential store implementation:

- **key-store-reference**
- **credential-store-reference**

- **clear-password**
- **key-pair**
- **certificate**
- **public-key-pem**
- **bearer-token**
- **oauth2-bearer-token**

Example: **protection-parameter-credentials** Configuration

```
<protection-parameter-credentials>
  <key-store-reference>...</key-store-reference>
  <credential-store-reference store="..." alias="..." clear-text="..." />
  <clear-password password="..." />
  <key-pair public-key-pem="..." private-key-pem="..." />
  <certificate private-key-pem="..." pem="..." />
  <public-key-pem>...</public-key-pem>
  <bearer-token value="..." />
  <oauth2-bearer-token token-endpoint-uri="...">...</oauth2-bearer-token>
</protection-parameter-credentials>
```

key-store-reference

This element, which is not currently used by any authentication mechanisms in JBoss EAP, defines a reference to a keystore.
It has the following attributes.

Attribute Name	Attribute Description
key-store-name	The keystore name. This attribute is required.
alias	The alias of the entry to load from the referenced keystore. This can be omitted only for keystores that contain just a single entry.

It can contain one and only one of the following child elements.

- **key-store-clear-password**
- **credential-store-reference**
- **key-store-credential**

Example: **key-store-reference** Configuration

```
<key-store-reference key-store-name="..." alias="...">
  <key-store-clear-password password="..." />
  <key-store-credential>...</key-store-credential>
</key-store-reference>
```

credential-store-reference

This element defines a reference to a credential store.
It has the following attributes.

Attribute Name	Attribute Description
store	The credential store name.
alias	The alias of the entry to load from the referenced credential store. This can be omitted only for keystores that contain just a single entry.
clear-text	The clear text password.

clear-password

This element defines a clear text password.

key-pair

This element, which is not currently used by any authentication mechanisms in JBoss EAP, defines a public and private key pair.
It can contain the following child elements.

- public-key-pem
- private-key-pem

public-key-pem

This element, which is not currently used by any authentication mechanisms in JBoss EAP, defines the PEM-encoded public key.

private-key-pem

This element defines the PEM-encoded private key.

certificate

This element, which is not currently used by any authentication mechanisms in JBoss EAP, specifies a certificate.
It has the following attributes.

Attribute Name	Attribute Description
private-key-pem	A PEM-encoded private key.
pem	The corresponding certificate.

bearer-token

This element defines a bearer token.

oauth2-bearer-token

This element defines an OAuth 2 bearer token.
It has the following attribute.

Attribute Name	Attribute Description
token-endpoint-uri	The URI of the token endpoint.

It can contain one and only one of each of the following child elements.

- **client-credentials**
- **resource-owner-credentials**

client-credentials

This element defines the client credentials.
It has the following attributes.

Attribute Name	Attribute Description
client-id	The client ID. This attribute is required.
client-secret	The client secret. This attribute is required.

resource-owner-credentials

This element defines the resource owner credentials.
It has the following attributes.

Attribute Name	Attribute Description
name	The resource name. This attribute is required.
pasword	The password. This attribute is required.

key-stores

This optional element defines keystores that are referenced from elsewhere in the configuration.

Example: key-stores Configuration

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <key-stores>
      <key-store name="...">
        <!-- The following 3 elements specify where to load the keystore from. -->
        <file name="..." />
```

```
      <load-from uri="..." />
      <resource name="..." />
      <!-- One of the following to specify the protection parameter to unlock the keystore. -->
      <key-store-clear-password password="..." />
      <key-store-credential>...</key-store-credential>
    </key-store>
  </key-stores>
  ...
</authentication-client>
</configuration>
```

key-store

This optional element defines a keystore that is referenced from elsewhere in the configuration. The **key-store** has the following attributes.

Attribute Name	Attribute Description
name	The name of the keystore. This attribute is required.
type	The keystore type, for example, **JCEKS**. This attribute is required.
provider	The name of the **java.security.Provider** to use to load the credential store. This attribute is optional.
wrap-passwords	If true, **passwords** will wrap. The passwords are stored by taking the clear password contents, encoding them in UTF-8, and storing the resultant bytes as a secret key. Defaults to **false**.

It must contain exactly one of the following elements, which define where to load the keystore from.

- **file**
- **load-from**
- **resource**

It must also contain one of the following elements, which specifies the protection parameter to use when initializing the keystore.

- **key-store-clear-password**
- **key-store-credential**

file

This element specifies the name of the keystore file.
It has the following attribute.

Attribute Name	Attribute Description
name	The fully qualified file path and name of the file.

load-from

This element specifies the URI of the keystore file.
It has the following attribute.

Attribute Name	Attribute Description
uri	The URI for the keystore file.

resource

This element specifies the name of the resource to load from the **Thread** context class loader.
It has the following attribute.

Attribute Name	Attribute Description
name	The name of the resource.

key-store-clear-password

This element specifies the clear text password.
It has the following attribute.

Attribute Name	Attribute Description
password	The clear text password.

key-store-credential

This element specifies a reference to another keystore that obtains an entry to use as the protection parameter to access this keystore.
The **key-store-credential** element has the following attributes.

Attribute Name	Attribute Description
key-store-name	The keystore name. This attribute is required.
alias	The alias of the entry to load from the referenced keystore. This can be omitted only for keystores that contain just a single entry.

It can contain one and only one of the following child elements.

- key-store-clear-password
- credential-store-reference
- key-store-credential

Example: **key-store-credential** Configuration

```
<key-store-credential key-store-name="..." alias="...">
  <key-store-clear-password password="..." />
  <key-store-credential>...</key-store-credential>
</key-store-credential>
```

authentication-rules

This element defines the rules to match against the outbound connection to apply the appropriate authentication configuration. When an **authentication-configuration** is required, the URI of the accessed resources as well as an optional abstract type and abstract type authority are matched against the rules defined in the configuration to identify which **authentication-configuration** should be used.
This element can contain one or more child **rule** elements.

Example: **authentication-rules** Configuration

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    ...
    <authentication-rules>
      <rule use-configuration="...">
        ...
      </rule>
     </authentication-rules>
     ...
  </authentication-client>
</configuration>
```

rule

This element defines the rules to match against the outbound connection to apply the appropriate authentication configuration.
It has the following attribute.

Attribute Name	Attribute Description
use-configuration	The authentication configuration that is chosen when rules match.

Authentication configuration rule matching is independent of SSL context rule matching. The authentication rule structure is identical to the SSL context rule structure, except that it references an authentication configuration, while the SSL context rule references an SSL context.

It can contain the following child elements.

- match-no-user
- match-user
- match-protocol
- match-host
- match-path
- match-port
- match-urn
- match-domain-name
- match-abstract-type

Example: rule Configuration for Authentication

```
<rule use-configuration="...">
  <!-- At most one of the following two can be defined. -->
  <match-no-user />
  <match-user name="..." />
  <!-- Each of the following can be defined at most once. -->
  <match-protocol name="..." />
  <match-host name="..." />
  <match-path name="..." />
  <match-port number="..." />
  <match-urn name="..." />
  <match-domain name="..." />
  <match-abstract-type name="..." authority="..." />
</rule>
```

match-no-user

This rule matches when there is no **user-info** embedded within the URI.

match-user

This rule matches when the **user-info** embedded in the URI matches the **name** attribute specified in this element.

match-protocol

This rule matches when the protocol within the URI matches the protocol **name** attribute specified in this element.

match-host

This rule matches when the host name specified within the URI matches the host **name** attribute specified in this element.

match-path

This rule matches when the path specified within the URI matches the path **name** attribute specified in this element.

match-port

This rule matches when the port number specified within the URI matches the port **number** attribute specified in this element. This only matches against the number specified within the URI and not against any default port number derived from the protocol.

match-urn

This rule matches when the scheme specific part of the URI matches the **name** attribute specified in this element.

match-domain-name

This rule matches when the protocol of the URI is **domain** and the scheme specific part of the URI matches the **name** attribute specified in this element.

match-abstract-type

This rule matches when the abstract type matches the **name** attribute and the authority matches the **authority** attribute specified in this element.

authentication-configurations

This element defines named authentication configurations that are to be chosen by the authentication rules.
It can contain one or more **configuration** elements.

Example: **authentication-configurations** Configuration

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <authentication-configurations>
      <configuration name="...">
        <!-- Destination Overrides. -->
        <set-host name="..." />
        <set-port number="..." />
        <set-protocol name="..." />
        <!-- At most one of the following two elements. -->
        <set-user-name name="..." />
        <set-anonymous />
        <set-mechanism-realm name="..." />
        <rewrite-user-name-regex pattern="..." replacement="..." />
        <sasl-mechanism-selector selector="..." />
        <set-mechanism-properties>
          <property key="..." value="..." />
        </set-mechanism-properties>
        <credentials>...</credentials>
        <set-authorization-name name="..." />
        <providers>...</providers>
        <!-- At most one of the following two elements. -->
        <use-provider-sasl-factory />
        <use-service-loader-sasl-factory module-name="..." />
      </configuration>
    </authentication-configurations>
  </authentication-client>
</configuration>
```

configuration

This element defines named authentication configurations that are to be chosen by the authentication rules.
It can contain the following child elements.

- The optional **set-host-name**, **set-port-number**, and **set-protocol** elements can override the destination.
- The optional **set-user-name** and **set-anonymous** elements are mutually exclusive and can be used to set the name for authentication or switch to anonymous authentication.
- Next are the **set-mechanism-realm-name**, **rewrite-user-name-regex**, **sasl-mechanism-selector**, **set-mechanism-properties**, **credentials**, **set-authorization-name**, and **providers** optional elements.
- The final two optional **use-provider-sasl-factory** and **use-service-loader-sasl-factory** elements are mutually exclusive and define how the SASL mechanism factories are discovered for authentication.

set-host-name

This element overrides the host name for the authenticated call.
It has the following attribute.

Attribute Name	Attribute Description
name	The host name.

set-port-number

This element overrides the port number for the authenticated call.
It has the following attribute.

Attribute Name	Attribute Description
number	The port number.

set-protocol

This element overrides the protocol for the authenticated call.
It has the following attribute.

Attribute Name	Attribute Description
name	The protocol.

set-user-name

This element sets the user name to use for the authentication. It should not be used with the **set-anonymous** element.

It has the following attribute.

Attribute Name	Attribute Description
name	The user name to use for authentication.

set-anonymous

The element is used to switch to anonymous authentication. It should not be used with the **set-user-name** element.

set-mechanism-realm-name

This element specifies the name of the realm that will be selected by the SASL mechanism if required.
It has the following attribute.

Attribute Name	Attribute Description
name	The name of the realm.

rewrite-user-name-regex

This element defines a regular expression pattern and replacement to rewrite the user name used for authentication.
It has the following attributes.

Attribute Name	Attribute Description
pattern	A regular expression pattern.
replacement	The replacement to use to rewrite the user name used for authentication.

sasl-mechanism-selector

This element specifies a SASL mechanism selector using the syntax from the **org.wildfly.security.sasl.SaslMechanismSelector.fromString(string)** method.
It has the following attribute.

Attribute Name	Attribute Description
selector	The SASL mechanism selector.

For more information about the grammar required for the **sasl-mechanism-selector**, see **sasl-mechanism-selector** Grammar in *How to Configure Server Security* for JBoss EAP.

set-mechanism-properties

This element can contain one or more **property** elements that are to be passed to the authentication mechanisms.

property

This element defines a property to be passed to the authentication mechanisms.
It has the following attributes.

Attribute Name	Attribute Description
key	The property name.
value	The property value.

credentials

This element defines one or more credentials available for use during authentication.
It can contain one or more of the following child elements, which are dependent on the credential store implementation:

- **key-store-reference**
- **credential-store-reference**
- **clear-password**
- **key-pair**
- **certificate**
- **public-key-pem**
- **bearer-token**
- **oauth2-bearer-token**.

These are the same child elements as those contained in the **protection-parameter-credentials** element. See the **protection-parameter-credentials** element for details and an example configuration.

set-authorization-name

This element specifies the name that should be used for authorization if it is different from the authentication identity.
It has the following attributes.

Attribute Name	Attribute Description
name	The name that should be used for authorization.

use-provider-sasl-factory

This element specifies the **java.security.Provider** instances that are either inherited or defined in this configuration and that are to be used to locate the available SASL client factories. This element should not be used with the **use-service-loader-sasl-factory** element.

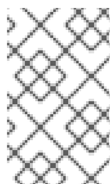
use-service-loader-sasl-factory

This element specifies the module that is to be used to discover the SASL client factories using the service loader discovery mechanism. If no module is specified, the class loader that loaded the configuration is used. This element should not be used with the **use-provider-sasl-factory** element. It has the following attribute.

Attribute Name	Attribute Description
module-name	The name of the module.

net-authenticator

This element contains no configuration. If present, the **org.wildfly.security.auth.util.ElytronAuthenticator** is registered with **java.net.Authenticator.setDefault(Authenticator)**. This allows the Elytron authentication client configuration to be used for authentication when JDK APIs are used for HTTP calls that require authentication.



NOTE

Because the JDK caches the authentication on the first call across the JVM, it is better to use this approach only on standalone processes that do not require different credentials for different calls to the same URI.

ssl-context-rules

This optional element defines the SSL context rules. When an **ssl-context** is required, the URI of the accessed resources as well as an optional abstract type and abstract type authority are matched against the rules defined in the configuration to identify which **ssl-context** should be used.
This element can contain one or more child **rule** elements.

Example: ssl-context-rules Configuration

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <ssl-context-rules>
      <rule use-ssl-context="...">
        ...
      </rule>
    </ssl-context-rules>
    ...
  </authentication-client>
</configuration>
```

rule

This element defines the rule to match on the SSL context definitions.

It has the following attribute.

Attribute Name	Attribute Description
use-ssl-context	The SSL context definition that is chosen when rules match.

SSL context rule matching is independent of authentication rule matching. The SSL context rule structure is identical to the authentication configuration rule structure, except that it references an SSL context, while the authentication rule references an authentication configuration.

It can contain the following child elements.

- match-no-user
- match-user
- match-protocol
- match-host
- match-path
- match-port
- match-urn
- match-domain-name
- match-abstract-type

Example: rule Configuration for SSL Context

```
<rule use-ssl-context="...">
  <!-- At most one of the following two can be defined. -->
  <match-no-user />
  <match-user name="..." />
  <!-- Each of the following can be defined at most once. -->
  <match-protocol name="..." />
  <match-host name="..." />
  <match-path name="..." />
  <match-port number="..." />
  <match-urn name="..." />
  <match-domain name="..." />
  <match-abstract-type name="..." authority="..." />
</rule>
```

ssl-contexts

This optional element defines SSL context definitions that are to be chosen by the SSL context rules.

Example: ssl-contexts Configuration

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
```

```
    <ssl-contexts>
      <default-ssl-context name="..."/>
      <ssl-context name="...">
        <key-store-ssl-certificate>...</key-store-ssl-certificate>
        <trust-store key-store-name="..." />
        <cipher-suite selector="..." />
        <protocol names="... ..." />
        <provider-name name="..." />
        <providers>...</providers>
        <certificate-revocation-list path="..." maximum-cert-path="..." />
      </ssl-context>
    </ssl-contexts>
  </authentication-client>
</configuration>
```

default-ssl-context

This element takes the **SSLContext** returned by **javax.net.ssl.SSLContext.getDefault()** and assigns it a name so it can referenced from the **ssl-context-rules**. This element can be repeated, meaning the default SSL context can be referenced using different names.

ssl-context

This element defines an SSL context to use for connections.
It can optionally contain one of each of the following child elements.

- **key-store-ssl-certificate**
- **trust-store**
- **cipher-suite**
- **protocol**
- **provider-name**
- **providers**
- **certificate-revocation-list**

key-store-ssl-certificate

This element defines a reference to an entry within a keystore for the key and certificate to use for this SSL context.
It has the following attributes.

Attribute Name	Attribute Description
key-store-name	The keystore name. This attribute is required.
alias	The alias of the entry to load from the referenced keystore. This can be omitted only for keystores that contain just a single entry.

It can contain the following child elements:

- **key-store-clear-password**
- **credential-store-reference**
- **key-store-credential**

This structure is nearly identical to the structure used in the **key-store-credential** configuration with the exception that here it obtains the entry for the key and for the certificate. However, the nested **key-store-clear-password** and **key-store-credential** elements still provide the protection parameter to unlock the entry.

Example: key-store-ssl-certificate Configuration

```
<key-store-ssl-certificate key-store-name="..." alias="...">
  <key-store-clear-password password="..." />
  <key-store-credential>...</key-store-credential>
</key-store-ssl-certificate>
```

trust-store

This element is a reference to the keystore that is to be used to initialize the **TrustManager**.
It has the following attribute.

Attribute Name	Attribute Description
key-store-name	The keystore name. This attribute is required.

cipher-suite

This element configures the filter for the enabled cipher suites.
It has the following attribute.

Attribute Name	Attribute Description
selector	The selector to filter the cipher suites. The selector uses the format of the OpenSSL-style cipher list string created by the **org.wildfly.security.ssl.CipherSuiteSelector.fromString(selector)** method.

Example: cipher-suite Configuration Using Default Filtering

```
<cipher-suite selector="DEFAULT" />
```

protocol

This element defines a space separated list of the protocols to be supported. See the client-ssl-context Attributes table in *How to Configure Server Security* for JBoss EAP for the list of available protocols. Red Hat recommends that you use **TLSv1.2**.

provider-name

Once the available providers have been identified, only the provider with the name defined on this element is used.

certificate-revocation-list

This element defines both the path to the certificate revocation list and the maximum number of non-self-issued intermediate certificates that can exist in a certification path. The presence of this element enables checking the peer's certificate against the certificate revocation list.
It has the following attributes.

Attribute Name	Attribute Description
path	The path to the certification list. This attribute is optional.
maximum-cert-path	The maximum number of non-self-issued intermediate certificates that can exist in a certification path. This attribute is optional.

providers

This element defines how **java.security.Provider** instances are located when required.
It can contain the following child elements.

- **global**
- **use-service-loader**

Because the configuration sections of **authentication-client** are independent of each other, this element can be configured in the following locations.

Example: Locations of providers Configuration

```
<configuration>
 <authentication-client xmlns="urn:elytron:client:1.2">
  <providers />
  ...
  <credential-stores>
   <credential-store name="...">
    ...
    <providers />
   </credential-store>
  </credential-stores>
  ...
  <authentication-configurations>
   <authentication-configuration name="...">
    ...
    <providers />
   </authentication-configuration>
  </authentication-configurations>
  ...
  <ssl-contexts>
   <ssl-context name="...">
    ...
    <providers />
   </ssl-context>
```

```
    </ssl-contexts>
  </authentication-client>
</configuration>
```

The **providers** configuration applies to the element in which it is defined and to any of its child elements unless it is overridden. The specification of a **providers** in a child element overrides a **providers** specified in any of its parent elements. If no **providers** configuration is specified, the default behavior is the equivalent of the following, which gives the Elytron provider priority over any globally registered providers, but also allows for the use of globally registered providers.

Example: providers Configuration

```
<providers>
  <use-service-loader />
  <global />
</providers>
```

global

This empty element specifies to use the global providers loaded by the **java.security.Security.getProviders()** method call.

use-service-loader

This empty element specifies to use the providers that are loaded by the specified module. If no module is specified, the class loader that loaded the authentication client is used.

IMPORTANT

Elements Not Currently Used By Any JBoss EAP Authentication Mechanisms

The following child elements of the **credentials** element in the Elytron client configuration are not currently used by any authentication mechanisms in JBoss EAP. They can be used in your own custom implementations of authentication mechanism; however, they are not supported.

1. **key-pair**
2. **public-key-pem**
3. **key-store-reference**
4. **certificate**

19.1.2. EJB Client Configuration Using the wildfly-config.xml File

You can use the **jboss-ejb-client** element, which is in the **urn:jboss:wildfly-client-ejb:3.0** namespace, to configure EJB client connections, global interceptors, and invocation timeouts using the **wildfly-config.xml** file. This section describes how to configure an EJB client using this element.

jboss-ejb-client Elements and Attributes

The **jboss-ejb-client** element can optionally contain the following three top level child elements, along with their child elements:

- **invocation-timeout**

- **global-interceptors**
  - **interceptor**
- **connections**
  - **connection**
    - **interceptors**
      - **interceptor**

invocation-timeout

This optional element specifies the EJB invocation timeout. It has the following attribute.

Attribute Name	Attribute Description
seconds	The timeout, in seconds, for the EJB handshake or the method invocation request/response cycle. This attribute is required. If the execution of a method takes longer than the timeout period, the invocation throws a **java.util.concurrent.TimeoutException**; however, the server side will not be interrupted.

global-interceptors

This optional element specifies the global EJB client interceptors. It can contain any number of **interceptor** elements.

interceptor

This element is used to specify an EJB client interceptor. It has the following attributes.

Attribute Name	Attribute Description
class	The name of a class that implements the **org.jboss.ejb.client.EJBClientInterceptor** interface. This attribute is required.
module	The name of the module that should be used to load the interceptor class. This attribute is optional.

connections

This element is used to specify EJB client connections. It can contain any number of **connection** elements.

connection

This element is used to specify an EJB client connection. It can optionally contain an **interceptors** element. It has the following attribute.

Attribute Name	Attribute Description
uri	The destination URI for the connection. This attribute is required.

Attribute Name	Attribute Description

interceptors

This element is used to specify EJB client interceptors and can contain any number of **interceptor** elements.

Example EJB Client Configuration in the wildfly-config.xml File

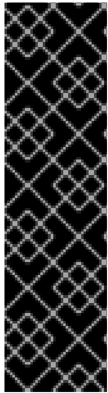
The following is an example that configures the EJB client connections, global interceptors, and invocation timeout using the **jboss-ejb-client** element in the **wildfly-config.xml** file.

```
<configuration>
...
    <jboss-ejb-client xmlns="urn:jboss:wildfly-client-ejb:3.0">
        <invocation-timeout seconds="10"/>
        <connections>
            <connection uri="remote+http://10.20.30.40:8080"/>
        </connections>
        <global-interceptors>
            <interceptor class="org.jboss.example.ExampleInterceptor"/>
        </global-interceptors>
    </jboss-ejb-client>
...
</configuration>
```

19.1.3. HTTP Client Configuration Using the wildfly-config.xml File

The following is an example of how to configure HTTP clients using the **wildfly-config.xml** file.

```
<configuration>
...
    <http-client xmlns="urn:wildfly-http-client:1.0">
        <defaults>
            <eagerly-acquire-session value="true" />
            <buffer-pool buffer-size="2000" max-size="10" direct="true" thread-local-size="1" />
        </defaults>
    </http-client>
...
</configuration>
```



IMPORTANT

HTTP client configuration using the **wildfly-config.xml** file is provided as Technology Preview only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs), might not be functionally complete, and Red Hat does not recommend to use them for production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

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

19.1.4. Remoting Client Configuration Using the wildfly-config.xml File

You can use the **endpoint** element, which is in the **urn:jboss-remoting:5.0** namespace, to configure a remoting client using the **wildfly-config.xml** file. This section describes how to configure a remoting client using this element.

endpoint Elements and Attributes

The **endpoint** element can optionally contain the following two top level child elements, along with their child elements.

- **providers**
  - **provider**
- **connections**
  - **connection**

It also has the following attribute:

Attribute Name	Attribute Description
name	The endpoint name. This attribute is optional. If not provided, an endpoint name is derived from the system's host name, if possible.

providers

This optional element specifies transport providers for the remote endpoint. It can contain any number of **provider** elements.

provider

This element defines a remote transport provider. It has the following attributes.

Attribute Name	Attribute Description
scheme	The primary URI scheme that corresponds to this provider. This attribute is required.
aliases	The space-separated list of of other URI scheme names that are also recognized for this provider. This attribute is optional.

Attribute Name	Attribute Description
module	The name of the module that contains the provider implementation. This attribute is optional. If not provided, the class loader that loads JBoss Remoting searches for the provider class.
class	The name of the class that implements the transport provider. This attribute is optional. If not provided, the **java.util.ServiceLoader** facility is used to search for the provider class.

connections

This optional element specifies connections for the remote endpoint. It can contain any number of **connection** elements.

connection

This element defines a connection for the remote endpoint. It has the following attributes.

Attribute Name	Attribute Description
destination	The destination URI for the endpoint. This attribute is required.
read-timeout	The timeout, in seconds, for read operations on the corresponding socket. This attribute is optional; however, it should be provided only if a **heartbeat-interval** is defined.
write-timeout	The timeout, in seconds, for a write operation. This attribute is optional; however, it should be provided only if a **heartbeat-interval** is defined.
ip-traffic-class	Defines the numeric IP traffic class to use for this connection’s traffic. This attribute is optional.
tcp-keepalive	Boolean setting that determines whether to use TCP keepalive. This attribute is optional.
heartbeat-interval	The interval, in milliseconds, to use when checking for a connection heartbeat. This attribute is optional.

Example Remoting Client Configuration in the**wildfly-config.xml** File

The following is an example that configures a remoting client using the **wildfly-config.xml** file.

```
<configuration>
  ...
  <endpoint xmlns="urn:jboss-remoting:5.0">
    <connections>
      <connection destination="remote+http://10.20.30.40:8080" read-timeout="50" write-timeout="50"
heartbeat-interval="10000"/>
    </connections>
```

```
  </endpoint>
  ...
</configuration>
```

19.1.5. Default XNIO Worker Configuration Using the wildfly-config.xml File

You can use the **worker** element, which is in the **urn:xnio:3.5** namespace, to configure an XNIO worker using the **wildfly-config.xml** file. This section describes how to configure an XNIO worker client using this element.

worker Elements and Attributes
The **worker** element can optionally contain the following top level child elements, along with their child elements:

- daemon-threads
- worker-name
- pool-size
- task-keepalive
- io-threads
- stack-size
- outbound-bind-addresses
  - bind-address

daemon-threads

This optional element specifies whether worker and task threads should be daemon threads. This element has no content. It has the following attribute.

Attribute Name	Attribute Description
value	A boolean value that specifies whether worker and task threads should be daemon threads. A value of **true** indicates that worker and task threads should be daemon threads. A value of **false** indicates that they should not be daemon threads. This attribute is required. If this element is not provided, a value of **true** is assumed.

worker-name

This element defines the name of the worker. The worker name appears in thread dumps and in JMX. This element has no content. It has the following attribute.

Attribute Name	Attribute Description
value	The name of the worker. This attribute is required.

pool-size

This optional element defines the maximum size of the worker's task thread pool. This element has no content. It has the following attribute.

Attribute Name	Attribute Description
max-threads	A positive integer that specifies the maximum number of threads that should be created. This attribute is required.

task-keepalive

This optional element establishes the keep-alive time of task threads before they can be expired. It has the following attribute.

Attribute Name	Attribute Description
value	A positive integer that specifies the minimum number of seconds to keep idle threads alive. This attribute is required.

io-threads

This optional element determines how many I/O selector threads should be maintained. Generally this number should be a small constant that is a multiple of the number of available cores. It has the following attribute.

Attribute Name	Attribute Description
value	A positive integer that specifies the number of I/O threads. This attribute is required.

stack-size

This optional element establishes the desired minimum thread stack size for worker threads. This element should only be defined in very specialized situations where density is at a premium. It has the following attribute.

Attribute Name	Attribute Description
value	A positive integer that specifies the requested stack size, in bytes. This attribute is required.

outbound-bind-addresses

This optional element specifies the bind addresses to use for outbound connections. Each bind address mapping consists of a destination IP address block, and a bind address and optional port number to use for connections to destinations within that block. It can contain any number of **bind-address** elements.

bind-address

This optional element defines an individual bind address mapping. It has the following attributes.

Attribute Name	Attribute Description
match	The IP address block, in CIDR notation, to match.
bind-address	The IP address to bind to if the address block matches. This attribute is required.
bind-port	The port number to bind to if the address block matches. This value defauts to **0**, meaning it binds to any port. This attribute is optional.

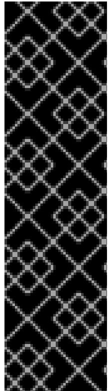
Example XNIO Worker Configuration in thewildfly-config.xml File

The following is an example of how to configure the default XNIO worker using the **wildfly-config.xml** file.

```
<configuration>
  ...
  <worker xmlns="urn:xnio:3.5">
    <io-threads value="10"/>
    <task-keepalive value="100"/>
  </worker>
  ...
</configuration>
```

CHAPTER 20. ECLIPSE MICROPROFILE

20.1. USING ECLIPSE MICROPROFILE OPENTRACING TO TRACE REQUESTS



IMPORTANT

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

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

Eclipse Microprofile OpenTracing functionality is provided by the **microprofile-opentracing-smallrye** subsystem. This subsystem ships with the Jaeger Client as the default tracer and provides a set of instrumentation libraries for components commonly used in Jakarta EE applications, such as Jakarta RESTful Web Services and Contexts and Dependency Injection. See Tracing Requests with the MicroProfile OpenTracing SmallRye Subsystem in the *Configuration Guide* for more information about this subsystem.

The following sections describe how to customize tracing for Jakarta Contexts and Dependency Injection beans and other Jakarta RESTful Web Services endpoints, and how to implement a custom tracer.

20.1.1. Enable or Disable Tracing for Jakarta Contexts and Dependency Injection Beans

As defined by the Eclipse MicroProfile OpenTracing specification, Jakarta Contexts and Dependency Injection beans are traced if the **org.eclipse.microprofile.opentracing.Traced** annotation is present, either at the class or at the method level. Tracing can be disabled by setting the annotation value to **false**. Similarly, a custom operation name can be set by specifying the parameter **operationName** for that annotation. The semantics are defined by the MicroProfile OpenTracing specification.

The following example demonstrates how to configure tracing for a Jakarta Contexts and Dependency Injection bean. Note that tracing can be specified at the method level.

```
import org.eclipse.microprofile.opentracing.Traced;

@Traced
public class TracedBean {
   public void doSomething() {
   }

   @Traced(true)
   public void doSomethingTraced() {
   }

   @Traced(false)
```

```
    public void doSomethingNotTraced() {
    }
}
```

The following example demonstrates how to specify an operation name for the OpenTracing Span for this trace point.

```
import org.eclipse.microprofile.opentracing.Traced;

@Traced(operationName = "my-custom-class-operation-name")
public class CustomOperationNameBean {

    @Traced(operationName = "my-custom-method-operation-name")
    public void doSomething() {
    }

    @Traced
    public void doSomethingElse() {
    }
}
```

20.1.2. Enable or Disable Tracing for Jakarta RESTful Web Services Endpoints

Jakarta RESTful Web Services endpoints are traced by default if the **microprofile-opentracing-smallrye** subsystem is present in the server configuration.

To disable tracing for Jakarta RESTful Web Services endpoints, add the **@Traced(false)** annotation to the Jakarta RESTful Web Services endpoint at the class or method level as described in Enable or Disable Tracing for Jakarta Contexts and Dependency Injection Beans above.

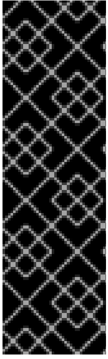
20.1.3. Implement a Custom Tracer

If you need something more complex than what is provided by the default Jaeger Client, you can provide your own tracer by implementing a **TracerResolver** that returns the **Tracer** with the desired state. In this case, the default tracer will *not* be used.

The following example demonstrates how to create a new implementation of **TracerResolver** that returns a custom implementation of the **Tracer** class.

```
import io.opentracing.Tracer;
import io.opentracing.contrib.tracerresolver.TracerResolver;
import org.myproject.opentracing.CustomTracer;

public static class MyTracerResolver extends TracerResolver {
    @Override
    protected Tracer resolve() {
        return new CustomTracer();
    }
}
```

20.2. USING ECLIPSE MICROPROFILE HEALTH TO MONITOR SERVER HEALTH



IMPORTANT

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

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

Eclipse Microprofile Health functionality is provided by the **microprofile-health-smallrye** subsystem. See Monitor Server Health Using Eclipse MicroProfile Health in the *Configuration Guide* for more information about this subsystem.

The following section describe how to implement a custom health check.

20.2.1. Implement a Custom Health Check

The default implementation provided by the **microprofile-health-smallrye** subsystem performs a basic health check. For more detailed information, on either the server or application status, custom health checks may be included. Any Jakarta Contexts and Dependency Injection beans that include the **org.eclipse.microprofile.health.Health** annotation at the class level are automatically discovered and invoked at runtime.

The following example demonstrates how to create a new implementation of a health check that returns an **UP** state.

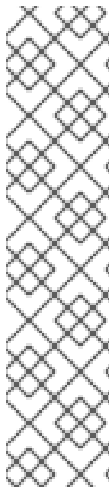
```
import org.eclipse.microprofile.health.Health;
import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;

@Health
public class HealthTest implements HealthCheck {

    @Override
    public HealthCheckResponse call() {
        return HealthCheckResponse.named("health-test").up().build();
    }
}
```

Once deployed, any subsequent health check queries will include the custom checks, as seen below.

```
/subsystem=microprofile-health-smallrye:check
{
    "outcome" => "success",
    "result" => {
        "outcome" => "UP",
        "checks" => [{
            "name" => "health-test",
            "state" => "UP"
        }]
    }
}
```

APPENDIX A. REFERENCE MATERIAL

A.1. PROVIDED UNDERTOW HANDLERS



NOTE

For the complete list of handlers, you must check the source JAR file of the Undertow core in the version that matches the Undertow core in your JBoss EAP installation. You can download the Undertow core source JAR file from the JBoss EAP Maven Repository, and then refer to the available handlers in the **/io/undertow/server/handlers/** directory.

You can verify the Undertow core version used in your current installation of JBoss EAP by searching the **server.log** file for the **INFO** message that is printed during JBoss EAP server startup, similar to the one shown in the example below:

```
INFO [org.wildfly.extension.undertow] (MSC service thread 1-1) WFLYUT0003:
Undertow 1.4.18.Final-redhat-1 starting
```

AccessControlListHandler
Class Name: **io.undertow.server.handlers.AccessControlListHandler**

Name: **access-control**

Handler that can accept or reject a request based on an attribute of the remote peer.

Table A.1. Parameters

Name	Description
acl	ACL rules. This parameter is required.
attribute	Exchange attribute string. This parameter is required.
default-allow	Boolean specifying whether handler accepts or rejects a request by default. Defaults to **false**.

AccessLogHandler
Class Name: **io.undertow.server.handlers.accesslog.AccessLogHandler**

Name: **access-log**

Access log handler. This handler generates access log messages based on the provided format string and pass these messages into the provided **AccessLogReceiver**.

This handler can log any attribute that is provided via the **ExchangeAttribute** mechanism.

This factory produces token handlers for the following patterns.

Table A.2. Patterns

Pattern	Description
%a	Remote IP address
%A	Local IP address
%b	Bytes sent, excluding HTTP headers or **-** if no bytes were sent
%B	Bytes sent, excluding HTTP headers
%h	Remote host name
%H	Request protocol
%l	Remote logical username from **identd** (always returns **-**)
%m	Request method
%p	Local port
%q	Query string (excluding the **?** character)
%r	First line of the request
%s	HTTP status code of the response
%t	Date and time, in Common Log Format format
%u	Remote user that was authenticated
%U	Requested URL path
%v	Local server name
%D	Time taken to process the request, in milliseconds
%T	Time taken to process the request, in seconds
%I	Current Request thread name (can compare later with stack traces)
common	**%h %l %u %t "%r" %s %b**
combined	**%h %l %u %t "%r" %s %b "%{i,Referer}" "%{i,User-Agent}"**

There is also support to write information from the cookie, incoming header, or the session.

It is modeled after the Apache syntax:

- **%{i,xxx}** for incoming headers
- **%{o,xxx}** for outgoing response headers
- **%{c,xxx}** for a specific cookie
- **%{r,xxx}** where **xxx** is an attribute in the **ServletRequest**
- **%{s,xxx}** where **xxx** is an attribute in the **HttpSession**

Table A.3. Parameters

Name	Description
format	Format used to generate the log messages. This is the default parameter.

AllowedMethodsHandler
Handler that whitelists certain HTTP methods. Only requests with a method in the allowed methods set are allowed to continue.

Class Name: **io.undertow.server.handlers.AllowedMethodsHandler**

Name: **allowed-methods**

Table A.4. Parameters

Name	Description
methods	Methods to allow, for example **GET**, **POST**, **PUT**, and so on. This is the default parameter.

BlockingHandler
An HttpHandler that initiates a blocking request. If the thread is currently running in the I/O thread it is dispatched.

Class Name: **io.undertow.server.handlers.BlockingHandler**

Name: **blocking**

This handler has no parameters.

ByteRangeHandler
Handler for range requests. This is a generic handler that can handle range requests to any resource of a fixed content length, for example, any resource where the **content-length** header has been set. This is not necessarily the most efficient way to handle range requests, as the full content is generated and then discarded. At present this handler can only handle simple, single range requests. If multiple ranges are requested the **Range** header is ignored.

Class Name: **io.undertow.server.handlers.ByteRangeHandler**

Name: **byte-range**

Table A.5. Parameters

Name	Description
send-accept-ranges	Boolean value on whether or not to send accept ranges. This is the default parameter.

CanonicalPathHandler
This handler transforms a relative path to a canonical path.

Class Name: **io.undertow.server.handlers.CanonicalPathHandler**

Name: **canonical-path**

This handler has no parameters.

DisableCacheHandler
Handler that disables response caching by browsers and proxies.

Class Name: **io.undertow.server.handlers.DisableCacheHandler**

Name: **disable-cache**

This handler has no parameters.

DisallowedMethodsHandler
Handler that blacklists certain HTTP methods.

Class Name: **io.undertow.server.handlers.DisallowedMethodsHandler**

Name: **disallowed-methods**

Table A.6. Parameters

Name	Description
methods	Methods to disallow, for example **GET**, **POST**, **PUT**, and so on. This is the default parameter.

EncodingHandler
This handler serves as the basis for content encoding implementations. Encoding handlers are added as delegates to this handler, with a specified server side priority.

The **q** value will be used to determine the correct handler. If a request comes in with no **q** value then the server picks the handler with the highest priority as the encoding to use.

If no handler matches then the identity encoding is assumed. If the identity encoding has been specifically disallowed due to a **q** value of **0** then the handler sets the response code **406 (Not Acceptable)** and returns.

Class Name: **io.undertow.server.handlers.encoding.EncodingHandler**

Name: **compress**

This handler has no parameters.

FileErrorPageHandler
Handler that serves up a file from disk to serve as an error page. This handler does not serve up any response codes by default, you must configure the response codes it responds to.

Class Name: **io.undertow.server.handlers.error.FileErrorPageHandler**

Name: **error-file**

Table A.7. Parameters

Name	Description
file	Location of file to serve up as an error page.
response-codes	List of response codes that result in a redirect to the defined error page file.

HttpTraceHandler
A handler that handles HTTP trace requests.

Class Name: **io.undertow.server.handlers.HttpTraceHandler**

Name: **trace**

This handler has no parameters.

IPAddressAccessControlHandler
Handler that can accept or reject a request based on the IP address of the remote peer.

Class Name: **io.undertow.server.handlers.IPAddressAccessControlHandler**

Name: **ip-access-control**

Table A.8. Parameters

Name	Description
acl	String representing the access control list. This is the default parameter.
failure-status	Integer representing the status code to return on rejected requests.
default-allow	Boolean representing whether or not to allow by default.

JDBCLogHandler
Class Name: **io.undertow.server.handlers.JDBCLogHandler**

Name: **jdbc-access-log**

Table A.9. Parameters

Name	Description
format	Specifies the JDBC Log pattern. Default value is **common**. You can also use **combined**, which adds the VirtualHost, request method, referrer, and user agent information to the log message.
datasource	Name of the datasource to log. This parameter is required and is the default parameter.
tableName	Table name.
remoteHostField	Remote Host address.
userField	Username.
timestampField	Timestamp.
virtualHostField	VirtualHost.
methodField	Method.
queryField	Query.
statusField	Status.
bytesField	Bytes.
refererField	Referrer.
userAgentField	UserAgent.

LearningPushHandler
Handler that builds up a cache of resources that a browser requests, and uses server push to push them when supported.

Class Name: **io.undertow.server.handlers.LearningPushHandler**

Name: **learning-push**

Table A.10. Parameters

Name	Description

Name	Description
max-age	Integer representing the maximum time of a cache entry.
max-entries	Integer representing the maximum number of cache entries

LocalNameResolvingHandler
A handler that performs DNS lookup to resolve a local address. Unresolved local address can be created when a front end server has sent a **X-forwarded-host** header or AJP is in use.

Class Name: **io.undertow.server.handlers.LocalNameResolvingHandler**

Name: **resolve-local-name**

This handler has no parameters.

PathSeparatorHandler
A handler that translates non-slash separator characters in the URL into a slash. In general this will translate backslash into slash on Windows systems.

Class Name: **io.undertow.server.handlers.PathSeparatorHandler**

Name: **path-separator**

This handler has no parameters.

PeerNameResolvingHandler
A handler that performs reverse DNS lookup to resolve a peer address.

Class Name: **io.undertow.server.handlers.PeerNameResolvingHandler**

Name: **resolve-peer-name**

This handler has no parameters.

ProxyPeerAddressHandler
Handler that sets the peer address to the value of the **X-Forwarded-For** header. This should only be used behind a proxy that always sets this header, otherwise it is possible for an attacker to forge their peer address.

Class Name: **io.undertow.server.handlers.ProxyPeerAddressHandler**

Name: **proxy-peer-address**

This handler has no parameters.

RedirectHandler
A redirect handler that redirects to the specified location via a **302** redirect. The location is specified as an exchange attribute string.

Class Name: **io.undertow.server.handlers.RedirectHandler**

Name: **redirect**

Table A.11. Parameters

Name	Description
value	Destination for the redirect. This is the default parameter.

RequestBufferingHandler
Handler that buffers all request data.

Class Name: **io.undertow.server.handlers.RequestBufferingHandler**

Name: **buffer-request**

Table A.12. Parameters

Name	Description
buffers	Integer that defines the maximum number of buffers. This is the default parameter.

RequestDumpingHandler
Handler that dumps an exchange to a log.

Class Name: **io.undertow.server.handlers.RequestDumpingHandler**

Name: **dump-request**

This handler has no parameters.

RequestLimitingHandler
A handler that limits the maximum number of concurrent requests. Requests beyond the limit will block until the previous request is complete.

Class Name: **io.undertow.server.handlers.RequestLimitingHandler**

Name: **request-limit**

Table A.13. Parameters

Name	Description
requests	Integer that represents the maximum number of concurrent requests. This is the default parameter and is required.

ResourceHandler
A handler for serving resources.

Class Name: **io.undertow.server.handlers.resource.ResourceHandler**

Name: **resource**

Table A.14. Parameters

Name	Description
location	Location of resources. This is the default parameter and is required.
allow-listing	Boolean value to determine whether or not to allow directory listings.

ResponseRateLimitingHandler

Handler that limits the download rate to a set number of bytes/time.

Class Name: **io.undertow.server.handlers.ResponseRateLimitingHandler**

Name: **response-rate-limit**

Table A.15. Parameters

Name	Description
bytes	Number of bytes to limit the download rate. This parameter is required.
time	Time in seconds to limit the download rate. This parameter is required.

SetHeaderHandler

A handler that sets a fixed response header.

Class Name: **io.undertow.server.handlers.SetHeaderHandler**

Name: **header**

Table A.16. Parameters

Name	Description
header	Name of header attribute. This parameter is required.
value	Value of header attribute. This parameter is required.

SSLHeaderHandler

Handler that sets SSL information on the connection based on the following headers:

- SSL_CLIENT_CERT
- SSL_CIPHER
- SSL_SESSION_ID

If this handler is present in the chain it always overrides the SSL session information, even if these headers are not present.

This handler *must* only be used on servers that are behind a reverse proxy, where the reverse proxy has been configured to always set these headers for *every* request or to strip existing headers with these names if no SSL information is present. Otherwise it might be possible for a malicious client to spoof an SSL connection.

Class Name: **io.undertow.server.handlers.SSLHeaderHandler**

Name: **ssl-headers**

This handler has no parameters.

StuckThreadDetectionHandler

This handler detects requests that take a long time to process, which might indicate that the thread that is processing it is stuck.

Class Name: **io.undertow.server.handlers.StuckThreadDetectionHandler**

Name: **stuck-thread-detector**

Table A.17. Parameters

Name	Description
threshhold	Integer value in seconds that determines the threshold for how long a request should take to process. Default value is **600** (10 minutes). This is the default parameter.

URLDecodingHandler

A handler that decodes the URL and query parameters to the specified charset. If you are using this handler you must set the UndertowOptions.DECODE_URL parameter to **false**.

This is not as efficient as using the parser's built in UTF-8 decoder. Unless you need to decode to something other than UTF-8 you should rely on the parsers decoding instead.

Class Name: **io.undertow.server.handlers.URLDecodingHandler**

Name: **url-decoding**

Table A.18. Parameters

Name	Description
charset	Charset to decode. This is the default parameter and it is required.

A.2. PERSISTENCE UNIT PROPERTIES

Persistence unit definition supports the following properties, which can be configured from the **persistence.xml** file.

Property	Description
jboss.as.jpa.providerModule	Name of the persistence provider module. Default is **org.hibernate**. Should be the application name if a persistence provider is packaged with the application.
jboss.as.jpa.adapterModule	Name of the integration classes that help JBoss EAP to work with the persistence provider.
jboss.as.jpa.adapterClass	Class name of the integration adapter.
jboss.as.jpa.managed	Set to **false** to disable container-managed Jakarta Persistence access to the persistence unit. The default is **true**.
jboss.as.jpa.classtransformer	Set to **false** to disable class transformers for the persistence unit. The default is **true**, which allows class transforming. Hibernate also needs persistence unit property **hibernate.ejb.use_class_enhancer** to be **true** for class transforming to be enabled.
jboss.as.jpa.scopedname	Specify the qualified application-scoped persistence unit name to be used. By default, this is set to the application name and persistence unit name, collectively. The **hibernate.cache.region_prefix** defaults to whatever you set **jboss.as.jpa.scopedname** to. Make sure you set the **jboss.as.jpa.scopedname** value to a value not already in use by other applications deployed on the same application server instance.
jboss.as.jpa.deferdetach	Controls whether transaction-scoped persistence context used in non-Jakarta Transactions transaction thread, will detach loaded entities after each **EntityManager** invocation or when the persistence context is closed. The default value is **false**. If set to **true**, the detach is deferred until the context is closed.
wildfly.jpa.default-unit	Set to **true** to choose the default persistence unit in an application. This is useful if you inject a persistence context without specifying the **unitName**, but have multiple persistence units specified in your **persistence.xml** file.
wildfly.jpa.twophasebootstrap	Persistence providers allow a two-phase persistence unit bootstrap, which improves Jakarta Persistence integration with Contexts and Dependency Injection. Setting the **wildfly.jpa.twophasebootstrap** value to **false** disables the two-phase bootstrap for the persistence unit that contains the value.
wildfly.jpa.allowdefaultdatasourceuse	Set to **false** to prevent persistence unit from using the default datasource. The default value is **true**. This is only important for persistence units that do not specify a datasource.
wildfly.jpa.hibernate.search.module	Controls which version of Hibernate Search to include on the classpath. The default is **auto**; other valid values are **none** or a full module identifier to use an alternative version.

A.3. POLICY PROVIDER PROPERTIES

Table A.19. policy-provider Attributes

Property	Description
custom-policy	A custom policy provider definition.
jacc-policy	A policy provider definition that sets up Jakarta Authorization and related services.

Table A.20. custom-policy Attributes

Property	Description
class-name	The name of a **java.security.Policy** implementation referencing a policy provider.
module	The name of the module to load the provider from.

Table A.21. jacc-policy Attributes

Property	Description
policy	The name of a **java.security.Policy** implementation referencing a policy provider.
configuration-factory	The name of a **javax.security.jacc.PolicyConfigurationFactory** implementation referencing a policy configuration factory provider.
module	The name of the module to load the provider from.

A.4. JAVA EE SPECIFICATIONS RELEVANT FOR JBOSS EAP AND THE CORRESPONDING JAKARTA EE SPECIFICATIONS

Table A.22. Java EE Specifications Relevant for JBoss EAP and the Corresponding Jakarta EE Specifications

Java EE 8 Specification	Java EE 8 Abbreviation	Jakarta EE 8 Specification
Java Servlet		Jakarta Servlet
JavaServer Faces	JSF	Jakarta Server Faces
Java API for WebSocket		Jakarta WebSocket
Concurrency Utilities for Java EE		Jakarta Concurrency

Java EE 8 Specification	Java EE 8 Abbreviation	Jakarta EE 8 Specification
Interceptors		Jakarta Interceptors
Java Authentication Service Provider Interface for Containers	JASPIC	Jakarta Authentication
Java Authorization Contract for Containers	JACC	Jakarta Authorization
Java EE Security API		Jakarta Security
Java Message Service	JMS	Jakarta Messaging
Java Persistence API	JPA	Jakarta Persistence
Java Transaction API	JTA	Jakarta Transactions
Batch Applications for the Java Platform	JBatch	Jakarta Batch
JavaMail API		Jakarta Mail
Java EE Connector Architecture	JCA	Jakarta Connectors
Common Annotations for Java Platform		Jakarta Annotations
JavaBeans Activation Framework	JAF	Jakarta Activation
Bean Validation	JBV	Jakarta Bean Validation
Expression Language	JEL	Jakarta Expression Language
Enterprise JavaBeans	EJB	Jakarta Enterprise Beans
Java Architecture for XML Binding	JAXB	JAXB
Java API for JSON Binding	JSON-B	Jakarta JSON Binding
JavaServer Pages	JSP	Jakarta Server Pages
Java API for XML-Based Web Services	JAX-WS	Jakarta XML Web Services
Java API for RESTful Web Services	JAX-RS	Jakarta RESTful Web Services

Java EE 8 Specification	Java EE 8 Abbreviation	Jakarta EE 8 Specification
JavaServer Pages Standard Tag Library	JSTL	Jakarta Standard Tag Library
Contexts and Dependency Injections	CDI	Jakarta Contexts and Dependency Injection
Java API for JSON Processing	JSON-P	Jakarta JSON Processing
Java API for XML-Based RPC	JAX-RPC	Jakarta XML RPC
Java API for XML Registries	JAXR	Jakarta XML Registries

A.5. JAKARTA EE PROFILES AND TECHNOLOGIES REFERENCE

The following tables list the Jakarta EE technologies by category and note whether they are included in the Web Profile or Full Platform profiles.

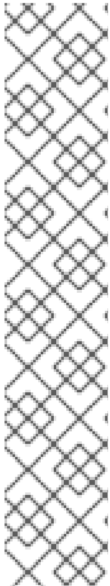
- Jakarta EE Web Application Technologies
- Jakarta EE Enterprise Application Technologies
- Jakarta EE Web Services Technologies
- Jakarta EE Management and Security Technologies

See Jakarta EE Specification for the specifications.

Table A.23. Jakarta EE Web Application Technologies

Technology	Web Profile	Full Platform
Jakarta WebSocket 1.1	✔	✔
Jakarta JSON Binding 1.0	✔	✔
Jakarta JSON Processing 1.1	✔	✔
Jakarta Servlet 4.0	✔	✔
Jakarta Server Faces 2.3	✔	✔
Jakarta Expression Language 3.0	✔	✔
Jakarta Server Pages 2.3	✔	✔
Jakarta Standard Tag Library 1.2 [1]	✔	✔

[1] Additional Jakarta Standard Tag Library information:



NOTE

A known security risk in JBoss EAP exists where the Jakarta Standard Tag Library allows the processing of external entity references in untrusted XML documents which could access resources on the host system and, potentially, allow arbitrary code execution.

To avoid this, the JBoss EAP server has to be run with system property **org.apache.taglibs.standard.xml.accessExternalEntity** correctly set, usually with an empty string as value. This can be done in two ways:

- Configuring the system properties and restarting the server.

```
org.apache.taglibs.standard.xml.accessExternalEntity
```

- Passing **-Dorg.apache.taglibs.standard.xml.accessExternalEntity=""** as an argument to the **standalone.sh** or **domain.sh** scripts.

Table A.24. Jakarta EE Enterprise Application Technologies

Technology	Web Profile	Full Platform
Jakarta Batch 1.0		✔
Jakarta Concurrency 1.0		✔
Jakarta Contexts and Dependency Injection 2.0	✔	✔
Jakarta Contexts and Dependency Injection 1.0	✔	✔
Jakarta Bean Validation 2.0	✔	✔
Jakarta Managed Beans 1.0	✔	✔
Jakarta Enterprise Beans 3.2		✔
Jakarta Interceptors 1.2	✔	✔
Jakarta Connectors 1.7		✔
Jakarta Persistence 2.2	✔	✔
Jakarta Annotations 1.3		✔
Jakarta Messaging 2.0		✔
Jakarta Transactions 1.2	✔	✔

Technology	Web Profile	Full Platform
Jakarta Mail 1.6		✔

Table A.25. Jakarta EE Web Services Technologies

Technology	Web Profile	Full Platform
Jakarta RESTful Web Services 2.1		✔
Jakarta Enterprise Web Services 1.3		✔
Web Services Metadata for the Java Platform 2.1		✔
Jakarta XML RPC 1.1 (Optional)		
Jakarta XML Registries 1.0 (Optional)		

Table A.26. Jakarta EE Management and Security Technologies

Technology	Web Profile	Full Platform
Jakarta Security 1.0	✔	✔
Jakarta Authentication 1.1	✔	✔
Jakarta Authorization 1.5		✔
Jakarta Deployment 1.2 (Optional)		✔
Jakarta Management 1.1		✔
Jakarta Debugging Support for Other Languages 1.0		✔

Revised on 2022-02-01 13:02:48 UTC