JBoss Enterprise Application Platform

4.2.0

Configuration Guide

ISBN: N/A
Publication date:

This book is a guide to the configuration of the Jboss Application Server.

JBoss Enterprise Application Platform : Configuration Guide

Copyright © 2007 Red Hat, Inc.

Copyright © 2007 by Red Hat, Inc. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, V1.0 or later (the latest version is presently available at *http://www.opencontent.org/openpub/*).

Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder.

Distribution of the work or derivative of the work in any standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from the copyright holder.

Red Hat and the Red Hat "Shadow Man" logo are registered trademarks of Red Hat, Inc. in the United States and other countries.

All other trademarks referenced herein are the property of their respective owners.

The GPG fingerprint of the security@redhat.com key is:

CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E

1801 Varsity Drive
Raleigh, NC 27606-2072
USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588
Research Triangle Park, NC 27709
USA

What this Book Covers xiii
About JBoss xv
About Open Source xvii
About Professional Open Source xix
I. Java EE 5 Application Configuration 1
1. Enterprise Applications with EJB3 Services 3
1. Session Beans 3
2. Entity Beans (a.k.a. Java Persistence API) 5
2.1. The persistence.xml file 8
2.2. Use Alternative Databases 9
2.3. Default Hibernate options 10
3. Message Driven Beans 11
4. Package and Deploy EJB3 Services 12
4.1. Deploy the EJB3 JAR 12
4.2. Deploy EAR with EJB3 JAR 13
2. Deployment 17
1. Deployable Application Types 17
2. Standard Server Configurations 18
2.1. The production Configuration 18
2.2. Further Tuning from the production Configuration 19
II. JBoss AS Infrastructure 21
3. The JBoss JMX Microkernel 23
1. An Introduction to JMX 23
1.1. Instrumentation Level 25
1.2. Agent Level 25
1.3. Distributed Services Level 26
1.4. JMX Component Overview 27
2. JBoss JMX Implementation Architecture 30
2.1. The JBoss ClassLoader Architecture 30
2.2. Class Loading and Types in Java 30
2.3. JBoss XMBeans 51
3. Connecting to the JMX Server 59
3.1. Inspecting the Server - the JMX Console Web Application 59
3.2. Connecting to JMX Using RMI 64
3.3. Command Line Access to JMX 67
3.4. Connecting to JMX Using Any Protocol 72
4. Using JMX as a Microkernel 72
4.1. The Startup Process 72
4.2. JBoss MBean Services 74
4.3. Writing JBoss MBean Services 84
4.4. Deployment Ordering and Dependencies 99
5. JBoss Deployer Architecture 110
5.1. Deployers and ClassLoaders 111
6. Remote Access to Services, Detached Invokers 113
6.1. A Detached Invoker Example, the MBeanServer Invoker Adaptor Service 115

6.2. Detached Invoker Reference 121
4. Naming on JBoss 129
1. An Overview of JNDI 129
1.1. Names 129
1.2. Contexts 130
2. The JBossNS Architecture 132
3. The Naming InitialContext Factories 134
3.1. The standard naming context factory 134
3.2. The org.jboss.naming.NamingContextFactory 136
3.3. Naming Discovery in Clustered Environments 136
3.4. The HTTP InitialContext Factory Implementation 137
3.5. The Login InitialContext Factory Implementation 138
3.6. The ORBInitialContextFactory 138
4. JNDI over HTTP 139
4.1. Accessing JNDI over HTTP 139
4.2. Accessing JNDI over HTTPS 143
4.3. Securing Access to JNDI over HTTP 145
4.4. Securing Access to JNDI with a Read-Only Unsecured Context 147
5. Additional Naming MBeans 149
5.1. JNDI Binding Manager 149
5.2. The org.jboss.naming.NamingAlias MBean 151
5.3. org.jboss.naming.ExternalContext MBean 151
5.4. The org.jboss.naming.JNDIView MBean 154
6. J2EE and JNDI - The Application Component Environment 156
6.1. ENC Usage Conventions 158
5. Connectors on JBoss 171
1. JCA Overview 171
2. An Overview of the JBossCX Architecture 174
2.1. BaseConnectionManager2 MBean 174
2.2. RARDeployment MBean 175
2.3. JBossManagedConnectionPool MBean 176
2.4. CachedConnectionManager MBean 177
2.5. A Sample Skeleton JCA Resource Adaptor 177
3. Configuring JDBC DataSources 184
4. Configuring Generic JCA Adaptors 196
6. Transactions on JBoss 201
1. Transaction/JTA Overview 201
1.1. Pessimistic and optimistic locking 202
1.2. The components of a distributed transaction 202
1.3. The two-phase XA protocol 203
1.4. Heuristic exceptions 204
1.5. Transaction IDs and branches 204
2. JTS support 205
3. Web Services Transactions 205
4. Configuring JBoss Transactions 205
5. Local versus distributed transactions 206
7. Messaging on JBoss 207

1. JMS Examples 207
1.1. A Point-To-Point Example 207
1.2. A Pub-Sub Example 210
1.3. A Pub-Sub With Durable Topic Example 216
1.4. A Point-To-Point With MDB Example 219
2. JBoss MQ Overview 226
2.1. Invocation Layer 226
2.2. Security Manager 227
2.3. Destination Manager 227
2.4. Message Cache 227
2.5. State Manager 227
2.6. Persistence Manager 227
2.7. Destinations 228
3. JBoss MQ Configuration and MBeans 228
3.1. org.jboss.mq.il.jvm.JVMServerILService 229
3.2. org.jboss.mq.il.uil2.UILServerILService 230
3.3. org.jboss.mq.il.http.HTTPServerILService 233
3.4. org.jboss.mq.server.jmx.Invoker 233
3.5. org.jboss.mq.server.jmx.InterceptorLoader 234
3.6. org.jboss.mq.sm.jdbc.JDBCStateManager 234
3.7. org.jboss.mq.security.SecurityManager 234
3.8. org.jboss.mq.server.jmx.DestinationManager 235
3.9. org.jboss.mq.server.MessageCache 236
3.10. org.jboss.mq.pm.jdbc2.PersistenceManager 238
3.11. Destination MBeans 240
4. Specifying the MDB JMS Provider 244
4.1. org.jboss.jms.jndi.JMSProviderLoader MBean 245
4.2. org.jboss.jms.asf.ServerSessionPoolLoader MBean 247
4.3. Integrating non-JBoss JMS Providers 247
8. Security on JBoss 249
1. J2EE Declarative Security Overview 249
1.1. Security References 249
1.2. Security Identity 251
1.3. Security roles 253
1.4. EJB method permissions 254
1.5. Web Content Security Constraints 258
1.6. Enabling Declarative Security in JBoss 260
2. An Introduction to JAAS 261
2.1. What is JAAS? 261
3. The JBoss Security Model 265
3.1. Enabling Declarative Security in JBoss Revisited 268
4. The JBoss Security Extension Architecture 273
4.1. How the JaasSecurityManager Uses JAAS 275
4.2. The JaasSecurityManagerService MBean 278
4.3. The JaasSecurityDomain MBean 281
5. Defining Security Domains 282
5.1. Loading Security Domains 284

5.2. The DynamicLoginConfig service 286
5.3. Using JBoss Login Modules 287
5.4. Writing Custom Login Modules 301
6. The Secure Remote Password (SRP) Protocol 310
6.1. Providing Password Information for SRP 314
6.2. Inside of the SRP algorithm 316
7. Running JBoss with a Java 2 security manager 322
8. Using SSL with JBoss using JSSE 324
9. Configuring JBoss for use Behind a Firewall 329
10. How to Secure the JBoss Server 329
10.1. The JMX Console 330
10.2. The Web Console 330
10.3. The HTTP Invokers 330
10.4. The JMX Invoker 330
9. Web Services 331
1. JAX-RPC Service Endpoints 331
2. EJB Endpoints 336
3. Web Services Clients 339
3.1. A JAX-RPC client 339
3.2. Service references 341
10. Additional Services 347
1. Memory and Thread Monitoring 347
2. The Log4j Service 347
3. System Properties Management 348
4. Property Editor Management 349
5. Services Binding Management 349
5.1. AttributeMappingDelegate 351
5.2. XSLTConfigDelegate 352
5.3. XSLTFileDelegate 353
5.4. The Sample Bindings File 355
6. RMI Dynamic Class Loading 356
7. Scheduling Tasks 356
7.1. org.jboss.varia.scheduler.Scheduler 357
8. The Timer Service 359
9. The BarrierController Service 362
10. Exposing MBean Events via SNMP 365
III. Clustering Configuration 367
11. Quick Tutorial to Setup a Clustered Web Application 369
1. Setup the simple web cluster 369
1.1. Setup the load balancer 370
1.2. Configure JBoss AS nodes 373
1.3. Shared Database 374
2. Optional improvements to the simple cluster 374
2.1. Failover support 374
2.2. Database cache 375
3. Basic optimization 376
12. JBossCache and JGroups Services 379

1. JGroups Configuration 379
1.1. Transport Protocols 380
1.2. Discovery Protocols 383
1.3. Failure Detection Protocols 386
1.4. Reliable Delivery Protocols 388
1.5. Other Configuration Options 389
2. JBossCache Configuration 392
13. Clustering 397
1. Introduction 397
1.1. Cluster Definition 397
1.2. Service Architectures 400
1.3. Load-Balancing Policies 402
1.4. Farming Deployment 403
1.5. Distributed state replication services 405
2. Clustered JNDI Services 406
2.1. How it works 406
2.2. Client configuration 408
2.3. JBoss configuration 409
3. Clustered Session EJBs 411
3.1. Stateless Session Bean in EJB 2.x 412
3.2. Stateful Session Bean in EJB 2.x 414
3.3. Stateless Session Bean in EJB 3.0 416
3.4. Stateful Session Bean in EJB 3.0 417
4. Clustered Entity EJBs 419
4.1. Entity Bean in EJB 2.x 419
4.2. Entity Bean in EJB 3.0 420
5. HTTP Services 423
5.1. Download the software 424
5.2. Configure Apache to load mod_jk 424
5.3. Configure worker nodes in mod_jk 426
5.4. Configure JBoss 428
5.5. Configure HTTP session state replication 429
5.6. Enabling session replication in your application 431
5.7. Use FIELD level replication 432
5.8. Monitoring session replication 434
5.9. Using Single Sign On 435
6. Clustered JMS Services 435
6.1. High Availability Singleton Fail-over 435
IV. Legacy EJB Support 439
14. EJBs on JBoss 441
1. The EJB Client Side View 441
1.1. Specifying the EJB Proxy Configuration 444
2. The EJB Server Side View 450
2.1. Detached Invokers - The Transport Middlemen 450
2.2. The HA JRMPInvoker - Clustered RMI/JRMP Transport 453
2.3. The HA HttpInvoker - Clustered RMI/HTTP Transport 454
3. The EJB Container 455

3.1. EJBDeployer MBean 455
3.2. Container Plug-in Framework 470
4. Entity Bean Locking and Deadlock Detection 482
4.1. Why JBoss Needs Locking 482
4.2. Entity Bean Lifecycle 482
4.3. Default Locking Behavior 483
4.4. Pluggable Interceptors and Locking Policy 483
4.5. Deadlock 484
4.6. Advanced Configurations and Optimizations 487
4.7. Running Within a Cluster 489
4.8. Troubleshooting 489
5. EJB Timer Configuration 490
15. The CMP Engine 493
1. Example Code 493
1.1. Enabling CMP Debug Logging 494
1.2. Running the examples 495
2. The jbosscmp-jdbc Structure 497
3. Entity Beans 498
3.1. Entity Mapping 501
4. CMP Fields 505
4.1. CMP Field Declaration 505
4.2. CMP Field Column Mapping 505
4.3. Read-only Fields 508
4.4. Auditing Entity Access 509
4.5. Dependent Value Classes (DVCs) 510
5. Container Managed Relationships 515
5.1. CMR-Field Abstract Accessors 515
5.2. Relationship Declaration 516
5.3. Relationship Mapping 517
6. Queries 524
6.1. Finder and select Declaration 525
6.2. EJB-QL Declaration 525
6.3. Overriding the EJB-QL to SQL Mapping 526
6.4. JBossQL 528
6.5. DynamicQL 529
6.6. DeclaredSQL 530
6.7. EJBQL 2.1 and SQL92 queries 535
6.8. BMP Custom Finders 535
7. Optimized Loading 536
7.1. Loading Scenario 536
7.2. Load Groups 538
7.3. Read-ahead 539
8. Loading Process 546
8.1. Commit Options 546
8.2. Eager-loading Process 547
8.3. Lazy loading Process 548
8.4. Lazy loading result sets 552

9. Transactions 553
10. Optimistic Locking 555
11. Entity Commands and Primary Key Generation 560
11.1. Existing Entity Commands 560
12. Defaults 563
12.1. A sample jbosscmp-jdbc.xml defaults declaration 565
13. Datasource Customization 567
13.1. Type Mapping 567
13.2. Function Mapping 570
13.3. Mapping 571
13.4. User Type Mappings 572
A. Book Example Installation 575
B. Use Alternative Databases with JBoss AS 577
1. How to Use Alternative Databases 577
2. Install JDBC Drivers 577
2.1. Special notes on Sybase 578
3. Creating a DataSource for the External Database 579
4. Change Database for the JMS Services 580
5. Support Foreign Keys in CMP Services 581
6. Specify Database Dialect for Java Persistence API 581
7. Change Other JBoss AS Services to Use the External Database 582
7.1. The Easy Way 582
7.2. The More Flexible Way 583
8. A Special Note About Oracle DataBases 584

What this Book Covers

The primary focus of this book is the presentation of the standard JBoss 4.2 architecture components from both the perspective of their configuration and architecture. As a user of a standard JBoss distribution you will be given an understanding of how to configure the standard components. Note that this book is not an introduction to J2EE or how to use J2EE in applications. It focuses on the internal details of the JBoss server architecture and how our implementation of a given J2EE container can be configured and extended.

As a JBoss developer, you will be given a good understanding of the architecture and integration of the standard components to enable you to extend or replace the standard components for your infrastructure needs. We also show you how to obtain the JBoss source code, along with how to build and debug the JBoss server.

About JBoss

JBoss, a division of Red Hat, is the global leader in open source middleware software, combining enterprise-class JEMS open source software with the industry's leading services and tools to provide simply a better way to transform your business to Service-Oriented Architecture (SOA).

JBoss, pioneered the disruptive Professional Open Source model, which combines the best of the open source and proprietary software worlds to make open source a safe choice for the enterprise and give CIOs peace of mind. This includes the royalty-free software, transparent development and active community inherent in open source and the accountability and professional support services expected of a traditional software vendor. The company finds innovative open source projects and professionalizes the project from a hobby into a livelihood by hiring the lead developer(s), often the founders themselves. JBoss provides the resources, core development and support services to enable popular open source projects to scale into enterprise-class software.

Coverage: North America and Europe on a direct basis. JBoss provides coverage worldwide via our extensive authorized partner network.

Mission Statement: JBoss' mission is to revolutionize the way enterprise middleware software is built, distributed, and supported through the Professional Open Source model. We are committed to delivering innovative and high quality technology and services that make JBoss the safe choice for enterprises and software providers.

Customers: Enterprise customers deploying JBoss technologies in mission-critical applications with professional services support from JBoss include Aviva Canada, Continental Airlines, La Quinta, NLG, MCI, Nielsen Media Research and Travelocity. For a current list of customer success stories, please visit the *Customers* [http://www.jboss.com/customers/index] section of our website.

Partners: JBoss works with software and hardware vendors, systems integrators and OEMs to deliver implementation services, frontline support, and certification for products embedded with JBoss technologies. For more information on the JBoss Certified Partner Program, please visit the *Partners* [http://www.jboss.com/partners/index] section of our website.

Professional Open Source(tm) from JBoss Inc. offers you:

- Standards-based and stable Java Middleware technology
- No cost open source product licenses
- Backed by a professional and expert support staff
- Comprehensive services including *Professional Support* [http://www.jboss.com/services/profsupport], *Training* [http://www.jboss.com/services/certification], and *Consulting* [http://www.jboss.com/services/consulting]

- A very large and active community of developers
- An extensive worldwide network of authorized and certified *partners* [http://www.jboss.com/partners/index]

Benefits of Professional Open Source from JBoss Inc.:

- Lowest possible total cost of ownership
- Reliable and safe technology
- Support, accountability, and trust from a stable company
- Expedited problem resolution compared to commercial software vendors

About Open Source

The basic idea behind open source is very simple: When programmers can read, redistribute, and modify the source code for a piece of software, the software evolves. People improve it, people adapt it, people fix bugs. And this can happen at a speed that, if one is used to the slow pace of conventional software development, seems astonishing. Open Source is an often-misunderstood term relating to free software. The Open Source Initiative (OSI) web site provides a number of resources that define the various aspects of Open Source including an Open Source Definition at: *http://www.opensource.org/docs/definition.html*. The following quote from the OSI home page summarizes the key aspects as they relate to JBoss nicely:

> We in the open source community have learned that this rapid evolutionary process produces better software than the traditional closed model, in which only very few programmers can see the source and everybody else must blindly use an opaque block of bits.
>
> Open Source Initiative exists to make this case to the commercial world.
>
> Open source software is an idea whose time has finally come. For twenty years it has been building momentum in the technical cultures that built the Internet and the World Wide Web. Now it's breaking out into the commercial world, and that's changing all the rules. Are you ready?
>
> —The Open Source Initiative

About Professional Open Source

JBoss is the leader in the second generation of open source, which we have termed Professional Open Source. The Professional Open Source methodology is based on the following:

1. We hire and pay experts in the open source community to write exceptional and innovative software full-time.
2. We only use open source licenses that are friendly to end-user IT shops, independent software vendors, and the community itself.
3. Directly and through our authorized partners, we deliver the best support services available; all of which are backed up by the real product experts.
4. Unlike first generation open source providers, we control the direction and source code for our projects. We can ensure that all bug fixes and patches are rolled into future versions of our products.
5. By combining enterprise-proven technology, business-friendly open source licenses, and world-class support services, we have made Professional Open Source the safe choice for end-user enterprises and independent software vendors alike.

Part I. Java EE 5 Application Configuration

Enterprise Applications with EJB3 Services

In the previous chapter, we discussed how to write simple web applications packaged in WAR files. Using JavaServer Faces (JSF), you can write web applications that conform to strict MVC-based component models. However, for most JBoss platform users, the web applcation in the WAR is just the UI frontend. The real business logic of the application, such as database access, transactions, secuirty etc., is handled by enterprise business components such as Enterprise JavaBeans (EJBs).

EJB3 (Enterprise JavaBean 3.0) provides the core component model for Java EE 5 applications. An EJB3 bean is a managed component that is automatically wired to take advantage of all services the J2EE server container provides, such as transaction, security, persistence, naming, dependency injection, etc. The managed component allows developers to focus on the business logic, and leave the cross-cutting concerns to the container as configurations. As an application developer, you need not create or destroy the components yourself. You only need to ask for an EJB3 bean from the Java EE container by its name, and then you can call its methods with all configured container services applied. You can get access to an EJB3 bean from either inside or outside of the J2EE container.

JBoss AS 4.2 supports EJB3 out of the box. For JBoss AS 4.0.x, you can add support for EJB3 by manually installing the JBoss EJB3 deployer, or installing the server from a JEMS GUI installer and select a EJB3 profile.

The details of the EJB3 component programming model is beyond the scope of this guide. Most EJB3 interfaces and annotations are part of the Java EE 5 standard and hence they are the same for all Java EE 5 compliant application servers. Interested readers should refer to the EJB3 specification or numerous EJB3 books to learn more about EJB3 programming.

In this chapter, we only cover EJB3 configuration issues that are specific to the JBoss AS. For instance, we discuss the JNDI naming conventions for EJB3 components inside the JBoss AS, the optional configurations for the Hibernate persistence engine for entity beans, as well as custom options in the JBoss EJB3 deployer.

1. Session Beans

Session beans are widely used to provide transactional services for local and remote clients. To write a session bean, you need an interface and an implementation class.

```
@Local
public interface MyBeanInt {
  public String doSomething (String para1, int para2);
}

@Stateless
```

```
public class MyBean implements MyBeanInt {

  public String doSomething (String para1, int para2) {
    ... implement the logic ...
  }

}
```

When you invoke a session bean method, the method execution is automatically managed by the transaction manager and the security manager in the server. You can specify the transactional or security properties for each method using annotations on the method. A session bean instance can be reused by many clients. Depending on whether the server maintains the bean's internal state between two clients, the session bean can be stateless or stateful. Depending on whether the bean is available to remote clients (i.e., clients outside of the current JVM for the server), the session bean can be local or remote. All these are configurable via standard annotations on the beans.

After you define a session bean, how does the client get access to it? As we discussed, the client does not create or destroy EJB3 components, it merely asks the server for a reference of an existing instance managed by the server. That is done via JNDI. In JBoss AS, the default local JNDI name for a session bean is dependent on the deployment packaging of the bean class.

- If the bean is deployed in a standalone JAR file in the `jboss-as/production/deploy` directory, the bean is accessible via local JNDI name `MyBean/local`, where `MyBean` is the implementation class name of the bean as we showed earlier. The "local" JNDI in JBoss AS means that the JNDI name is relative to `java:comp/env/`.
- If the JAR file containing the bean is packaged in an EAR file, the local JNDI name for the bean is `myapp/MyBean/local`, where `myapp` is the root name of the EAR archive file (e.g., `myapp.ear`, see later for the EAR packaging of EJB3 beans).

Of course, you should change `local` to `remote` if the bean interface is annotated with `@Remote` and the bean is accessed from outside of the server it is deployed on. Below is the code snippet to get a reference of the MyBean bean in a web application (e.g., in a servlet or a JSF backing bean) packaged in `myapp.ear`, and then invoke a managed method.

```
try {
  InitialContext ctx = new InitialContext();
  MyBeanInt bean = (MyBeanInt) ctx.lookup("myapp/MyBean/local");
} catch (Exception e) {
  e.printStackTrace ();
}
```

```
... ...

String result = bean.doSomething("have fun", 1);

... ...
```

What the client gets from the JNDI is essentially a "stub" or "proxy" of the bean instance. When the client invokes a method, the proxy figures out how to route the request to the server and marshal together the response.

If you do not like the default JNDI names, you can always specify your own JNDI binding for any bean via the `@LocalBinding` annotation on the bean implementation class. The JNDI binding is always "local" under the `java:comp/env/` space. For instance, the following bean class definition results in the bean instances available under JNDI name `java:comp/env/MyService/MyOwnName`.

```
@Stateless
@LocalBinding (jndiBinding="MyService/MyOwnName")
public class MyBean implements MyBeanInt {

  public String doSomething (String para1, int para2) {
    ... implement the logic ...
  }

}
```

Injecting EJB3 Beans into the Web Tier

Java EE 5 allows you to inject EJB3 bean instances directly into the web application via annotations without explicit JNDI lookup. This behavior is not yet supported in JBoss AS 4.2. However, the JBoss Enterprise Platform provides an integration framework called JBoss Seam. JBoss Seam brings EJB3 / JSF integration to new heights far beyond what Java EE 5 provides. Please see more details in the JBoss Seam reference guide bundled with the platform.

2. Entity Beans (a.k.a. Java Persistence API)

EJB3 session beans allow you to implement data accessing business logic in transactional methods. To actually access the database, you will need EJB3 entity beans and the entity manager API. They are collectively called the Java Persistence API (JPA).

EJB3 Entity Beans are Plain Old Java Objects (POJOs) that map to relational database tables. For instance, the following entity bean class maps to a relational table named customer. The table has three columns: name, age, and signupdate. Each instance of the bean corresponds to a row of data in the table.

```
@Entity
public class Customer {

  String name;

  public String getName () {
    return name;
  }

  public void setName (String name) {
    this.name = name;
  }

  int age;

  public int getAge () {
    return age;
  }

  public void setAge (int age) {
    this.age = age;
  }

  Date signupdate;

  public Date getSignupdate () {
    return signupdate;
  }

  public void setSignupdate (Date signupdate) {
    this.signupdate = signupdate;
  }
}
```

Besides simple data properties, the entity bean can also contain references to other entity beans with relational mapping annotations such as @OneToOne, @OneToMany, @ManyToMany etc. The relationships of those entity objects will be automatically set up in the database as foreign keys. For instance, the following example shows that each record in the Customer table has one corresponding record in the Account table, multiple corresponding records in the Order table, and each record in the Employee table has multiple corresponding records in the Customer table.

```

```

```
@Entity
public class Customer {

  ... ...

  Account account;

  @OneToOne
  public Account getAccount () {
    return account;
  }

  public void setAccount (Accout account) {
    this.account = account;
  }

  Employee salesRep;

  @ManyToOne
  public Employee getSalesRep () {
    return salesRep;
  }

  public void setSalesRep (Employee salesRep) {
    this.salesRep = salesRep;
  }

  Vector <Order> orders;

  @OneToMany
  public Vector <Order> getOrders () {
    return orders;
  }

  public void setOrders (Vector <Order> orders) {
    this.orders = orders;
  }
```

Using the EntityManager API, you can create, update, delete, and query entity objects. The EntityManager transparently updates the underlying database tables in the process. You can obtain an EntityManager object in your EJB3 session bean via the @PersistenceContext annotation.

```
@PersistenceContext
EntityManager em;

Customer customer = new Cutomer ();
// populate data in customer
```

```
// Save the newly created customer object to DB
em.persist (customer);

// Increase age by 1 and auto save to database
customer.setAge (customer.getAge() + 1);

// delete the customer and its related objects from the DB
em.remove (customer);

// Get all customer records with age > 30 from the DB
List <Customer> customers = em.query (
     "select c from Customer where c.age > 30");
```

The detailed use of the EntityManager API is beyond the scope of this book. Interested readers should refer to the JPA documentation or Hibernate EntityManager documentation.

2.1. The persistence.xml file

The EntityManager API is great, but how does the server know which database it is supposed to save / update / query the entity objects? How do we configure the underlying object-relational-mapping engine and cache for better performance and trouble shooting? The persistence.xml file gives you complete flexibility to configure the EntityManager.

The persistence.xml file is a standard configuration file in JPA. It has to be included in the META-INF directory inside the JAR file that contains the entity beans. The persistence.xml file must define a persistence-unit with a unique name in the current scoped classloader. The provider attribute specifies the underlying implementation of the JPA EntityManager. In JBoss AS, the default and only supported / recommended JPA provider is Hibernate. The jta-data-source points to the JNDI name of the database this persistence unit maps to. The java:/DefaultDS here points to the HSQL DB embedded in the JBoss AS. Please refer to *Appendix B, Use Alternative Databases with JBoss AS* on how to setup alternative databases for JBoss AS.

```
<persistence>
   <persistence-unit name="myapp">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
      <jta-data-source>java:/DefaultDS</jta-data-source>
      <properties>
         ... ...
      </properties>
   </persistence-unit>
</persistence>
```

Inject EntityManager by persistence-unit name

Since you might have multiple instances of persistence-unit defined in the same application, you typically need to explicitly tell the @PersistenceContext annotation which unit you want to inject. For instance, @PersistenceContext(name="myapp") injects the EntityManager from the persistence-unit named "myapp".

However, if you deploy your EAR application in its own scoped classloader and have only one persistence-unit defined in the whole application, you can omit the "name" on @PersistenceContext. See later in this chapter for EAR packaging and deployment.

The properties element in the persistence.xml can contain any configuration properties for the underlying persistence provider. Since JBoss AS uses Hibernate as the EJB3 persistence provider, you can pass in any Hibernate options here. Please refer to the Hibernate and Hibernate EntityManager documentation for more details. Here we will just give an example to set the SQL dialect of the persistence engine to HSQL, and to create tables from the entity beans when the application starts and drop those tables when the application stops.

```
<persistence>
   <persistence-unit name="myapp">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
      <jta-data-source>java:/DefaultDS</jta-data-source>
      <properties>
         property name="hibernate.dialect"
                  value="org.hibernate.dialect.HSQLDialect"/>
         <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      </properties>
   </persistence-unit>
</persistence>
```

2.2. Use Alternative Databases

To use an alternative database other than the built-in HSQL DB to back your entity beans, you need to first define the data source for the database and register it in the JNDI. This is done via the *-ds.xml files in the deploy directory. Please see *Section 3, “Configuring JDBC DataSources”* for more details. Examples of *-ds.xml files for various databases are available in jboss-as/docs/examples/jca directory in the server.

Then, in the persistence.xml, you need to change the jta-data-source attribute to point to the new data source in JNDI (e.g., java:/MysqlDS if you are using the default mysql-ds.xml to setup

a MySQL external database).

In most cases, Hibernate tries to automatically detect the database it connects to and then automatically selects an appropriate SQL dialect for the database. However, we have found that this detection does not always work, especially for less used database servers. We recommend you to set the hibernate.dialect property explicitly in persistence.xml. Here are the Hibernate dialect for database servers officially supported on the JBoss platform.

- Oracle 9i and 10g: org.hibernate.dialect.Oracle9Dialect
- Microsoft SQL Server 2005: org.hibernate.dialect.SQLServerDialect
- PostgresSQL 8.1: org.hibernate.dialect.PostgreSQLDialect
- MySQL 5.0: org.hibernate.dialect.MySQL5Dialect
- DB2 8.0: org.hibernate.dialect.DB2Dialect
- Sybase ASE 12.5: org.hibernate.dialect.SybaseDialect

2.3. Default Hibernate options

Hibernate has many configuration properties. For the properties that you do not specify in the persistence.xml file, JBoss AS will provide a reasonable set of default values. The default Hibernate property values are specified in the `jboss-as/server/production/deploy/ejb3.deployer/MEAT-INF/persistence.properties` file. Below is the `persistence.properties` file bundled in JBoss AS 4.2. Notice the options that are commented out. They give you an idea of available properties in your `persistence.xml` file.

```
hibernate.transaction.manager_lookup_class=org.hibernate.transaction.JBossTransactionManagerL
#hibernate.connection.release_mode=after_statement
#hibernate.transaction.flush_before_completion=false
#hibernate.transaction.auto_close_session=false
#hibernate.query.factory_class=org.hibernate.hql.ast.ASTQueryTranslatorFactory
#hibernate.hbm2ddl.auto=create-drop
#hibernate.hbm2ddl.auto=create
hibernate.cache.provider_class=org.hibernate.cache.HashtableCacheProvider
# Clustered cache with TreeCache
#hibernate.cache.provider_class=org.jboss.ejb3.entity.TreeCacheProviderHook
#hibernate.treecache.mbean.object_name=jboss.cache:service=EJB3EntityTreeCache
#hibernate.dialect=org.hibernate.dialect.HSQLDialect
hibernate.jndi.java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
hibernate.jndi.java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
hibernate.bytecode.use_reflection_optimizer=false
# I don't think this is honored, but EJB3Deployer uses it
hibernate.bytecode.provider=javassist
```

3. Message Driven Beans

Messaging driven beans are specialized EJB3 beans that receive service requests via JMS messages instead of proxy method calls from the "stub". So, a crucial configuration parameter for the message driven bean is to specify which JMS message queue its listens to. When there is an incoming message in the queue, the server invokes the beans's `onMessage()` method, and passes in the message itself for processing. The bean class specifies the JMS queue it listens to in the @MessageDriven annotation. The queue is registered under the local JNDI java:comp/env/ name space.

```
@MessageDriven(activationConfig =
{
  @ActivationConfigProperty(propertyName="destinationType",
    propertyValue="javax.jms.Queue"),
  @ActivationConfigProperty(propertyName="destination",
    propertyValue="queue/MyQueue")
})
public class MyJmsBean implements MessageListener {

  public void onMessage (Message msg) {
    // ... do something with the msg ...
  }

  // ... ...
}
```

When a message driven bean is deployed, its incoming message queue is automatically created if it does not exist already. To send a message to the bean, you can use the standard JMS API.

```
try {
    InitialContext ctx = new InitialContext();
    queue = (Queue) ctx.lookup("queue/MyQueue");
    QueueConnectionFactory factory =
        (QueueConnectionFactory) ctx.lookup("ConnectionFactory");
    cnn = factory.createQueueConnection();
    sess = cnn.createQueueSession(false,
            QueueSession.AUTO_ACKNOWLEDGE);

} catch (Exception e) {
    e.printStackTrace ();
}

TextMessage msg = sess.createTextMessage(...);
```

```
sender = sess.createSender(queue);
sender.send(msg);
```

Please refer to the JMS specification or books to learn how to program in the JMS API.

4. Package and Deploy EJB3 Services

EJB3 bean classes are packaged in regular JAR files. The standard configuration files, such as ejb-jar.xml for session beans, and persistence.xml for entity beans, are in the META-INF directory inside the JAR. You can deploy EJB3 beans as standalone services in JBoss AS or as part of an enterprise application (i.e., in an EAR archive). In this section, we discuss those two deployment options.

4.1. Deploy the EJB3 JAR

When you drop JAR files into the `jboss-as/server/production/deploy/` directory, it will be automatically picked up and processed by the server. All the EJB3 beans defined in the JAR file will then be available to other applications deployed inside or outside of the server via JNDI names like `MyBean/local`, where `MyBean` is the implementation class name for the session bean. The deployment is done via the JBoss EJB3 deployer in jboss-as/server/production/ejb3.deployer/. The META-INF/persistence.properties file we discussed earlier to configure the default behavior of EJB3 entity manager is located in the EJB3 deployer.

The EJB3 deployer automatically scans JARs on the classpath to look for EJB3 annotations. When it finds classes with EJB3 annotations, it would deploy them as EJB3 services. However, scanning all JARs on the classpath could be very time-consuming if you have large applications with many JARs deployed. In the jboss-as/server/production/ejb3.deployer/META-INF/jboss-service.xml file, you can tell the EJB3 deployer to ignore JARs you know do not contain EJB3 beans. The non-EJB3 JAR files shipped with the JBoss AS are already listed in the jboss.ejb3:service=JarsIgnoredForScanning MBean service:

```
... ...
<mbean code="org.jboss.ejb3.JarsIgnoredForScanning"
       name="jboss.ejb3:service=JarsIgnoredForScanning">
    <attribute name="IgnoredJars">
       snmp-adaptor.jar,
       otherimages.jar,
       applet.jar,
       jcommon.jar,
       console-mgr-classes.jar,
       jfreechart.jar,
       juddi-service.jar,
```

```
         wsdl4j.jar,
         ... ...
         servlets-webdav.jar
      </attribute>
   </mbean>
  ... ...
```

You can add any non-EJB3 JARs from your application to this list so that the server do not have to waste time scanning them. This could significantly improve the application startup time in some cases.

4.2. Deploy EAR with EJB3 JAR

Most Java EE applications are deployed as EAR archives. An EAR archive is a JAR file that typically contains a WAR archive for the web pages, servlets, and other web-related components, one or several EJB3 JARs that provide services (e.g., data access and transaction) to the WAR components, and some other support library JARs required by the application. An EAR file also have deployment descriptors such as application.xml and jboss-app.xml. Below is the basic structure of a typical EAR application.

```
myapp.ear
|+ META-INF
   |+ applications.xml and jboss-app.xml
|+ myapp.war
   |+ web pages and JSP /JSF pages
   |+ WEB-INF
      |+ web.xml, jboss-web.xml, faces-config.xml etc.
      |+ lib
         |+ tag library JARs
      |+ classes
         |+ servlets and other classes used by web pages
|+ myapp.jar
   |+ EJB3 bean classes
   |+ META-INF
      |+ ejb-jar.xml and persistence.xml
|+ lib
   |+ Library JARs for the EAR
```

Notice that in JBoss AS, unlike in many other application servers, you do not need to declare EJB references in the web.xml file in order for the components in the WAR file to access EJB3 services. You can obtain the references directly via JNDI as we discussed earlier in the chapter.

A typical application.xml file is as follows. It declares the WAR and EJB3 JAR archives in the EAR, and defines the web content root for the application. Of course, you can have multiple

EJB3 modules in the same EAR application. The application.xml file could also optionally define a shared classpath for JAR files used in this application. The JAR file location defaults to lib in JBoss AS -- but it might be different in other application servers.

```
<application>
  <display-name>My Application</display-name>

  <module>
    <web>
      <web-uri>myapp.war</web-uri>
      <context-root>/myapp</context-root>
    </web>
  </module>

  <module>
    <ejb>myapp.jar</ejb>
  </module>

  <library-directory>lib</library-directory>

</application>
```

The jboss-app.xml file provides JBoss-specific deployment configuration for the EAR application. For instance, it can specify the deployment order of modules in the EAR, deploy JBoss-specific application modules in the EAR, such as SARs (Service ARchive for MBeans) and HARs (Hibernate ARchive for Hibernate objects), provide security domain and JMX MBeans that can be used with this application, etc. You can lear more about the possible attributes in jboss-app.xml in its DTD: http://www.jboss.org/j2ee/dtd/jboss-app_4_2.dtd.

A common use case for jboss-app.xml is to configure whether this EAR file should be deployed in its own scoped classloader to avoid naming conflicts with other applications. If your EAR application is deployed in its own scoped classloader and it only has one persistence-unit defined in its EJB3 JARs, you will be able to use @PersistenceContext EntotyManager em to inject EntityManager to session beans without worrying about passing the persistence unit name to the @PersistenceContext annotation. The following jboss-app.xml specifies a scoped classloader myapp:archive=myapp.ear for the EAR application.

```
<jboss-app>
      <loader-repository>
      myapp:archive=myapp.ear
      </loader-repository>
</jboss-app>
```

The EAR deployment is configured by the jboss-as/server/production/deploy/ear-deploy.xml file. This file contains three attributes as follows.

```
<server>
   <mbean code="org.jboss.deployment.EARDeployer"
          name="jboss.j2ee:service=EARDeployer">
      <!--
          A flag indicating if ear deployments should
           have their own scoped class loader to isolate
           their classes from other deployments.
      -->
      <attribute name="Isolated">false</attribute>

      <!--
          A flag indicating if the ear components should
          have in VM call optimization disabled.
      -->
      <attribute name="CallByValue">false</attribute>

      <!--
          A flag the enables the default behavior of
          the ee5 library-directory. If true, the lib
          contents of an ear are assumed to be the default
          value for library-directory in the absence of
          an explicit library-directory. If false, there
          must be an explicit library-directory.
      -->
      <attribute name="EnablelibDirectoryByDefault">true</attribute>
   </mbean>
</server>
```

If you set the Isolated parameter to true, all EAR deployment will have scoped classloaders by default. There will be no need to define the classloader in jboss-app.xml. The CallByValue attribute specifies whether we should treat all EJB calls as remote calls. Remote calls have a large additional performance penalty compared with local call-by-reference calls, because objects involved in remote calls have to be serialized and de-serialized. For most of our applications, the WAR and EJB3 JARs are deployed on the same server, hence this value should be default to false and the server uses local call-by-reference calls to invoke EJB methods in the same JVM. The EnablelibDirectoryByDefault attribute specifies whether the lib directory in the EAR archive should be the default location for shared library JARs.

Deployment

Deploying applications on JBoss AS is very easy. You just need to copy the application into the jboss-as/server/production/deploy directory. You can replace default with different server profiles such as all or minimal or production. We will cover those later in this chapter. JBoss AS constantly scans the deploy directory to pick up new applications or any changes to existing applications. So, you can "hot deploy" application on the fly while JBoss AS is still running.

1. Deployable Application Types

You can deploy several different types of enterprise applications in JBoss AS:

- The WAR application archive (e.g., myapp.war) packages a Java EE web application in a JAR file. It contains servlet classes, view pages, libraries, and deployment descriptors such as web.xml, faces-config.xml, and jboss-web.xml etc..
- The EAR application archive (e.g., myapp.ear) packages a Java EE enterprise application in a JAR file. It typically contains a WAR file for the web module, JAR files for EJB modules, as well as deployment descriptors such as application.xml and jboss-app.xml etc..
- The SAR application archive (e.g., myservice.sar) packages a JBoss service in a JAR file. It is mostly used by JBoss internal services. Please see more in *Chapter 3, The JBoss JMX Microkernel*.
- The *-ds.xml file defines connections to external databases. The data source can then be reused by all applications and services in JBoss AS via the internal JNDI.
- You can deploy XML files with MBean service definitions. If you have the appropriate JAR files available in the deploy or lib directories, the MBeans specified in the XML files will be started. This is the way how you start many JBoss AS internal services, such as the JMS queues.
- You can also deploy JAR files containing EJBs or other service objects directly in JBoss AS.

Exploded Deployment

The WAR, EAR, and SAR deployment packages are really just JAR files with special XML deployment descriptors in directories like META-INF and WEB-INF. JBoss AS allows you to deploy those archives as expanded directories instead of JAR files. That allows you to make changes to web pages etc on the fly without re-deploying the entire application. If you do need to re-deploy the exploded directory without re-start the server, you can just "touch" the deployment descriptors (e.g., the WEB-INF/web.xml in a WAR and the META-INF/application.xml in an EAR) to update their timestamps.

2. Standard Server Configurations

The JBoss Enterprise Platform ships with four server configurations. You can choose which configuration to start by passing the -c parameter to the server startup script. For instance, command run.sh -c all would start the server in the all configuration. Each configuration is contained in a directory named jboss-as/server/[config name]/. You can look into each server configuration's directory to see the default services, applications, and libraries supported in the configuration.

- The minimal configuration starts the core server container without any of the enterprise services. It is a good starting point if you want to build a customized version of JBoss AS that only contains the servers you need.
- The default configuration is the mostly common used configuration for application developers. It supports the standard J2EE 1.4 and most of the Java EE 5.0 programming APIs (e.g., JSF and EJB3).
- The all configuration is the default configuration with clustering support and other enterprise extensions.
- The production configuration is based on the all configuration but with key parameters pre-tuned for production deployment.

The detailed services and APIs supported in each of those configurations will be discussed throughout this book. In this section, we focus on the optimization we did for the production configuration.

2.1. The production Configuration

To start the server in the production configuration, you can use the following command under Linux / Unix:

```
cd /path/to/jboss-as
RUN_CONF=server/production/run.conf bin/run.sh -c production
```

Or, you can simply copy the jboss-as/server/production/run.conf file to jboss-as/bin directory and start the server with run.sh -c production command. Below is a list of optimizations we specifically did for the production configuration:

- In the jboss-as/server/production/run.conf file, we expanded the memory size of the server to 1.7 GB. We added the -server tag to JVM startup command on all platforms except for Darwin (Mac OS X). If the JVM is BEA jRockit, the -Xgc:gencon parameter is also added.
- We configured the key generation algorithm to use the database to generate HiLo keys in

order to generate the correct keys in a cluster environment (see deploy/uuid-key-generator.sar/META-INF/jboss-service.xml).

- We removed the test JMS queues from deploy-hasingleton/jms/jbossmq-destinations-service.xml. Those queues are setup primarily for ease of application development. Production applications should configure their own JMS queues.
- We set the ScanPeriod parameter to 60000 in conf/jboss-minimal.xml and conf/jboss-service.xml, so that JBoss AS does not spend too much time constantly scanning the deploy directory for new or updated deployments.
- We removed the connection monitoring in deploy/jbossjca-service.xml. The connection monitoring feature helps catch unclosed connections that would otherwise cause leaks in the connection pools in development. However, it is a global point of contention that should be turned off (false) in production.
- Logging is a big contention point in many production applications. In the production configuration, we removed the console logging and increased the logging level to WARN and ERROR for most packages. Please see details in conf/jboss-log4j.xml.

2.2. Further Tuning from the production Configuration

In addition to the standard optimization in the production configuration, there are a couple of simple techniques you can use to improve the performance and stability of your server.

The production configuration increases the JVM heap memory size to 1.7 GB. You should probably change it to fit your own server. For instance, if have a 64 bit server with several GBs of RAM, you can probably increase this value as long as you also use a 64 bit JVM. If your server has less than 2 GB RAM, you should decrease that value accordingly. In the production/run.conf file, the -Xmx and -Xms parameters specify the maximum and minimum heap sizes respectively. It is recommended that you set the -Xmx and -Xms to the same value to avoid dynamic re-sizing of the heap, which is a source of instability in many JVMs. You could also consider turing on parallel GC options if you are using the Sun JVM on a multi-core machine. The following is an example setup you might use a reference. Please see the Sun JVM documentation for more details on this startup parameters.

```
JAVA_OPTS="-Xms1740m -Xmx1740m -XX:PermSize=256m -XX:MaxPermSize=512
  -XX:+UseConcMarkSweepGC -XX:+CMSPermGenSweepingEnabled
  -XX:+CMSClassUnloadingEnabled"
```

In the embedded Tomcat module, you can turn off the development mode so that the server does not constantly monitor the changes in JSP files. To do that, edit the deploy/jboss-web.deployer/conf/web.xml file and add the development attribute to the JspServlet.

```
<servlet>
  <servlet-name>jsp</servlet-name>
  <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
  ... ...
  <init-param>
    <param-name>development</param-name>
    <param-value>false</param-value>
  </init-param>
  ... ...
```

In Tomcat, you could adjust the size of the thread pool. If you have multi-core CPUs or more than one CPUs on your server, it might be beneficial to increase the thread pool beyond the default 250. On the other hand, if you have a slow server, decreasing the thread pool will decrease the overhead on the server. The thread pool size can be adjusted via the deploy/jboss-web.deployer/server.xml file.

```
... ...
<Connector port="8080" address="${jboss.bind.address}"
       maxThreads="250" maxHttpHeaderSize="8192"
       emptySessionPath="true" protocol="HTTP/1.1"
       enableLookups="false" redirectPort="8443" acceptCount="100"
       connectionTimeout="20000" disableUploadTimeout="true" />
... ...
```

In addition, JBoss AS needs to use a relational database to store runtime data. In a production environment, you should use a production quality database to replace the embedded HSQL database. Please see *Appendix B, Use Alternative Databases with JBoss AS* for more information on how to setup alternative databases for the JBoss AS.

Part II. JBoss AS Infrastructure

The JBoss JMX Microkernel

Modularly developed from the ground up, the JBoss server and container are completely implemented using component-based plug-ins. The modularization effort is supported by the use of JMX, the Java Management Extension API. Using JMX, industry-standard interfaces help manage both JBoss/Server components and the applications deployed on it. Ease of use is still the number one priority, and the JBoss Server architecture sets a new standard for modular, plug-in design as well as ease of server and application management.

This high degree of modularity benefits the application developer in several ways. The already tight code can be further trimmed down to support applications that must have a small footprint. For example, if EJB passivation is unnecessary in your application, simply take the feature out of the server. If you later decide to deploy the same application under an Application Service Provider (ASP) model, simply enable the server's passivation feature for that web-based deployment. Another example is the freedom you have to drop your favorite object to relational database (O-R) mapping tool, such as TOPLink, directly into the container.

This chapter will introduce you to JMX and its role as the JBoss server component bus. You will also be introduced to the JBoss MBean service notion that adds life cycle operations to the basic JMX management component.

1. An Introduction to JMX

The success of the full Open Source J2EE stack lies with the use of JMX (Java Management Extension). JMX is the best tool for integration of software. It prov ides a common spine that allows the user to integrate modules, containers, and plug-ins. *Figure 3.1, “The JBoss JMX integration bus and the standard JBoss components”* shows the role of JMX as an integration spine or bus into which components plug. Components are declared as MBean services that are then loaded into JBoss. The components may subsequently be administered using JMX.

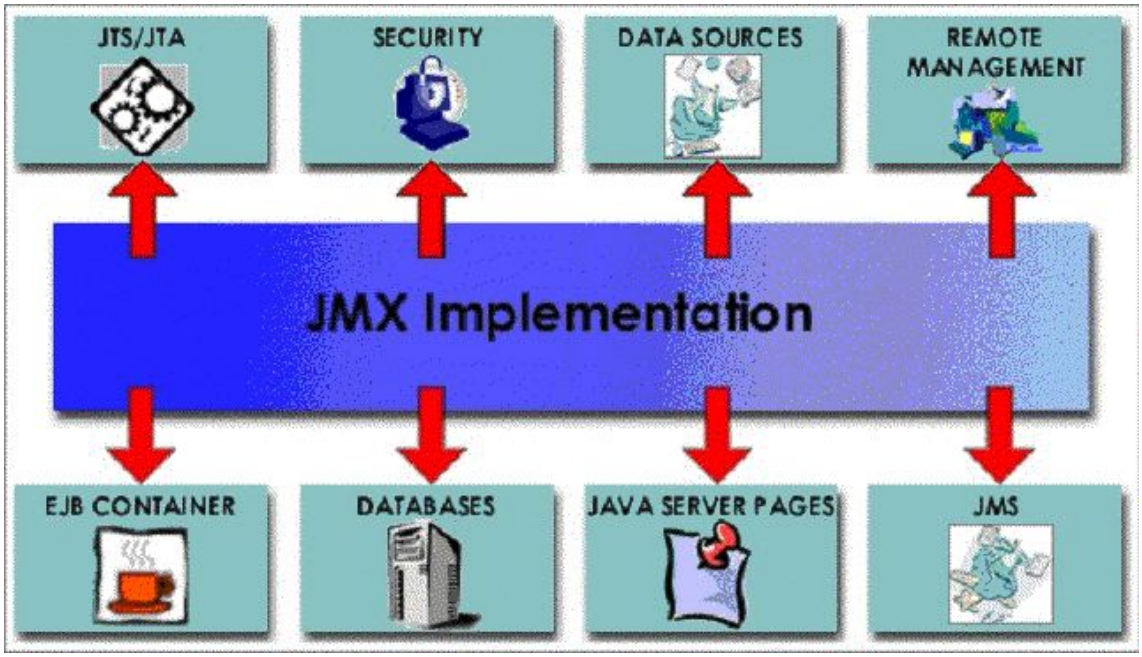


Figure 3.1. The JBoss JMX integration bus and the standard JBoss components

Before looking at how JBoss uses JMX as its component bus, it would help to get a basic overview what JMX is by touching on some of its key aspects.

JMX components are defined by the Java Management Extensions Instrumentation and Agent Specification, v1.2, which is available from the JSR003 Web page at *http://jcp.org/en/jsr/detail?id=3*. The material in this JMX overview section is derived from the JMX instrumentation specification, with a focus on the aspects most used by JBoss. A more comprehensive discussion of JMX and its application can be found in *JMX: Managing J2EE with Java Management Extensions* written by Juha Lindfors (Sams, 2002).

JMX is a standard for managing and monitoring all varieties of software and hardware components from Java. Further, JMX aims to provide integration with the large number of existing management standards. *Figure 3.2, "The Relationship between the components of the JMX architecture"* shows examples of components found in a JMX environment, and illustrates the relationship between them as well as how they relate to the three levels of the JMX model. The three levels are:

- **Instrumentation**, which are the resources to manage
- **Agents**, which are the controllers of the instrumentation level objects
- **Distributed services**, the mechanism by which administration applications interact with agents and their managed objects

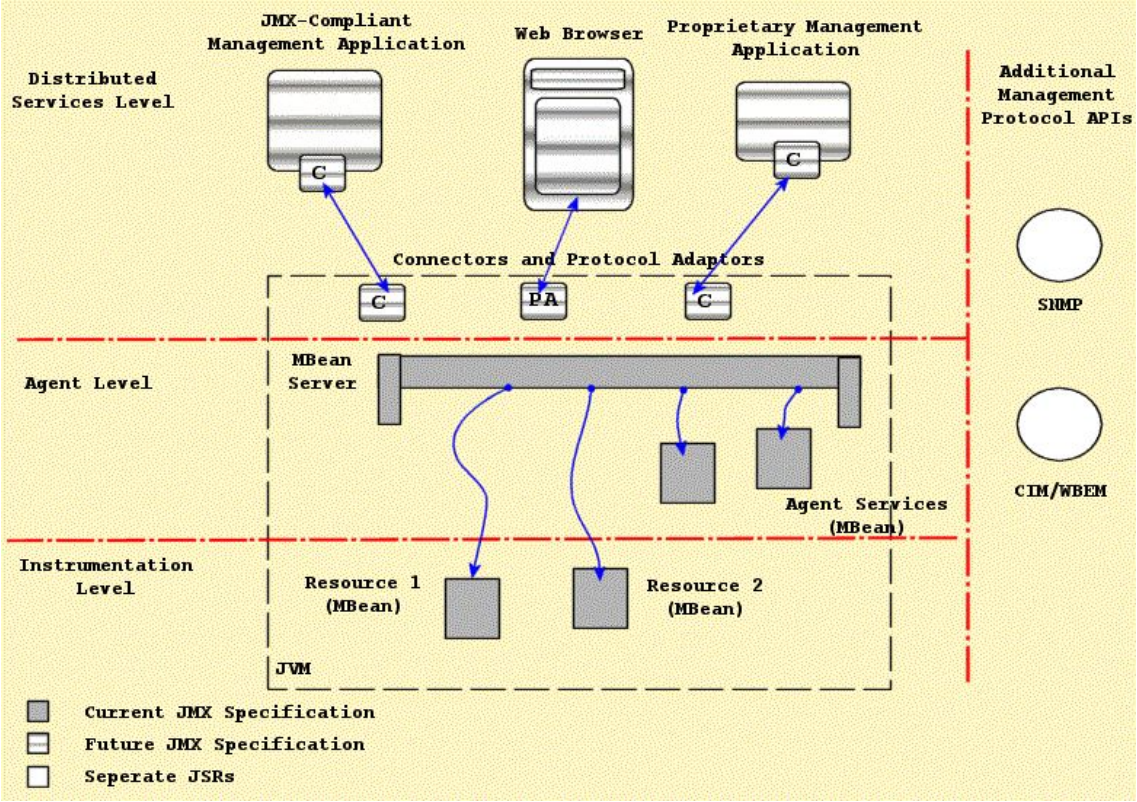


Figure 3.2. The Relationship between the components of the JMX architecture

1.1. Instrumentation Level

The instrumentation level defines the requirements for implementing JMX manageable resources. A JMX manageable resource can be virtually anything, including applications, service components, devices, and so on. The manageable resource exposes a Java object or wrapper that describes its manageable features, which makes the resource instrumented so that it can be managed by JMX-compliant applications.

The user provides the instrumentation of a given resource using one or more managed beans, or MBeans. There are four varieties of MBean implementations: standard, dynamic, model, and open. The differences between the various MBean types is discussed in Managed Beans or MBeans.

The instrumentation level also specifies a notification mechanism. The purpose of the notification mechanism is to allow MBeans to communicate changes with their environment. This is similar to the JavaBean property change notification mechanism, and can be used for attribute change notifications, state change notifications, and so on.

1.2. Agent Level

The agent level defines the requirements for implementing agents. Agents are responsible for controlling and exposing the managed resources that are registered with the agent. By default, management agents are located on the same hosts as their resources. This collocation is not a requirement.

The agent requirements make use of the instrumentation level to define a standard MBeanServer management agent, supporting services, and a communications connector. JBoss provides both an html adaptor as well as an RMI adaptor.

The JMX agent can be located in the hardware that hosts the JMX manageable resources when a Java Virtual Machine (JVM) is available. This is how the JBoss server uses the MBeanServer. A JMX agent does not need to know which resources it will serve. JMX manageable resources may use any JMX agent that offers the services it requires.

Managers interact with an agent's MBeans through a protocol adaptor or connector, as described in the *Section 1.3, "Distributed Services Level"* in the next section. The agent does not need to know anything about the connectors or management applications that interact with the agent and its MBeans.

1.3. Distributed Services Level

The JMX specification notes that a complete definition of the distributed services level is beyond the scope of the initial version of the JMX specification. This was indicated by the component boxes with the horizontal lines in *Figure 3.2, "The Relationship between the components of the JMX architecture"*. The general purpose of this level is to define the interfaces required for implementing JMX management applications or managers. The following points highlight the intended functionality of the distributed services level as discussed in the current JMX specification.

- Provide an interface for management applications to interact transparently with an agent and its JMX manageable resources through a connector
- Exposes a management view of a JMX agent and its MBeans by mapping their semantic meaning into the constructs of a data-rich protocol (for example HTML or SNMP)
- Distributes management information from high-level management platforms to numerous JMX agents
- Consolidates management information coming from numerous JMX agents into logical views that are relevant to the end user's business operations
- Provides security

It is intended that the distributed services level components will allow for cooperative management of networks of agents and their resources. These components can be expanded to provide a complete management application.

1.4. JMX Component Overview

This section offers an overview of the instrumentation and agent level components. The instrumentation level components include the following:

- MBeans (standard, dynamic, open, and model MBeans)
- Notification model elements
- MBean metadata classes

The agent level components include:

- MBean server
- Agent services

1.4.1. Managed Beans or MBeans

An MBean is a Java object that implements one of the standard MBean interfaces and follows the associated design patterns. The MBean for a resource exposes all necessary information and operations that a management application needs to control the resource.

The scope of the management interface of an MBean includes the following:

- Attribute values that may be accessed by name
- Operations or functions that may be invoked
- Notifications or events that may be emitted
- The constructors for the MBean's Java class

JMX defines four types of MBeans to support different instrumentation needs:

- **Standard MBeans**: These use a simple JavaBean style naming convention and a statically defined management interface. This is the most common type of MBean used by JBoss.
- **Dynamic MBeans**: These must implement the `javax.management.DynamicMBean` interface, and they expose their management interface at runtime when the component is instantiated for the greatest flexibility. JBoss makes use of Dynamic MBeans in circumstances where the components to be managed are not known until runtime.
- **Open MBeans**: These are an extension of dynamic MBeans. Open MBeans rely on basic, self-describing, user-friendly data types for universal manageability.
- **Model MBeans**: These are also an extension of dynamic MBeans. Model MBeans must implement the `javax.management.modelmbean.ModelMBean` interface. Model MBeans simplify the instrumentation of resources by providing default behavior. JBoss XMBeans are an implementation of Model MBeans.

We will present an example of a Standard and a Model MBean in the section that discusses

extending JBoss with your own custom services.

1.4.2. Notification Model

JMX Notifications are an extension of the Java event model. Both the MBean server and MBeans can send notifications to provide information. The JMX specification defines the `javax.management` package `Notification` event object, `NotificationBroadcaster` event sender, and `NotificationListener` event receiver interfaces. The specification also defines the operations on the MBean server that allow for the registration of notification listeners.

1.4.3. MBean Metadata Classes

There is a collection of metadata classes that describe the management interface of an MBean. Users can obtain a common metadata view of any of the four MBean types by querying the MBean server with which the MBeans are registered. The metadata classes cover an MBean's attributes, operations, notifications, and constructors. For each of these, the metadata includes a name, a description, and its particular characteristics. For example, one characteristic of an attribute is whether it is readable, writable, or both. The metadata for an operation contains the signature of its parameter and return types.

The different types of MBeans extend the metadata classes to be able to provide additional information as required. This common inheritance makes the standard information available regardless of the type of MBean. A management application that knows how to access the extended information of a particular type of MBean is able to do so.

1.4.4. MBean Server

A key component of the agent level is the managed bean server. Its functionality is exposed through an instance of the `javax.management.MBeanServer`. An MBean server is a registry for MBeans that makes the MBean management interface available for use by management applications. The MBean never directly exposes the MBean object itself; rather, its management interface is exposed through metadata and operations available in the MBean server interface. This provides a loose coupling between management applications and the MBeans they manage.

MBeans can be instantiated and registered with the MBeanServer by the following:

- Another MBean
- The agent itself
- A remote management application (through the distributed services)

When you register an MBean, you must assign it a unique object name. The object name then becomes the unique handle by which management applications identify the object on which to perform management operations. The operations available on MBeans through the MBean server include the following:

- Discovering the management interface of MBeans

- Reading and writing attribute values
- Invoking operations defined by MBeans
- Registering for notifications events
- Querying MBeans based on their object name or their attribute values

Protocol adaptors and connectors are required to access the MBeanServer from outside the agent's JVM. Each adaptor provides a view via its protocol of all MBeans registered in the MBean server the adaptor connects to. An example adaptor is an HTML adaptor that allows for the inspection and editing of MBeans using a Web browser. As was indicated in *Figure 3.2, "The Relationship between the components of the JMX architecture"*, there are no protocol adaptors defined by the current JMX specification. Later versions of the specification will address the need for remote access protocols in standard ways.

A connector is an interface used by management applications to provide a common API for accessing the MBean server in a manner that is independent of the underlying communication protocol. Each connector type provides the same remote interface over a different protocol. This allows a remote management application to connect to an agent transparently through the network, regardless of the protocol. The specification of the remote management interface will be addressed in a future version of the JMX specification.

Adaptors and connectors make all MBean server operations available to a remote management application. For an agent to be manageable from outside of its JVM, it must include at least one protocol adaptor or connector. JBoss currently includes a custom HTML adaptor implementation and a custom JBoss RMI adaptor.

1.4.5. Agent Services

The JMX agent services are objects that support standard operations on the MBeans registered in the MBean server. The inclusion of supporting management services helps you build more powerful management solutions. Agent services are often themselves MBeans, which allow the agent and their functionality to be controlled through the MBean server. The JMX specification defines the following agent services:

- **A dynamic class loading MLet (management applet) service**: This allows for the retrieval and instantiation of new classes and native libraries from an arbitrary network location.
- **Monitor services**: These observe an MBean attribute's numerical or string value, and can notify other objects of several types of changes in the target.
- **Timer services**: These provide a scheduling mechanism based on a one-time alarm-clock notification or on a repeated, periodic notification.
- **The relation service**: This service defines associations between MBeans and enforces consistency on the relationships.

Any JMX-compliant implementation will provide all of these agent services. However, JBoss does not rely on any of these standard agent services.

2. JBoss JMX Implementation Architecture

2.1. The JBoss ClassLoader Architecture

JBoss employs a class loading architecture that facilitates sharing of classes across deployment units and hot deployment of services and applications. Before discussing the JBoss specific class loading model, we need to understand the nature of Java's type system and how class loaders fit in.

2.2. Class Loading and Types in Java

Class loading is a fundamental part of all server architectures. Arbitrary services and their supporting classes must be loaded into the server framework. This can be problematic due to the strongly typed nature of Java. Most developers know that the type of a class in Java is a function of the fully qualified name of the class. However the type is also a function of the `java.lang.ClassLoader` that is used to define that class. This additional qualification of type is necessary to ensure that environments in which classes may be loaded from arbitrary locations would be type-safe.

However, in a dynamic environment like an application server, and especially JBoss with its support for hot deployment are that class cast exceptions, linkage errors and illegal access errors can show up in ways not seen in more static class loading contexts. Let's take a look at the meaning of each of these exceptions and how they can happen.

2.2.1. ClassCastExceptions - I'm Not Your Type

A `java.lang.ClassCastException` results whenever an attempt is made to cast an instance to an incompatible type. A simple example is trying to obtain a `String` from a `List` into which a `URL` was placed:

```
ArrayList array = new ArrayList();
array.add(new URL("file:/tmp"));
String url = (String) array.get(0);

java.lang.ClassCastException: java.net.URL
at org.jboss.book.jmx.ex0.ExCCEa.main(Ex1CCE.java:16)
```

The `ClassCastException` tells you that the attempt to cast the array element to a `String` failed because the actual type was `URL`. This trivial case is not what we are interested in however. Consider the case of a JAR being loaded by different class loaders. Although the classes loaded through each class loader are identical in terms of the bytecode, they are completely different types as viewed by the Java type system. An example of this is illustrated by the code shown in *Example 3.1, “The ExCCEc class used to demonstrate ClassCastException due to duplicate class loaders”*.

```
package org.jboss.book.jmx.ex0;
```

```
import java.io.File;
import java.net.URL;
import java.net.URLClassLoader;
import java.lang.reflect.Method;

import org.apache.log4j.Logger;

import org.jboss.util.ChapterExRepository;
import org.jboss.util.Debug;

/**
 * An example of a ClassCastException that
 * results from classes loaded through
 * different class loaders.
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.9 $
 */
public class ExCCEc
{
    public static void main(String[] args) throws Exception
    {
        ChapterExRepository.init(ExCCEc.class);

        String chapDir = System.getProperty("j2eechapter.dir");
        Logger ucl0Log = Logger.getLogger("UCL0");
        File jar0 = new File(chapDir+"/j0.jar");
        ucl0Log.info("jar0 path: "+jar0.toString());
        URL[] cp0 = {jar0.toURL()};
        URLClassLoader ucl0 = new URLClassLoader(cp0);
        Thread.currentThread().setContextClassLoader(ucl0);
        Class objClass = ucl0.loadClass("org.jboss.book.jmx.ex0.ExObj");
        StringBuffer buffer = new
            StringBuffer("ExObj Info");
        Debug.displayClassInfo(objClass, buffer, false);
        ucl0Log.info(buffer.toString());
        Object value = objClass.newInstance();

        File jar1 = new File(chapDir+"/j0.jar");
        Logger ucl1Log = Logger.getLogger("UCL1");
        ucl1Log.info("jar1 path: "+jar1.toString());
        URL[] cp1 = {jar1.toURL()};
        URLClassLoader ucl1 = new URLClassLoader(cp1);
        Thread.currentThread().setContextClassLoader(ucl1);
        Class ctxClass2 = ucl1.loadClass("org.jboss.book.jmx.ex0.ExCtx");
        buffer.setLength(0);
        buffer.append("ExCtx Info");
        Debug.displayClassInfo(ctxClass2, buffer, false);
        ucl1Log.info(buffer.toString());
        Object ctx2 = ctxClass2.newInstance();

        try {
            Class[] types = {Object.class};
            Method useValue =
                ctxClass2.getMethod("useValue", types);
            Object[] margs = {value};
            useValue.invoke(ctx2, margs);
```

```
        } catch(Exception e) {
            ucl1Log.error("Failed to invoke ExCtx.useValue", e);
            throw e;
        }
    }
}
```

Example 3.1. The ExCCEc class used to demonstrate ClassCastException due to duplicate class loaders

```
package org.jboss.book.jmx.ex0;

import java.io.IOException;
import org.apache.log4j.Logger;
import org.jboss.util.Debug;

/**
 * A classes used to demonstrate various class
 * loading issues
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.9 $
 */
public class ExCtx
{
    ExObj value;

    public ExCtx()
        throws IOException
    {
        value = new ExObj();
        Logger log = Logger.getLogger(ExCtx.class);
        StringBuffer buffer = new StringBuffer("ctor.ExObj");
        Debug.displayClassInfo(value.getClass(), buffer, false);
        log.info(buffer.toString());
        ExObj2 obj2 = value.ivar;
        buffer.setLength(0);
        buffer = new StringBuffer("ctor.ExObj.ivar");
        Debug.displayClassInfo(obj2.getClass(), buffer, false);
        log.info(buffer.toString());
    }

    public Object getValue()
    {
        return value;
    }

    public void useValue(Object obj)
        throws Exception
    {
        Logger log = Logger.getLogger(ExCtx.class);
        StringBuffer buffer = new
```

```
            StringBuffer("useValue2.arg class");
        Debug.displayClassInfo(obj.getClass(), buffer, false);
        log.info(buffer.toString());
        buffer.setLength(0);
        buffer.append("useValue2.ExObj class");
        Debug.displayClassInfo(ExObj.class, buffer, false);
        log.info(buffer.toString());
        ExObj ex = (ExObj) obj;
    }

    void pkgUseValue(Object obj)
        throws Exception
    {
        Logger log = Logger.getLogger(ExCtx.class);
        log.info("In pkgUseValue");
    }
}
```

```
package org.jboss.book.jmx.ex0;

import java.io.Serializable;

/**
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.9 $
 */
public class ExObj
    implements Serializable
{
    public ExObj2 ivar = new ExObj2();
}
```

```
package org.jboss.book.jmx.ex0;

import java.io.Serializable;

/**
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.9 $
 */
public class ExObj2
    implements Serializable
{
}
```

Example 3.2. The ExCtx, ExObj, and ExObj2 classes used by the examples

The `ExCCEc.main` method uses reflection to isolate the classes that are being loaded by the class loaders `ucl0` and `ucl1` from the application class loader. Both are setup to load classes

from the `output/jmx/j0.jar`, the contents of which are:

```
[examples]$ jar -tf output/jmx/j0.jar
...
org/jboss/book/jmx/ex0/ExCtx.class
org/jboss/book/jmx/ex0/ExObj.class
org/jboss/book/jmx/ex0/ExObj2.class
```

We will run an example that demonstrates how a class cast exception can occur and then look at the specific issue with the example. See *Appendix A, Book Example Installation* for instructions on installing the examples accompanying the book, and then run the example from within the examples directory using the following command:

```
[examples]$ ant -Dchap=jmx -Dex=0c run-example
...
     [java] java.lang.reflect.InvocationTargetException
     [java]     at sun.reflect.NativeMethodAccessorImpl.invoke0(Native
Method)
     [java]     at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
     [java]     at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl
                .java:25)
     [java]     at java.lang.reflect.Method.invoke(Method.java:585)
     [java]     at org.jboss.book.jmx.ex0.ExCCEc.main(ExCCEc.java:58)
     [java] Caused by: java.lang.ClassCastException:
org.jboss.book.jmx.ex0.ExObj
     [java]     at org.jboss.book.jmx.ex0.ExCtx.useValue(ExCtx.java:44)
     [java]     ... 5 more
```

Only the exception is shown here. The full output can be found in the `logs/jmx-ex0c.log` file. At line 55 of `ExCCEc.java` we are invoking `ExcCCECtx.useValue(Object)` on the instance loaded and created in lines 37-48 using `ucl1`. The `ExObj` passed in is the one loaded and created in lines 25-35 via `ucl0`. The exception results when the `ExCtx.useValue` code attempts to cast the argument passed in to a `ExObj`. To understand why this fails consider the debugging output from the `jmx-ex0c.log` file shown in *Example 3.3, "The jmx-ex0c.log debugging output for the ExObj classes seen"*.

```
[INFO,UCL0] ExObj Info
org.jboss.book.jmx.ex0.ExObj(f8968f).ClassLoader=java.net.URLClassLoader@2611a7
..java.net.URLClassLoader@2611a7
....file:/Users/orb/proj/jboss/jboss-docs/jbossas/j2ee/examples/output/jmx/j0.jar
++++CodeSource:
(file:/Users/orb/proj/jboss/jboss-docs/jbossas/j2ee/examples/output/
                jmx/j0.jar <no signer certificates>)
Implemented Interfaces:
++interface java.io.Serializable(41b571)
++++ClassLoader: null
++++Null CodeSource
[INFO,ExCtx] useValue2.ExObj class
org.jboss.book.jmx.ex0.ExObj(bc8e1e).ClassLoader=java.net.URLClassLoader@6bd8ea
```

```
..java.net.URLClassLoader@6bd8ea
....file:/Users/orb/proj/jboss/jboss-docs/jbossas/j2ee/examples/output/jmx/j0.jar
++++CodeSource:
(file:/Users/orb/proj/jboss/jboss-docs/jbossas/j2ee/examples/output/
                jmx/j0.jar <no signer certificates>)
Implemented Interfaces:
++interface java.io.Serializable(41b571)
++++ClassLoader: null
++++Null CodeSource
```

Example 3.3. The jmx-ex0c.log debugging output for the ExObj classes seen

The first output prefixed with `[INFO,UCL0]` shows that the `ExObj` class loaded at line `ExCCEc.java:31` has a hash code of `f8968f` and an associated `URLClassLoader` instance with a hash code of `2611a7`, which corresponds to ucl0. This is the class used to create the instance passed to the `ExCtx.useValue` method. The second output prefixed with `[INFO,ExCtx]` shows that the `ExObj` class as seen in the context of the `ExCtx.useValue` method has a hash code of `bc8e1e` and a `URLClassLoader` instance with an associated hash code of `6bd8ea`, which corresponds to `ucl1`. So even though the `ExObj` classes are the same in terms of actual bytecode since it comes from the same `j0.jar`, the classes are different as seen by both the `ExObj` class hash codes, and the associated `URLClassLoader` instances. Hence, attempting to cast an instance of `ExObj` from one scope to the other results in the `ClassCastException`.

This type of error is common when redeploying an application to which other applications are holding references to classes from the redeployed application. For example, a standalone WAR accessing an EJB. If you are redeploying an application, all dependent applications must flush their class references. Typically this requires that the dependent applications themselves be redeployed.

An alternate means of allowing independent deployments to interact in the presence of redeployment would be to isolate the deployments by configuring the EJB layer to use the standard call-by-value semantics rather than the call-by-reference JBoss will default to for components collocated in the same VM. An example of how to enable call-by-value semantics is presented in *Chapter 14, EJBs on JBoss*

2.2.2. IllegalAccessException - Doing what you should not

A `java.lang.IllegalAccessException` is thrown when one attempts to access a method or member that visibility qualifiers do not allow. Typical examples are attempting to access private or protected methods or instance variables. Another common example is accessing package protected methods or members from a class that appears to be in the correct package, but is really not due to caller and callee classes being loaded by different class loaders. An example of this is illustrated by the code shown in *Example 3.4, "The ExIAEd class used to demonstrate IllegalAccessException due to duplicate class loaders"*.

```
package org.jboss.book.jmx.ex0;

import java.io.File;
import java.net.URL;
import java.net.URLClassLoader;
import java.lang.reflect.Method;

import org.apache.log4j.Logger;

import org.jboss.util.ChapterExRepository;
import org.jboss.util.Debug;

/**
 * An example of IllegalAccessExceptions due to
 * classes loaded by two class loaders.
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.9 $
 */
public class ExIAEd
{
    public static void main(String[] args) throws Exception
    {
        ChapterExRepository.init(ExIAEd.class);

        String chapDir = System.getProperty("j2eechapter.dir");
        Logger ucl0Log = Logger.getLogger("UCL0");
        File jar0 = new File(chapDir+"/j0.jar");
        ucl0Log.info("jar0 path: "+jar0.toString());
        URL[] cp0 = {jar0.toURL()};
        URLClassLoader ucl0 = new URLClassLoader(cp0);
        Thread.currentThread().setContextClassLoader(ucl0);

        StringBuffer buffer = new
            StringBuffer("ExIAEd Info");
        Debug.displayClassInfo(ExIAEd.class, buffer, false);
        ucl0Log.info(buffer.toString());

        Class ctxClass1 = ucl0.loadClass("org.jboss.book.jmx.ex0.ExCtx");
        buffer.setLength(0);
        buffer.append("ExCtx Info");
        Debug.displayClassInfo(ctxClass1, buffer, false);
        ucl0Log.info(buffer.toString());
        Object ctx0 = ctxClass1.newInstance();

        try {
            Class[] types = {Object.class};
            Method useValue =
                ctxClass1.getDeclaredMethod("pkgUseValue", types);
            Object[] margs = {null};
            useValue.invoke(ctx0, margs);
        } catch(Exception e) {
            ucl0Log.error("Failed to invoke ExCtx.pkgUseValue", e);
        }
    }
}
```

Example 3.4. The ExIAEd class used to demonstrate IllegalAccessException due to duplicate class loaders

The `ExIAEd.main` method uses reflection to load the `ExCtx` class via the `ucl0` class loader while the `ExIEAd` class was loaded by the application class loader. We will run this example to demonstrate how the `IllegalAccessException` can occur and then look at the specific issue with the example. Run the example using the following command:

```
[examples]$ ant -Dchap=jmx -Dex=0d run-example
Buildfile: build.xml
...
[java] java.lang.IllegalAccessException: Class org.jboss.book.jmx.ex0.ExIAEd
  can not access a member of class org.jboss.book.jmx.ex0.ExCtx with
modifiers ""
[java]     at sun.reflect.Reflection.ensureMemberAccess(Reflection.java:65)
[java]     at java.lang.reflect.Method.invoke(Method.java:578)
[java]     at org.jboss.book.jmx.ex0.ExIAEd.main(ExIAEd.java:48)
```

The truncated output shown here illustrates the `IllegalAccessException`. The full output can be found in the `logs/jmx-ex0d.log` file. At line 48 of `ExIAEd.java` the `ExCtx.pkgUseValue(Object)` method is invoked via reflection. The `pkgUseValue` method has package protected access and even though both the invoking class `ExIAEd` and the `ExCtx` class whose method is being invoked reside in the `org.jboss.book.jmx.ex0` package, the invocation is seen to be invalid due to the fact that the two classes are loaded by different class loaders. This can be seen by looking at the debugging output from the `jmx-ex0d.log file`.

```
[INFO,UCL0] ExIAEd Info
org.jboss.book.jmx.ex0.ExIAEd(7808b9).ClassLoader=sun.misc.Launcher$AppClassLoader@a9c85c
..sun.misc.Launcher$AppClassLoader@a9c85c
...
[INFO,UCL0] ExCtx Info
org.jboss.book.jmx.ex0.ExCtx(64c34e).ClassLoader=java.net.URLClassLoader@a9c85c
..java.net.URLClassLoader@5d88a
...
```

The ExIAEd class is seen to have been loaded via the default application class loader instance `sun.misc.Launcher$AppClassLoader@a9c85c`, while the `ExCtx` class was loaded by the `java.net.URLClassLoader@a9c85c` instance. Because the classes are loaded by different class loaders, access to the package protected method is seen to be a security violation. So, not only is type a function of both the fully qualified class name and class loader, the package scope is as well.

An example of how this can happen in practice is to include the same classes in two different SAR deployments. If classes in the deployment have a package protected relationship, users of the SAR service may end up loading one class from SAR class loading at one point, and then load another class from the second SAR at a later time. If the two classes in question have a

protected access relationship an `IllegalAccessError` will result. The solution is to either include the classes in a separate jar that is referenced by the SARs, or to combine the SARs into a single deployment. This can either be a single SAR, or an EAR that includes both SARs.

2.2.3. LinkageErrors - Making Sure You Are Who You Say You Are

Loading constraints validate type expectations in the context of class loader scopes to ensure that a class `X` is consistently the same class when multiple class loaders are involved. This is important because Java allows for user defined class loaders. Linkage errors are essentially an extension of the class cast exception that is enforced by the VM when classes are loaded and used.

To understand what loading constraints are and how they ensure type-safety we will first introduce the nomenclature of the Liang and Bracha paper along with an example from this paper. There are two type of class loaders, initiating and defining. An initiating class loader is one that a `ClassLoader.loadClass` method has been invoked on to initiate the loading of the named class. A defining class loader is the loader that calls one of the `ClassLoader.defineClass` methods to convert the class byte code into a `Class` instance. The most complete expression of a class is given by $\langle C, Ld \rangle^{Li}$, where `C` is the fully qualified class name, `Ld` is the defining class loader, and `Li` is the initiating class loader. In a context where the initiating class loader is not important the type may be represented by $\langle C, Ld \rangle$, while when the defining class loader is not important, the type may be represented by C^{Li}. In the latter case, there is still a defining class loader, it's just not important what the identity of the defining class loader is. Also, a type is completely defined by $\langle C, Ld \rangle$. The only time the initiating loader is relevant is when a loading constraint is being validated. Now consider the classes shown in *Example 3.5, "Classes demonstrating the need for loading constraints"*.

```
class <C,L1> {
    void f() {
        <Spoofed, L1>^L1 x = <Delegated, L2>^L2
        x.secret_value = 1; // Should not be allowed
    }
}
```

```
class <Delegated,L2> {
    static <Spoofed, L2>^L3 g() {...}
    }
}
```

```
class <Spoofed, L1> {
    public int secret_value;
}
```

```
class <Spoofed, L2> {
    private int secret_value;
}
```

Example 3.5. Classes demonstrating the need for loading constraints

The class `C` is defined by `L1` and so `L1` is used to initiate loading of the classes `Spoofed` and `Delegated` referenced in the `C.f()` method. The `Spoofed` class is defined by `L1`, but `Delegated` is defined by `L2` because `L1` delegates to `L2`. Since `Delegated` is defined by `L2`, `L2` will be used to initiate loading of `Spoofed` in the context of the `Delegated.g()` method. In this example both `L1` and `L2` define different versions of `Spoofed` as indicated by the two versions shown at the end of *Example 3.5, "Classes demonstrating the need for loading constraints"*. Since `C.f()` believes `x` is an instance of `<Spoofed,L1>` it is able to access the private field `secret_value` of `<Spoofed,L2>` returned by `Delegated.g()` due to the 1.1 and earlier Java VM's failure to take into account that a class type is determined by both the fully qualified name of the class and the defining class loader.

Java addresses this problem by generating loader constraints to validate type consistency when the types being used are coming from different defining class loaders. For the *Example 3.5, "Classes demonstrating the need for loading constraints"* example, the VM generates a constraint $\texttt{Spoofed}^{L1}=\texttt{Spoofed}^{L2}$ when the first line of method `C.f()` is verified to indicate that the type `Spoofed` must be the same regardless of whether the load of `Spoofed` is initiated by `L1` or `L2`. It does not matter if `L1` or `L2`, or even some other class loader defines `Spoofed`. All that matters is that there is only one `Spoofed` class defined regardless of whether `L1` or `L2` was used to initiate the loading. If `L1` or `L2` have already defined separate versions of `Spoofed` when this check is made a `LinkageError` will be generated immediately. Otherwise, the constraint will be recorded and when `Delegated.g()` is executed, any attempt to load a duplicate version of `Spoofed` will result in a `LinkageError`.

Now let's take a look at how a `LinkageError` can occur with a concrete example. *Example 3.6, "A concrete example of a LinkageError"* gives the example main class along with the custom class loader used.

```
package org.jboss.book.jmx.ex0;
import java.io.File;
import java.net.URL;

import org.apache.log4j.Logger;
import org.jboss.util.ChapterExRepository;
import org.jboss.util.Debug;

/**
 * An example of a LinkageError due to classes being defined by more
 * than one class loader in a non-standard class loading environment.
 *
 * @author Scott.Stark@jboss.orgn
 * @version $Revision: 1.9 $
 */
public class ExLE
{
```

```
    public static void main(String[] args)
        throws Exception
    {
        ChapterExRepository.init(ExLE.class);

        String chapDir = System.getProperty("j2eechapter.dir");
        Logger ucl0Log = Logger.getLogger("UCL0");
        File jar0 = new File(chapDir+"/j0.jar");
        ucl0Log.info("jar0 path: "+jar0.toString());
        URL[] cp0 = {jar0.toURL()};
        Ex0URLClassLoader ucl0 = new Ex0URLClassLoader(cp0);
        Thread.currentThread().setContextClassLoader(ucl0);
        Class ctxClass1 = ucl0.loadClass("org.jboss.book.jmx.ex0.ExCtx");
        Class obj2Class1 = ucl0.loadClass("org.jboss.book.jmx.ex0.ExObj2");
        StringBuffer buffer = new StringBuffer("ExCtx Info");
        Debug.displayClassInfo(ctxClass1, buffer, false);
        ucl0Log.info(buffer.toString());
        buffer.setLength(0);
        buffer.append("ExObj2 Info, UCL0");
        Debug.displayClassInfo(obj2Class1, buffer, false);
        ucl0Log.info(buffer.toString());

        File jar1 = new File(chapDir+"/j1.jar");
        Logger ucl1Log = Logger.getLogger("UCL1");
        ucl1Log.info("jar1 path: "+jar1.toString());
        URL[] cp1 = {jar1.toURL()};
        Ex0URLClassLoader ucl1 = new Ex0URLClassLoader(cp1);
        Class obj2Class2 = ucl1.loadClass("org.jboss.book.jmx.ex0.ExObj2");
        buffer.setLength(0);
        buffer.append("ExObj2 Info, UCL1");
        Debug.displayClassInfo(obj2Class2, buffer, false);
        ucl1Log.info(buffer.toString());

        ucl0.setDelegate(ucl1);
        try {
            ucl0Log.info("Try ExCtx.newInstance()");
            Object ctx0 = ctxClass1.newInstance();
            ucl0Log.info("ExCtx.ctor succeeded, ctx0: "+ctx0);
        } catch(Throwable e) {
            ucl0Log.error("ExCtx.ctor failed", e);
        }
    }
}
```

```
package org.jboss.book.jmx.ex0;

import java.net.URLClassLoader;
import java.net.URL;

import org.apache.log4j.Logger;

/**
 * A custom class loader that overrides the standard parent delegation
 * model
```

```
 *
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.9 $
 */
public class Ex0URLClassLoader extends URLClassLoader
{
    private static Logger log = Logger.getLogger(Ex0URLClassLoader.class);
    private Ex0URLClassLoader delegate;

    public Ex0URLClassLoader(URL[] urls)
    {
        super(urls);
    }

    void setDelegate(Ex0URLClassLoader delegate)
    {
        this.delegate = delegate;
    }

    protected synchronized Class loadClass(String name, boolean resolve)
        throws ClassNotFoundException
    {
        Class clazz = null;
        if (delegate != null) {
            log.debug(Integer.toHexString(hashCode()) +
                      "; Asking delegate to loadClass: " + name);
            clazz = delegate.loadClass(name, resolve);
            log.debug(Integer.toHexString(hashCode()) +
                      "; Delegate returned: "+clazz);
        } else {
            log.debug(Integer.toHexString(hashCode()) +
                      "; Asking super to loadClass: "+name);
            clazz = super.loadClass(name, resolve);
            log.debug(Integer.toHexString(hashCode()) +
                      "; Super returned: "+clazz);
        }
        return clazz;
    }

    protected Class findClass(String name)
        throws ClassNotFoundException
    {
        Class clazz = null;
        log.debug(Integer.toHexString(hashCode()) +
                  "; Asking super to findClass: "+name);
        clazz = super.findClass(name);
        log.debug(Integer.toHexString(hashCode()) +
                  "; Super returned: "+clazz);
        return clazz;
    }
}
```

Example 3.6. A concrete example of a LinkageError

The key component in this example is the `URLClassLoader` subclass `Ex0URLClassLoader`. This class loader implementation overrides the default parent delegation model to allow the `ucl0` and `ucl1` instances to both load the `ExObj2` class and then setup a delegation relationship from `ucl0` to `ucl1`. At lines 30 and 31. the `ucl0Ex0URLClassLoader` is used to load the `ExCtx` and `ExObj2` classes. At line 45 of `ExLE.main` the `ucl1Ex0URLClassLoader` is used to load the `ExObj2` class again. At this point both the `ucl0` and `ucl1` class loaders have defined the `ExObj2` class. A delegation relationship from `ucl0` to `ucl1` is then setup at line 51 via the `ucl0.setDelegate(ucl1)` method call. Finally, at line 54 of `ExLE.main` an instance of `ExCtx` is created using the class loaded via `ucl0`. The `ExCtx` class is the same as presented in *Example 3.4, "The ExIAEd class used to demonstrate IllegalAccessException due to duplicate class loaders"*, and the constructor was:

```
public ExCtx()
    throws IOException
{
    value = new ExObj();
    Logger log = Logger.getLogger(ExCtx.class);
    StringBuffer buffer = new StringBuffer("ctor.ExObj");
    Debug.displayClassInfo(value.getClass(), buffer, false);
    log.info(buffer.toString());
    ExObj2 obj2 = value.ivar;
    buffer.setLength(0);
    buffer = new StringBuffer("ctor.ExObj.ivar");
    Debug.displayClassInfo(obj2.getClass(), buffer, false);
    log.info(buffer.toString());
}
```

Now, since the `ExCtx` class was defined by the `ucl0` class loader, and at the time the `ExCtx` constructor is executed, `ucl0` delegates to `ucl1`, line 24 of the `ExCtx` constructor involves the following expression which has been rewritten in terms of the complete type expressions:

<ExObj2,ucl0>^{ucl0} obj2 = <ExObj,ucl1>^{ucl0} value * ivar

This generates a loading constraint of `ExObj2`^{ucl0} = `ExObj2`^{ucl1} since the `ExObj2` type must be consistent across the `ucl0` and `ucl1` class loader instances. Because we have loaded `ExObj2` using both `ucl0` and `ucl1` prior to setting up the delegation relationship, the constraint will be violated and should generate a `LinkageError` when run. Run the example using the following command:

```
[examples]$ ant -Dchap=jmx -Dex=0e run-example
Buildfile: build.xml
...
[java] java.lang.LinkageError: loader constraints violated when linking
           org/jboss/book/jmx/ex0/ExObj2 class
[java]     at org.jboss.book.jmx.ex0.ExCtx.<init>(ExCtx.java:24)
[java]     at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native
Method)
[java]     at
sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessor
           Impl.java:39)
[java]     at
```

```
sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructor
          AccessorImpl.java:27)
[java]     at
java.lang.reflect.Constructor.newInstance(Constructor.java:494)
[java]     at java.lang.Class.newInstance0(Class.java:350)
[java]     at java.lang.Class.newInstance(Class.java:303)
[java]     at org.jboss.book.jmx.ex0.ExLE.main(ExLE.java:53)
```

As expected, a LinkageError is thrown while validating the loader constraints required by line 24 of the `ExCtx` constructor.

2.2.3.1. Debugging Class Loading Issues

Debugging class loading issues comes down to finding out where a class was loaded from. A useful tool for this is the code snippet shown in *Example 3.7, "Obtaining debugging information for a Class"* taken from the org.jboss.util.Debug class of the book examples.

```
Class clazz =...;
StringBuffer results = new StringBuffer();

ClassLoader cl = clazz.getClassLoader();
results.append("\n" + clazz.getName() + "(" +
               Integer.toHexString(clazz.hashCode()) + ").ClassLoader=" +
cl);
ClassLoader parent = cl;

while (parent != null) {
    results.append("\n.."+parent);
    URL[] urls = getClassLoaderURLs(parent);

    int length = urls != null ? urls.length : 0;
    for(int u = 0; u < length; u ++) {
        results.append("\n...."+urls[u]);
    }

    if (showParentClassLoaders == false) {
        break;
    }
    if (parent != null) {
        parent = parent.getParent();
    }
}

CodeSource clazzCS = clazz.getProtectionDomain().getCodeSource();
if (clazzCS != null) {
    results.append("\n++++CodeSource: "+clazzCS);
} else {
    results.append("\n++++Null CodeSource");
}
```

Example 3.7. Obtaining debugging information for a Class

The key items are shown in bold. The first is that every Class object knows its defining `ClassLoader` and this is available via the `getClassLoader()` method. The defines the scope in which the `Class` type is known as we have just seen in the previous sections on class cast exceptions, illegal access exceptions and linkage errors. From the `ClassLoader` you can view the hierarchy of class loaders that make up the parent delegation chain. If the class loader is a `URLClassLoader` you can also see the URLs used for class and resource loading.

The defining `ClassLoader` of a `Class` cannot tell you from what location that `Class` was loaded. To determine this you must obtain the `java.security.ProtectionDomain` and then the `java.security.CodeSource`. It is the `CodeSource` that has the URL p location from which the class originated. Note that not every `Class` has a `CoPdeSource`. If a class is loaded by the bootstrap class loader then its `CodeSource` will be null. This will be the case for all classes in the `java.*` and `javax.*` packages, for example.

Beyond that it may be useful to view the details of classes being loaded into the JBoss server. You can enable verbose logging of the JBoss class loading layer using a Log4j configuration fragment like that shown in *Example 3.8, "An example log4j.xml configuration fragment for enabling verbose class loading logging"*.

```
<appender name="UCL" class="org.apache.log4j.FileAppender">
    <param name="File" value="${jboss.server.home.dir}/log/ucl.log"/>
    <param name="Append" value="false"/>
    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="[%r,%c{1},%t] %m%n"/>
    </layout>
</appender>

<category name="org.jboss.mx.loading" additivity="false">
    <priority value="TRACE" class="org.jboss.logging.XLevel"/>
    <appender-ref ref="UCL"/>
</category>
```

Example 3.8. An example log4j.xml configuration fragment for enabling verbose class loading logging

This places the output from the classes in the `org.jboss.mx.loading` package into the `ucl.log` file of the server configurations log directory. Although it may not be meaningful if you have not looked at the class loading code, it is vital information needed for submitting bug reports or questions regarding class loading problems.

2.2.4. Inside the JBoss Class Loading Architecture

Now that we have the role of class loaders in the Java type system defined, let's take a look at the JBoss class loading architecture. *Figure 3.3, "The core JBoss class loading components"*.

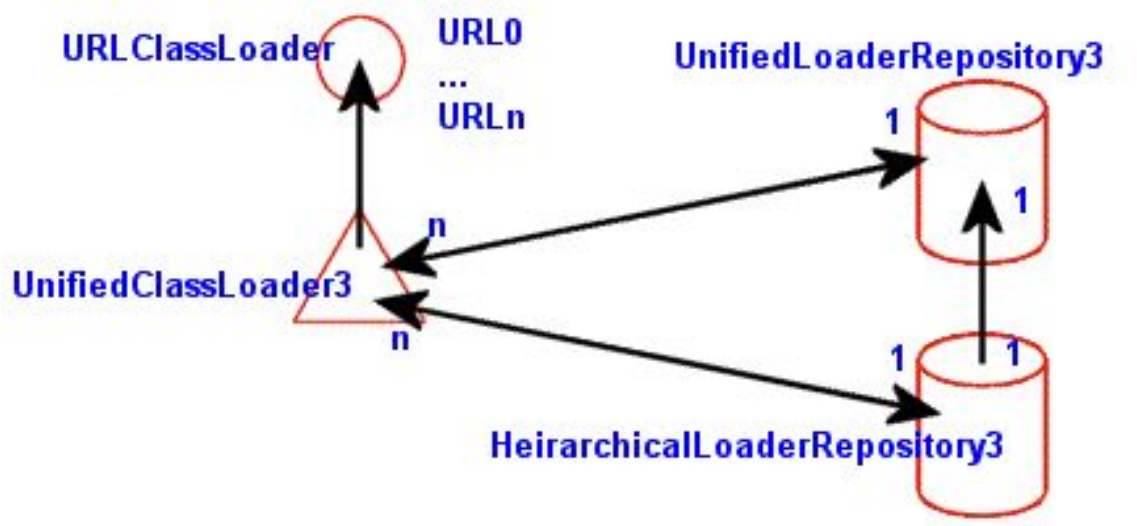


Figure 3.3. The core JBoss class loading components

The central component is the `org.jboss.mx.loading.UnifiedClassLoader3` (UCL) class loader. This is an extension of the standard `java.net.URLClassLoader` that overrides the standard parent delegation model to use a shared repository of classes and resources. This shared repository is the `org.jboss.mx.loading.UnifiedLoaderRepository3`. Every UCL is associated with a single `UnifiedLoaderRepository3`, and a `UnifiedLoaderRepository3` typically has many UCLs. A UCL may have multiple URLs associated with it for class and resource loading. Deployers use the top-level deployment's UCL as a shared class loader and all deployment archives are assigned to this class loader. We will talk about the JBoss deployers and their interaction with the class loading system in more detail later in *Section 4.2, "JBoss MBean Services"*.

When a UCL is asked to load a class, it first looks to the repository cache it is associated with to see if the class has already been loaded. Only if the class does not exist in the repository will it be loaded into the repository by the UCL. By default, there is a single `UnifiedLoaderRepository3` shared across all UCL instances. This means the UCLs form a single flat class loader namespace. The complete sequence of steps that occur when a `UnfiedClassLoader3.loadClass(String, boolean)` method is called is:

1. Check the `UnifiedLoaderRepository3` classes cache associated with the `UnifiedClassLoader3`. If the class is found in the cache it is returned.
2. Else, ask the `UnfiedClassLoader3` if it can load the class. This is essentially a call to the superclass `URLClassLoader.loadClass(String, boolean)` method to see if the class is among the URLs associated with the class loader, or visible to the parent class loader. If the class is found it is placed into the repository classes cache and returned.
3. Else, the repository is queried for all UCLs that are capable of providing the class based on the repository package name to UCL map. When a UCL is added to a repository an

association between the package names available in the URLs associated with the UCL is made, and a mapping from package names to the UCLs with classes in the package is updated. This allows for a quick determination of which UCLs are capable of loading the class. The UCLs are then queried for the requested class in the order in which the UCLs were added to the repository. If a UCL is found that can load the class it is returned, else a `java.lang.ClassNotFoundException` is thrown.

2.2.4.1. Viewing Classes in the Loader Repository

Another useful source of information on classes is the UnifiedLoaderRepository itself. This is an MBean that contains operations to display class and package information. The default repository is located under a standard JMX name of `JMImplementation:name=Default,service=LoaderRepository`, and its MBean can be accessed via the JMX console by following its link from the front page. The JMX console view of this MBean is shown in *Figure 3.4, "The default class LoaderRepository MBean view in the JMX console"*.

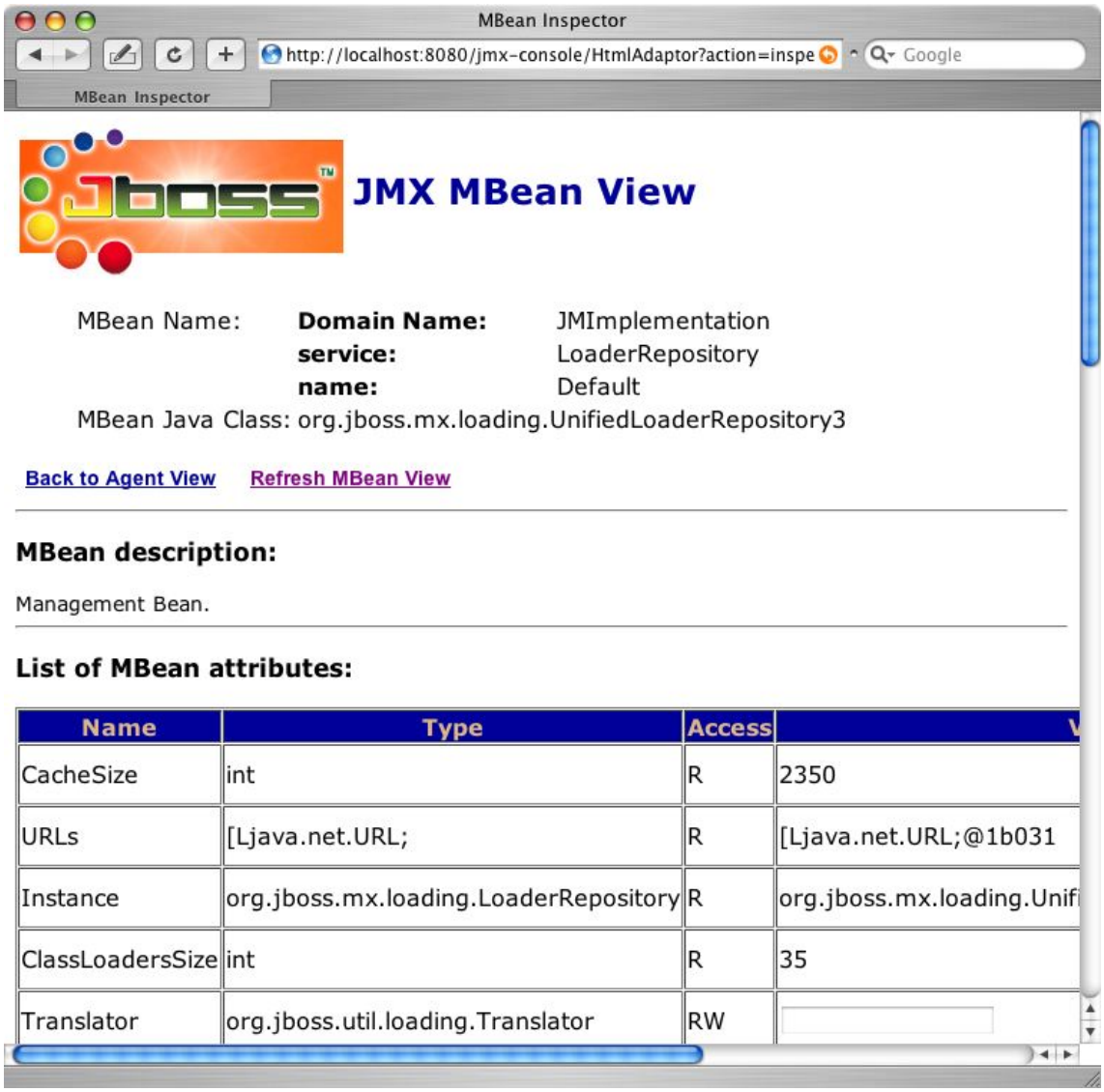


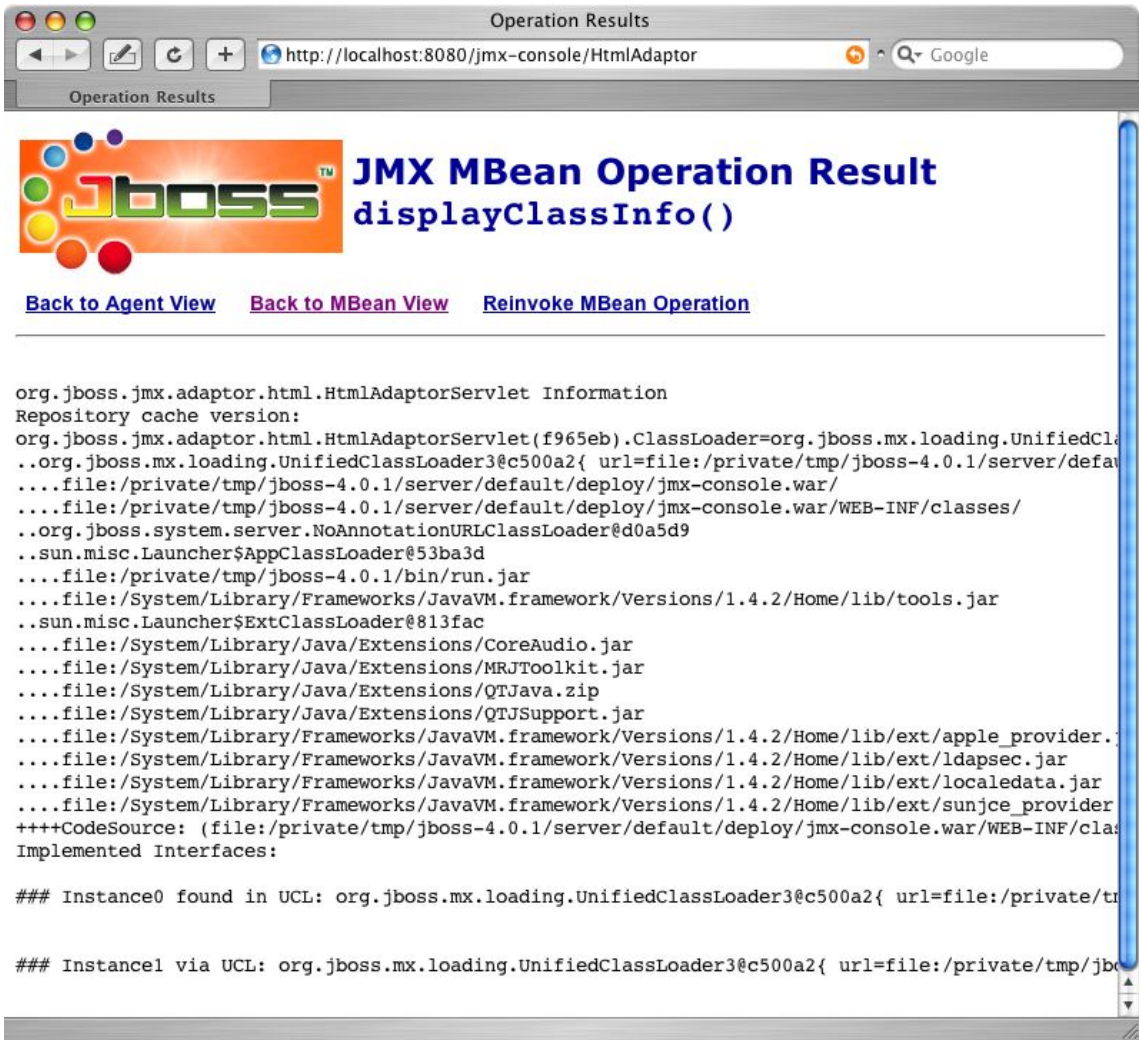
Figure 3.4. The default class LoaderRepository MBean view in the JMX console

Two useful operations you will find here are `getPackageClassLoaders(String)` and `displayClassInfo(String)`. The `getPackageClassLoaders` operation returns a set of class loaders that have been indexed to contain classes or resources for the given package name. The package name must have a trailing period. If you type in the package name `org.jboss.ejb.`, the following information is displayed:

```
[org.jboss.mx.loading.UnifiedClassLoader3@e26ae7{
url=file:/private/tmp/jboss-4.2.0/server/production/tmp/deploy/tmp11895jboss-service.xml,
  addedOrder=2}]
```

This is the string representation of the set. It shows one `UnifiedClassLoader3` instance with a primary URL pointing to the `jboss-service.xml` descriptor. This is the second class loader added to the repository (shown by `addedOrder=2`). It is the class loader that owns all of the JARs in the `lib` directory of the server configuration (e.g., `server/production/lib`).

The view the information for a given class, use the `displayClassInfo` operation, passing in the fully qualified name of the class to view. For example, if we use `org.jboss.jmx.adaptor.html.HtmlAdaptorServlet` which is from the package we just looked at, the following description is displayed:



The information is a dump of the information for the Class instance in the loader repository if one has been loaded, followed by the class loaders that are seen to have the class file available. If a class is seen to have more than one class loader associated with it, then there is the potential for class loading related errors.

2.2.4.2. Scoping Classes

If you need to deploy multiple versions of an application you need to use deployment based scoping. With deployment based scoping, each deployment creates its own class loader repository in the form of a `HeirarchicalLoaderRepository3` that looks first to the `UnifiedClassLoader3` instances of the deployment units included in the EAR before delegating to the default `UnifiedLoaderRepository3`. To enable an EAR specific loader repository, you need to create a `META-INF/jboss-app.xml` descriptor as shown in *Example 3.9, "An example jboss-app.xml descriptor for enabled scoped class loading at the EAR level."*.

```
<jboss-app>
    <loader-repository>some.dot.com:loader=webtest.ear</loader-repository>
</jboss-app>
```

Example 3.9. An example jboss-app.xml descriptor for enabled scoped class loading at the EAR level.

The value of the `loader-repository` element is the JMX object name to assign to the repository created for the EAR. This must be unique and valid JMX ObjectName, but the actual name is not important.

2.2.4.3. The Complete Class Loading Model

The previous discussion of the core class loading components introduced the custom `UnifiedClassLoader3` and `UnifiedLoaderRepository3` classes that form a shared class loading space. The complete class loading picture must also include the parent class loader used by `UnifiedClassLoader3`s as well as class loaders introduced for scoping and other specialty class loading purposes. *Figure 3.5, "A complete class loader view"* shows an outline of the class hierarchy that would exist for an EAR deployment containing EJBs and WARs.

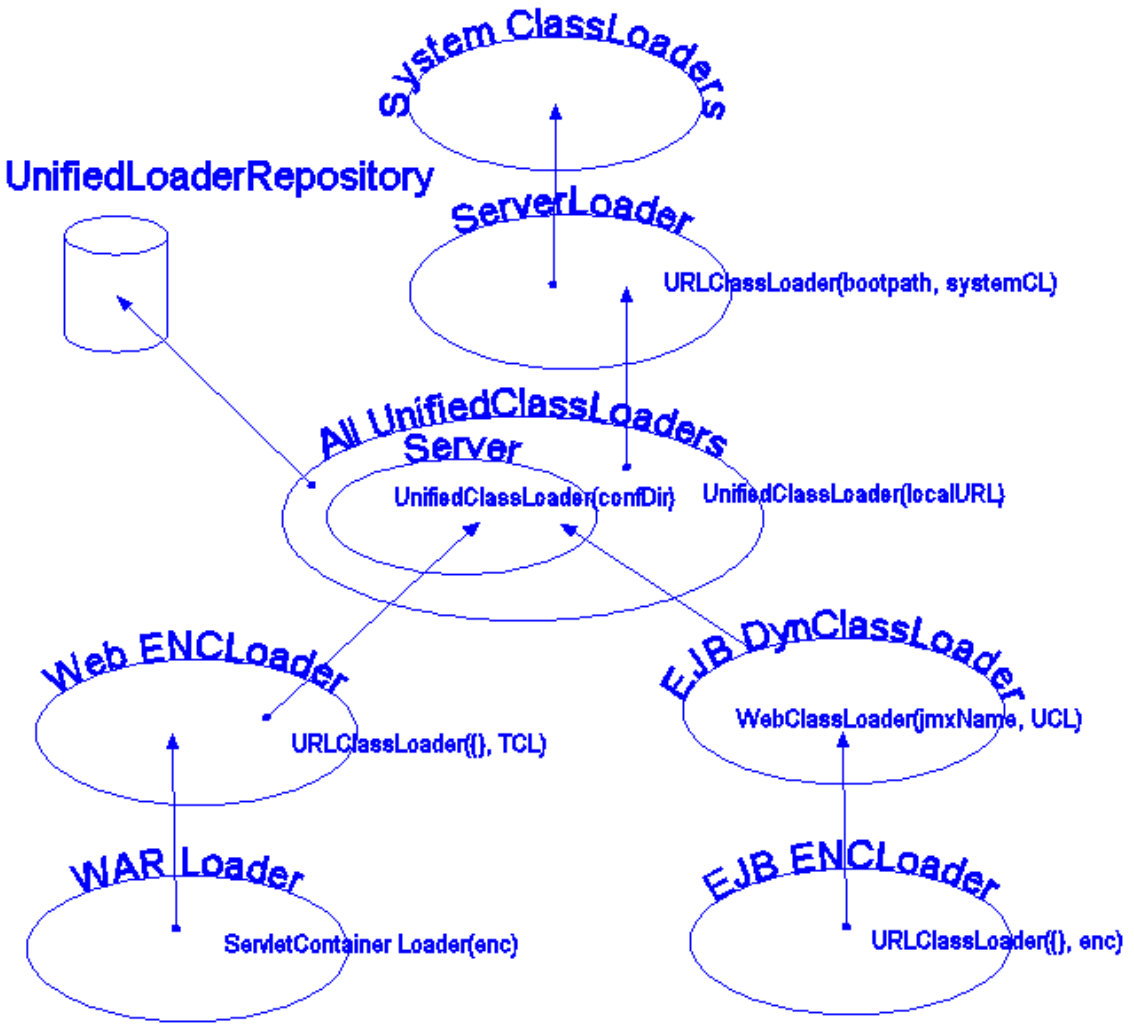


Figure 3.5. A complete class loader view

The following points apply to this figure:

- **System ClassLoaders**: The System ClassLoaders node refers to either the thread context class loader (TCL) of the VM main thread or of the thread of the application that is loading the JBoss server if it is embedded.
- **ServerLoader**: The ServerLoader node refers to the a `URLClassLoader` that delegates to the System ClassLoaders and contains the following boot URLs:
  - All URLs referenced via the `jboss.boot.library.list` system property. These are path specifications relative to the `libraryURL` defined by the `jboss.lib.url` property. If there is no `jboss.lib.url` property specified, it defaults to `jboss.home.url + /lib/`. If there is no `jboss.boot.library` property specified, it defaults to `jaxp.jar`, `log4j-boot.jar`, `jboss-common.jar`, and `jboss-system.jar`.
  - The JAXP JAR which is either `crimson.jar` or `xerces.jar` depending on the `-j` option to the `Main` entry point. The default is `crimson.jar`.
  - The JBoss JMX jar and GNU regex jar, `jboss-jmx.jar` and `gnu-regexp.jar`.
  - Oswego concurrency classes JAR, `concurrent.jar`
  - Any JARs specified as libraries via `-L` command line options
  - Any other JARs or directories specified via `-C` command line options
- **Server**: The Server node represent a collection of UCLs created by the `org.jboss.system.server.Server` interface implementation. The default implementation creates UCLs for the `patchDir` entries as well as the server `conf` directory. The last UCL created is set as the JBoss main thread context class loader. This will be combined into a single UCL now that multiple URLs per UCL are supported.
- **All UnifiedClassLoader3s**: The *All UnifiedClassLoader3* node represents the UCLs created by deployers. This covers EARs, jars, WARs, SARs and directories seen by the deployment scanner as well as JARs referenced by their manifests and any nested deployment units they may contain. This is a flat namespace and there should not be multiple instances of a class in different deployment JARs. If there are, only the first loaded will be used and the results may not be as expected. There is a mechanism for scoping visibility based on EAR deployment units that we discussed in *Section 2.2.4.2, "Scoping Classes"*. Use this mechanism if you need to deploy multiple versions of a class in a given JBoss server.
- **EJB DynClassLoader**: The EJB `DynClassLoader` node is a subclass of `URLClassLoader` that is used to provide RMI dynamic class loading via the simple HTTP WebService. It specifies an empty `URL[]` and delegates to the TCL as its parent class loader. If the WebService is configured to allow system level classes to be loaded, all classes in the

`UnifiedLoaderRepository3` as well as the system classpath are available via HTTP.

- **EJB ENCLoader**: The *EJB ENCLoader* node is a `URLClassLoader` that exists only to provide a unique context for an EJB deployment's `java:comp` JNDI context. It specifies an empty `URL[]` and delegates to the EJB `DynClassLoader` as its parent class loader.
- **Web ENCLoader**: The Web `ENCLoader` node is a URLClassLoader that exists only to provide a unique context for a web deployment's `java:comp` JNDI context. It specifies an empty `URL[]` and delegates to the TCL as its parent class loader.
- **WAR Loader**: The *WAR Loader* is a servlet container specific classloader that delegates to the Web ENCLoader as its parent class loader. The default behavior is to load from its parent class loader and then the WAR `WEB-INFclasses` and `lib` directories. If the servlet 2.3 class loading model is enabled it will first load from the its `WEB-INF` directories and then the parent class loader.

In its current form there are some advantages and disadvantages to the JBoss class loading architecture. Advantages include:

- Classes do not need to be replicated across deployment units in order to have access to them.
- Many future possibilities including novel partitioning of the repositories into domains, dependency and conflict detection, etc.

Disadvantages include:

- Existing deployments may need to be repackaged to avoid duplicate classes. Duplication of classes in a loader repository can lead to class cast exceptions and linkage errors depending on how the classes are loaded.
- Deployments that depend on different versions of a given class need to be isolated in separate EARs and a unique `HeirarchicalLoaderRepository3` defined using a `jboss-app.xml` descriptor.

2.3. JBoss XMBeans

XMBeans are the JBoss JMX implementation version of the JMX model MBean. XMBeans have the richness of the dynamic MBean metadata without the tedious programming required by a direct implementation of the `DynamicMBean` interface. The JBoss model MBean implementation allows one to specify the management interface of a component through a XML descriptor, hence the X in XMBean. In addition to providing a simple mechanism for describing the metadata required for a dynamic MBean, XMBeans also allow for the specification of attribute persistence, caching behavior, and even advanced customizations like the MBean implementation interceptors. The high level elements of the `jboss_xmbean_1_2.dtd` for the XMBean descriptor is given in *Figure 3.6, “The JBoss 1.0 XMBean DTD Overview*

(jboss_xmbean_1_2.dtd)".

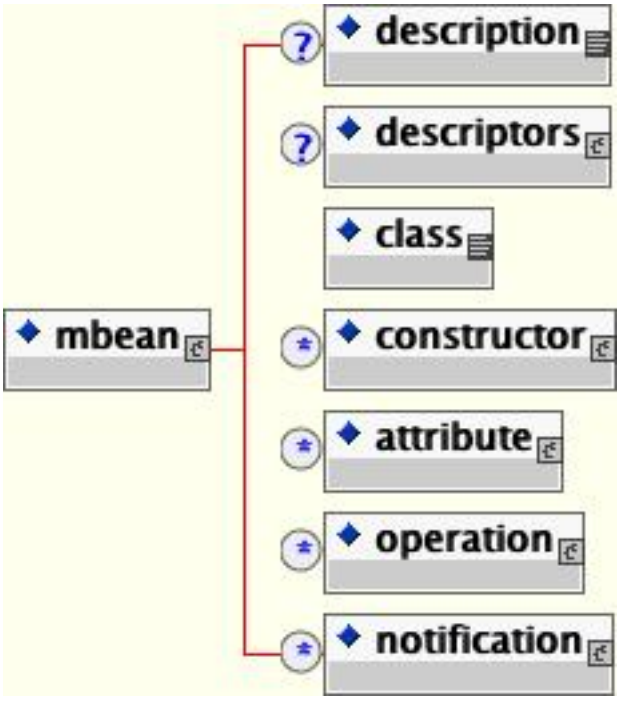


Figure 3.6. The JBoss 1.0 XMBean DTD Overview (jboss_xmbean_1_2.dtd)

The `mbean` element is the root element of the document containing the required elements for describing the management interface of one MBean (constructors, attributes, operations and notifications). It also includes an optional description element, which can be used to describe the purpose of the MBean, as well as an optional descriptors element which allows for persistence policy specification, attribute caching, etc.

2.3.1. Descriptors

The descriptors element contains all the descriptors for a containing element, as subelements. The descriptors suggested in the JMX specification as well as those used by JBoss have predefined elements and attributes, whereas custom descriptors have a generic descriptor element with name and value attributes as show in *Figure 3.7, " The descriptors element content model"*.

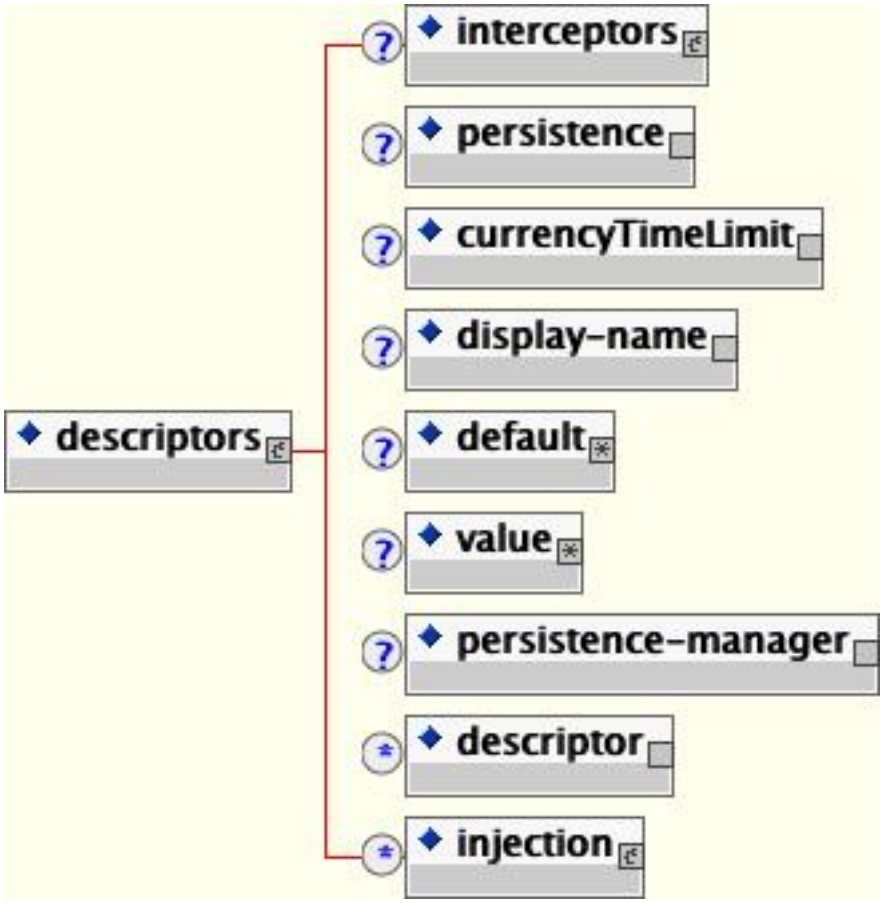


Figure 3.7. The descriptors element content model

The key descriptors child elements include:

- **interceptors**: The `interceptors` element specifies a customized stack of interceptors that will be used in place of the default stack. Currently this is only used when specified at the MBean level, but it could define a custom attribute or operation level interceptor stack in the future. The content of the interceptors element specifies a custom interceptor stack. If no interceptors element is specified the standard `ModelMBean` interceptors will be used. The standard interceptors are:
  - org.jboss.mx.interceptor.PersistenceInterceptor
  - org.jboss.mx.interceptor.MBeanAttributeInterceptor
  - org.jboss.mx.interceptor.ObjectReferenceInterceptor

  When specifying a custom interceptor stack you would typically include the standard interceptors along with your own unless you are replacing the corresponding standard interceptor.

  Each interceptor element content value specifies the fully qualified class name of the interceptor implementation. The class must implement the

`org.jboss.mx.interceptor.Interceptor` interface. The interceptor class must also have either a no-arg constructor, or a constructor that accepts a `javax.management.MBeanInfo`.

The interceptor elements may have any number of attributes that correspond to JavaBean style properties on the interceptor class implementation. For each `interceptor` element attribute specified, the interceptor class is queried for a matching setter method. The attribute value is converted to the true type of the interceptor class property using the `java.beans.PropertyEditor` associated with the type. It is an error to specify an attribute for which there is no setter or `PropertyEditor`.

- **persistence**: The `persistence` element allows the specification of the `persistPolicy`, `persistPeriod`, `persistLocation`, and `persistName` persistence attributes suggested by the JMX specification. The persistence element attributes are:
  - **persistPolicy**: The `persistPolicy` attribute defines when attributes should be persisted and its value must be one of
    - **Never**: attribute values are transient values that are never persisted
    - **OnUpdate**: attribute values are persisted whenever they are updated
    - **OnTimer**: attribute values are persisted based on the time given by the `persistPeriod`.
    - **NoMoreOftenThan**: attribute values are persisted when updated but no more often than the `persistPeriod`.
  - **persistPeriod**: The `persistPeriod` attribute gives the update frequency in milliseconds if the `perisitPolicy` attribute is `NoMoreOftenThan` or `OnTimer`.
  - **persistLocation**: The `persistLocation` attribute specifies the location of the persistence store. Its form depends on the JMX persistence implementation. Currently this should refer to a directory into which the attributes will be serialized if using the default JBoss persistence manager.
  - **persistName**: The `persistName` attribute can be used in conjunction with the `persistLocation` attribute to further qualify the persistent store location. For a directory `persistLocation` the `persistName` specifies the file to which the attributes are stored within the directory.
- **currencyTimeLimit**: The `currencyTimeLimit` element specifies the time in seconds that a cached value of an attribute remains valid. Its value attribute gives the time in seconds. A value of 0 indicates that an attribute value should always be retrieved from the MBean and never cached. A value of -1 indicates that a cache value is always valid.
- **display-name**: The `display-name` element specifies the human friendly name of an item.
- **default**: The `default` element specifies a default value to use when a field has not been set. Note that this value is not written to the MBean on startup as is the case with the `jboss-service.xml` attribute element content value. Rather, the default value is used only if there is no attribute accessor defined, and there is no value element defined.

- **value**: The `value` element specifies a management attribute's current value. Unlike the `default` element, the `value` element is written through to the MBean on startup provided there is a setter method available.
- **persistence-manager**: The `persistence-manager` element gives the name of a class to use as the persistence manager. The `value` attribute specifies the class name that supplies the `org.jboss.mx.persistence.PersistenceManager` interface implementation. The only implementation currently supplied by JBoss is the `org.jboss.mx.persistence.ObjectStreamPersistenceManager` which serializes the `ModelMBeanInfo` content to a file using Java serialization.
- **descriptor**: The `descriptor` element specifies an arbitrary descriptor not known to JBoss. Its `name` attribute specifies the type of the descriptor and its `value` attribute specifies the descriptor value. The `descriptor` element allows for the attachment of arbitrary management metadata.
- **injection**: The `injection` element describes an injection point for receiving information from the microkernel. Each injection point specifies the type and the set method to use to inject the information into the resource. The `injection` element supports type attributes:
  - **id**: The `id` attribute specifies the injection point type. The current injection point types are:
    - **MBeanServerType**: An *MBeanServerType* injection point receives a reference to the *MBeanServer* that the XMBean is registered with.
    - **MBeanInfoType**: An *MBeanInfoType* injection point receives a reference to the XMBean *ModelMBeanInfo* metadata.
    - **ObjectNameType**: The *ObjectName* injection point receives the *ObjectName* that the XMBean is registered under.
- **setMethod**: The *setMethod* attribute gives the name of the method used to set the injection value on the resource. The set method should accept values of the type corresponding to the injection point type.

Note that any of the constructor, attribute, operation or notification elements may have a `descriptors` element to specify the specification defined descriptors as well as arbitrary extension descriptor settings.

2.3.2. The Management Class

The `class` element specifies the fully qualified name of the managed object whose management interface is described by the XMBean descriptor.

2.3.3. The Constructors

The `constructor` element(s) specifies the constructors available for creating an instance of the managed object. The constructor element and its content model are shown in *Figure 3.8, “The XMBean constructor element and its content model”*.

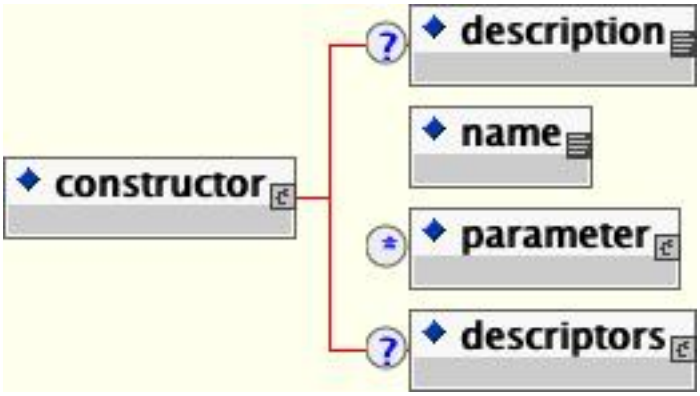


Figure 3.8. The XMBean constructor element and its content model

The key child elements are:

- **description**: A description of the constructor.
- **name**: The name of the constructor, which must be the same as the implementation class.
- **parameter**: The parameter element describes a constructor parameter. The parameter element has the following attributes:
  - **description**: An optional description of the parameter.
  - **name**: The required variable name of the parameter.
  - **type**: The required fully qualified class name of the parameter type.
- **descriptors**: Any descriptors to associate with the constructor metadata.

2.3.4. The Attributes

The `attribute` element(s) specifies the management attributes exposed by the MBean. The attribute element and its content model are shown in *Figure 3.9, "The XMBean attribute element and its content model"*.

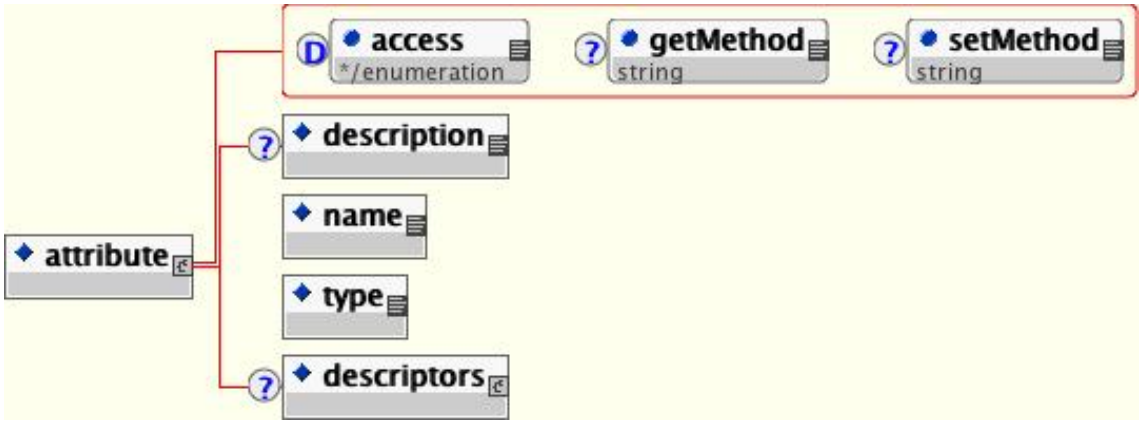


Figure 3.9. The XMBean attribute element and its content model

The `attribute` element supported attributes include:

- **access**: The optional `access` attribute defines the read/write access modes of an attribute. It must be one of:
  - **read-only**: The attribute may only be read.
  - **write-only**: The attribute may only be written.
  - **read-write**: The attribute is both readable and writable. This is the implied default.
- **getMethod**: The `getMethod` attribute defines the name of the method which reads the named attribute. This must be specified if the managed attribute should be obtained from the MBean instance.
- **setMethod**: The `setMethod` attribute defines the name of the method which writes the named attribute. This must be specified if the managed attribute should be obtained from the MBean instance.

The key child elements of the attribute element include:

- **description**: A description of the attribute.
- **name**: The name of the attribute as would be used in the `MBeanServer.getAttribute()` operation.
- **type**: The fully qualified class name of the attribute type.
- **descriptors**: Any additional descriptors that affect the attribute persistence, caching, default value, etc.

2.3.5. The Operations

The management operations exposed by the XMBean are specified via one or more operation elements. The operation element and its content model are shown in *Figure 3.10, "The XMBean operation element and its content model"*.

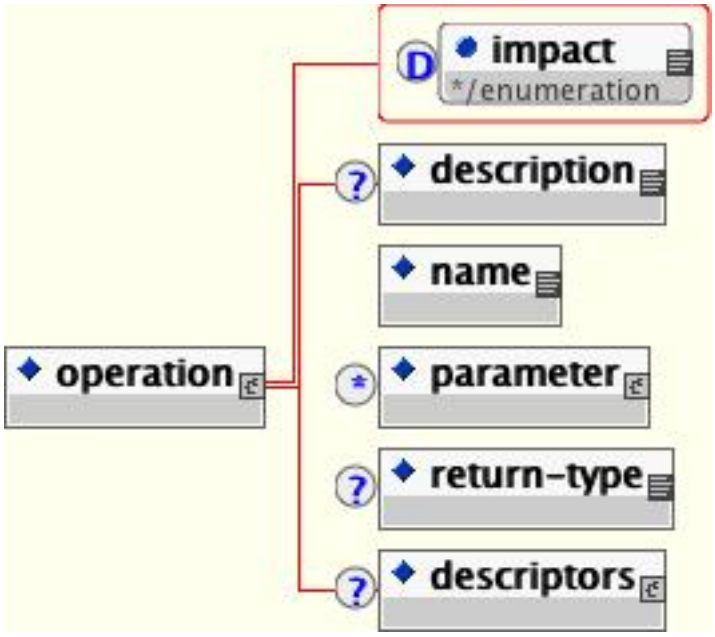


Figure 3.10. The XMBean operation element and its content model

The impact attribute defines the impact of executing the operation and must be one of:

- **ACTION**: The operation changes the state of the MBean component (write operation)
- **INFO**: The operation should not alter the state of the MBean component (read operation).
- **ACTION_INFO**: The operation behaves like a read/write operation.

The child elements are:

- **description**: This element specifies a human readable description of the operation.
- **name**: This element contains the operation's name
- **parameter**: This element describes the operation's signature.
- **return-type**: This element contains a fully qualified class name of the return type from this operation. If not specified, it defaults to void.

- **descriptors**: Any descriptors to associate with the operation metadata.

2.3.6. Notifications

The `notification` element(s) describes the management notifications that may be emitted by the XMBean. The notification element and its content model is shown in *Figure 3.11, "The XMBean notification element and content model"*.

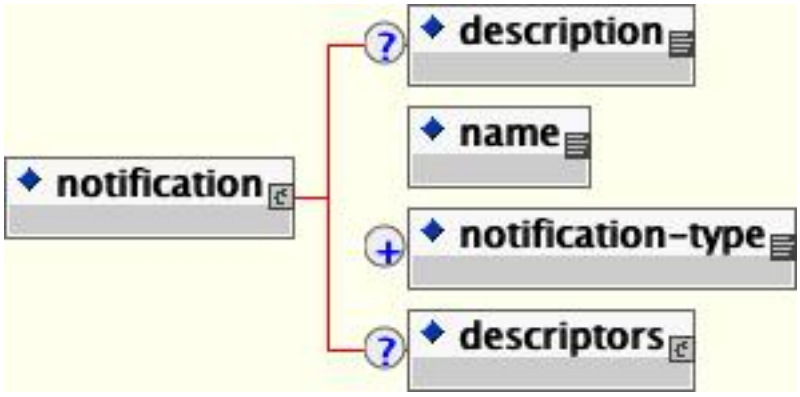


Figure 3.11. The XMBean notification element and content model

The child elements are:

- **description**: This element gives a human readable description of the notification.
- **name**: This element contains the fully qualified name of the notification class.
- **notification-type**: This element contains the dot-separated notification type string.
- **descriptors**: Any descriptors to associate with the notification metadata.

3. Connecting to the JMX Server

JBoss includes adaptors that allow access to the JMX MBeanServer from outside of the JBoss server VM. The current adaptors include HTML, an RMI interface, and an EJB.

3.1. Inspecting the Server - the JMX Console Web Application

JBoss comes with its own implementation of a JMX HTML adaptor that allows one to view the server's MBeans using a standard web browser. The default URL for the console web application is *http://localhost:8080/jmx-console/*. If you browse this location you will see something similar to that presented in *Figure 3.12, "The JBoss JMX console web application agent view"*.

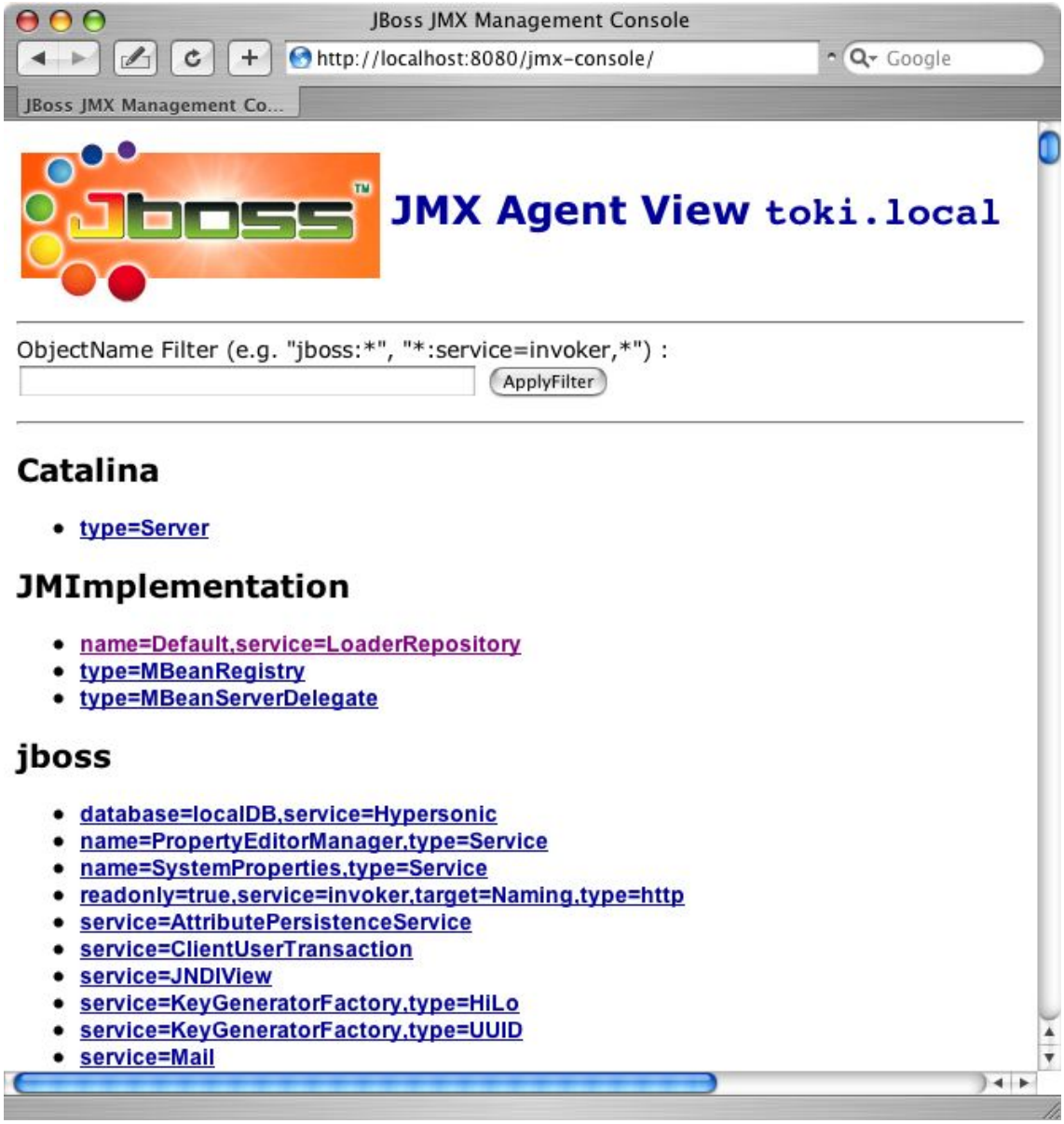


Figure 3.12. The JBoss JMX console web application agent view

The top view is called the agent view and it provides a listing of all MBeans registered with the `MBeanServer` sorted by the domain portion of the MBean's `ObjectName`. Under each domain are the MBeans under that domain. When you select one of the MBeans you will be taken to the MBean view. This allows one to view and edit an MBean's attributes as well as invoke operations. As an example, *Figure 3.13, "The MBean view for the "jboss.system:type=Server" MBean"* shows the MBean view for the `jboss.system:type=Server` MBean.

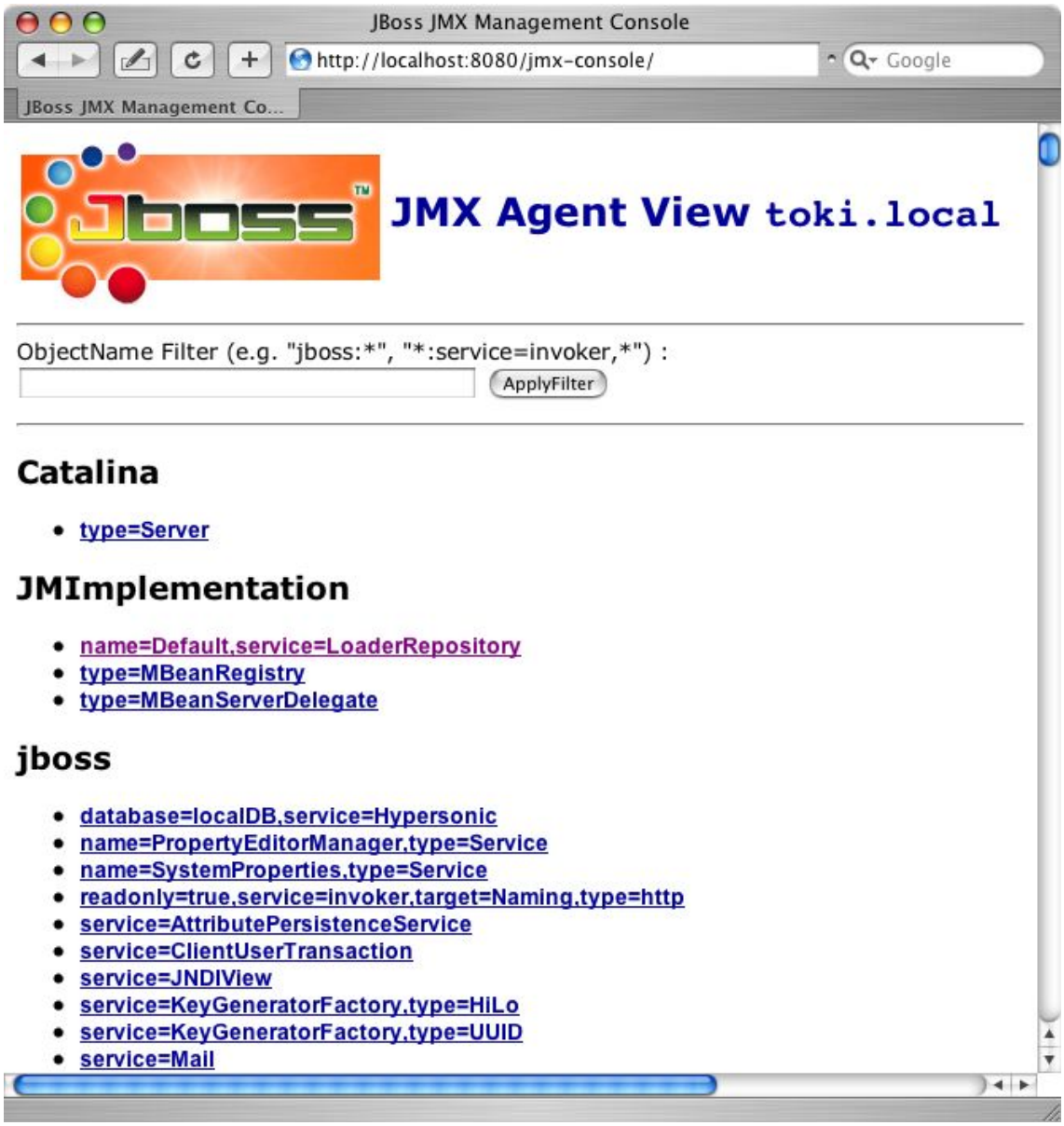


Figure 3.13. The MBean view for the "jboss.system:type=Server" MBean

The source code for the JMX console web application is located in the `varia` module under the `src/main/org/jboss/jmx` directory. Its web pages are located under `varia/src/resources/jmx`. The application is a simple MVC servlet with JSP views that utilize the MBeanServer.

3.1.1. Securing the JMX Console

Since the JMX console web application is just a standard servlet, it may be secured using standard J2EE role based security. The `jmx-console.war` that is deployed as an unpacked WAR that includes template settings for quickly enabling simple username and password based

access restrictions. If you look at the `jmx-console.war` in the `server/production/deploy` directory you will find the `web.xml` and `jboss-web.xml` descriptors in the `WEB-INF` directory. The `jmx-console-roles.properties` and `jmx-console-users.properties` files are located in the `server/production/conf/props` directory.

By uncommenting the security sections of the `web.xml` and `jboss-web.xml` descriptors as shown in *Example 3.10, "The jmx-console.war web.xml descriptors with the security elements uncommented."*, you enable HTTP basic authentication that restricts access to the JMX Console application to the user `admin` with password `admin`. The username and password are determined by the `admin=admin` line in the `jmx-console-users.properties` file.

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC
          "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
          "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <!-- ... -->

    <!-- A security constraint that restricts access to the HTML JMX console
         to users with the role JBossAdmin. Edit the roles to what you want
and
         uncomment the WEB-INF/jboss-web.xml/security-domain element to
enable
         secured access to the HTML JMX console.
    -->
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>HtmlAdaptor</web-resource-name>
            <description> An example security config that only allows users
with
                the role JBossAdmin to access the HTML JMX console web
                application </description>
            <url-pattern>/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>JBossAdmin</role-name>
        </auth-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>JBoss JMX Console</realm-name>
    </login-config>
    <security-role>
        <role-name>JBossAdmin</role-name>
    </security-role>
</web-app>
```

Example 3.10. The jmx-console.war web.xml descriptors with the security elements uncommented.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss-web
    PUBLIC "-//JBoss//DTD Web Application 2.3//EN"
    "http://www.jboss.org/j2ee/dtd/jboss-web_3_0.dtd">
<jboss-web>
    <!--
        Uncomment the security-domain to enable security. You will
        need to edit the htmladaptor login configuration to setup the
        login modules used to authentication users.
    -->
    <security-domain>java:/jaas/jmx-console</security-domain>
</jboss-web>
```

Example 3.11. The jmx-console.war jboss-web.xml descriptors with the security elements uncommented.

Make these changes and then when you try to access the JMX Console URL. You will see a dialog similar to that shown in *Figure 3.14, "The JMX Console basic HTTP login dialog."*.

Figure 3.14. The JMX Console basic HTTP login dialog.

You probably to use the properties files for securing access to the JMX console application. To see how to properly configure the security settings of web applications see *Chapter 8, Security on JBoss*.

3.2. Connecting to JMX Using RMI

JBoss supplies an RMI interface for connecting to the JMX MBeanServer. This interface is `org.jboss.jmx.adaptor.rmi.RMIAdaptor`. The `RMIAdaptor` interface is bound into JNDI in the default location of `jmx/invoker/RMIAdaptor` as well as `jmx/rmi/RMIAdaptor` for backwards compatibility with older clients.

Example 3.12, " A JMX client that uses the RMIAdaptor" shows a client that makes use of the `RMIAdaptor` interface to query the `MBeanInfo` for the `JNDIView` MBean. It also invokes the MBean's `list(boolean)` method and displays the result.

```
public class JMXBrowser
{
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws Exception
    {
        InitialContext ic = new InitialContext();
        RMIAdaptor server = (RMIAdaptor)
ic.lookup("jmx/invoker/RMIAdaptor");

        // Get the MBeanInfo for the JNDIView MBean
        ObjectName name = new ObjectName("jboss:service=JNDIView");
        MBeanInfo  info = server.getMBeanInfo(name);
        System.out.println("JNDIView Class: " + info.getClassName());

        MBeanOperationInfo[] opInfo = info.getOperations();
        System.out.println("JNDIView Operations: ");
        for(int o = 0; o < opInfo.length; o ++) {
            MBeanOperationInfo op = opInfo[o];

            String returnType = op.getReturnType();
            String opName     = op.getName();
            System.out.print(" + " + returnType + " " + opName + "(");

            MBeanParameterInfo[] params = op.getSignature();
            for(int p = 0; p < params.length; p++)  {
                MBeanParameterInfo paramInfo = params[p];

                String pname = paramInfo.getName();
                String type  = paramInfo.getType();

                if (pname.equals(type)) {
                    System.out.print(type);
                } else {
                    System.out.print(type + " " + name);
                }

                if (p < params.length-1) {
                    System.out.print(',');
                }
            }
            System.out.println(")");
        }
```

```
        // Invoke the list(boolean) op
        String[] sig    = {"boolean"};
        Object[] opArgs = {Boolean.TRUE};
        Object   result = server.invoke(name, "list", opArgs, sig);

        System.out.println("JNDIView.list(true) output:\n"+result);
    }
}
```

Example 3.12. A JMX client that uses the RMIAdaptor

To test the client access using the `RMIAdaptor`, run the following:

```
[examples]$ ant -Dchap=jmx -Dex=4 run-example
...

run-example4:
     [java] JNDIView Class: org.jboss.mx.modelmbean.XMBean
     [java] JNDIView Operations:
     [java]  + java.lang.String list(boolean jboss:service=JNDIView)
     [java]  + java.lang.String listXML()
     [java]  + void create()
     [java]  + void start()
     [java]  + void stop()
     [java]  + void destroy()
     [java]  + void jbossInternalLifecycle(java.lang.String
jboss:service=JNDIView)
     [java]  + java.lang.String getName()
     [java]  + int getState()
     [java]  + java.lang.String getStateString()
     [java] JNDIView.list(true) output:
     [java] <h1>java: Namespace</h1>
     [java] <pre>
     [java]   +- XAConnectionFactory (class:
org.jboss.mq.SpyXAConnectionFactory)
     [java]   +- DefaultDS (class: javax.sql.DataSource)
     [java]   +- SecurityProxyFactory (class:
org.jboss.security.SubjectSecurityProxyFactory)
     [java]   +- DefaultJMSProvider (class:
org.jboss.jms.jndi.JNDIProviderAdapter)
     [java]   +- comp (class: javax.naming.Context)
     [java]   +- JmsXA (class:
org.jboss.resource.adapter.jms.JmsConnectionFactoryImpl)
     [java]   +- ConnectionFactory (class:
org.jboss.mq.SpyConnectionFactory)
     [java]   +- jaas (class: javax.naming.Context)
     [java]   |   +- JmsXARealm (class:
org.jboss.security.plugins.SecurityDomainContext)
     [java]   |   +- jbossmq (class:
org.jboss.security.plugins.SecurityDomainContext)
     [java]   |   +- HsqlDbRealm (class:
```

```
org.jboss.security.plugins.SecurityDomainContext)
     [java]   +- timedCacheFactory (class: javax.naming.Context)
     [java] Failed to lookup: timedCacheFactory, errmsg=null
     [java]   +- TransactionPropagationContextExporter (class:
org.jboss.tm.TransactionPropag
ationContextFactory)
     [java]   +- StdJMSPool (class:
org.jboss.jms.asf.StdServerSessionPoolFactory)
     [java]   +- Mail (class: javax.mail.Session)
     [java]   +- TransactionPropagationContextImporter (class:
org.jboss.tm.TransactionPropag
ationContextImporter)
     [java]   +- TransactionManager (class: org.jboss.tm.TxManager)
     [java] </pre>
     [java] <h1>Global JNDI Namespace</h1>
     [java] <pre>
     [java]   +- XAConnectionFactory (class:
org.jboss.mq.SpyXAConnectionFactory)
     [java]   +- UIL2ConnectionFactory[link -> ConnectionFactory] (class:
javax.naming.Lin
kRef)
     [java]   +- UserTransactionSessionFactory (proxy: $Proxy11 implements
interface org.jbos
s.tm.usertx.interfaces.UserTransactionSessionFactory)
     [java]   +- HTTPConnectionFactory (class:
org.jboss.mq.SpyConnectionFactory)
     [java]   +- console (class: org.jnp.interfaces.NamingContext)
     [java]   |   +- PluginManager (proxy: $Proxy36 implements interface
org.jboss.console.ma
nager.PluginManagerMBean)
     [java]   +- UIL2XAConnectionFactory[link -> XAConnectionFactory]
(class: javax.naming
.LinkRef)
     [java]   +- UUIDKeyGeneratorFactory (class:
org.jboss.ejb.plugins.keygenerator.uuid.UUID
KeyGeneratorFactory)
     [java]   +- HTTPXAConnectionFactory (class:
org.jboss.mq.SpyXAConnectionFactory)
     [java]   +- topic (class: org.jnp.interfaces.NamingContext)
     [java]   |   +- testDurableTopic (class: org.jboss.mq.SpyTopic)
     [java]   |   +- testTopic (class: org.jboss.mq.SpyTopic)
     [java]   |   +- securedTopic (class: org.jboss.mq.SpyTopic)
     [java]   +- queue (class: org.jnp.interfaces.NamingContext)
     [java]   |   +- A (class: org.jboss.mq.SpyQueue)
     [java]   |   +- testQueue (class: org.jboss.mq.SpyQueue)
     [java]   |   +- ex (class: org.jboss.mq.SpyQueue)
     [java]   |   +- DLQ (class: org.jboss.mq.SpyQueue)
     [java]   |   +- D (class: org.jboss.mq.SpyQueue)
     [java]   |   +- C (class: org.jboss.mq.SpyQueue)
     [java]   |   +- B (class: org.jboss.mq.SpyQueue)
     [java]   +- ConnectionFactory (class:
org.jboss.mq.SpyConnectionFactory)
     [java]   +- UserTransaction (class:
org.jboss.tm.usertx.client.ClientUserTransaction)
     [java]   +- jmx (class: org.jnp.interfaces.NamingContext)
     [java]   |   +- invoker (class: org.jnp.interfaces.NamingContext)
     [java]   |   |   +- RMIAdaptor (proxy: $Proxy35 implements interface
```

```
org.jboss.jmx.adapt
or.rmi.RMIAdaptor,interface org.jboss.jmx.adaptor.rmi.RMIAdaptorExt)
     [java]   |   +- rmi (class: org.jnp.interfaces.NamingContext)
     [java]   |   |   +- RMIAdaptor[link -> jmx/invoker/RMIAdaptor] (class:
javax.naming.L
inkRef)
     [java]   +- HiLoKeyGeneratorFactory (class:
org.jboss.ejb.plugins.keygenerator.hilo.HiLo
KeyGeneratorFactory)
     [java]   +- UILXAConnectionFactory[link -> XAConnectionFactory] (class:
javax.naming.
LinkRef)
     [java]   +- UILConnectionFactory[link -> ConnectionFactory] (class:
javax.naming.Link
Ref)
     [java] </pre>
```

3.3. Command Line Access to JMX

JBoss provides a simple command line tool that allows for interaction with a remote JMX server instance. This tool is called twiddle (for twiddling bits via JMX) and is located in the `bin` directory of the distribution. Twiddle is a command execution tool, not a general command shell. It is run using either the `twiddle.sh` or `twiddle.bat` scripts, and passing in a `-h`(`--help`) argument provides the basic syntax, and `--help-commands` shows what you can do with the tool:

```
[bin]$ ./twiddle.sh -h
A JMX client to 'twiddle' with a remote JBoss server.

usage: twiddle.sh [options] <command> [command_arguments]

options:
    -h, --help                Show this help message
        --help-commands       Show a list of commands
    -H=<command>              Show command specific help
    -c=command.properties     Specify the command.properties file to use
    -D<name>[=<value>]        Set a system property
    --                        Stop processing options
    -s, --server=<url>        The JNDI URL of the remote server
    -a, --adapter=<name>      The JNDI name of the RMI adapter to use
    -q, --quiet               Be somewhat more quiet
```

3.3.1. Connecting twiddle to a Remote Server

By default the twiddle command will connect to the localhost at port 1099 to lookup the default `jmx/rmi/RMIAdaptor` binding of the `RMIAdaptor` service as the connector for communicating with the JMX server. To connect to a different server/port combination you can use the `-s` (`--server`) option:

```
[bin]$ ./twiddle.sh -s toki serverinfo -d jboss
[bin]$ ./twiddle.sh -s toki:1099 serverinfo -d jboss
```

To connect using a different RMIAdaptor binding use the `-a` (`--adapter`) option:

```
[bin]$ ./twiddle.sh -s toki -a jmx/rmi/RMIAdaptor serverinfo -d jboss
```

3.3.2. Sample twiddle Command Usage

To access basic information about a server, use the `serverinfo` command. This currently supports:

```
[bin]$ ./twiddle.sh -H serverinfo
Get information about the MBean server

usage: serverinfo [options]

options:
    -d, --domain    Get the default domain
    -c, --count     Get the MBean count
    -l, --list      List the MBeans
    --              Stop processing options
[bin]$ ./twiddle.sh --server=toki serverinfo --count
460
[bin]$ ./twiddle.sh --server=toki serverinfo --domain
jboss
```

To query the server for the name of MBeans matching a pattern, use the `query` command. This currently supports:

```
[bin]$ ./twiddle.sh -H query
Query the server for a list of matching MBeans

usage: query [options] <query>
options:
    -c, --count    Display the matching MBean count
    --             Stop processing options
Examples:
 query all mbeans: query '*:*'
 query all mbeans in the jboss.j2ee domain: query 'jboss.j2ee:*'
[bin]$ ./twiddle.sh -s toki query 'jboss:service=invoker,*'
jboss:readonly=true,service=invoker,target=Naming,type=http
jboss:service=invoker,type=jrmp
jboss:service=invoker,type=local
jboss:service=invoker,type=pooled
jboss:service=invoker,type=http
jboss:service=invoker,target=Naming,type=http
```

To get the attributes of an MBean, use the get command:

```
[bin]$ ./twiddle.sh -H get
Get the values of one or more MBean attributes

usage: get [options] <name> [<attr>+]
```

```
  If no attribute names are given all readable attributes are retrieved
options:
    --noprefix    Do not display attribute name prefixes
    --            Stop processing options
[bin]$ ./twiddle.sh get jboss:service=invoker,type=jrmp RMIObjectPort
StateString
RMIObjectPort=4444
StateString=Started
[bin]$ ./twiddle.sh get jboss:service=invoker,type=jrmp
ServerAddress=0.0.0.0
RMIClientSocketFactoryBean=null
StateString=Started
State=3
RMIServerSocketFactoryBean=org.jboss.net.sockets.DefaultSocketFactory@ad093076
EnableClassCaching=false
SecurityDomain=null
RMIServerSocketFactory=null
Backlog=200
RMIObjectPort=4444
Name=JRMPInvoker
RMIClientSocketFactory=null
```

To query the MBeanInfo for an MBean, use the info command:

```
[bin]$ ./twiddle.sh -H info
Get the metadata for an MBean

usage: info <mbean-name>
  Use '*' to query for all attributes
[bin]$ Description: Management Bean.
+++ Attributes:
 Name: ServerAddress
 Type: java.lang.String
 Access: rw
 Name: RMIClientSocketFactoryBean
 Type: java.rmi.server.RMIClientSocketFactory
 Access: rw
 Name: StateString
 Type: java.lang.String
 Access: r-
 Name: State
 Type: int
 Access: r-
 Name: RMIServerSocketFactoryBean
 Type: java.rmi.server.RMIServerSocketFactory
 Access: rw
 Name: EnableClassCaching
 Type: boolean
 Access: rw
 Name: SecurityDomain
 Type: java.lang.String
 Access: rw
 Name: RMIServerSocketFactory
 Type: java.lang.String
 Access: rw
```

```
 Name: Backlog
 Type: int
 Access: rw
 Name: RMIObjectPort
 Type: int
 Access: rw
 Name: Name
 Type: java.lang.String
 Access: r-
 Name: RMIClientSocketFactory
 Type: java.lang.String
 Access: rw
+++ Operations:
 void start()
 void jbossInternalLifecycle(java.lang.String java.lang.String)
 void create()
 void stop()
 void destroy()
```

To invoke an operation on an MBean, use the invoker command:

```
[bin]$ ./twiddle.sh -H invoke
Invoke an operation on an MBean

usage: invoke [options] <query> <operation> (<arg>)*

options:
    -q, --query-type[=<type>]    Treat object name as a query
    --                           Stop processing options

query type:
    f[irst]    Only invoke on the first matching name [default]
    a[ll]      Invoke on all matching names
[bin]$ ./twiddle.sh invoke jboss:service=JNDIView list true
<h1>java: Namespace</h1>
<pre>
  +- XAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
  +- DefaultDS (class: javax.sql.DataSource)
  +- SecurityProxyFactory (class:
org.jboss.security.SubjectSecurityProxyFactory)
  +- DefaultJMSProvider (class: org.jboss.jms.jndi.JNDIProviderAdapter)
  +- comp (class: javax.naming.Context)
  +- JmsXA (class: org.jboss.resource.adapter.jms.JmsConnectionFactoryImpl)
  +- ConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
  +- jaas (class: javax.naming.Context)
  |   +- JmsXARealm (class:
org.jboss.security.plugins.SecurityDomainContext)
  |   +- jbossmq (class: org.jboss.security.plugins.SecurityDomainContext)
  |   +- HsqlDbRealm (class:
org.jboss.security.plugins.SecurityDomainContext)
  +- timedCacheFactory (class: javax.naming.Context)
Failed to lookup: timedCacheFactory, errmsg=null
  +- TransactionPropagationContextExporter (class:
org.jboss.tm.TransactionPropagationContext
Factory)
```

```
  +- StdJMSPool (class: org.jboss.jms.asf.StdServerSessionPoolFactory)
  +- Mail (class: javax.mail.Session)
  +- TransactionPropagationContextImporter (class:
org.jboss.tm.TransactionPropagationContext
Importer)
  +- TransactionManager (class: org.jboss.tm.TxManager)
</pre>
<h1>Global JNDI Namespace</h1>
<pre>
  +- XAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
  +- UIL2ConnectionFactory[link -> ConnectionFactory] (class:
javax.naming.LinkRef)
  +- UserTransactionSessionFactory (proxy: $Proxy11 implements interface
org.jboss.tm.usertx.
interfaces.UserTransactionSessionFactory)
  +- HTTPConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
  +- console (class: org.jnp.interfaces.NamingContext)
  |   +- PluginManager (proxy: $Proxy36 implements interface
org.jboss.console.manager.Plugin
ManagerMBean)
  +- UIL2XAConnectionFactory[link -> XAConnectionFactory] (class:
javax.naming.LinkRef)
  +- UUIDKeyGeneratorFactory (class:
org.jboss.ejb.plugins.keygenerator.uuid.UUIDKeyGenerator
Factory)
  +- HTTPXAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
  +- topic (class: org.jnp.interfaces.NamingContext)
  |   +- testDurableTopic (class: org.jboss.mq.SpyTopic)
  |   +- testTopic (class: org.jboss.mq.SpyTopic)
  |   +- securedTopic (class: org.jboss.mq.SpyTopic)
  +- queue (class: org.jnp.interfaces.NamingContext)
  |   +- A (class: org.jboss.mq.SpyQueue)
  |   +- testQueue (class: org.jboss.mq.SpyQueue)
  |   +- ex (class: org.jboss.mq.SpyQueue)
  |   +- DLQ (class: org.jboss.mq.SpyQueue)
  |   +- D (class: org.jboss.mq.SpyQueue)
  |   +- C (class: org.jboss.mq.SpyQueue)
  |   +- B (class: org.jboss.mq.SpyQueue)
  +- ConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
  +- UserTransaction (class:
org.jboss.tm.usertx.client.ClientUserTransaction)
  +- jmx (class: org.jnp.interfaces.NamingContext)
  |   +- invoker (class: org.jnp.interfaces.NamingContext)
  |   |   +- RMIAdaptor (proxy: $Proxy35 implements interface
org.jboss.jmx.adaptor.rmi.RMIAd
aptor,interface org.jboss.jmx.adaptor.rmi.RMIAdaptorExt)
  |   +- rmi (class: org.jnp.interfaces.NamingContext)
  |   |   +- RMIAdaptor[link -> jmx/invoker/RMIAdaptor] (class:
javax.naming.LinkRef)
  +- HiLoKeyGeneratorFactory (class:
org.jboss.ejb.plugins.keygenerator.hilo.HiLoKeyGenerator
Factory)
  +- UILXAConnectionFactory[link -> XAConnectionFactory] (class:
javax.naming.LinkRef)
  +- UILConnectionFactory[link -> ConnectionFactory] (class:
javax.naming.LinkRef)
</pre>
```

3.4. Connecting to JMX Using Any Protocol

With the detached invokers and a somewhat generalized proxy factory capability, you can really talk to the JMX server using the `InvokerAdaptorService` and a proxy factory service to expose an `RMIAdaptor` or similar interface over your protocol of choice. We will introduce the detached invoker notion along with proxy factories in *Section 6, "Remote Access to Services, Detached Invokers"*. See *Section 6.1, "A Detached Invoker Example, the MBeanServer Invoker Adaptor Service"* for an example of an invoker service that allows one to access the MBean server using to the `RMIAdaptor` interface over any protocol for which a proxy factory service exists.

4. Using JMX as a Microkernel

When JBoss starts up, one of the first steps performed is to create an MBean server instance (`javax.management.MBeanServer`). The JMX MBean server in the JBoss architecture plays the role of a microkernel. All other manageable MBean components are plugged into JBoss by registering with the MBean server. The kernel in that sense is only an framework, and not a source of actual functionality. The functionality is provided by MBeans, and in fact all major JBoss components are manageable MBeans interconnected through the MBean server.

4.1. The Startup Process

In this section we will describe the JBoss server startup process. A summary of the steps that occur during the JBoss server startup sequence is:

1. The run start script initiates the boot sequence using the `org.jboss.Main.main(String[])` method entry point.
2. The `Main.main` method creates a thread group named `jboss` and then starts a thread belonging to this thread group. This thread invokes the Main.boot method.
3. The `Main.boot` method processes the `Main.main` arguments and then creates an `org.jboss.system.server.ServerLoader` using the system properties along with any additional properties specified as arguments.
4. The XML parser libraries, `jboss-jmx.jar`, `concurrent.jar` and extra libraries and classpaths given as arguments are registered with the `ServerLoader` .
5. The JBoss server instance is created using the `ServerLoader.load(ClassLoader)` method with the current thread context class loader passed in as the `ClassLoader` argument. The returned server instance is an implementation of the `org.jboss.system.server.Server` interface. The creation of the server instance entails:
   - Creating a `java.net.URLClassLoader` with the URLs of the jars and directories registered with the `ServerLoader` . This `URLClassLoader` uses the `ClassLoader` passed in as its

parent and it is pushed as the thread context class loader.

- The class name of the implementation of the `Server` interface to use is determined by the `jboss.server.type` property. This defaults to `org.jboss.system.server.ServerImpl`.
- The `Server` implementation class is loaded using the `URLClassLoader` created in step 6 and instantiated using its no-arg constructor. The thread context class loader present on entry into the `ServerLoader.load` method is restored and the server instance is returned.

6. The server instance is initialized with the properties passed to the `ServerLoader` constructor using the `Server.init(Properties)` method.
7. The server instance is then started using the `Server.start()` method. The default implementation performs the following steps:
   - Set the thread context class loader to the class loader used to load the `ServerImpl` class.
   - Create an `MBeanServer` under the `jboss` domain using the `MBeanServerFactory.createMBeanServer(String)` method.
   - Register the `ServerImpl` and `ServerConfigImpl` MBeans with the MBean server.
   - Initialize the unified class loader repository to contain all JARs in the optional patch directory as well as the server configuration file conf directory, for example, `server/production/conf`. For each JAR and directory an `org.jboss.mx.loading.UnifiedClassLoader` is created and registered with the unified repository. One of these `UnifiedClassLoader` is then set as the thread context class loader. This effectively makes all `UnifiedClassLoaders` available through the thread context class loader.
   - The `org.jboss.system.ServiceController` MBean is created. The `ServiceController` manages the JBoss MBean services life cycle. We will discuss the JBoss MBean services notion in detail in *Section 4.2, "JBoss MBean Services"*.
   - The `org.jboss.deployment.MainDeployer` is created and started. The `MainDeployer` manages deployment dependencies and directing deployments to the correct deployer.
   - The `org.jboss.deployment.JARDeployer` is created and started. The `JARDeployer` handles the deployment of JARs that are simple library JARs.
   - The `org.jboss.deployment.SARDeployer` is created and started. The SARDeployer handles the deployment of JBoss MBean services.
   - The `MainDeployer` is invoked to deploy the services defined in the `conf/jboss-service.xml` of the current server file set.
   - Restore the thread context class loader.

The JBoss server starts out as nothing more than a container for the JMX MBean server, and then loads its personality based on the services defined in the `jboss-service.xml` MBean

configuration file from the named configuration set passed to the server on the command line. Because MBeans define the functionality of a JBoss server instance, it is important to understand how the core JBoss MBeans are written, and how you should integrate your existing services into JBoss using MBeans. This is the topic of the next section.

4.2. JBoss MBean Services

As we have seen, JBoss relies on JMX to load in the MBean services that make up a given server instance's personality. All of the bundled functionality provided with the standard JBoss distribution is based on MBeans. The best way to add services to the JBoss server is to write your own JMX MBeans.

There are two classes of MBeans: those that are independent of JBoss services, and those that are dependent on JBoss services. MBeans that are independent of JBoss services are the trivial case. They can be written per the JMX specification and added to a JBoss server by adding an mbean tag to the `deploy/user-service.xml` file. Writing an MBean that relies on a JBoss service such as naming requires you to follow the JBoss service pattern. The JBoss MBean service pattern consists of a set of life cycle operations that provide state change notifications. The notifications inform an MBean service when it can create, start, stop, and destroy itself. The management of the MBean service life cycle is the responsibility of three JBoss MBeans: `SARDeployer`, `ServiceConfigurator` and `ServiceController`.

4.2.1. The SARDeployer MBean

JBoss manages the deployment of its MBean services via a custom MBean that loads an XML variation of the standard JMX MLet configuration file. This custom MBean is implemented in the `org.jboss.deployment.SARDeployer` class. The `SARDeployer` MBean is loaded when JBoss starts up as part of the bootstrap process. The SAR acronym stands for *service archive*.

The `SARDeployer` handles services archives. A service archive can be either a jar that ends with a `.sar` suffix and contains a `META-INF/jboss-service.xml` descriptor, or a standalone XML descriptor with a naming pattern that matches `*-service.xml`. The DTD for the service descriptor is `jboss-service_4.2.dtd` and is shown in *Figure 3.15, “The DTD for the MBean service descriptor parsed by the SARDeployer”*.

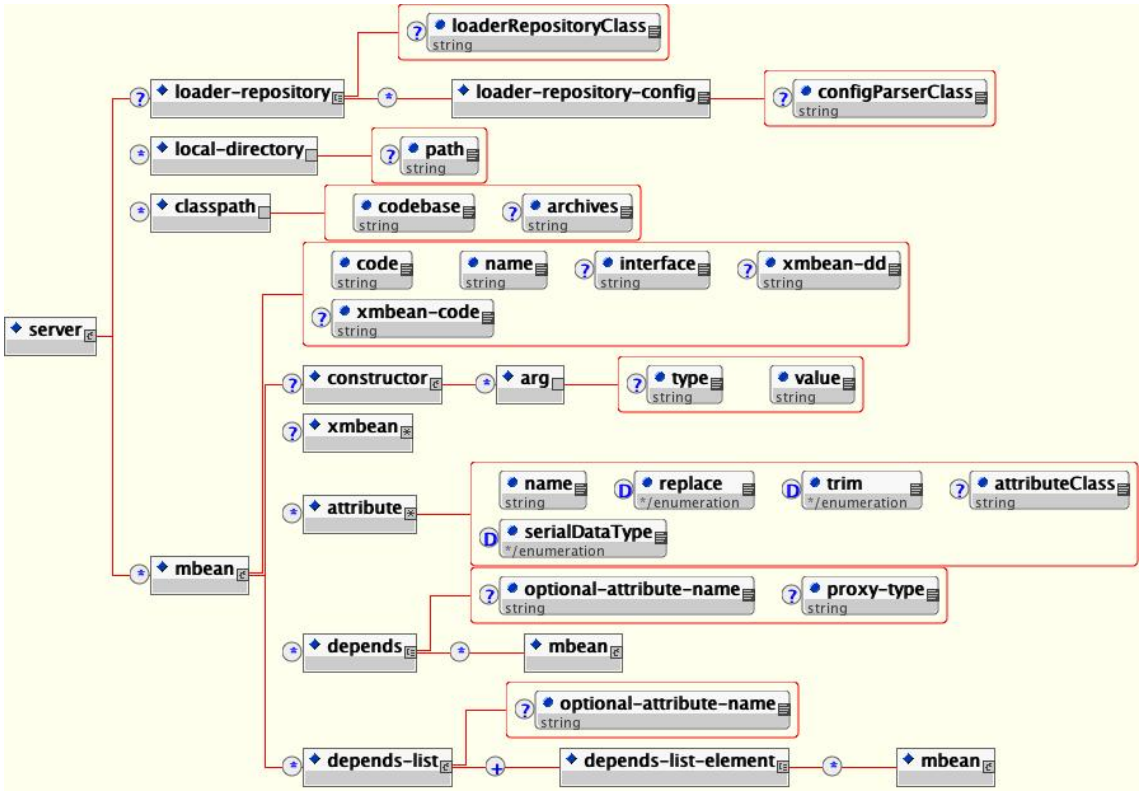


Figure 3.15. The DTD for the MBean service descriptor parsed by the SARDeployer

The elements of the DTD are:

- **loader-repository**: This element specifies the name of the `UnifiedLoaderRepository` MBean to use for the SAR to provide SAR level scoping of classes deployed in the sar. It is a unique JMX `ObjectName` string. It may also specify an arbitrary configuration by including a `loader-repository-config` element. The optional `loaderRepositoryClass` attribute specifies the fully qualified name of the loader repository implementation class. It defaults to `org.jboss.mx.loading.HeirachicalLoaderRepository3`.
  - **loader-repository-config**: This optional element specifies an arbitrary configuration that may be used to configure the `loadRepositoryClass`. The optional `configParserClass` attribute gives the fully qualified name of the `org.jboss.mx.loading.LoaderRepositoryFactory.LoaderRepositoryConfigParser` implementation to use to parse the `loader-repository-config` content.
- **local-directory**: This element specifies a path within the deployment archive that should be copied to the `server/<config>/db` directory for use by the MBean. The path attribute is the name of an entry within the deployment archive.

- **classpath**: This element specifies one or more external JARs that should be deployed with the MBean(s). The optional archives attribute specifies a comma separated list of the JAR names to load, or the * wild card to signify that all jars should be loaded. The wild card only works with file URLs, and http URLs if the web server supports the WEBDAV protocol. The codebase attribute specifies the URL from which the JARs specified in the archive attribute should be loaded. If the codebase is a path rather than a URL string, the full URL is built by treating the codebase value as a path relative to the JBoss distribution `server/<config>` directory. The order of JARs specified in the archives as well as the ordering across multiple classpath element is used as the classpath ordering of the JARs. Therefore, if you have patches or inconsistent versions of classes that require a certain ordering, use this feature to ensure the correct ordering.

- **mbean**: This element specifies an MBean service. The required code attribute gives the fully qualified name of the MBean implementation class. The required name attribute gives the JMX `ObjectName` of the MBean. The optional `xmbean-dd` attribute specifies the path to the XMBean resource if this MBean service uses the JBoss XMBean descriptor to define a Model MBean management interface.

  - **constructor**: The `constructor` element defines a non-default constructor to use when instantiating the MBean The `arg` element specify the constructor arguments in the order of the constructor signature. Each `arg` has a `type` and `value` attribute.

  - **attribute**: Each attribute element specifies a name/value pair of the attribute of the MBean. The name of the attribute is given by the name attribute, and the attribute element body gives the value. The body may be a text representation of the value, or an arbitrary element and child elements if the type of the MBean attribute is `org.w3c.dom.Element`. For text values, the text is converted to the attribute type using the JavaBean `java.beans.PropertyEditor` mechanism.

  - **server/mbean/depends** and **server/mbean/depends-list**: these elements specify a dependency from the MBean using the element to the MBean(s) named by the `depends` or `depends-list` elements. *Section 4.2.4, “Specifying Service Dependencies”*. Note that the dependency value can be another mbean element which defines a nested mbean.

MBean attribute values don't need to be hardcoded literal strings. Service files may contain references to system properties using the `${name}` notation, where `name` is the name of a Java system property. The value of this system property, as would be returned from the call `System.getProperty("name")`. Multiple properties can be specified separated by commas like `${name1,name2,name3}`. If there is no system property named `name1`, `name2` will be tried and then `name3`. This allows multiple levels of substitution to be used. Finally, a default value can be added using a colon separator. The substitution `${name:default value}` would substitute the the text "`default value`" if the system property `name` didn't exist. If none of the listed properties exist and no default value is given, no substitution will occur.

When the `SARDeployer` is asked to deploy a service performs several steps. *Figure 3.16, “A sequence diagram highlighting the main activities performed by the SARDeployer to start a JBoss MBean service”* is a sequence diagram that shows the init through start phases of a service.

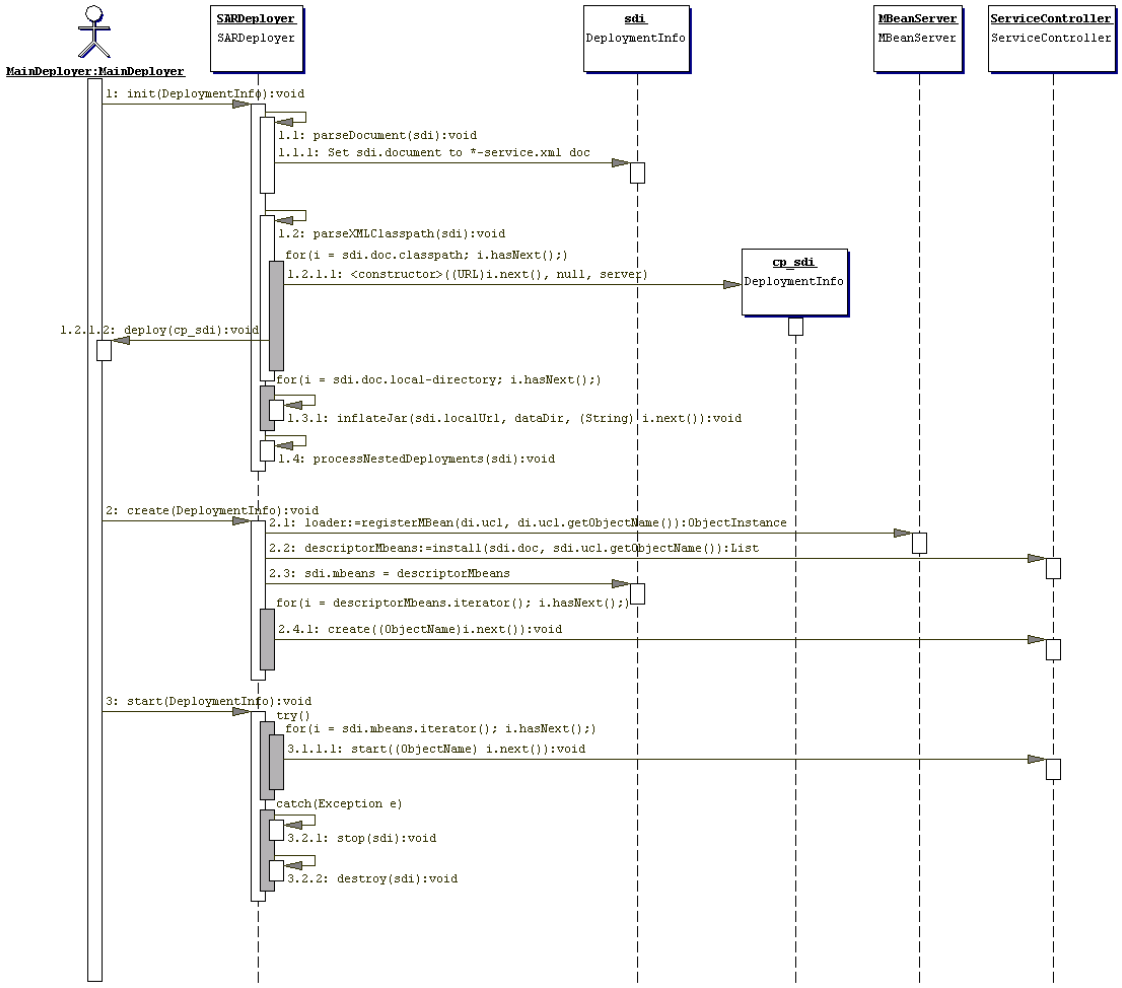


Figure 3.16. A sequence diagram highlighting the main activities performed by the SARDeployer to start a JBoss MBean service

In *Figure 3.16, "A sequence diagram highlighting the main activities performed by the SARDeployer to start a JBoss MBean service"* the following is illustrated:

- Methods prefixed with 1.1 correspond to the load and parse of the XML service descriptor.
- Methods prefixed with 1.2 correspond to processing each classpath element in the service descriptor to create an independent deployment that makes the jar or directory available through a `UnifiedClassLoader` registered with the unified loader repository.
- Methods prefixed with 1.3 correspond to processing each `local-directory` element in the service descriptor. This does a copy of the SAR elements specified in the path attribute to the `server/<config>/db` directory.
- Method 1.4. Process each deployable unit nested in the service a child deployment is created

and added to the service deployment info subdeployment list.

- Method 2.1. The `UnifiedClassLoader` of the SAR deployment unit is registered with the MBean Server so that is can be used for loading of the SAR MBeans.
- Method 2.2. For each MBean element in the descriptor, create an instance and initialize its attributes with the values given in the service descriptor. This is done by calling the `ServiceController.install` method.
- Method 2.4.1. For each MBean instance created, obtain its JMX `ObjectName` and ask the ServiceController to handle the create step of the service life cycle. The `ServiceController` handles the dependencies of the MBean service. Only if the service's dependencies are satisfied is the service create method invoked.
- Methods prefixed with 3.1 correspond to the start of each MBean service defined in the service descriptor. For each MBean instance created, obtain its JMX ObjectName and ask the `ServiceController` to handle the start step of the service life cycle. The `ServiceController` handles the dependencies of the MBean service. Only if the service's dependencies are satisfied is the service start method invoked.

4.2.2. The Service Life Cycle Interface

The JMX specification does not define any type of life cycle or dependency management for MBeans. The JBoss ServiceController MBean introduces this notion. A JBoss MBean is an extension of the JMX MBean in that an MBean is expected to decouple creation from the life cycle of its service duties. This is necessary to implement any type of dependency management. For example, if you are writing an MBean that needs a JNDI naming service to be able to function, your MBean needs to be told when its dependencies are satisfied. This ranges from difficult to impossible to do if the only life cycle event is the MBean constructor. Therefore, JBoss introduces a service life cycle interface that describes the events a service can use to manage its behavior. The following listing shows the `org.jboss.system.Service` interface:

```
package org.jboss.system;
public interface Service
{
    public void create() throws Exception;
    public void start() throws Exception;
    public void stop();
    public void destroy();
}
```

The `ServiceController` MBean invokes the methods of the `Service` interface at the appropriate times of the service life cycle. We'll discuss the methods in more detail in the `ServiceController` section.

4.2.3. The ServiceController MBean

JBoss manages dependencies between MBeans via the

`org.jboss.system.ServiceController` custom MBean. The SARDeployer delegates to the ServiceController when initializing, creating, starting, stopping and destroying MBean services. *Figure 3.17, "The interaction between the SARDeployer and ServiceController to start a service"* shows a sequence diagram that highlights interaction between the `SARDeployer` and `ServiceController`.

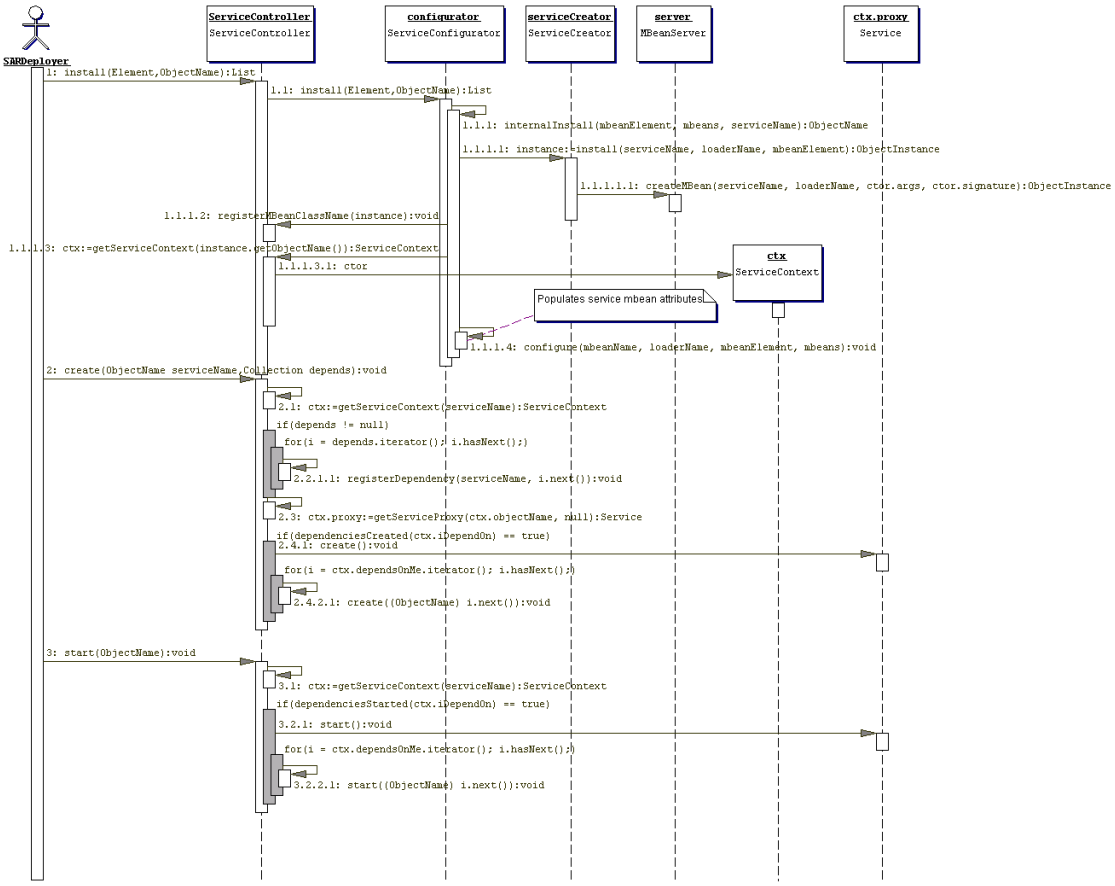


Figure 3.17. The interaction between the SARDeployer and ServiceController to start a service

The `ServiceController` MBean has four key methods for the management of the service life cycle: `create`, `start`, `stop` and `destroy`.

4.2.3.1. The create(ObjectName) method

The `create(ObjectName)` method is called whenever an event occurs that affects the named services state. This could be triggered by an explicit invocation by the `SARDeployer`, a notification of a new class, or another service reaching its created state.

When a service's `create` method is called, all services on which the service depends have also had their create method invoked. This gives an MBean an opportunity to check that required MBeans or resources exist. A service cannot utilize other MBean services at this point, as most JBoss MBean services do not become fully functional until they have been started via their

start method. Because of this, service implementations often do not implement create in favor of just the start method because that is the first point at which the service can be fully functional.

4.2.3.2. The start(ObjectName) method

The start(ObjectName) method is called whenever an event occurs that affects the named services state. This could be triggered by an explicit invocation by the SARDeployer, a notification of a new class, or another service reaching its started state.

When a service's start method is called, all services on which the service depends have also had their start method invoked. Receipt of a start method invocation signals a service to become fully operational since all services upon which the service depends have been created and started.

4.2.3.3. The stop(ObjectName) method

The stop(ObjectName) method is called whenever an event occurs that affects the named services state. This could be triggered by an explicit invocation by the SARDeployer, notification of a class removal, or a service on which other services depend reaching its stopped state.

4.2.3.4. The destroy(ObjectName) method

The destroy(ObjectName) method is called whenever an event occurs that affects the named services state. This could be triggered by an explicit invocation by the SARDeployer, notification of a class removal, or a service on which other services depend reaching its destroyed state.

Service implementations often do not implement destroy in favor of simply implementing the stop method, or neither stop nor destroy if the service has no state or resources that need cleanup.

4.2.4. Specifying Service Dependencies

To specify that an MBean service depends on other MBean services you need to declare the dependencies in the mbean element of the service descriptor. This is done using the depends and depends-list elements. One difference between the two elements relates to the optional-attribute-name attribute usage. If you track the ObjectNames of dependencies using single valued attributes you should use the depends element. If you track the ObjectNames of dependencies using java.util.List compatible attributes you would use the depends-list element. If you only want to specify a dependency and don't care to have the associated service ObjectName bound to an attribute of your MBean then use whatever element is easiest. The following listing shows example service descriptor fragments that illustrate the usage of the dependency related elements.

```
<mbean code="org.jboss.mq.server.jmx.Topic"
       name="jms.topic:service=Topic,name=testTopic">
    <!-- Declare a dependency on the "jboss.mq:service=DestinationManager"
and
         bind this name to the DestinationManager attribute -->
    <depends optional-attribute-name="DestinationManager">
```

```
        jboss.mq:service=DestinationManager
    </depends>

    <!-- Declare a dependency on the "jboss.mq:service=SecurityManager" and
         bind this name to the SecurityManager attribute -->
    <depends optional-attribute-name="SecurityManager">
        jboss.mq:service=SecurityManager
    </depends>

    <!-- ... -->

    <!-- Declare a dependency on the
         "jboss.mq:service=CacheManager" without
         any binding of the name to an attribute-->
    <depends>jboss.mq:service=CacheManager</depends>
</mbean>

<mbean code="org.jboss.mq.server.jmx.TopicMgr"
       name="jboss.mq.destination:service=TopicMgr">
    <!-- Declare a dependency on the given topic destination mbeans and
         bind these names to the Topics attribute -->
    <depends-list optional-attribute-name="Topics">
<depends-list-element>jms.topic:service=Topic,name=A</depends-list-element>
<depends-list-element>jms.topic:service=Topic,name=B</depends-list-element>
<depends-list-element>jms.topic:service=Topic,name=C</depends-list-element>
    </depends-list>
</mbean>
```

Another difference between the `depends` and `depends-list` elements is that the value of the depends element may be a complete MBean service configuration rather than just the `ObjectName` of the service. *Example 3.13, "An example of using the depends element to specify the complete configuration of a depended on service."* shows an example from the `hsqldb-service.xml` descriptor. In this listing the `org.jboss.resource.connectionmanager.RARDeployment` service configuration is defined using a nested `mbean` element as the `depends` element value. This indicates that the `org.jboss.resource.connectionmanager.LocalTxConnectionManager` MBean depends on this service. The `jboss.jca:service=LocalTxDS,name=hsqldbDSObjectName` will be bound to the `ManagedConnectionFactoryName` attribute of the `LocalTxConnectionManager` class.

```
<mbean code="org.jboss.resource.connectionmanager.LocalTxConnectionManager"
       name="jboss.jca:service=LocalTxCM,name=hsqldbDS">
    <depends optional-attribute-name="ManagedConnectionFactoryName">
        <!--embedded mbean-->
        <mbean code="org.jboss.resource.connectionmanager.RARDeployment"
               name="jboss.jca:service=LocalTxDS,name=hsqldbDS">
            <attribute name="JndiName">DefaultDS</attribute>
            <attribute name="ManagedConnectionFactoryProperties">
                <properties>
                    <config-property name="ConnectionURL"
                                     type="java.lang.String">
                        jdbc:hsqldb:hsql://localhost:1476
                    </config-property>
```

```
                    <config-property name="DriverClass"
type="java.lang.String">
                        org.hsqldb.jdbcDriver
                    </config-property>
                    <config-property name="UserName"
type="java.lang.String">
                        sa
                    </config-property>
                    <config-property name="Password"
type="java.lang.String"/>
                </properties>
            </attribute>
            <!-- ... -->
        </mbean>
    </depends>
    <!-- ... -->
</mbean>
```

Example 3.13. An example of using the depends element to specify the complete configuration of a depended on service.

4.2.5. Identifying Unsatisfied Dependencies

The `ServiceController` MBean supports two operations that can help determine which MBeans are not running due to unsatisfied dependencies. The first operation is `listIncompletelyDeployed`. This returns a `java.util.List` of `org.jboss.system.ServiceContext` objects for the MBean services that are not in the `RUNNING` state.

The second operation is `listWaitingMBeans`. This operation returns a `java.util.List` of the JMX `ObjectName`s of MBean services that cannot be deployed because the class specified by the code attribute is not available.

4.2.6. Hot Deployment of Components, the URLDeploymentScanner

The `URLDeploymentScanner` MBean service provides the JBoss hot deployment capability. This service watches one or more URLs for deployable archives and deploys the archives as they appear or change. It also undeploys previously deployed applications if the archive from which the application was deployed is removed. The configurable attributes include:

- **URLs**: A comma separated list of URL strings for the locations that should be watched for changes. Strings that do not correspond to valid URLs are treated as file paths. Relative file paths are resolved against the server home URL, for example, `JBOSS_DIST/server/production` for the production config file set. If a URL represents a file then the file is deployed and watched for subsequent updates or removal. If a URL ends in `/`

to represent a directory, then the contents of the directory are treated as a collection of deployables and scanned for content that are to be watched for updates or removal. The requirement that a URL end in a `/` to identify a directory follows the RFC2518 convention and allows discrimination between collections and directories that are simply unpacked archives.

The default value for the URLs attribute is `deploy/` which means that any SARs, EARs, JARs, WARs, RARs, etc. dropped into the `server/<name>/deploy` directory will be automatically deployed and watched for updates.

Example URLs include:

- **deploy/** scans `${jboss.server.url}/deploy/`, which is local or remote depending on the URL used to boot the server
- **${jboss.server.home.dir}/deploy/** scans *${jboss.server.home.dir)/deploy*, which is always local
- **file:/var/opt/myapp.ear** deploys `myapp.ear` from a local location
- **file:/var/opt/apps/** scans the specified directory
- **http://www.test.com/netboot/myapp.ear** deploys `myapp.ear` from a remote location
- **http://www.test.com/netboot/apps/** scans the specified remote location using WebDAV. This will only work if the remote http server supports the WebDAV PROPFIND command.

- **ScanPeriod**: The time in milliseconds between runs of the scanner thread. The default is 5000 (5 seconds).
- **URLComparator**: The class name of a `java.util.Comparator` implementation used to specify a deployment ordering for deployments found in a scanned directory. The implementation must be able to compare two `java.net.URL` objects passed to its compare method. The default setting is the `org.jboss.deployment.DeploymentSorter` class which orders based on the deployment URL suffix. The ordering of suffixes is: `deployer`, `deployer.xml`, `sar`, `rar`, `ds.xml`, `service.xml`, `har`, `jar`, `war`, `wsr`, `ear`, `zip`, `bsh`, `last`.

  An alternate implementation is the `org.jboss.deployment.scanner.PrefixDeploymentSorter` class. This orders the URLs based on numeric prefixes. The prefix digits are converted to an int (ignoring leading zeroes), smaller prefixes are ordered ahead of larger numbers. Deployments that do not start with any digits will be deployed after all numbered deployments. Deployments with the same prefix value are further sorted by the `DeploymentSorter` logic.
- **Filter**: The class name of a `java.io.FileFilter` implementation that is used to filter the contents of scanned directories. Any file not accepted by this filter will not be deployed. The default is `org.jboss.deployment.scanner.DeploymentFilter` which is an implementation that rejects the following patterns:

  `"#*", "%*", ",*", ".*", "_$*", "*#", "*$", "*%", "*.BAK", "*.old", "*.orig", "*.rej", "*.bak", "*.sh", "*,v", "*~", ".make.state", ".nse_depinfo", "CVS", "CVS.admin", "RCS", "RCSLOG",`

SCCS", "TAGS", "core", "tags"

- **RecursiveSearch**: This property indicates whether or not deploy subdirectories are seen to be holding deployable content. If this is false, deploy subdirectories that do not contain a dot (.) in their name are seen to be unpackaged JARs with nested subdeployments. If true, then deploy subdirectories are just groupings of deployable content. The difference between the two views shows is related to the depth first deployment model JBoss supports. The false setting which treats directories as unpackaged JARs with nested content triggers the deployment of the nested content as soon as the JAR directory is deployed. The true setting simply ignores the directory and adds its content to the list of deployable packages and calculates the order based on the previous filter logic. The default is true.

- **Deployer**: The JMX `ObjectName` string of the MBean that implements the `org.jboss.deployment.Deployer` interface operations. The default setting is to use the `MainDeployer` created by the bootstrap startup process.

4.3. Writing JBoss MBean Services

Writing a custom MBean service that integrates into the JBoss server requires the use of the `org.jboss.system.Service` interface pattern if the custom service is dependent on other services. When a custom MBean depends on other MBean services you cannot perform any service dependent initialization in any of the `javax.management.MBeanRegistration` interface methods since JMX has no dependency notion. Instead, you must manage dependency state using the `Service` interface `create` and/or `start` methods. You can do this using any one of the following approaches:

- Add any of the `Service` methods that you want called on your MBean to your MBean interface. This allows your MBean implementation to avoid dependencies on JBoss specific interfaces.

- Have your MBean interface extend the `org.jboss.system.Service` interface.

- Have your MBean interface extend the `org.jboss.system.ServiceMBean` interface. This is a subinterface of `org.jboss.system.Service` that adds `getName()`, `getState()`, `getStateString()` methods.

Which approach you choose depends on whether or not you want your code to be coupled to JBoss specific code. If you don't, then you would use the first approach. If you don't care about dependencies on JBoss classes, the simplest approach is to have your MBean interface extend from `org.jboss.system.ServiceMBean` and your MBean implementation class extend from the abstract `org.jboss.system.ServiceMBeanSupport` class. This class implements the `org.jboss.system.ServiceMBean` interface. `ServiceMBeanSupport` provides implementations of the `create`, `start`, `stop`, and `destroy` methods that integrate logging and JBoss service state management tracking. Each method delegates any subclass specific work to `createService`, `startService`, `stopService`, and `destroyService` methods respectively. When subclassing `ServiceMBeanSupport`, you would override one or more of the

`createService`, `startService`, `stopService`, and `destroyService` methods as required

4.3.1. A Standard MBean Example

This section develops a simple MBean that binds a `HashMap` into the JBoss JNDI namespace at a location determined by its `JndiName` attribute to demonstrate what is required to create a custom MBean. Because the MBean uses JNDI, it depends on the JBoss naming service MBean and must use the JBoss MBean service pattern to be notified when the naming service is available.

Version one of the classes, shown in *Example 3.14, "JNDIMapMBean interface and implementation based on the service interface method pattern"*, is based on the service interface method pattern. This version of the interface declares the `start` and `stop` methods needed to start up correctly without using any JBoss-specific classes.

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

// The JNDIMap MBean interface
import javax.naming.NamingException;

public interface JNDIMapMBean
{
    public String getJndiName();
    public void setJndiName(String jndiName) throws NamingException;
    public void start() throws Exception;
    public void stop() throws Exception;
}
```

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

// The JNDIMap MBean implementation
import java.util.HashMap;
import javax.naming.InitialContext;
import javax.naming.Name;
import javax.naming.NamingException;
import org.jboss.naming.NonSerializableFactory;

public class JNDIMap implements JNDIMapMBean
{
    private String jndiName;
    private HashMap contextMap = new HashMap();
    private boolean started;

    public String getJndiName()
    {
        return jndiName;
    }
    public void setJndiName(String jndiName) throws NamingException
    {
        String oldName = this.jndiName;
        this.jndiName = jndiName;
        if (started) {
            unbind(oldName);
```

```
            try {
                rebind();
            } catch(Exception e) {
                NamingException ne = new NamingException("Failedto update
jndiName");
                ne.setRootCause(e);
                throw ne;
            }
        }
    }

    public void start() throws Exception
    {
        started = true;
        rebind();
    }

    public void stop()
    {
        started = false;
        unbind(jndiName);
    }

    private void rebind() throws NamingException
    {
        InitialContext rootCtx = new InitialContext();
        Name fullName = rootCtx.getNameParser("").parse(jndiName);
        System.out.println("fullName="+fullName);
        NonSerializableFactory.rebind(fullName, contextMap, true);
    }

    private void unbind(String jndiName)
    {
        try {
            InitialContext rootCtx = new InitialContext();
            rootCtx.unbind(jndiName);
            NonSerializableFactory.unbind(jndiName);
        } catch(NamingException e) {
            e.printStackTrace();
        }
    }
}
```

Example 3.14. JNDIMapMBean interface and implementation based on the service interface method pattern

Version two of the classes, shown in *Example 3.14, "JNDIMapMBean interface and implementation based on the service interface method pattern"*, use the JBoss `ServiceMBean` interface and `ServiceMBeanSupport` class. In this version, the implementation class extends the `ServiceMBeanSupport` class and overrides the `startService` and `stopService` methods. `JNDIMapMBean` also implements the abstract `getName` method to return a descriptive name for

the MBean. The `JNDIMapMBean` interface extends the `org.jboss.system.ServiceMBean` interface and only declares the setter and getter methods for the `JndiName` attribute because it inherits the service life cycle methods from `ServiceMBean`. This is the third approach mentioned at the start of the *Section 4.2, "JBoss MBean Services"*.

```
package org.jboss.book.jmx.ex2;

// The JNDIMap MBean interface
import javax.naming.NamingException;

public interface JNDIMapMBean extends org.jboss.system.ServiceMBean
{
    public String getJndiName();
    public void setJndiName(String jndiName) throws NamingException;
}
```

```
package org.jboss.book.jmx.ex2;
// The JNDIMap MBean implementation
import java.util.HashMap;
import javax.naming.InitialContext;
import javax.naming.Name;
import javax.naming.NamingException;
import org.jboss.naming.NonSerializableFactory;

public class JNDIMap extends org.jboss.system.ServiceMBeanSupport
    implements JNDIMapMBean
{
    private String jndiName;
    private HashMap contextMap = new HashMap();

    public String getJndiName()
    {
        return jndiName;
    }

    public void setJndiName(String jndiName)
        throws NamingException
    {
        String oldName = this.jndiName;
        this.jndiName = jndiName;
        if (super.getState() == STARTED) {
            unbind(oldName);
            try {
                rebind();
            } catch(Exception e) {
                NamingException ne = new NamingException("Failed to update
jndiName");
                ne.setRootCause(e);
                throw ne;
            }
        }
    }

    public void startService() throws Exception
```

```
    {
        rebind();
    }

    public void stopService()
    {
        unbind(jndiName);
    }

    private void rebind() throws NamingException
    {
        InitialContext rootCtx = new InitialContext();
        Name fullName = rootCtx.getNameParser("").parse(jndiName);
        log.info("fullName="+fullName);
        NonSerializableFactory.rebind(fullName, contextMap, true);
    }

    private void unbind(String jndiName)
    {
        try {
            InitialContext rootCtx = new InitialContext();
            rootCtx.unbind(jndiName);
            NonSerializableFactory.unbind(jndiName);
        } catch(NamingException e) {
            log.error("Failed to unbind map", e);
        }
    }
}
```

Example 3.15. JNDIMap MBean interface and implementation based on the ServiceMBean interface and ServiceMBeanSupport class

The source code for these MBeans along with the service descriptors is located in the `examples/src/main/org/jboss/book/jmx/{ex1,ex2}` directories.

The jboss-service.xml descriptor for the first version is shown below.

```
<!-- The SAR META-INF/jboss-service.xml descriptor -->
<server>
    <mbean code="org.jboss.book.jmx.ex1.JNDIMap"
           name="j2eechap2.ex1:service=JNDIMap">
        <attribute name="JndiName">inmemory/maps/MapTest</attribute>
        <depends>jboss:service=Naming</depends>
    </mbean>
</server>
```

The JNDIMap MBean binds a `HashMap` object under the `inmemory/maps/MapTest` JNDI name and the client code fragment demonstrates retrieving the HashMap object from the `inmemory/maps/MapTest` location. The corresponding client code is shown below.

```
// Sample lookup code
InitialContext ctx = new InitialContext();
HashMap map = (HashMap) ctx.lookup("inmemory/maps/MapTest");
```

4.3.2. XMBean Examples

In this section we will develop a variation of the `JNDIMap` MBean introduced in the preceding section that exposes its management metadata using the JBoss XMBean framework. Our core managed component will be exactly the same core code from the `JNDIMap` class, but it will not implement any specific management related interface. We will illustrate the following capabilities not possible with a standard MBean:

- The ability to add rich descriptions to attribute and operations
- The ability to expose notification information
- The ability to add persistence of attributes
- The ability to add custom interceptors for security and remote access through a typed interface

4.3.2.1. Version 1, The Annotated JNDIMap XMBean

Let's start with a simple XMBean variation of the standard MBean version of the JNDIMap that adds the descriptive information about the attributes and operations and their arguments. The following listing shows the `jboss-service.xml` descriptor and the `jndimap-xmbean1.xml` XMBean descriptor. The source can be found in the `src/main/org/jboss/book/jmx/xmbean` directory of the book examples.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE server PUBLIC
                     "-//JBoss//DTD MBean Service 3.2//EN"
                     "http://www.jboss.org/j2ee/dtd/jboss-service_3_2.dtd">
<server>
    <mbean code="org.jboss.book.jmx.xmbean.JNDIMap"
           name="j2eechap2.xmbean:service=JNDIMap"
           xmbean-dd="META-INF/jndimap-xmbean.xml">
        <attribute name="JndiName">inmemory/maps/MapTest</attribute>
        <depends>jboss:service=Naming</depends>
    </mbean>
</server>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mbean PUBLIC
          "-//JBoss//DTD JBOSS XMBEAN 1.0//EN"
          "http://www.jboss.org/j2ee/dtd/jboss_xmbean_1_0.dtd">
<mbean>
    <description>The JNDIMap XMBean Example Version 1</description>
```

```
    <descriptors>
        <persistence persistPolicy="Never" persistPeriod="10"
            persistLocation="data/JNDIMap.data" persistName="JNDIMap"/>
        <currencyTimeLimit value="10"/>
        <state-action-on-update value="keep-running"/>
    </descriptors>
    <class>org.jboss.test.jmx.xmbean.JNDIMap</class>
    <constructor>
        <description>The default constructor</description>
        <name>JNDIMap</name>
    </constructor>
    <!-- Attributes -->
    <attribute access="read-write" getMethod="getJndiName"
setMethod="setJndiName">
        <description>
            The location in JNDI where the Map we manage will be bound
        </description>
        <name>JndiName</name>
        <type>java.lang.String</type>
        <descriptors>
            <default value="inmemory/maps/MapTest"/>
        </descriptors>
    </attribute>
    <attribute access="read-write" getMethod="getInitialValues"
               setMethod="setInitialValues">
        <description>The array of initial values that will be placed into
the
            map associated with the service. The array is a collection of
            key,value pairs with elements[0,2,4,...2n] being the keys and
            elements [1,3,5,...,2n+1] the associated values. The
            "[Ljava.lang.String;" type signature is the VM representation of
the
            java.lang.String[] type. </description>
        <name>InitialValues</name>
        <type>[Ljava.lang.String;</type>
        <descriptors>
            <default value="key0,value0"/>
        </descriptors>
    </attribute>
    <!-- Operations -->
    <operation>
        <description>The start lifecycle operation</description>
        <name>start</name>
    </operation>
    <operation>
        <description>The stop lifecycle operation</description>
        <name>stop</name>
    </operation>
    <operation impact="ACTION">
        <description>Put a value into the map</description>
        <name>put</name>
        <parameter>
            <description>The key the value will be store under</description>
            <name>key</name>
            <type>java.lang.Object</type>
        </parameter>
        <parameter>
```

```
            <description>The value to place into the map</description>
            <name>value</name>
            <type>java.lang.Object</type>
        </parameter>
    </operation>
    <operation impact="INFO">
        <description>Get a value from the map</description>
        <name>get</name>
        <parameter>
            <description>The key to lookup in the map</description>
            <name>get</name>
            <type>java.lang.Object</type>
        </parameter>
        <return-type>java.lang.Object</return-type>
    </operation>
    <!-- Notifications -->
    <notification>
        <description>The notification sent whenever a value is get into the
map
            managed by the service</description>
        <name>javax.management.Notification</name>
<notification-type>org.jboss.book.jmx.xmbean.JNDIMap.get</notification-type>
    </notification>
    <notification>
        <description>The notification sent whenever a value is put into the
map
            managed by the service</description>
        <name>javax.management.Notification</name>
<notification-type>org.jboss.book.jmx.xmbean.JNDIMap.put</notification-type>
    </notification>
</mbean>
```

You can build, deploy and test the XMBean as follows:

```
[examples]$ ant -Dchap=jmx -Dex=xmbean1 run-example
...
run-examplexmbean1:
     [java] JNDIMap Class: org.jboss.mx.modelmbean.XMBean
     [java] JNDIMap Operations:
     [java]  + void start()
     [java]  + void stop()
     [java]  + void put(java.lang.Object
chap2.xmbean:service=JNDIMap,java.lang.Object
                        chap2.xmbean:service=JNDIMap)
     [java]  + java.lang.Object get(java.lang.Object
chap2.xmbean:service=JNDIMap)
     [java] name=chap2.xmbean:service=JNDIMap
     [java] listener=org.jboss.book.jmx.xmbean.TestXMBean1$Listener@f38cf0
     [java] key=key0, value=value0
     [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:
service=JNDIMap][type=org.jboss.book.jmx.xmbean.JNDIMap.put][message=]
     [java] JNDIMap.put(key1, value1) successful
     [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:
```

```
service=JNDIMap][type=org.jboss.book.jmx.xmbean.JNDIMap.get][message=]
     [java] JNDIMap.get(key0): null
     [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:
service=JNDIMap][type=org.jboss.book.jmx.xmbean.JNDIMap.get][message=]
     [java] JNDIMap.get(key1): value1
     [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:
service=JNDIMap][type=org.jboss.book.jmx.xmbean.JNDIMap.put][message=]
     [java] handleNotification, event:
javax.management.AttributeChangeNotification[source
=chap2.xmbean:service=JNDIMap][type=jmx.attribute.change][message=InitialValues
            changed from javax.management.Attribute@82a72a to
            javax.management.Attribute@acdb96]
```

The functionality is largely the same as the Standard MBean with the notable exception of the JMX notifications. A Standard MBean has no way of declaring that it will emit notifications. An XMBean may declare the notifications it emits using notification elements as is shown in the version 1 descriptor. We see the notifications from the get and put operations on the test client console output. Note that there is also an `jmx.attribute.change notification` emitted when the `InitialValues` attribute was changed. This is because the `ModelMBean` interface extends the `ModelMBeanNotificationBroadcaster` which supports `AttributeChangeNotificationListeners`.

The other major difference between the Standard and XMBean versions of JNDIMap is the descriptive metadata. Look at the `chap2.xmbean:service=JNDIMap` in the JMX Console, and you will see the attributes section as shown in *Figure 3.18, "The Version 1 JNDIMapXMBean jmx-console view"*.

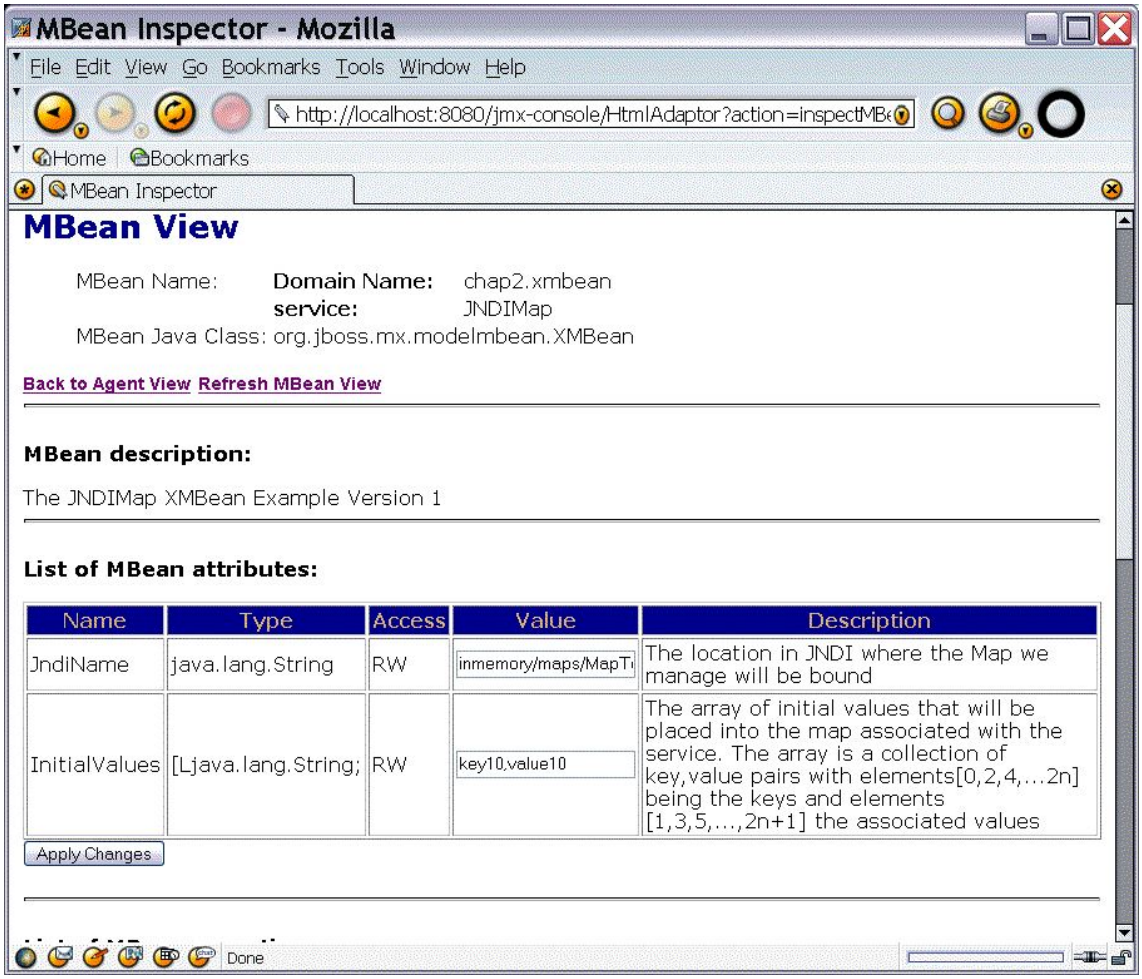


Figure 3.18. The Version 1 JNDIMapXMBean jmx-console view

Notice that the JMX Console now displays the full attribute description as specified in the XMBean descriptor rather than `MBean Attribute` text seen in standard MBean implementations. Scroll down to the operations and you will also see that these now also have nice descriptions of their function and parameters.

4.3.2.2. Version 2, Adding Persistence to the JNDIMap XMBean

In version 2 of the XMBean we add support for persistence of the XMBean attributes. The updated XMBean deployment descriptor is given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mbean PUBLIC
          "-//JBoss//DTD JBOSS XMBEAN 1.0//EN"
          "http://www.jboss.org/j2ee/dtd/jboss_xmbean_1_0.dtd">
<mbean>
    <description>The JNDIMap XMBean Example Version 2</description>
    <descriptors>
        <persistence persistPolicy="OnUpdate" persistPeriod="10"
```

```
            persistLocation="${jboss.server.data.dir}"
persistName="JNDIMap.ser"/>
        <currencyTimeLimit value="10"/>
        <state-action-on-update value="keep-running"/>
        <persistence-manager
value="org.jboss.mx.persistence.ObjectStreamPersistenceManager"/>
    </descriptors>  <class>org.jboss.test.jmx.xmbean.JNDIMap</class>
    <constructor>
        <description>The default constructor</description>
        <name>JNDIMap</name>
    </constructor>
    <!-- Attributes -->
    <attribute access="read-write" getMethod="getJndiName"
setMethod="setJndiName">
        <description>
            The location in JNDI where the Map we manage will be bound
        </description>
        <name>JndiName</name>
        <type>java.lang.String</type>
        <descriptors>
            <default value="inmemory/maps/MapTest"/>
        </descriptors>
    </attribute>
    <attribute access="read-write" getMethod="getInitialValues"
               setMethod="setInitialValues">
        <description>The array of initial values that will be placed into
the
            map associated with the service. The array is a collection of
            key,value pairs with elements[0,2,4,...2n] being the keys and
            elements [1,3,5,...,2n+1] the associated values</description>
        <name>InitialValues</name>
        <type>[Ljava.lang.String;</type>
        <descriptors>
            <default value="key0,value0"/>
        </descriptors>
    </attribute>
    <!-- Operations -->
    <operation>
        <description>The start lifecycle operation</description>
        <name>start</name>
    </operation>
    <operation>
        <description>The stop lifecycle operation</description>
        <name>stop</name>
    </operation>
    <operation impact="ACTION">
        <description>Put a value into the nap</description>
        <name>put</name>
        <parameter>
            <description>The key the value will be store under</description>
            <name>key</name>
            <type>java.lang.Object</type>
        </parameter>
        <parameter>
            <description>The value to place into the map</description>
            <name>value</name>
            <type>java.lang.Object</type>
```

```
        </parameter>
    </operation>
    <operation impact="INFO">
        <description>Get a value from the map</description>
        <name>get</name>
        <parameter>
            <description>The key to lookup in the map</description>
            <name>get</name>
            <type>java.lang.Object</type>
        </parameter>
        <return-type>java.lang.Object</return-type>
    </operation>
    <!-- Notifications -->
    <notification>
        <description>The notification sent whenever a value is get into the
map
            managed by the service</description>
        <name>javax.management.Notification</name>
<notification-type>org.jboss.book.jmx.xmbean.JNDIMap.get</notification-type>
    </notification>
    <notification>
        <description>The notification sent whenever a value is put into the
map
            managed by the service</description>
        <name>javax.management.Notification</name>
<notification-type>org.jboss.book.jmx.xmbean.JNDIMap.put</notification-type>
    </notification>
</mbean>
```

Build, deploy and test the version 2 XMBean as follows:

```
[examples]$ ant -Dchap=jmx -Dex=xmbean2 -Djboss.deploy.conf=rmi-adaptor
run-example
...
run-examplexmbean2:
     [java] JNDIMap Class: org.jboss.mx.modelmbean.XMBean
     [java] JNDIMap Operations:
     [java]  + void start()
     [java]  + void stop()
     [java]  + void put(java.lang.Object
chap2.xmbean:service=JNDIMap,java.lang.Object cha
p2.xmbean:service=JNDIMap)
     [java]  + java.lang.Object get(java.lang.Object
chap2.xmbean:service=JNDIMap)
     [java]  + java.lang.String getJndiName()
     [java]  + void setJndiName(java.lang.String
chap2.xmbean:service=JNDIMap)
     [java]  + [Ljava.lang.String; getInitialValues()
     [java]  + void setInitialValues([Ljava.lang.String;
chap2.xmbean:service=JNDIMap)
     [java] handleNotification, event: null
     [java] key=key10, value=value10
     [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.chap2.xmbean.JNDIMap.put,sequenceNumber=7,timeStamp=10986326
```

```
93716,message=null,userData=null]
     [java] JNDIMap.put(key1, value1) successful
     [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.chap2.xmbean.JNDIMap.get,sequenceNumber=8,timeStamp=10986326
93857,message=null,userData=null]
     [java] JNDIMap.get(key0): null
     [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.chap2.xmbean.JNDIMap.get,sequenceNumber=9,timeStamp=10986326
93896,message=null,userData=null]
     [java] JNDIMap.get(key1): value1
     [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.chap2.xmbean.JNDIMap.put,sequenceNumber=10,timeStamp=1098632
693925,message=null,userData=null]
```

There is nothing manifestly different about this version of the XMBean at this point because we have done nothing to test that changes to attribute value are actually persisted. Perform this test by running example xmbean2a several times:

```
[examples] ant -Dchap=jmx -Dex=xmbean2a run-example
...
     [java] InitialValues.length=2
     [java] key=key10, value=value10
```

```
[examples] ant -Dchap=jmx -Dex=xmbean2a run-example
...
     [java] InitialValues.length=4
     [java] key=key10, value=value10
     [java] key=key2, value=value2
```

```
[examples] ant -Dchap=jmx -Dex=xmbean2a run-example
...
     [java] InitialValues.length=6
     [java] key=key10, value=value10
     [java] key=key2, value=value2
     [java] key=key3, value=value3
```

The `org.jboss.book.jmx.xmbean.TestXMBeanRestart` used in this example obtains the current `InitialValues` attribute setting, and then adds another key/value pair to it. The client code is shown below.

```
package org.jboss.book.jmx.xmbean;

import javax.management.Attribute;
import javax.management.ObjectName;
```

```
import javax.naming.InitialContext;

import org.jboss.jmx.adaptor.rmi.RMIAdaptor;

/**
 *  A client that demonstrates the persistence of the xmbean
 *  attributes. Every time it run it looks up the InitialValues
 *  attribute, prints it out and then adds a new key/value to the
 *  list.
 *
 *  @author Scott.Stark@jboss.org
 *  @version $Revision: 1.9 $
 */
public class TestXMBeanRestart
{
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws Exception
    {
        InitialContext ic = new InitialContext();
        RMIAdaptor server = (RMIAdaptor) ic.lookup("jmx/rmi/RMIAdaptor");

        // Get the InitialValues attribute
        ObjectName name = new
ObjectName("j2eechap2.xmbean:service=JNDIMap");
        String[] initialValues = (String[])
            server.getAttribute(name, "InitialValues");
        System.out.println("InitialValues.length="+initialValues.length);
        int length = initialValues.length;
        for (int n = 0; n < length; n += 2) {
            String key = initialValues[n];
            String value = initialValues[n+1];

            System.out.println("key="+key+", value="+value);
        }
        // Add a new key/value pair
        String[] newInitialValues = new String[length+2];
        System.arraycopy(initialValues, 0, newInitialValues,
                         0, length);
        newInitialValues[length] = "key"+(length/2+1);
        newInitialValues[length+1] = "value"+(length/2+1);

        Attribute ivalues = new
            Attribute("InitialValues", newInitialValues);
        server.setAttribute(name, ivalues);
    }
}
```

At this point you may even shutdown the JBoss server, restart it and then rerun the initial example to see if the changes are persisted across server restarts:

```
[examples]$ ant -Dchap=jmx -Dex=xmbean2 run-example
...
```

```
run-examplexmbean2:
     [java] JNDIMap Class: org.jboss.mx.modelmbean.XMBean
     [java] JNDIMap Operations:
     [java]  + void start()
     [java]  + void stop()
     [java]  + void put(java.lang.Object
chap2.xmbean:service=JNDIMap,java.lang.Object cha
p2.xmbean:service=JNDIMap)
     [java]  + java.lang.Object get(java.lang.Object
chap2.xmbean:service=JNDIMap)
     [java]  + java.lang.String getJndiName()
     [java]  + void setJndiName(java.lang.String
chap2.xmbean:service=JNDIMap)
     [java]  + [Ljava.lang.String; getInitialValues()
     [java]  + void setInitialValues([Ljava.lang.String;
chap2.xmbean:service=JNDIMap)
     [java] handleNotification, event: null
     [java] key=key10, value=value10
     [java] key=key2, value=value2
     [java] key=key3, value=value3
     [java] key=key4, value=value4
     [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.book.jmx.xmbean.JNDIMap.put,sequenceNumber=3,timeStamp=10986
33664712,message=null,userData=null]
     [java] JNDIMap.put(key1, value1) successful
     [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.book.jmx.xmbean.JNDIMap.get,sequenceNumber=4,timeStamp=10986
33664821,message=null,userData=null]
     [java] JNDIMap.get(key0): null
     [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.book.jmx.xmbean.JNDIMap.get,sequenceNumber=5,timeStamp=10986
33664860,message=null,userData=null]
     [java] JNDIMap.get(key1): value1
     [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.book.jmx.xmbean.JNDIMap.put,sequenceNumber=6,timeStamp=10986
33664877,message=null,userData=null]
     [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.book.jmx.xmbean.JNDIMap.put,sequenceNumber=7,timeStamp=10986
33664895,message=null,userData=null]
     [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.book.jmx.xmbean.JNDIMap.put,sequenceNumber=8,timeStamp=10986
33664899,message=null,userData=null]
     [java] handleNotification, event:
javax.management.Notification[source=chap2.xmbean:s
ervice=JNDIMap,type=org.jboss.book.jmx.xmbean.JNDIMap.put,sequenceNumber=9,timeStamp=10986
33665614,message=null,userData=null]
```

You see that the last `InitialValues` attribute setting is in fact visible.

4.4. Deployment Ordering and Dependencies

We have seen how to manage dependencies using the service descriptor `depends` and `depends-list` tags. The deployment ordering supported by the deployment scanners provides a coarse-grained dependency management in that there is an order to deployments. If dependencies are consistent with the deployment packages then this is a simpler mechanism than having to enumerate the explicit MBean-MBean dependencies. By writing your own filters you can change the coarse grained ordering performed by the deployment scanner.

When a component archive is deployed, its nested deployment units are processed in a depth first ordering. Structuring of components into an archive hierarchy is yet another way to manage deployment ordering.You will need to explicitly state your MBean dependencies if your packaging structure does not happen to resolve the dependencies. Let's consider an example component deployment that consists of an MBean that uses an EJB. Here is the structure of the example EAR.

```
output/jmx/jmx-ex3.ear
+- META-INF/MANIFEST.MF
+- META-INF/jboss-app.xml
+- jmx-ex3.jar (archive) [EJB jar]
| +- META-INF/MANIFEST.MF
| +- META-INF/ejb-jar.xml
| +- org/jboss/book/jmx/ex3/EchoBean.class
| +- org/jboss/book/jmx/ex3/EchoLocal.class
| +- org/jboss/book/jmx/ex3/EchoLocalHome.class
+- jmx-ex3.sar (archive) [MBean sar]
| +- META-INF/MANIFEST.MF
| +- META-INF/jboss-service.xml
| +- org/jboss/book/jmx/ex3/EjbMBeanAdaptor.class
+- META-INF/application.xml
```

The EAR contains a `jmx-ex3.jar` and `jmx-ex3.sar`. The `jmx-ex3.jar` is the EJB archive and the `jmx-ex3.sar` is the MBean service archive. We have implemented the service as a Dynamic MBean to provide an illustration of their use.

```
package org.jboss.book.jmx.ex3;

import java.lang.reflect.Method;
import javax.ejb.CreateException;
import javax.management.Attribute;
import javax.management.AttributeList;
import javax.management.AttributeNotFoundException;
import javax.management.DynamicMBean;
import javax.management.InvalidAttributeValueException;
import javax.management.JMRuntimeException;
import javax.management.MBeanAttributeInfo;
import javax.management.MBeanConstructorInfo;
import javax.management.MBeanInfo;
import javax.management.MBeanNotificationInfo;
import javax.management.MBeanOperationInfo;
import javax.management.MBeanException;
import javax.management.MBeanServer;
```

```
import javax.management.ObjectName;
import javax.management.ReflectionException;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import org.jboss.system.ServiceMBeanSupport;

/**
 *  An example of a DynamicMBean that exposes select attributes and
 *  operations of an EJB as an MBean.
 *  @author Scott.Stark@jboss.org
 *  @version $Revision: 1.9 $
 */
public class EjbMBeanAdaptor extends ServiceMBeanSupport
    implements DynamicMBean
{
    private String helloPrefix;
    private String ejbJndiName;
    private EchoLocalHome home;

    /** These are the mbean attributes we expose
     */
    private MBeanAttributeInfo[] attributes = {
        new MBeanAttributeInfo("HelloPrefix", "java.lang.String",
                               "The prefix message to append to the session
echo reply",
                               true, // isReadable
                               true, // isWritable
                               false), // isIs
        new MBeanAttributeInfo("EjbJndiName", "java.lang.String",
                               "The JNDI name of the session bean local
home",
                               true, // isReadable
                               true, // isWritable
                               false) // isIs
    };

    /**
     * These are the mbean operations we expose
     */
    private MBeanOperationInfo[] operations;

    /**
     * We override this method to setup our echo operation info. It
     * could also be done in a ctor.
     */
    public ObjectName preRegister(MBeanServer server,
                                  ObjectName name)
        throws Exception
    {
        log.info("preRegister notification seen");

        operations = new MBeanOperationInfo[5];

        Class thisClass = getClass();
        Class[] parameterTypes = {String.class};
        Method echoMethod =
```

```
            thisClass.getMethod("echo", parameterTypes);
        String desc = "The echo op invokes the session bean echo method and"
            + " returns its value prefixed with the helloPrefix attribute
value";
        operations[0] = new MBeanOperationInfo(desc, echoMethod);

        // Add the Service interface operations from our super class
        parameterTypes = new Class[0];
        Method createMethod =
            thisClass.getMethod("create", parameterTypes);
        operations[1] = new MBeanOperationInfo("The
                JBoss Service.create", createMethod);
        Method startMethod =
            thisClass.getMethod("start", parameterTypes);
        operations[2] = new MBeanOperationInfo("The
                JBoss Service.start", startMethod);
        Method stopMethod =
            thisClass.getMethod("stop", parameterTypes);
        operations[3] = new MBeanOperationInfo("The
                JBoss Service.stop", startMethod);
        Method destroyMethod =
            thisClass.getMethod("destroy", parameterTypes);
        operations[4] = new MBeanOperationInfo("The
                JBoss Service.destroy", startMethod);
        return name;
    }


    // --- Begin ServiceMBeanSupport overides
    protected void createService() throws Exception
    {
        log.info("Notified of create state");
    }

    protected void startService() throws Exception
    {
        log.info("Notified of start state");
        InitialContext ctx = new InitialContext();
        home = (EchoLocalHome) ctx.lookup(ejbJndiName);
    }

    protected void stopService()
    {
        log.info("Notified of stop state");
    }

    // --- End ServiceMBeanSupport overides

    public String getHelloPrefix()
    {
        return helloPrefix;
    }
    public void setHelloPrefix(String helloPrefix)
    {
        this.helloPrefix = helloPrefix;
    }
```

```
    public String getEjbJndiName()
    {
        return ejbJndiName;
    }
    public void setEjbJndiName(String ejbJndiName)
    {
        this.ejbJndiName = ejbJndiName;
    }

    public String echo(String arg)
        throws CreateException, NamingException
    {
        log.debug("Lookup EchoLocalHome@"+ejbJndiName);
        EchoLocal bean = home.create();
        String echo = helloPrefix + bean.echo(arg);
        return echo;
    }

    // --- Begin DynamicMBean interface methods
    /**
     *  Returns the management interface that describes this dynamic
     *  resource.  It is the responsibility of the implementation to
     *  make sure the description is accurate.
     *
     * @return the management interface descriptor.
     */
    public MBeanInfo getMBeanInfo()
    {
        String classname = getClass().getName();
        String description = "This is an MBean that uses a session bean in
the"
            + " implementation of its echo operation.";
        MBeanInfo[] constructors = null;
        MBeanNotificationInfo[] notifications = null;
        MBeanInfo mbeanInfo = new MBeanInfo(classname,
                                            description, attributes,
                                            constructors, operations,
                                            notifications);
        // Log when this is called so we know when in the
        lifecycle this is used
            Throwable trace = new Throwable("getMBeanInfo trace");
        log.info("Don't panic, just a stack
                trace", trace);
        return mbeanInfo;
    }

    /**
     *  Returns the value of the attribute with the name matching the
     *  passed string.
     *
     * @param attribute the name of the attribute.
     * @return the value of the attribute.
     * @exception AttributeNotFoundException when there is no such
     * attribute.
     * @exception MBeanException wraps any error thrown by the
     * resource when
     * getting the attribute.
```

```
     * @exception ReflectionException wraps any error invoking the
     * resource.
     */
    public Object getAttribute(String attribute)
        throws AttributeNotFoundException,
               MBeanException,
               ReflectionException
    {
        Object value = null;
        if (attribute.equals("HelloPrefix")) {
            value = getHelloPrefix();
        } else if(attribute.equals("EjbJndiName")) {
            value = getEjbJndiName();
        } else {
            throw new AttributeNotFoundException("Unknown
                attribute("+attribute+") requested");
        }
        return value;
    }

    /**
     * Returns the values of the attributes with names matching the
     * passed string array.
     *
     * @param attributes the names of the attribute.
     * @return an {@link AttributeList AttributeList} of name
     * and value pairs.
     */
    public AttributeList getAttributes(String[] attributes)
    {
        AttributeList values = new AttributeList();
        for (int a = 0; a < attributes.length; a++) {
            String name = attributes[a];
            try {
                Object value = getAttribute(name);
                Attribute attr = new Attribute(name, value);
                values.add(attr);
            } catch(Exception e) {
                log.error("Failed to find attribute: "+name, e);
            }
        }
        return values;
    }

    /**
     *  Sets the value of an attribute. The attribute and new value
     *  are passed in the name value pair {@link Attribute
     *  Attribute}.
     *
     * @see javax.management.Attribute
     *
     * @param attribute the name and new value of the attribute.
     * @exception AttributeNotFoundException when there is no such
     * attribute.
     * @exception InvalidAttributeValueException when the new value
     * cannot be converted to the type of the attribute.
     * @exception MBeanException wraps any error thrown by the
```

```
     * resource when setting the new value.
     * @exception ReflectionException wraps any error invoking the
     * resource.
     */
    public void setAttribute(Attribute attribute)
        throws AttributeNotFoundException,
               InvalidAttributeValueException,
               MBeanException,
               ReflectionException
    {
        String name = attribute.getName();
        if (name.equals("HelloPrefix")) {
            String value = attribute.getValue().toString();
            setHelloPrefix(value);
        } else if(name.equals("EjbJndiName")) {
            String value = attribute.getValue().toString();
            setEjbJndiName(value);
        } else {
            throw new AttributeNotFoundException("Unknown
attribute("+name+") requested");
        }
    }

    /**
     * Sets the values of the attributes passed as an
     * {@link AttributeList AttributeList} of name and new
     * value pairs.
     *
     * @param attributes the name an new value pairs.
     * @return an {@link AttributeList AttributeList} of name and
     * value pairs that were actually set.
     */
    public AttributeList setAttributes(AttributeList attributes)
    {
        AttributeList setAttributes = new AttributeList();
        for(int a = 0; a < attributes.size(); a++) {
            Attribute attr = (Attribute) attributes.get(a);
            try {
                setAttribute(attr);
                setAttributes.add(attr);
            } catch(Exception ignore) {
            }
        }
        return setAttributes;
    }

    /**
     *  Invokes a resource operation.
     *
     *  @param actionName the name of the operation to perform.
     *  @param params the parameters to pass to the operation.
     *  @param signature the signartures of the parameters.
     *  @return the result of the operation.
     *  @exception MBeanException wraps any error thrown by the
     *  resource when performing the operation.
     *  @exception ReflectionException wraps any error invoking the
     *  resource.
```

```
     */
    public Object invoke(String actionName, Object[] params,
                         String[] signature)
        throws MBeanException,
               ReflectionException
    {
        Object rtnValue = null;
        log.debug("Begin invoke, actionName="+actionName);
        try {
            if (actionName.equals("echo")) {
                String arg = (String) params[0];
                rtnValue = echo(arg);
                log.debug("Result: "+rtnValue);
            } else if (actionName.equals("create")) {
                super.create();
            } else if (actionName.equals("start")) {
                super.start();
            } else if (actionName.equals("stop")) {
                super.stop();
            } else if (actionName.equals("destroy")) {
                super.destroy();
            } else {
                throw new JMRuntimeException("Invalid state,
                don't know about op="+actionName);
            }
        } catch(Exception e) {
            throw new ReflectionException(e, "echo failed");
        }


        log.debug("End invoke, actionName="+actionName);
        return rtnValue;
    }

    // --- End DynamicMBean interface methods

}
```

Believe it or not, this is a very trivial MBean. The vast majority of the code is there to provide the MBean metadata and handle the callbacks from the MBean Server. This is required because a Dynamic MBean is free to expose whatever management interface it wants. A Dynamic MBean can in fact change its management interface at runtime simply by returning different metadata from the `getMBeanInfo` method. Of course, some clients may not be happy with such a dynamic object, but the MBean Server will do nothing to prevent a Dynamic MBean from changing its interface.

There are two points to this example. First, demonstrate how an MBean can depend on an EJB for some of its functionality and second, how to create MBeans with dynamic management interfaces. If we were to write a standard MBean with a static interface for this example it would look like the following.

```
public interface EjbMBeanAdaptorMBean
{
```

```
    public String getHelloPrefix();
    public void setHelloPrefix(String prefix);
    public String getEjbJndiName();
    public void setEjbJndiName(String jndiName);
    public String echo(String arg) throws CreateException, NamingException;
    public void create() throws Exception;
    public void start() throws Exception;
    public void stop();
    public void destroy();
}
```

Moving to lines 67-83, this is where the MBean operation metadata is constructed. The `echo(String)`, `create()`, `start()`, `stop()` and `destroy()` operations are defined by obtaining the corresponding java.lang.reflect.Method object and adding a description. Let's go through the code and discuss where this interface implementation exists and how the MBean uses the EJB. Beginning with lines 40-51, the two `MBeanAttributeInfo` instances created define the attributes of the MBean. These attributes correspond to the `getHelloPrefix`/`setHelloPrefix` and `getEjbJndiName`/`setEjbJndiName` of the static interface. One thing to note in terms of why one might want to use a Dynamic MBean is that you have the ability to associate descriptive text with the attribute metadata. This is not something you can do with a static interface.

Lines 88-103 correspond to the JBoss service life cycle callbacks. Since we are subclassing the `ServiceMBeanSupport` utility class, we override the `createService`, `startService`, and `stopService` template callbacks rather than the `create`, `start`, and `stop` methods of the service interface. Note that we cannot attempt to lookup the `EchoLocalHome` interface of the EJB we make use of until the `startService` method. Any attempt to access the home interface in an earlier life cycle method would result in the name not being found in JNDI because the EJB container had not gotten to the point of binding the home interfaces. Because of this dependency we will need to specify that the MBean service depends on the EchoLocal EJB container to ensure that the service is not started before the EJB container is started. We will see this dependency specification when we look at the service descriptor.

Lines 105-121 are the `HelloPrefix` and `EjbJndiName` attribute accessors implementations. These are invoked in response to `getAttribute`/`setAttribute` invocations made through the MBean Server.

Lines 123-130 correspond to the `echo(String)` operation implementation. This method invokes the `EchoLocal.echo(String)` EJB method. The local bean interface is created using the `EchoLocalHome` that was obtained in the `startService` method.

The remainder of the class makes up the Dynamic MBean interface implementation. Lines 133-152 correspond to the MBean metadata accessor callback. This method returns a description of the MBean management interface in the form of the `javax.management.MBeanInfo` object. This is made up of a `description`, the `MBeanAttributeInfo` and `MBeanOperationInfo` metadata created earlier, as well as constructor and notification information. This MBean does not need any special constructors or notifications so this information is null.

Lines 154-258 handle the attribute access requests. This is rather tedious and error prone code so a toolkit or infrastructure that helps generate these methods should be used. A Model MBean framework based on XML called XBeans is currently being investigated in JBoss. Other than this, no other Dynamic MBean frameworks currently exist.

Lines 260-310 correspond to the operation invocation dispatch entry point. Here the request operation action name is checked against those the MBean handles and the appropriate method is invoked.

The `jboss-service.xml` descriptor for the MBean is given below. The dependency on the EJB container MBean is highlighted in bold. The format of the EJB container MBean ObjectName is: `"jboss.j2ee:service=EJB,jndiName="` + `<home-jndi-name>` where the <home-jndi-name> is the EJB home interface JNDI name.

```
<server>
    <mbean code="org.jboss.book.jmx.ex3.EjbMBeanAdaptor"
           name="jboss.book:service=EjbMBeanAdaptor">
        <attribute name="HelloPrefix">AdaptorPrefix</attribute>
        <attribute name="EjbJndiName">local/j2ee_chap2.EchoBean</attribute>
<depends>jboss.j2ee:service=EJB,jndiName=local/j2ee_chap2.EchoBean</depends>
    </mbean>
</server>
```

Deploy the example ear by running:

```
[examples]$ ant -Dchap=jmx -Dex=3 run-example
```

On the server console there will be messages similar to the following:

```
14:57:12,906 INFO  [EARDeployer] Init J2EE application:
file:/private/tmp/jboss-4.2.0/server/
        production/deploy/j2ee_chap2-ex3.ear
14:57:13,044 INFO  [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
  at
org.jboss.book.jmx.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:153)
...
14:57:13,088 INFO  [EjbMBeanAdaptor] preRegister notification seen
14:57:13,093 INFO  [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
  at
org.jboss.book.jmx.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:153)
...
14:57:13,117 INFO  [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
  at
org.jboss.book.jmx.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:153)
...
14:57:13,140 WARN  [EjbMBeanAdaptor] Unexcepted error accessing MBeanInfo
for null
java.lang.NullPointerException
```

```
  at
org.jboss.system.ServiceMBeanSupport.postRegister(ServiceMBeanSupport.java:418)
...
14:57:13,203 INFO  [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
  at
org.jboss.book.jmx.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:153)
...
14:57:13,232 INFO  [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
  at
org.jboss.book.jmx.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:153)
...
14:57:13,420 INFO  [EjbModule] Deploying Chap2EchoInfoBean
14:57:13,443 INFO  [EjbModule] Deploying chap2.EchoBean
14:57:13,488 INFO  [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
  at
org.jboss.book.jmx.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:153)
...
14:57:13,542 INFO  [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
  at
org.jboss.book.jmx.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:153)
...
14:57:13,558 INFO  [EjbMBeanAdaptor] Begin invoke, actionName=create
14:57:13,560 INFO  [EjbMBeanAdaptor] Notified of create state
14:57:13,562 INFO  [EjbMBeanAdaptor] End invoke, actionName=create
14:57:13,604 INFO  [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
  at
org.jboss.book.jmx.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:153)
...
14:57:13,621 INFO  [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
  at
org.jboss.book.jmx.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:153)
14:57:13,641 INFO  [EjbMBeanAdaptor] Begin invoke, actionName=getState
14:57:13,942 INFO  [EjbMBeanAdaptor] Begin invoke, actionName=start
14:57:13,944 INFO  [EjbMBeanAdaptor] Notified of start state
14:57:13,951 INFO  [EjbMBeanAdaptor] Testing Echo
14:57:13,983 INFO  [EchoBean] echo, info=echo info, arg=, arg=startService
14:57:13,986 INFO  [EjbMBeanAdaptor] echo(startService) = startService
14:57:13,988 INFO  [EjbMBeanAdaptor] End invoke, actionName=start
14:57:13,991 INFO  [EJBDeployer] Deployed:
file:/tmp/jboss-4.2.0.GA/server/production/tmp/deploy
/tmp60550jmx-ex3.ear-contents/jmx-ex3.jar
14:57:14,075 INFO  [EARDeployer] Started J2EE application: ...
```

The stack traces are not exceptions. They are traces coming from the `EjbMBeanAdaptor` code to demonstrate that clients ask for the MBean interface when they want to discover the MBean's capabilities. Notice that the EJB container (lines with [EjbModule]) is started before the example MBean (lines with [EjbMBeanAdaptor]).

Now, let's invoke the echo method using the JMX console web application. Go to the JMX Console (*http://localhost:8080/jmx-console*) and find the *service=EjbMBeanAdaptor* in the *jboss.book* domain. Click on the link and scroll down to the *echo* operation section. The view should be like that shown in *Figure 3.19, "The EjbMBeanAdaptor MBean operations JMX console view"*.

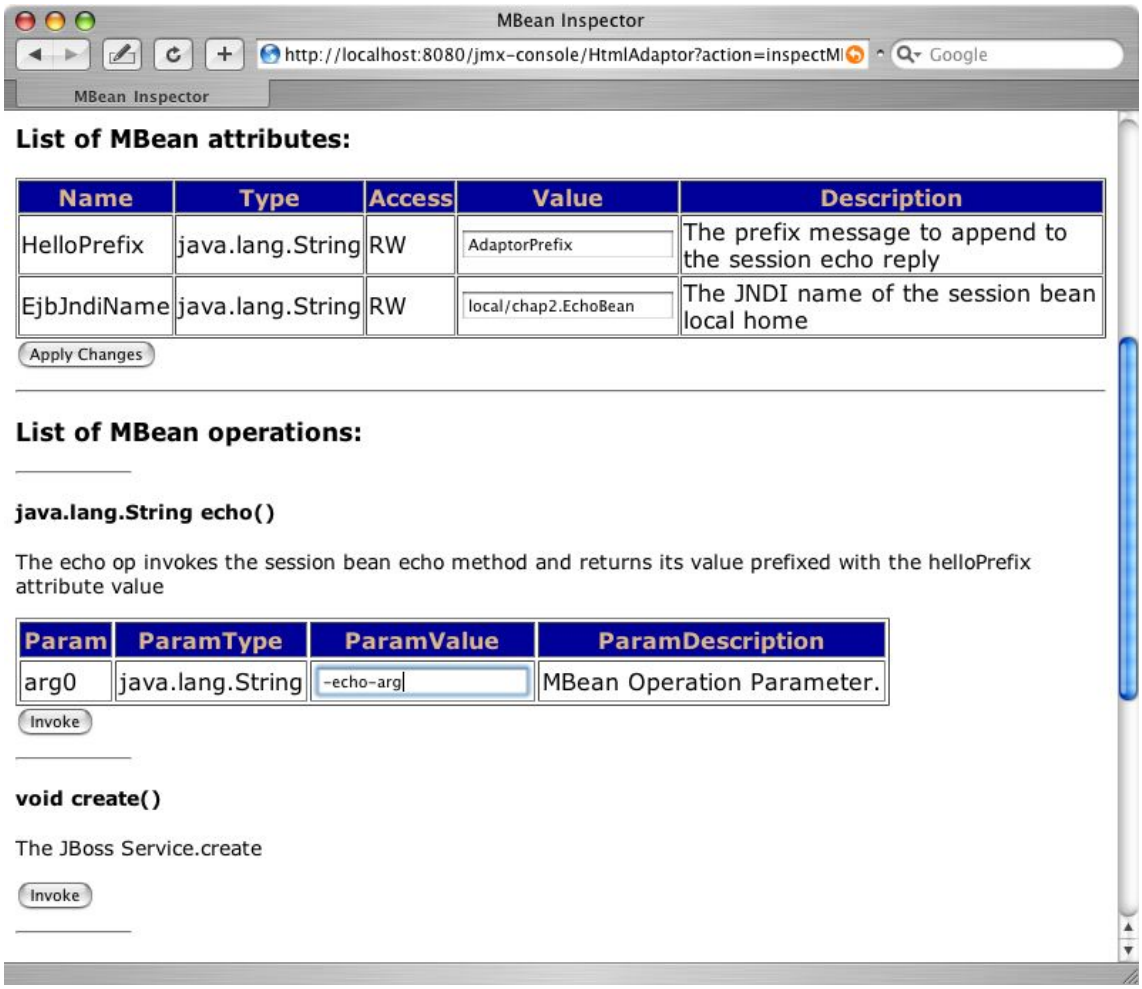


Figure 3.19. The EjbMBeanAdaptor MBean operations JMX console view

As shown, we have already entered an argument string of `-echo-arg` into the ParamValue text field. Press the Invoke button and a result string of `AdaptorPrefix-echo-arg` is displayed on the results page. The server console will show several stack traces from the various metadata queries issues by the JMX console and the MBean invoke method debugging lines:

```
10:51:48,671 INFO [EjbMBeanAdaptor] Begin invoke, actionName=echo
10:51:48,671 INFO [EjbMBeanAdaptor] Lookup
EchoLocalHome@local/j2ee_chap2.EchoBean
10:51:48,687 INFO [EchoBean] echo, info=echo info, arg=, arg=-echo-arg
10:51:48,687 INFO [EjbMBeanAdaptor] Result: AdaptorPrefix-echo-arg
10:51:48,687 INFO [EjbMBeanAdaptor] End invoke, actionName=echo
```

5. JBoss Deployer Architecture

JBoss has an extensible deployment architecture that allows one to incorporate components into the bare JBoss JMX microkernel. The `MainDeployer` is the deployment entry point. Requests to deploy a component are sent to the `MainDeployer` and it determines if there is a subdeployer capable of handling the deployment, and if there is, it delegates the deployment to the subdeployer. We saw an example of this when we looked at how the `MainDeployer` used the `SARDeployer` to deploy MBean services. Among the deployers provided with JBoss are:

- **AbstractWebDeployer**: This subdeployer handles web application archives (WARs). It accepts deployment archives and directories whose name ends with a `war` suffix. WARs must have a `WEB-INF/web.xml` descriptor and may have a `WEB-INF/jboss-web.xml` descriptor.
- **EARDeployer**: This subdeployer handles enterprise application archives (EARs). It accepts deployment archives and directories whose name ends with an `ear` suffix. EARs must have a `META-INF/application.xml` descriptor and may have a `META-INF/jboss-app.xml` descriptor.
- **EJBDeployer**: This subdeployer handles enterprise bean jars. It accepts deployment archives and directories whose name ends with a `jar` suffix. EJB jars must have a `META-INF/ejb-jar.xml` descriptor and may have a `META-INF/jboss.xml` descriptor.
- **JARDeployer**: This subdeployer handles library JAR archives. The only restriction it places on an archive is that it cannot contain a `WEB-INF` directory.
- **RARDeployer**: This subdeployer handles JCA resource archives (RARs). It accepts deployment archives and directories whose name ends with a `rar` suffix. RARs must have a `META-INF/ra.xml` descriptor.
- **SARDeployer**: This subdeployer handles JBoss MBean service archives (SARs). It accepts deployment archives and directories whose name ends with a `sar` suffix, as well as standalone XML files that end with `service.xml`. SARs that are jars must have a `META-INF/jboss-service.xml` descriptor.
- **XSLSubDeployer**: This subdeployer deploys arbitrary XML files. JBoss uses the XSLSubDeployer to deploy `ds.xml` files and transform them into `service.xml` files for the `SARDeployer`. However, it is not limited to just this task.
- **HARDeployer**: This subdeployer deploys hibernate archives (HARs). It accepts deployment archives and directories whose name ends with a `har` suffix. HARs must have a `META-INF/hibernate-service.xml` descriptor.
- **AspectDeployer**: This subdeployer deploys AOP archives. It accepts deployment archives and directories whose name ends with an `aop` suffix as well as `aop.xml` files. AOP archives must have a `META-INF/jboss-aop.xml` descriptor.
- **ClientDeployer**: This subdeployer deploys J2EE application clients. It accepts deployment archives and directories whose name ends with a `jar` suffix. J2EE clients must have a

`META-INF/application-client.xml` descriptor and may have a `META-INF/jboss-client.xml` descriptor.

- **BeanShellSubDeployer**: This subdeployer deploys bean shell scripts as MBeans. It accepts files whose name ends with a `bsh` suffix.

The MainDeployer, JARDeployer and SARDeployer are hard coded deployers in the JBoss server core. All other deployers are MBean services that register themselves as deployers with the MainDeployer using the `addDeployer(SubDeployer)` operation.

The `MainDeployer` communicates information about the component to be deployed the `SubDeployer` using a `DeploymentInfo` object. The `DeploymentInfo` object is a data structure that encapsulates the complete state of a deployable component.

When the `MainDeployer` receives a deployment request, it iterates through its registered subdeployers and invokes the `accepts(DeploymentInfo)` method on the subdeployer. The first subdeployer to return true is chosen. The MainDeployer will delegate the init, create, start, stop and destroy deployment life cycle operations to the subdeployer.

5.1. Deployers and ClassLoaders

Deployers are the mechanism by which components are brought into a JBoss server. Deployers are also the creators of the majority of UCL instances, and the primary creator is the MainDeployer. The MainDeployer creates the UCL for a deployment early on during its init method. The UCL is created by calling the DeploymentInfo.createClassLoaders() method. Only the topmost `DeploymentInfo` will actually create a UCL. All subdeployments will add their class paths to their parent `DeploymentInfo` UCL. Every deployment does have a standalone URLClassLoader that uses the deployment URL as its path. This is used to localize the loading of resources such as deployment descriptors. *Figure 3.20, “An illustration of the class loaders involved with an EAR deployment”* provides an illustration of the interaction between Deployers, DeploymentInfos and class loaders.

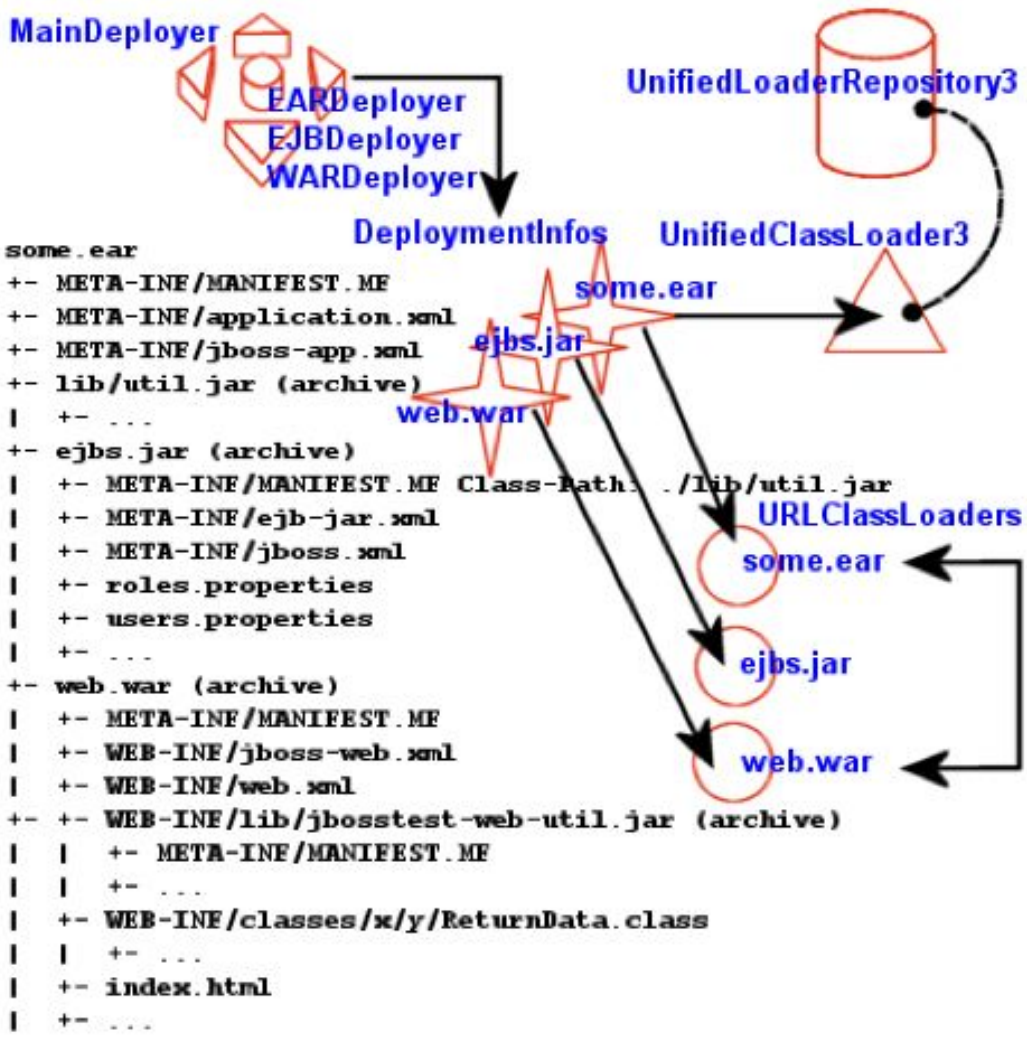


Figure 3.20. An illustration of the class loaders involved with an EAR deployment

The figure illustrates an EAR deployment with EJB and WAR subdeployments. The EJB deployment references the `lib/util.jar` utility jar via its manifest. The WAR includes classes in its `WEB-INF/classes` directory as well as the `WEB-INF/lib/jbosstest-web-util.jar`. Each deployment has a `DeploymentInfo` instance that has a `URLClassLoader` pointing to the deployment archive. The `DeploymentInfo` associated with `some.ear` is the only one to have a UCL created. The `ejbs.jar` and `web.warDeploymentInfo`s add their deployment archive to the `some.ear` UCL classpath, and share this UCL as their deployment UCL. The `EJBDeployer` also adds any manifest jars to the EAR UCL.

The `WARDeployer` behaves differently than other deployers in that it only adds its WAR archive to the `DeploymentInfo` UCL classpath. The loading of classes from the WAR `WEB-INF/classes` and `WEB-INF/lib` locations is handled by the servlet container class loader. The servlet container class loaders delegate to the WAR `DeploymentInfo` UCL as their parent

class loader, but the server container class loader is not part of the JBoss class loader repository. Therefore, classes inside of a WAR are not visible to other components. Classes that need to be shared between web application components and other components such as EJBs, and MBeans need to be loaded into the shared class loader repository either by including the classes into a SAR or EJB deployment, or by referencing a jar containing the shared classes through a manifest `Class-Path` entry. In the case of a SAR, the SAR classpath element in the service deployment serves the same purpose as a JAR manifest `Class-Path`.

6. Remote Access to Services, Detached Invokers

In addition to the MBean services notion that allows for the ability to integrate arbitrary functionality, JBoss also has a detached invoker concept that allows MBean services to expose functional interfaces via arbitrary protocols for remote access by clients. The notion of a detached invoker is that remoting and the protocol by which a service is accessed is a functional aspect or service independent of the component. Thus, one can make a naming service available for use via RMI/JRMP, RMI/HTTP, RMI/SOAP, or any arbitrary custom transport.

Let's begin our discussion of the detached invoker architecture with an overview of the components involved. The main components in the detached invoker architecture are shown in *Figure 3.21, "The main components in the detached invoker architecture"*.

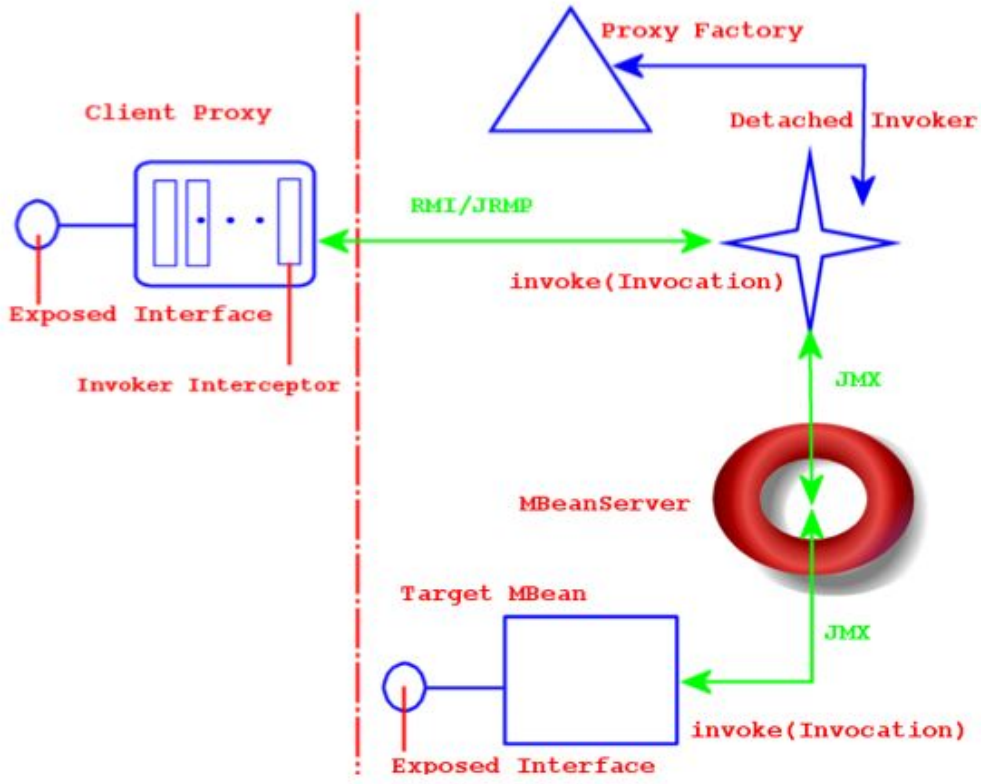


Figure 3.21. The main components in the detached invoker architecture

On the client side, there exists a client proxy which exposes the interface(s) of the MBean service. This is the same smart, compile-less dynamic proxy that we use for EJB home and remote interfaces. The only difference between the proxy for an arbitrary service and the EJB is the set of interfaces exposed as well as the client side interceptors found inside the proxy. The client interceptors are represented by the rectangles found inside of the client proxy. An interceptor is an assembly line type of pattern that allows for transformation of a method invocation and/or return values. A client obtains a proxy through some lookup mechanism, typically JNDI. Although RMI is indicated in *Figure 3.21, "The main components in the detached invoker architecture"*, the only real requirement on the exposed interface and its types is that they are serializable between the client server over JNDI as well as the transport layer.

The choice of the transport layer is determined by the last interceptor in the client proxy, which is referred to as the *Invoker Interceptor* in *Figure 3.21, "The main components in the detached invoker architecture"*. The invoker interceptor contains a reference to the transport specific stub of the server side *Detached Invoker* MBean service. The invoker interceptor also handles the optimization of calls that occur within the same VM as the target MBean. When the invoker interceptor detects that this is the case the call is passed to a call-by-reference invoker that simply passes the invocation along to the target MBean.

The detached invoker service is responsible for making a generic invoke operation available via the transport the detached invoker handles. The `Invoker` interface illustrates the generic invoke operation.

```
package org.jboss.invocation;

import java.rmi.Remote;
import org.jboss.proxy.Interceptor;
import org.jboss.util.id.GUID;


public interface Invoker
    extends Remote
{
    GUID ID = new GUID();

    String getServerHostName() throws Exception;

    Object invoke(Invocation invocation) throws Exception;
}
```

The Invoker interface extends `Remote` to be compatible with RMI, but this does not mean that an invoker must expose an RMI service stub. The detached invoker service simply acts as a transport gateway that accepts invocations represented as the `org.jboss.invocation.Invocation` object over its specific transport, unmarshalls the invocation, forwards the invocation onto the destination MBean service, represented by the *Target MBean* in *Figure 3.21, "The main components in the detached invoker architecture"*, and marshalls the return value or exception resulting from the forwarded call back to the client.

The `Invocation` object is just a representation of a method invocation context. This includes the target MBean name, the method, the method arguments, a context of information associated with the proxy by the proxy factory, and an arbitrary map of data associated with the invocation by the client proxy interceptors.

The configuration of the client proxy is done by the server side proxy factory MBean service, indicated by the *Proxy Factory* component in *Figure 3.21, "The main components in the detached invoker architecture"*. The proxy factory performs the following tasks:

- Create a dynamic proxy that implements the interface the target MBean wishes to expose.
- Associate the client proxy interceptors with the dynamic proxy handler.
- Associate the invocation context with the dynamic proxy. This includes the target MBean, detached invoker stub and the proxy JNDI name.
- Make the proxy available to clients by binding the proxy into JNDI.

The last component in *Figure 3.21, "The main components in the detached invoker architecture"* is the *Target MBean* service that wishes to expose an interface for invocations to remote clients. The steps required for an MBean service to be accessible through a given interface are:

- Define a JMX operation matching the signature: `public Object invoke(org.jboss.invocation.Invocation) throws Exception`
- Create a `HashMap<Long, Method>` mapping from the exposed interface `java.lang.reflect.Methods` to the long hash representation using the `org.jboss.invocation.MarshalledInvocation.calculateHash` method.
- Implement the `invoke(Invocation)` JMX operation and use the interface method hash mapping to transform from the long hash representation of the invoked method to the `java.lang.reflect.Method` of the exposed interface. Reflection is used to perform the actual invocation on the object associated with the MBean service that actually implements the exposed interface.

6.1. A Detached Invoker Example, the MBeanServer Invoker Adaptor Service

In the section on connecting to the JMX server we mentioned that there was a service that allows one to access the `javax.management.MBeanServer` via any protocol using an invoker service. In this section we present the `org.jboss.jmx.connector.invoker.InvokerAdaptorService` and its configuration for access via RMI/JRMP as an example of the steps required to provide remote access to an MBean service.

The `InvokerAdaptorService` is a simple MBean service that only exists to fulfill the target

MBean role in the detached invoker pattern.

```
package org.jboss.jmx.connector.invoker;
public interface InvokerAdaptorServiceMBean
    extends org.jboss.system.ServiceMBean
{
    Class getExportedInterface();
    void setExportedInterface(Class exportedInterface);

    Object invoke(org.jboss.invocation.Invocation invocation)
        throws Exception;
}

package org.jboss.jmx.connector.invoker;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.UndeclaredThrowableException;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

import javax.management.MBeanServer;
import javax.management.ObjectName;

import org.jboss.invocation.Invocation;
import org.jboss.invocation.MarshalledInvocation;
import org.jboss.mx.server.ServerConstants;
import org.jboss.system.ServiceMBeanSupport;
import org.jboss.system.Registry;

public class InvokerAdaptorService
    extends ServiceMBeanSupport
    implements InvokerAdaptorServiceMBean, ServerConstants
{
    private static ObjectName mbeanRegistry;

    static {
        try {
            mbeanRegistry = new ObjectName(MBEAN_REGISTRY);
        } catch (Exception e) {
            throw new RuntimeException(e.toString());
        }
    }

    private Map marshalledInvocationMapping = new HashMap();
    private Class exportedInterface;

    public Class getExportedInterface()
    {
        return exportedInterface;
    }

    public void setExportedInterface(Class exportedInterface)
    {
        this.exportedInterface = exportedInterface;
    }
```

```
    protected void startService()
        throws Exception
    {
        // Build the interface method map
        Method[] methods = exportedInterface.getMethods();
        HashMap tmpMap = new HashMap(methods.length);
        for (int m = 0; m < methods.length; m ++) {
            Method method = methods[m];
            Long hash = new
Long(MarshalledInvocation.calculateHash(method));
            tmpMap.put(hash, method);
        }

        marshalledInvocationMapping = Collections.unmodifiableMap(tmpMap);
        // Place our ObjectName hash into the Registry so invokers can
        // resolve it
        Registry.bind(new Integer(serviceName.hashCode()), serviceName);
    }

    protected void stopService()
        throws Exception
    {
        Registry.unbind(new Integer(serviceName.hashCode()));
    }


    public Object invoke(Invocation invocation)
        throws Exception
    {
        // Make sure we have the correct classloader before unmarshalling
        Thread thread = Thread.currentThread();
        ClassLoader oldCL = thread.getContextClassLoader();

        // Get the MBean this operation applies to
        ClassLoader newCL = null;
        ObjectName objectName = (ObjectName)
            invocation.getValue("JMX_OBJECT_NAME");
        if (objectName != null) {
            // Obtain the ClassLoader associated with the MBean deployment
            newCL = (ClassLoader)
                server.invoke(mbeanRegistry, "getValue",
                              new Object[] { objectName, CLASSLOADER },
                              new String[] { ObjectName.class.getName(),
                                             "java.lang.String" });
        }

        if (newCL != null && newCL != oldCL) {
            thread.setContextClassLoader(newCL);
        }

        try {
            // Set the method hash to Method mapping
            if (invocation instanceof MarshalledInvocation) {
                MarshalledInvocation mi = (MarshalledInvocation) invocation;
                mi.setMethodMap(marshalledInvocationMapping);
```

```
            }

            // Invoke the MBeanServer method via reflection
            Method method = invocation.getMethod();
            Object[] args = invocation.getArguments();
            Object value = null;
            try {
                String name = method.getName();
                Class[] sig = method.getParameterTypes();
                Method mbeanServerMethod =
                    MBeanServer.class.getMethod(name, sig);
                value = mbeanServerMethod.invoke(server, args);
            } catch(InvocationTargetException e) {
                Throwable t = e.getTargetException();
                if (t instanceof Exception) {
                    throw (Exception) t;
                } else {
                    throw new UndeclaredThrowableException(t,
method.toString());
                }
            }

            return value;
        } finally {
            if (newCL != null && newCL != oldCL) {
                thread.setContextClassLoader(oldCL);
            }
        }
    }
}
```

Example 3.16. The InvokerAdaptorService MBean

Let's go through the key details of this service. The `InvokerAdaptorServiceMBean` Standard MBean interface of the `InvokerAdaptorService` has a single `ExportedInterface` attribute and a single `invoke(Invocation)` operation. The `ExportedInterface` attribute allows customization of the type of interface the service exposes to clients. This has to be compatible with the `MBeanServer` class in terms of method name and signature. The `invoke(Invocation)` operation is the required entry point that target MBean services must expose to participate in the detached invoker pattern. This operation is invoked by the detached invoker services that have been configured to provide access to the `InvokerAdaptorService`.

Lines 54-64 of the InvokerAdaptorService build the HashMap<Long, Method> of the ExportedInterface Class using the `org.jboss.invocation.MarshalledInvocation.calculateHash(Method)` utility method. Because `java.lang.reflect.Method` instances are not serializable, a `MarshalledInvocation` version of the non-serializable `Invocation` class is used to marshall the invocation between the client and server. The `MarshalledInvocation` replaces the Method instances with their corresponding hash representation. On the server side, the `MarshalledInvocation` must be

told what the hash to Method mapping is.

Line 64 creates a mapping between the `InvokerAdaptorService` service name and its hash code representation. This is used by detached invokers to determine what the target MBean `ObjectName` of an `Invocation` is. When the target MBean name is store in the `Invocation`, its store as its hashCode because `ObjectName`s are relatively expensive objects to create. The `org.jboss.system.Registry` is a global map like construct that invokers use to store the hash code to `ObjectName` mappings in.

Lines 77-93 obtain the name of the MBean on which the MBeanServer operation is being performed and lookup the class loader associated with the MBean's SAR deployment. This information is available via the `org.jboss.mx.server.registry.BasicMBeanRegistry`, a JBoss JMX implementation specific class. It is generally necessary for an MBean to establish the correct class loading context because the detached invoker protocol layer may not have access to the class loaders needed to unmarshall the types associated with an invocation.

Lines 101-105 install the `ExposedInterface` class method hash to method mapping if the invocation argument is of type `MarshalledInvocation`. The method mapping calculated previously at lines 54-62 is used here.

Lines 107-114 perform a second mapping from the `ExposedInterface` Method to the matching method of the MBeanServer class. The `InvokerServiceAdaptor` decouples the `ExposedInterface` from the MBeanServer class in that it allows an arbitrary interface. This is needed on one hand because the standard `java.lang.reflect.Proxy` class can only proxy interfaces. It also allows one to only expose a subset of the MBeanServer methods and add transport specific exceptions like `java.rmi.RemoteException` to the `ExposedInterface` method signatures.

Line 115 dispatches the MBeanServer method invocation to the MBeanServer instance to which the `InvokerAdaptorService` was deployed. The server instance variable is inherited from the `ServiceMBeanSupport` superclass.

Lines 117-124 handle any exceptions coming from the reflective invocation including the unwrapping of any declared exception thrown by the invocation.

Line 126 is the return of the successful MBeanServer method invocation result.

Note that the `InvokerAdaptorService` MBean does not deal directly with any transport specific details. There is the calculation of the method hash to Method mapping, but this is a transport independent detail.

Now let's take a look at how the `InvokerAdaptorService` may be used to expose the same `org.jboss.jmx.adaptor.rmi.RMIAdaptor` interface via RMI/JRMP as seen in Connecting to JMX Using RMI. We will start by presenting the proxy factory and `InvokerAdaptorService` configurations found in the default setup in the `jmx-invoker-adaptor-service.sar` deployment. *Example 3.17, "The default jmx-invoker-adaptor-server.sar jboss-service.xml deployment descriptor"* shows the `jboss-service.xml` descriptor for this deployment.

```
<server>
```

```
    <!-- The JRMP invoker proxy configuration for the InvokerAdaptorService
-->
    <mbean code="org.jboss.invocation.jrmp.server.JRMPProxyFactory"
name="jboss.jmx:type=adaptor,name=Invoker,protocol=jrmp,service=proxyFactory">
        <!-- Use the standard JRMPInvoker from conf/jboss-service.xml -->
        <attribute
name="InvokerName">jboss:service=invoker,type=jrmp</attribute>
        <!-- The target MBean is the InvokerAdaptorService configured below
-->
        <attribute
name="TargetName">jboss.jmx:type=adaptor,name=Invoker</attribute>
        <!-- Where to bind the RMIAdaptor proxy -->
        <attribute name="JndiName">jmx/invoker/RMIAdaptor</attribute>
        <!-- The RMI compabitle MBeanServer interface -->
        <attribute
name="ExportedInterface">org.jboss.jmx.adaptor.rmi.RMIAdaptor</attribute>
        <attribute name="ClientInterceptors">
            <iterceptors>
<interceptor>org.jboss.proxy.ClientMethodInterceptor</interceptor>
                <interceptor>
org.jboss.jmx.connector.invoker.client.InvokerAdaptorClientInterceptor
                </interceptor>
<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
            </iterceptors>
        </attribute>
        <depends>jboss:service=invoker,type=jrmp</depends>
    </mbean>
    <!-- This is the service that handles the RMIAdaptor invocations by
routing
         them to the MBeanServer the service is deployed under. -->
    <mbean code="org.jboss.jmx.connector.invoker.InvokerAdaptorService"
           name="jboss.jmx:type=adaptor,name=Invoker">
        <attribute
name="ExportedInterface">org.jboss.jmx.adaptor.rmi.RMIAdaptor</attribute>
    </mbean>
</server>
```

Example 3.17. The default jmx-invoker-adaptor-server.sar jboss-service.xml deployment descriptor

The first MBean, `org.jboss.invocation.jrmp.server.JRMPProxyFactory`, is the proxy factory MBean service that creates proxies for the RMI/JRMP protocol. The configuration of this service as shown in *Example 3.17, "The default jmx-invoker-adaptor-server.sar jboss-service.xml deployment descriptor"* states that the JRMPInvoker will be used as the detached invoker, the `InvokerAdaptorService` is the target mbean to which requests will be forwarded, that the proxy will expose the `RMIAdaptor` interface, the proxy will be bound into JNDI under the name `jmx/invoker/RMIAdaptor`, and the proxy will contain 3 interceptors: `ClientMethodInterceptor`, `InvokerAdaptorClientInterceptor`, `InvokerInterceptor`. The configuration of the `InvokerAdaptorService` simply sets the RMIAdaptor interface that the

service is exposing.

The last piece of the configuration for exposing the `InvokerAdaptorService` via RMI/JRMP is the detached invoker. The detached invoker we will use is the standard RMI/JRMP invoker used by the EJB containers for home and remote invocations, and this is the `org.jboss.invocation.jrmp.server.JRMPInvoker` MBean service configured in the `conf/jboss-service.xml` descriptor. That we can use the same service instance emphasizes the detached nature of the invokers. The JRMPInvoker simply acts as the RMI/JRMP endpoint for all RMI/JRMP proxies regardless of the interface(s) the proxies expose or the service the proxies utilize.

6.2. Detached Invoker Reference

6.2.1. The JRMPInvoker - RMI/JRMP Transport

The `org.jboss.invocation.jrmp.server.JRMPInvoker` class is an MBean service that provides the RMI/JRMP implementation of the Invoker interface. The JRMPInvoker exports itself as an RMI server so that when it is used as the Invoker in a remote client, the JRMPInvoker stub is sent to the client instead and invocations use the RMI/JRMP protocol.

The JRMPInvoker MBean supports a number of attribute to configure the RMI/JRMP transport layer. Its configurable attributes are:

- **RMIObjectPort**: sets the RMI server socket listening port number. This is the port RMI clients will connect to when communicating through the proxy interface. The default setting in the `jboss-service.xml` descriptor is 4444, and if not specified, the attribute defaults to 0 to indicate an anonymous port should be used.
- **RMIClientSocketFactory**: specifies a fully qualified class name for the `java.rmi.server.RMIClientSocketFactory` interface to use during export of the proxy interface.
- **RMIServerSocketFactory**: specifies a fully qualified class name for the `java.rmi.server.RMIServerSocketFactory` interface to use during export of the proxy interface.
- **ServerAddress**: specifies the interface address that will be used for the RMI server socket listening port. This can be either a DNS hostname or a dot-decimal Internet address. Since the `RMIServerSocketFactory` does not support a method that accepts an InetAddress object, this value is passed to the `RMIServerSocketFactory` implementation class using reflection. A check for the existence of a `public void setBindAddress(java.net.InetAddress addr)` method is made, and if one exists the `RMIServerSocketAddr` value is passed to the `RMIServerSocketFactory` implementation. If the `RMIServerSocketFactory` implementation does not support such a method, the `ServerAddress` value will be ignored.
- **SecurityDomain**: specifies the JNDI name of an `org.jboss.security.SecurityDomain`

interface implementation to associate with the `RMIServerSocketFactory` implementation. The value will be passed to the `RMIServerSocketFactory` using reflection to locate a method with a signature of `public void setSecurityDomain(org.jboss.security.SecurityDomain d)`. If no such method exists the `SecurityDomain` value will be ignored.

6.2.2. The PooledInvoker - RMI/Socket Transport

The `org.jboss.invocation.pooled.server.PooledInvoker` is an MBean service that provides RMI over a custom socket transport implementation of the Invoker interface. The `PooledInvoker` exports itself as an RMI server so that when it is used as the `Invoker` in a remote client, the `PooledInvoker` stub is sent to the client instead and invocations use the custom socket protocol.

The `PooledInvoker` MBean supports a number of attribute to configure the socket transport layer. Its configurable attributes are:

- **NumAcceptThreads**: The number of threads that exist for accepting client connections. The default is 1.
- **MaxPoolSize**: The number of server threads for processing client. The default is 300.
- **SocketTimeout**: The socket timeout value passed to the `Socket.setSoTimeout()` method. The default is 60000.
- **ServerBindPort**: The port used for the server socket. A value of 0 indicates that an anonymous port should be chosen.
- **ClientConnectAddress**: The address that the client passes to the `Socket(addr, port)` constructor. This defaults to the server `InetAddress.getLocalHost()` value.
- **ClientConnectPort**: The port that the client passes to the `Socket(addr, port)` constructor. The default is the port of the server listening socket.
- **ClientMaxPoolSize**: The client side maximum number of threads. The default is 300.
- **Backlog**: The backlog associated with the server accept socket. The default is 200.
- **EnableTcpNoDelay**: A boolean flag indicating if client sockets will enable the `TcpNoDelay` flag on the socket. The default is false.
- **ServerBindAddress**: The address on which the server binds its listening socket. The default is an empty value which indicates the server should be bound on all interfaces.
- **TransactionManagerService**: The JMX ObjectName of the JTA transaction manager service.

6.2.3. The IIOPInvoker - RMI/IIOP Transport

The `org.jboss.invocation.iiop.IIOPInvoker` class is an MBean service that provides the RMI/IIOP implementation of the `Invoker` interface. The `IIOPInvoker` routes IIOP requests to CORBA servants. This is used by the `org.jboss.proxy.ejb.IORFactory` proxy factory to create RMI/IIOP proxies. However, rather than creating Java proxies (as the JRMP proxy factory does), this factory creates CORBA IORs. An `IORFactory` is associated to a given enterprise bean. It registers with the IIOP invoker two CORBA servants: `anEjbHomeCorbaServant` for the bean's `EJBHome` and an `EjbObjectCorbaServant` for the bean's `EJBObjects`.

The IIOPInvoker MBean has no configurable properties, since all properties are configured from the `conf/jacorb.properties` property file used by the JacORB CORBA service.

6.2.4. The JRMPProxyFactory Service - Building Dynamic JRMP Proxies

The `org.jboss.invocation.jrmp.server.JRMPProxyFactory` MBean service is a proxy factory that can expose any interface with RMI compatible semantics for access to remote clients using JRMP as the transport.

The JRMPProxyFactory supports the following attributes:

- **InvokerName**: The server side JRMPInvoker MBean service JMX ObjectName string that will handle the RMI/JRMP transport.
- **TargetName**: The server side MBean that exposes the `invoke(Invocation)` JMX operation for the exported interface. This is used as the destination service for any invocations done through the proxy.
- **JndiName**: The JNDI name under which the proxy will be bound.
- **ExportedInterface**: The fully qualified class name of the interface that the proxy implements. This is the typed view of the proxy that the client uses for invocations.
- **ClientInterceptors**: An XML fragment of interceptors/interceptor elements with each interceptor element body specifying the fully qualified class name of an `org.jboss.proxy.Interceptor` implementation to include in the proxy interceptor stack. The ordering of the interceptors/interceptor elements defines the order of the interceptors.

6.2.5. The HttpInvoker - RMI/HTTP Transport

The `org.jboss.invocation.http.server.HttpInvoker` MBean service provides support for making invocations into the JMX bus over HTTP. Unlike the `JRMPInvoker`, the `HttpInvoker` is not an implementation of `Invoker`, but it does implement the Invoker.invoke method. The HttpInvoker is accessed indirectly by issuing an HTTP POST against the `org.jboss.invocation.http.servlet.InvokerServlet`. The `HttpInvoker` exports a client side proxy in the form of the `org.jboss.invocation.http.interfaces.HttpInvokerProxy` class, which is an implementation of `Invoker`, and is serializable. The `HttpInvoker` is a drop in

replacement for the `JRMPInvoker` as the target of the `bean-invoker` and `home-invoker` EJB configuration elements. The `HttpInvoker` and `InvokerServlet` are deployed in the `http-invoker.sar` discussed in the JNDI chapter in the section entitled Accessing JNDI over HTTP

The HttpInvoker supports the following attributes:

- **InvokerURL**: This is either the http URL to the `InvokerServlet` mapping, or the name of a system property that will be resolved inside the client VM to obtain the http URL to the `InvokerServlet`.
- **InvokerURLPrefix**: If there is no `invokerURL` set, then one will be constructed via the concatenation of `invokerURLPrefix` + the local host + `invokerURLSuffix`. The default prefix is `http://`.
- **InvokerURLSuffix**: If there is no `invokerURL` set, then one will be constructed via the concatenation of `invokerURLPrefix` + the local host + `invokerURLSuffix`. The default suffix is `:8080/invoker/JMXInvokerServlet`.
- **UseHostName**: A boolean flag if the `InetAddress.getHostName()` or `getHostAddress()` method should be used as the host component of `invokerURLPrefix` + host + `invokerURLSuffix`. If true `getHostName()` is used, otherwise `getHostAddress()` is used.

6.2.6. The HA JRMPInvoker - Clustered RMI/JRMP Transport

The `org.jboss.proxy.generic.ProxyFactoryHA` service is an extension of the `ProxyFactoryHA` that is a cluster aware factory. The `ProxyFactoryHA` fully supports all of the attributes of the `JRMPProxyFactory`. This means that customized bindings of the port, interface and socket transport are available to clustered RMI/JRMP as well. In addition, the following cluster specific attributes are supported:

- **PartitionObjectName**: The JMX `ObjectName` of the cluster service to which the proxy is to be associated with.
- **LoadBalancePolicy**: The class name of the `org.jboss.ha.framework.interfaces.LoadBalancePolicy` interface implementation to associate with the proxy.

6.2.7. The HA HttpInvoker - Clustered RMI/HTTP Transport

The RMI/HTTP layer allows for software load balancing of the invocations in a clustered environment. The HA capable extension of the HTTP invoker borrows much of its functionality from the HA-RMI/JRMP clustering. To enable HA-RMI/HTTP you need to configure the invokers for the EJB container. This is done through either a `jboss.xml` descriptor, or the `standardjboss.xml` descriptor.

6.2.8. HttpProxyFactory - Building Dynamic HTTP Proxies

The `org.jboss.invocation.http.server.HttpProxyFactory` MBean service is a proxy factory that can expose any interface with RMI compatible semantics for access to remote clients using HTTP as the transport.

The HttpProxyFactory supports the following attributes:

- **InvokerName**: The server side MBean that exposes the invoke operation for the exported interface. The name is embedded into the `HttpInvokerProxy` context as the target to which the invocation should be forwarded by the `HttpInvoker`.
- **JndiName**: The JNDI name under which the `HttpInvokerProxy` will be bound. This is the name clients lookup to obtain the dynamic proxy that exposes the service interfaces and marshalls invocations over HTTP. This may be specified as an empty value to indicate that the proxy should not be bound into JNDI.
- **InvokerURL**: This is either the http URL to the `InvokerServlet` mapping, or the name of a system property that will be resolved inside the client VM to obtain the http URL to the `InvokerServlet`.
- **InvokerURLPrefix**: If there is no `invokerURL` set, then one will be constructed via the concatenation of `invokerURLPrefix` + the local host + `invokerURLSuffix`. The default prefix is `http://`.
- **InvokerURLSuffix**: If there is no `invokerURL` set, then one will be constructed via the concatenation of `invokerURLPrefix` + the local host + `invokerURLSuffix`. The default suffix is `:8080/invoker/JMXInvokerServlet`.
- **UseHostName**: A boolean flag indicating if the `InetAddress.getHostName()` or `getHostAddress()` method should be used as the host component of `invokerURLPrefix` + host + `invokerURLSuffix`. If true `getHostName()` is used, otherwise `getHostAddress()` is used.
- **ExportedInterface**: The name of the RMI compatible interface that the `HttpInvokerProxy` implements.

6.2.9. Steps to Expose Any RMI Interface via HTTP

Using the `HttpProxyFactory` MBean and JMX, you can expose any interface for access using HTTP as the transport. The interface to expose does not have to be an RMI interface, but it does have to be compatible with RMI in that all method parameters and return values are serializable. There is also no support for converting RMI interfaces used as method parameters or return values into their stubs.

The three steps to making your object invocable via HTTP are:

- Create a mapping of longs to the RMI interface methods using the `MarshalledInvocation.calculateHash` method. Here for example, is the procedure for an RMI `SRPRemoteServerInterface` interface:

```
import java.lang.reflect.Method;
import java.util.HashMap;
import org.jboss.invocation.MarshalledInvocation;

HashMap marshalledInvocationMapping = new HashMap();

// Build the Naming interface method map
Method[] methods = SRPRemoteServerInterface.class.getMethods();
for(int m = 0; m < methods.length; m ++) {
    Method method = methods[m];
    Long hash = new Long(MarshalledInvocation.calculateHash(method));
    marshalledInvocationMapping.put(hash, method);
}
```

- Either create or extend an existing MBean to support an invoke operation. Its signature is `Object invoke(Invocation invocation) throws Exception`, and the steps it performs are as shown here for the `SRPRemoteServerInterface` interface. Note that this uses the `marshalledInvocationMapping` from step 1 to map from the `Long` method hashes in the `MarshalledInvocation` to the `Method` for the interface.

```
import org.jboss.invocation.Invocation;
import org.jboss.invocation.MarshalledInvocation;

public Object invoke(Invocation invocation)
    throws Exception
{
    SRPRemoteServerInterface theServer = <the_actual_rmi_server_object>;
    // Set the method hash to Method mapping
    if (invocation instanceof MarshalledInvocation) {
        MarshalledInvocation mi = (MarshalledInvocation) invocation;
        mi.setMethodMap(marshalledInvocationMapping);
    }

    // Invoke the Naming method via reflection
    Method method = invocation.getMethod();
    Object[] args = invocation.getArguments();
    Object value = null;
    try {
        value = method.invoke(theServer, args);
    } catch(InvocationTargetException e) {
        Throwable t = e.getTargetException();
        if (t instanceof Exception) {
            throw (Exception) e;
        } else {
            throw new UndeclaredThrowableException(t, method.toString());
        }
    }

    return value;
```

```
}
```

- Create a configuration of the `HttpProxyFactory` MBean to make the RMI/HTTP proxy available through JNDI. For example:

```
<!-- Expose the SRP service interface via HTTP -->
<mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
       name="jboss.security.tests:service=SRP/HTTP">
    <attribute
name="InvokerURL">http://localhost:8080/invoker/JMXInvokerServlet</attribute>
    <attribute
name="InvokerName">jboss.security.tests:service=SRPService</attribute>
    <attribute name="ExportedInterface">
        org.jboss.security.srp.SRPRemoteServerInterface
    </attribute>
    <attribute name="JndiName">srp-test-http/SRPServerInterface</attribute>
</mbean>
```

Any client may now lookup the RMI interface from JNDI using the name specified in the `HttpProxyFactory` (e.g., `srp-test-http/SRPServerInterface`) and use the obtain proxy in exactly the same manner as the RMI/JRMP version.

Naming on JBoss

The JNDI Naming Service

The naming service plays a key role in enterprise Java applications, providing the core infrastructure that is used to locate objects or services in an application server. It is also the mechanism that clients external to the application server use to locate services inside the application server. Application code, whether it is internal or external to the JBoss instance, need only know that it needs to talk to the a message queue named `queue/IncomingOrders` and would not need to worry about any of the details of how the queue is configured. In a clustered environment, naming services are even more valuable. A client of a service would desire to look up the `ProductCatalog` session bean from the cluster without worrying which machine the service is residing. Whether it is a big clustered service, a local resource or just a simple application component that is needed, the JNDI naming service provides the glue that lets code find the objects in the system by name.

1. An Overview of JNDI

JNDI is a standard Java API that is bundled with JDK1.3 and higher. JNDI provides a common interface to a variety of existing naming services: DNS, LDAP, Active Directory, RMI registry, COS registry, NIS, and file systems. The JNDI API is divided logically into a client API that is used to access naming services, and a service provider interface (SPI) that allows the user to create JNDI implementations for naming services.

The SPI layer is an abstraction that naming service providers must implement to enable the core JNDI classes to expose the naming service using the common JNDI client interface. An implementation of JNDI for a naming service is referred to as a JNDI provider. JBoss naming is an example JNDI implementation, based on the SPI classes. Note that the JNDI SPI is not needed by J2EE component developers.

For a thorough introduction and tutorial on JNDI, which covers both the client and service provider APIs, see the Sun tutorial at *http://java.sun.com/products/jndi/tutorial/*.

The main JNDI API package is the `javax.naming` package. It contains five interfaces, 10 classes, and several exceptions. There is one key class, `InitialContext`, and two key interfaces, `Context` and `Name`

1.1. Names

The notion of a name is of fundamental importance in JNDI. The naming system determines the syntax that the name must follow. The syntax of the naming system allows the user to parse string representations of names into its components. A name is used with a naming system to locate objects. In the simplest sense, a naming system is just a collection of objects with unique names. To locate an object in a naming system you provide a name to the naming system, and the naming system returns the object store under the name.

As an example, consider the Unix file system's naming convention. Each file is named from its

path relative to the root of the file system, with each component in the path separated by the forward slash character ("/"). The file's path is ordered from left to right. The pathname/usr/jboss/readme.txt, for example, names a file readme.txt in the directory jboss, under the directory usr, located in the root of the file system. JBoss naming uses a UNIX-style namespace as its naming convention.

The javax.naming.Name interface represents a generic name as an ordered sequence of components. It can be a composite name (one that spans multiple namespaces), or a compound name (one that is used within a single hierarchical naming system). The components of a name are numbered. The indexes of a name with N components range from 0 up to, but not including, N. The most significant component is at index 0. An empty name has no components.

A composite name is a sequence of component names that span multiple namespaces. An example of a composite name would be the hostname and file combination commonly used with UNIX commands like scp. For example, the following command copies localfile.txt to the file remotefile.txt in the tmp directory on host ahost.someorg.org:

```
scp localfile.txt ahost.someorg.org:/tmp/remotefile.txt
```

A compound name is derived from a hierarchical namespace. Each component in a compound name is an atomic name, meaning a string that cannot be parsed into smaller components. A file pathname in the UNIX file system is an example of a compound name. ahost.someorg.org:/tmp/remotefile.txt is a composite name that spans the DNS and UNIX file system namespaces. The components of the composite name are ahost.someorg.org and /tmp/remotefile.txt. A component is a string name from the namespace of a naming system. If the component comes from a hierarchical namespace, that component can be further parsed into its atomic parts by using the javax.naming.CompoundName class. The JNDI API provides the javax.naming.CompositeName class as the implementation of the Name interface for composite names.

1.2. Contexts

The javax.naming.Context interface is the primary interface for interacting with a naming service. The Context interface represents a set of name-to-object bindings. Every context has an associated naming convention that determines how the context parses string names into javax.naming.Name instances. To create a name to object binding you invoke the bind method of a Context and specify a name and an object as arguments. The object can later be retrieved using its name using the Context lookup method. A Context will typically provide operations for binding a name to an object, unbinding a name, and obtaining a listing of all name-to-object bindings. The object you bind into a Context can itself be of type Context . The Context object that is bound is referred to as a subcontext of the Context on which the bind method was invoked.

As an example, consider a file directory with a pathname /usr, which is a context in the UNIX

file system. A file directory named relative to another file directory is a subcontext (commonly referred to as a subdirectory). A file directory with a pathname `/usr/jboss` names a `jboss` context that is a subcontext of `usr`. In another example, a DNS domain, such as `org`, is a context. A DNS domain named relative to another DNS domain is another example of a subcontext. In the DNS domain `jboss.org`, the DNS domain `jboss` is a subcontext of `org` because DNS names are parsed right to left.

1.2.1. Obtaining a Context using InitialContext

All naming service operations are performed on some implementation of the `Context` interface. Therefore, you need a way to obtain a `Context` for the naming service you are interested in using. The `javax.naming.IntialContext` class implements the `Context` interface, and provides the starting point for interacting with a naming service.

When you create an `InitialContext`, it is initialized with properties from the environment. JNDI determines each property's value by merging the values from the following two sources, in order.

- The first occurrence of the property from the constructor's environment parameter and (for appropriate properties) the applet parameters and system properties.
- All `jndi.properties` resource files found on the classpath.

For each property found in both of these two sources, the property's value is determined as follows. If the property is one of the standard JNDI properties that specify a list of JNDI factories, all of the values are concatenated into a single colon-separated list. For other properties, only the first value found is used. The preferred method of specifying the JNDI environment properties is through a `jndi.properties` file, which allows your code to externalize the JNDI provider specific information so that changing JNDI providers will not require changes to your code or recompilation.

The `Context` implementation used internally by the `InitialContext` class is determined at runtime. The default policy uses the environment property `java.naming.factory.initial`, which contains the class name of the `javax.naming.spi.InitialContextFactory` implementation. You obtain the name of the `InitialContextFactory` class from the naming service provider you are using.

Example 4.1, "A sample jndi.properties file" gives a sample `jndi.properties` file a client application would use to connect to a JBossNS service running on the local host at port 1099. The client application would need to have the `jndi.properties` file available on the application classpath. These are the properties that the JBossNS JNDI implementation requires. Other JNDI providers will have different properties and values.

```
### JBossNS properties
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

Example 4.1. A sample jndi.properties file

2. The JBossNS Architecture

The JBossNS architecture is a Java socket/RMI based implementation of the `javax.naming.Context` interface. It is a client/server implementation that can be accessed remotely. The implementation is optimized so that access from within the same VM in which the JBossNS server is running does not involve sockets. Same VM access occurs through an object reference available as a global singleton. *Figure 4.1, "Key components in the JBossNS architecture."* illustrates some of the key classes in the JBossNS implementation and their relationships.

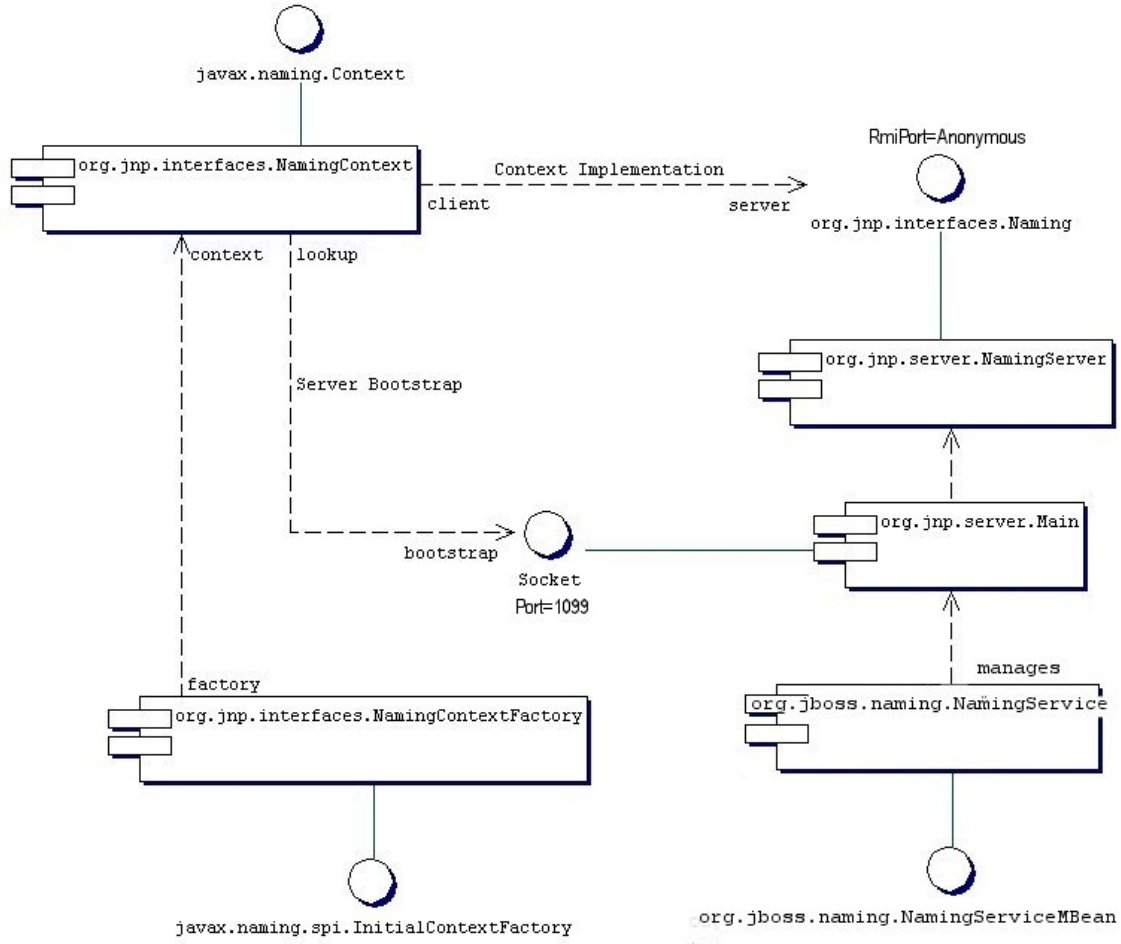


Figure 4.1. Key components in the JBossNS architecture.

We will start with the `NamingService` MBean. The `NamingService` MBean provides the JNDI naming service. This is a key service used pervasively by the J2EE technology components. The configurable attributes for the `NamingService` are as follows.

- **Port**: The jnp protocol listening port for the `NamingService`. If not specified default is 1099, the same as the RMI registry default port.
- **RmiPort**: The RMI port on which the RMI Naming implementation will be exported. If not specified the default is 0 which means use any available port.
- **BindAddress**: The specific address the `NamingService` listens on. This can be used on a multi-homed host for a `java.net.ServerSocket` that will only accept connect requests on one of its addresses.
- **RmiBindAddress**: The specific address the RMI server portion of the `NamingService` listens on. This can be used on a multi-homed host for a `java.net.ServerSocket` that will only accept connect requests on one of its addresses. If this is not specified and the `BindAddress` is, the `RmiBindAddress` defaults to the `BindAddress` value.
- **Backlog**: The maximum queue length for incoming connection indications (a request to connect) is set to the `backlog` parameter. If a connection indication arrives when the queue is full, the connection is refused.
- **ClientSocketFactory**: An optional custom `java.rmi.server.RMIClientSocketFactory` implementation class name. If not specified the default `RMIClientSocketFactory` is used.
- **ServerSocketFactory**: An optional custom `java.rmi.server.RMIServerSocketFactory` implementation class name. If not specified the default `RMIServerSocketFactory` is used.
- **JNPServerSocketFactory**: An optional custom `javax.net.ServerSocketFactory` implementation class name. This is the factory for the `ServerSocket` used to bootstrap the download of the JBossNS `Naming` interface. If not specified the `javax.net.ServerSocketFactory.getDefault()` method value is used.

The `NamingService` also creates the `java:comp` context such that access to this context is isolated based on the context class loader of the thread that accesses the `java:comp` context. This provides the application component private ENC that is required by the J2EE specs. This segregation is accomplished by binding a `javax.naming.Reference` to a context that uses the `org.jboss.naming.ENCFactory` as its `javax.naming.ObjectFactory`. When a client performs a lookup of `java:comp`, or any subcontext, the `ENCFactory` checks the thread context `ClassLoader`, and performs a lookup into a map using the `ClassLoader` as the key.

If a context instance does not exist for the class loader instance, one is created and associated with that class loader in the `ENCFactory` map. Thus, correct isolation of an application component's ENC relies on each component receiving a unique `ClassLoader` that is associated with the component threads of execution.

The `NamingService` delegates its functionality to an `org.jnp.server.Main` MBean. The reason for the duplicate MBeans is because JBossNS started out as a stand-alone JNDI implementation, and can still be run as such. The `NamingService` MBean embeds the `Main` instance into the JBoss server so that usage of JNDI with the same VM as the JBoss server does not incur any socket overhead. The configurable attributes of the NamingService are really the configurable attributes of the JBossNS `Main` MBean. The setting of any attributes on the

`NamingService` MBean simply set the corresponding attributes on the `Main` MBean the `NamingService` contains. When the `NamingService` is started, it starts the contained `Main` MBean to activate the JNDI naming service.

In addition, the `NamingService` exposes the `Naming` interface operations through a JMX detyped invoke operation. This allows the naming service to be accessed via JMX adaptors for arbitrary protocols. We will look at an example of how HTTP can be used to access the naming service using the invoke operation later in this chapter.

The details of threads and the thread context class loader won't be explored here, but the JNDI tutorial provides a concise discussion that is applicable. See *http://java.sun.com/products/jndi/tutorial/beyond/misc/classloader.html* for the details.

When the `Main` MBean is started, it performs the following tasks:

- Instantiates an `org.jnp.naming.NamingService` instance and sets this as the local VM server instance. This is used by any `org.jnp.interfaces.NamingContext` instances that are created within the JBoss server VM to avoid RMI calls over TCP/IP.
- Exports the `NamingServer` instance's `org.jnp.naming.interfaces.Naming` RMI interface using the configured `RmiPort`, `ClientSocketFactory`, `ServerSocketFactory`attributes.
- Creates a socket that listens on the interface given by the `BindAddress` and `Port` attributes.
- Spawns a thread to accept connections on the socket.

3. The Naming InitialContext Factories

The JBoss JNDI provider currently supports several different `InitialContext` factory implementations.

3.1. The standard naming context factory

The most commonly used factory is the `org.jnp.interfaces.NamingContextFactory` implementation. Its properties include:

- **java.naming.factory.initial**: The name of the environment property for specifying the initial context factory to use. The value of the property should be the fully qualified class name of the factory class that will create an initial context. If it is not specified, a `javax.naming.NoInitialContextException` will be thrown when an `InitialContext` object is created.
- **java.naming.provider.url**: The name of the environment property for specifying the location of the JBoss JNDI service provider the client will use. The `NamingContextFactory` class uses this information to know which JBossNS server to connect to. The value of the property should be a URL string. For JBossNS the URL format is `jnp://host:port/[jndi_path]`. The `jnp:` portion of the URL is the protocol and refers to the socket/RMI based protocol used

by JBoss. The `jndi_path` portion of the URL is an optional JNDI name relative to the root context, for example, `apps` or `apps/tmp`. Everything but the host component is optional. The following examples are equivalent because the default port value is 1099.

- `jnp://www.jboss.org:1099/`
- `www.jboss.org:1099`
- `www.jboss.org`

- **java.naming.factory.url.pkgs**: The name of the environment property for specifying the list of package prefixes to use when loading in URL context factories. The value of the property should be a colon-separated list of package prefixes for the class name of the factory class that will create a URL context factory. For the JBoss JNDI provider this must be `org.jboss.naming:org.jnp.interfaces`. This property is essential for locating the `jnp:` and `java:` URL context factories of the JBoss JNDI provider.

- **jnp.socketFactory**: The fully qualified class name of the `javax.net.SocketFactory` implementation to use to create the bootstrap socket. The default value is `org.jnp.interfaces.TimedSocketFactory`. The `TimedSocketFactory` is a simple `SocketFactory` implementation that supports the specification of a connection and read timeout. These two properties are specified by:

- **jnp.timeout**: The connection timeout in milliseconds. The default value is 0 which means the connection will block until the VM TCP/IP layer times out.

- **jnp.sotimeout**: The connected socket read timeout in milliseconds. The default value is 0 which means reads will block. This is the value passed to the `Socket.setSoTimeout` on the newly connected socket.

When a client creates an `InitialContext` with these JBossNS properties available, the `org.jnp.interfaces.NamingContextFactory` object is used to create the `Context` instance that will be used in subsequent operations. The `NamingContextFactory` is the JBossNS implementation of the `javax.naming.spi.InitialContextFactory` interface. When the `NamingContextFactory` class is asked to create a `Context`, it creates an `org.jnp.interfaces.NamingContext` instance with the `InitialContext` environment and name of the context in the global JNDI namespace. It is the `NamingContext` instance that actually performs the task of connecting to the JBossNS server, and implements the `Context` interface. The `Context.PROVIDER_URL` information from the environment indicates from which server to obtain a `NamingServer` RMI reference.

The association of the `NamingContext` instance to a `NamingServer` instance is done in a lazy fashion on the first `Context` operation that is performed. When a `Context` operation is performed and the `NamingContext` has no `NamingServer` associated with it, it looks to see if its environment properties define a `Context.PROVIDER_URL`. A `Context.PROVIDER_URL` defines the host and port of the JBossNS server the `Context` is to use. If there is a provider URL, the `NamingContext` first checks to see if a `Naming` instance keyed by the host and port pair has already been created by checking a `NamingContext` class static map. It simply uses the existing `Naming` instance if one for the host port pair has already been obtained. If no `Naming` instance has been created for the given host and port, the `NamingContext` connects to the host and port using a `java.net.Socket`, and retrieves a `Naming` RMI stub from the server by reading a

`java.rmi.MarshalledObject` from the socket and invoking its get method. The newly obtained Naming instance is cached in the `NamingContext` server map under the host and port pair. If no provider URL was specified in the JNDI environment associated with the context, the `NamingContext` simply uses the in VM Naming instance set by the `Main` MBean.

The `NamingContext` implementation of the `Context` interface delegates all operations to the `Naming` instance associated with the `NamingContext`. The `NamingServer` class that implements the `Naming` interface uses a `java.util.Hashtable` as the `Context` store. There is one unique `NamingServer` instance for each distinct JNDI Name for a given JBossNS server. There are zero or more transient `NamingContext` instances active at any given moment that refers to a `NamingServer` instance. The purpose of the `NamingContext` is to act as a `Context` to the `Naming` interface adaptor that manages translation of the JNDI names passed to the `NamingContext` . Because a JNDI name can be relative or a URL, it needs to be converted into an absolute name in the context of the JBossNS server to which it refers. This translation is a key function of the `NamingContext`.

3.2. The org.jboss.naming.NamingContextFactory

This version of the `InitialContextFactory` implementation is a simple extension of the jnp version which differs from the jnp version in that it stores the last configuration passed to its `InitialContextFactory.getInitialContext(Hashtable env)` method in a public thread local variable. This is used by EJB handles and other JNDI sensitive objects like the `UserTransaction` factory to keep track of the JNDI context that was in effect when they were created. If you want this environment to be bound to the object even after its serialized across vm boundaries, then you should the `org.jboss.naming.NamingContextFactory`. If you want the environment that is defined in the current VM `jndi.properties` or system properties, then you should use the `org.jnp.interfaces.NamingContextFactory` version.

3.3. Naming Discovery in Clustered Environments

When running in a clustered JBoss environment, you can choose not to specify a `Context.PROVIDER_URL` value and let the client query the network for available naming services. This only works with JBoss servers running with the `all` configuration, or an equivalent configuration that has `org.jboss.ha.framework.server.ClusterPartition` and `org.jboss.ha.jndi.HANamingService` services deployed. The discovery process consists of sending a multicast request packet to the discovery address/port and waiting for any node to respond. The response is a HA-RMI version of the `Naming` interface. The following `InitialContext` properties affect the discovery configuration:

- **jnp.partitionName**: The cluster partition name discovery should be restricted to. If you are running in an environment with multiple clusters, you may want to restrict the naming discovery to a particular cluster. There is no default value, meaning that any cluster response will be accepted.
- **jnp.discoveryGroup**: The multicast IP/address to which the discovery query is sent. The default is 230.0.0.4.

- **jnp.discoveryPort**: The port to which the discovery query is sent. The default is 1102.
- **jnp.discoveryTimeout**: The time in milliseconds to wait for a discovery query response. The default value is 5000 (5 seconds).
- **jnp.disableDiscovery**: A flag indicating if the discovery process should be avoided. Discovery occurs when either no `Context.PROVIDER_URL` is specified, or no valid naming service could be located among the URLs specified. If the `jnp.disableDiscovery` flag is true, then discovery will not be attempted.

3.4. The HTTP InitialContext Factory Implementation

The JNDI naming service can be accessed over HTTP. From a JNDI client's perspective this is a transparent change as they continue to use the JNDI `Context` interface. Operations through the `Context` interface are translated into HTTP posts to a servlet that passes the request to the NamingService using its JMX invoke operation. Advantages of using HTTP as the access protocol include better access through firewalls and proxies setup to allow HTTP, as well as the ability to secure access to the JNDI service using standard servlet role based security.

To access JNDI over HTTP you use the `org.jboss.naming.HttpNamingContextFactory` as the factory implementation. The complete set of support `InitialContext` environment properties for this factory are:

- **java.naming.factory.initial**: The name of the environment property for specifying the initial context factory, which must be `org.jboss.naming.HttpNamingContextFactory`.
- **java.naming.provider.url** (or `Context.PROVIDER_URL`): This must be set to the HTTP URL of the JNDI factory. The full HTTP URL would be the public URL of the JBoss servlet container plus `/invoker/JNDIFactory`. Examples include:
  - `http://www.jboss.org:8080/invoker/JNDIFactory`
  - `http://www.jboss.org/invoker/JNDIFactory`
  - `https://www.jboss.org/invoker/JNDIFactory`

  The first example accesses the servlet using the port 8080. The second uses the standard HTTP port 80, and the third uses an SSL encrypted connection to the standard HTTPS port 443.
- **java.naming.factory.url.pkgs**: For all JBoss JNDI provider this must be `org.jboss.naming:org.jnp.interfaces`. This property is essential for locating the `jnp:` and `java:` URL context factories of the JBoss JNDI provider.

The JNDI `Context` implementation returned by the `HttpNamingContextFactory` is a proxy that delegates invocations made on it to a bridge servlet which forwards the invocation to the `NamingService` through the JMX bus and marshalls the reply back over HTTP. The proxy needs to know what the URL of the bridge servlet is in order to operate. This value may have been bound on the server side if the JBoss web server has a well known public interface. If the JBoss web server is sitting behind one or more firewalls or proxies, the proxy cannot know what

URL is required. In this case, the proxy will be associated with a system property value that must be set in the client VM. For more information on the operation of JNDI over HTTP see *Section 4.1, "Accessing JNDI over HTTP"*.

3.5. The Login InitialContext Factory Implementation

JAAS is the preferred method for authenticating a remote client to JBoss. However, for simplicity and to ease the migration from other application server environment that do not use JAAS, JBoss allows you the security credentials to be passed through the `InitialContext`. JAAS is still used under the covers, but there is no manifest use of the JAAS interfaces in the client application.

The factory class that provides this capability is the `org.jboss.security.jndi.LoginInitialContextFactory`. The complete set of support `InitialContext` environment properties for this factory are:

- **java.naming.factory.initial**: The name of the environment property for specifying the initial context factory, which must be `org.jboss.security.jndi.LoginInitialContextFactory`.
- **java.naming.provider.url**: This must be set to a `NamingContextFactory` provider URL. The `LoginIntialContext` is really just a wrapper around the `NamingContextFactory` that adds a JAAS login to the existing `NamingContextFactory` behavior.
- **java.naming.factory.url.pkgs**: For all JBoss JNDI provider this must be `org.jboss.naming:org.jnp.interfaces`. This property is essential for locating the `jnp:` and `java:` URL context factories of the JBoss JNDI provider.
- **java.naming.security.principal** (or `Context.SECURITY_PRINCIPAL`): The principal to authenticate. This may be either a `java.security.Principal` implementation or a string representing the name of a principal.
- **java.naming.security.credentials** (or `Context.SECURITY_CREDENTIALS`), The credentials that should be used to authenticate the principal, e.g., password, session key, etc.
- **java.naming.security.protocol**: (`Context.SECURITY_PROTOCOL`) This gives the name of the JAAS login module to use for the authentication of the principal and credentials.

3.6. The ORBInitialContextFactory

When using Sun's CosNaming it is necessary to use a different naming context factory from the default. CosNaming looks for the ORB in JNDI instead of using the the ORB configured in `deploy/iiop-service.xml?`. It is necessary to set the global context factory to `org.jboss.iiop.naming.ORBInitialContextFactory`, which sets the ORB to JBoss's ORB. This is done in the `conf/jndi.properties` file:

```
# DO NOT EDIT THIS FILE UNLESS YOU KNOW WHAT YOU ARE DOING
#
java.naming.factory.initial=org.jboss.iiop.naming.ORBInitialContextFactory
```

```
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

It is also necessary to use `ORBInitialContextFactory` when using CosNaming in an application client.

4. JNDI over HTTP

In addition to the legacy RMI/JRMP with a socket bootstrap protocol, JBoss provides support for accessing its JNDI naming service over HTTP.

4.1. Accessing JNDI over HTTP

This capability is provided by `http-invoker.sar`. The structure of the `http-invoker.sar` is:

```
http-invoker.sar
+- META-INF/jboss-service.xml
+- invoker.war
| +- WEB-INF/jboss-web.xml
| +- WEB-INF/classes/org/jboss/invocation/http/servlet/InvokerServlet.class
| +-
WEB-INF/classes/org/jboss/invocation/http/servlet/NamingFactoryServlet.class
| +-
WEB-INF/classes/org/jboss/invocation/http/servlet/ReadOnlyAccessFilter.class
| +- WEB-INF/classes/roles.properties
| +- WEB-INF/classes/users.properties
| +- WEB-INF/web.xml
| +- META-INF/MANIFEST.MF
+- META-INF/MANIFEST.MF
```

The `jboss-service.xml` descriptor defines the `HttpInvoker` and `HttpInvokerHA` MBeans. These services handle the routing of methods invocations that are sent via HTTP to the appropriate target MBean on the JMX bus.

The `http-invoker.war` web application contains servlets that handle the details of the HTTP transport. The `NamingFactoryServlet` handles creation requests for the JBoss JNDI naming service `javax.naming.Context` implementation. The `InvokerServlet` handles invocations made by RMI/HTTP clients. The `ReadOnlyAccessFilter` allows one to secure the JNDI naming service while making a single JNDI context available for read-only access by unauthenticated clients.

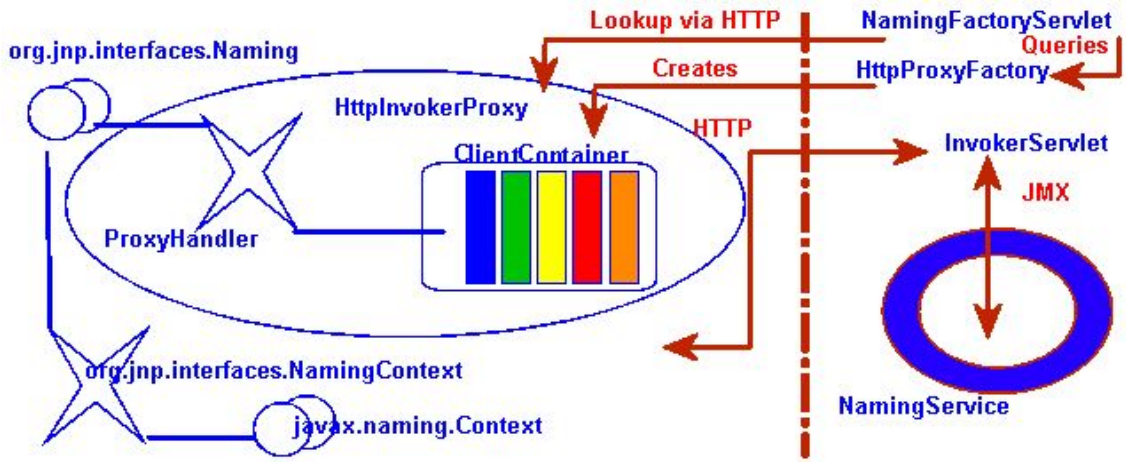


Figure 4.2. The HTTP invoker proxy/server structure for a JNDI Context

Before looking at the configurations let's look at the operation of the `http-invoker` services. *Figure 4.2, "The HTTP invoker proxy/server structure for a JNDI Context"* shows a logical view of the structure of a JBoss JNDI proxy and its relationship to the JBoss server side components of the `http-invoker`. The proxy is obtained from the `NamingFactoryServlet` using an `InitialContext` with the `Context.INITIAL_CONTEXT_FACTORY` property set to `org.jboss.naming.HttpNamingContextFactory`, and the `Context.PROVIDER_URL` property set to the HTTP URL of the `NamingFactoryServlet`. The resulting proxy is embedded in an `org.jnp.interfaces.NamingContext` instance that provides the `Context` interface implementation.

The proxy is an instance of `org.jboss.invocation.http.interfaces.HttpInvokerProxy`, and implements the `org.jnp.interfaces.Naming` interface. Internally the `HttpInvokerProxy` contains an invoker that marshalls the `Naming` interface method invocations to the `InvokerServlet` via HTTP posts. The `InvokerServlet` translates these posts into JMX invocations to the `NamingService`, and returns the invocation response back to the proxy in the HTTP post response.

There are several configuration values that need to be set to tie all of these components together and *Figure 4.3, "The relationship between configuration files and JNDI/HTTP component"* illustrates the relationship between configuration files and the corresponding components.

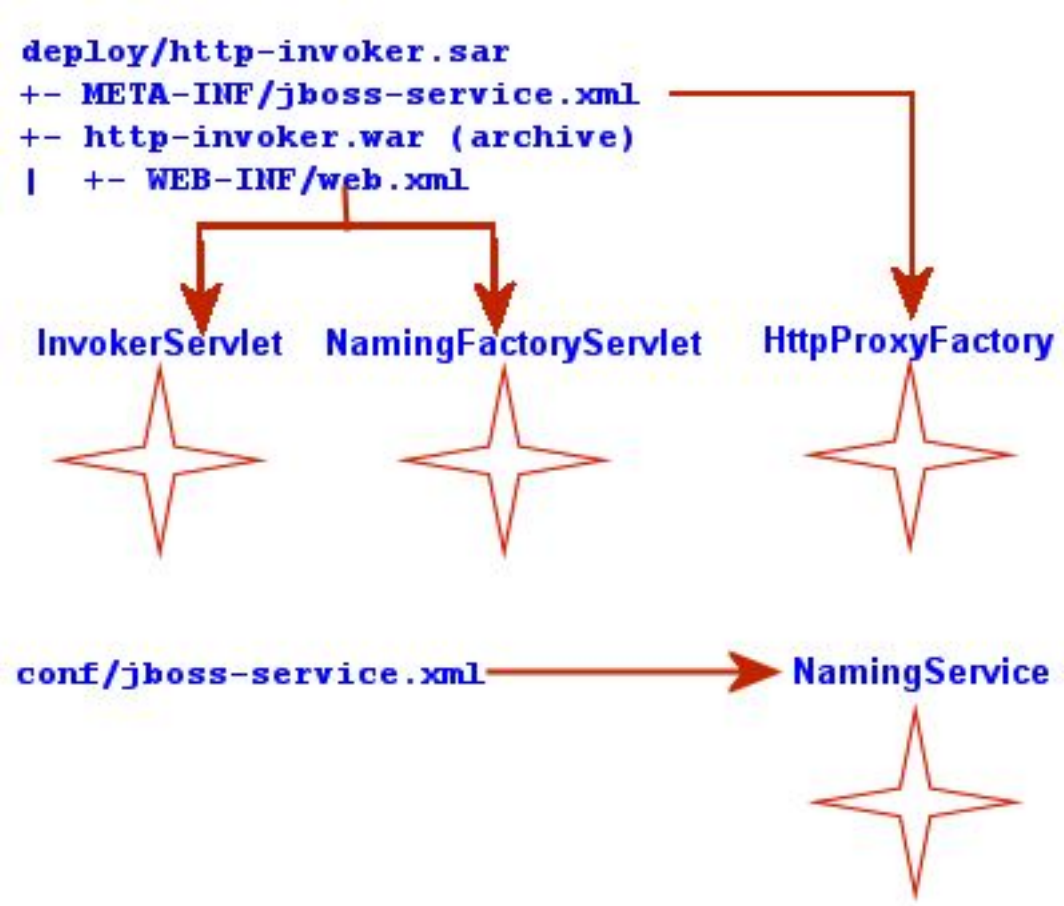


Figure 4.3. The relationship between configuration files and JNDI/HTTP component

The `http-invoker.sar/META-INF/jboss-service.xml` descriptor defines the `HttpProxyFactory` that creates the `HttpInvokerProxy` for the `NamingService`. The attributes that need to be configured for the `HttpProxyFactory` include:

- **InvokerName**: The JMX `ObjectName` of the `NamingService` defined in the `conf/jboss-service.xml` descriptor. The standard setting used in the JBoss distributions is `jboss:service=Naming`.
- **InvokerURL** or **InvokerURLPrefix** + InvokerURLSuffix + **UseHostName**. You can specify the full HTTP URL to the `InvokerServlet` using the `InvokerURL` attribute, or you can specify the hostname independent parts of the URL and have the `HttpProxyFactory` fill them in. An example `InvokerURL` value would be `http://jbosshost1.dot.com:8080/invoker/JMXInvokerServlet`. This can be broken down into:

- **InvokerURLPrefix**: the URL prefix prior to the hostname. Typically this will be `http://` or `https://` if SSL is to be used.
- **InvokerURLSuffix**: the URL suffix after the hostname. This will include the port number of the web server as well as the deployed path to the `InvokerServlet` . For the example `InvokerURL` value the `InvokerURLSuffix` would be `:8080/invoker/JMXInvokerServlet` without the quotes. The port number is determined by the web container service settings. The path to the `InvokerServlet` is specified in the `http-invoker.sar/invoker.war/WEB-INF/web.xml` descriptor.
- **UseHostName**: a flag indicating if the hostname should be used in place of the host IP address when building the hostname portion of the full `InvokerURL`. If true, `InetAddress.getLocalHost().getHostName` method will be used. Otherwise, the `InetAddress.getLocalHost().getHostAddress()` method is used.

- **ExportedInterface**: The `org.jnp.interfaces.Naming` interface the proxy will expose to clients. The actual client of this proxy is the JBoss JNDI implementation `NamingContext` class, which JNDI client obtain from `InitialContext` lookups when using the JBoss JNDI provider.
- **JndiName**: The name in JNDI under which the proxy is bound. This needs to be set to a blank/empty string to indicate the interface should not be bound into JNDI. We can't use the JNDI to bootstrap itself. This is the role of the `NamingFactoryServlet`.

The `http-invoker.sar/invoker.war/WEB-INF/web.xml` descriptor defines the mappings of the `NamingFactoryServlet` and `InvokerServlet` along with their initialization parameters. The configuration of the `NamingFactoryServlet` relevant to JNDI/HTTP is the `JNDIFactory` entry which defines:

- A `namingProxyMBean` initialization parameter that maps to the `HttpProxyFactory` MBean name. This is used by the `NamingFactoryServlet` to obtain the `Naming` proxy which it will return in response to HTTP posts. For the default `http-invoker.sar/META-INF/jboss-service.xml` settings the name `jboss:service=invoker,type=http,target=Naming`.
- A proxy initialization parameter that defines the name of the `namingProxyMBean` attribute to query for the Naming proxy value. This defaults to an attribute name of `Proxy`.
- The servlet mapping for the `JNDIFactory` configuration. The default setting for the unsecured mapping is `/JNDIFactory/*`. This is relative to the context root of the `http-invoker.sar/invoker.war`, which by default is the WAR name minus the `.war` suffix.

The configuration of the `InvokerServlet` relevant to JNDI/HTTP is the `JMXInvokerServlet` which defines:

- The servlet mapping of the `InvokerServlet`. The default setting for the unsecured mapping is `/JMXInvokerServlet/*`. This is relative to the context root of the `http-invoker.sar/invoker.war`, which by default is the WAR name minus the `.war` suffix.

4.2. Accessing JNDI over HTTPS

To be able to access JNDI over HTTP/SSL you need to enable an SSL connector on the web container. The details of this are covered in the Integrating Servlet Containers for Tomcat. We will demonstrate the use of HTTPS with a simple example client that uses an HTTPS URL as the JNDI provider URL. We will provide an SSL connector configuration for the example, so unless you are interested in the details of the SSL connector setup, the example is self contained.

We also provide a configuration of the `HttpProxyFactory` setup to use an HTTPS URL. The following example shows the section of the `http-invoker.sarjboss-service.xml` descriptor that the example installs to provide this configuration. All that has changed relative to the standard HTTP configuration are the `InvokerURLPrefix` and `InvokerURLSuffix` attributes, which setup an HTTPS URL using the 8443 port.

```
<!-- Expose the Naming service interface via HTTPS -->
<mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
       name="jboss:service=invoker,type=https,target=Naming">
    <!-- The Naming service we are proxying -->
    <attribute name="InvokerName">jboss:service=Naming</attribute>
    <!-- Compose the invoker URL from the cluster node address -->
    <attribute name="InvokerURLPrefix">https://</attribute>
    <attribute name="InvokerURLSuffix">:8443/invoker/JMXInvokerServlet
</attribute>
    <attribute name="UseHostName">true</attribute>
    <attribute name="ExportedInterface">org.jnp.interfaces.Naming
</attribute>
    <attribute name="JndiName"/>
    <attribute name="ClientInterceptors">
        <interceptors>
            <interceptor>org.jboss.proxy.ClientMethodInterceptor
</interceptor>
            <interceptor>org.jboss.proxy.SecurityInterceptor
</interceptor>
            <interceptor>org.jboss.naming.interceptors.ExceptionInterceptor
</interceptor>
            <interceptor>org.jboss.invocation.InvokerInterceptor
</interceptor>
        </interceptors>
    </attribute>
</mbean>
```

At a minimum, a JNDI client using HTTPS requires setting up a HTTPS URL protocol handler. We will be using the Java Secure Socket Extension (JSSE) for HTTPS. The JSSE documentation does a good job of describing what is necessary to use HTTPS, and the following steps were needed to configure the example client shown in *Example 4.2, “A JNDI*

client that uses HTTPS as the transport":

- A protocol handler for HTTPS URLs must be made available to Java. The JSSE release includes an HTTPS handler in the `com.sun.net.ssl.internal.www.protocol` package. To enable the use of HTTPS URLs you include this package in the standard URL protocol handler search property, `java.protocol.handler.pkgs`. We set the `java.protocol.handler.pkgs` property in the Ant script.
- The JSSE security provider must be installed in order for SSL to work. This can be done either by installing the JSSE jars as an extension package, or programatically. We use the programatic approach in the example since this is less intrusive. Line 18 of the `ExClient` code demonstrates how this is done.
- The JNDI provider URL must use HTTPS as the protocol. Lines 24-25 of the `ExClient` code specify an HTTP/SSL connection to the localhost on port 8443. The hostname and port are defined by the web container SSL connector.
- The validation of the HTTPS URL hostname against the server certificate must be disabled. By default, the JSSE HTTPS protocol handler employs a strict validation of the hostname portion of the HTTPS URL against the common name of the server certificate. This is the same check done by web browsers when you connect to secured web site. We are using a self-signed server certificate that uses a common name of "`Chapter 8 SSL Example`" rather than a particular hostname, and this is likely to be common in development environments or intranets. The JBoss `HttpInvokerProxy` will override the default hostname checking if a `org.jboss.security.ignoreHttpsHost` system property exists and has a value of true. We set the `org.jboss.security.ignoreHttpsHost` property to true in the Ant script.

```
package org.jboss.chap3.ex1;

import java.security.Security;
import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;

public class ExClient
{
    public static void main(String args[]) throws Exception
    {
        Properties env = new Properties();
        env.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                        "org.jboss.naming.HttpNamingContextFactory");
        env.setProperty(Context.PROVIDER_URL,
                        "https://localhost:8443/invoker/JNDIFactorySSL");

        Context ctx = new InitialContext(env);
        System.out.println("Created InitialContext, env=" + env);

        Object data = ctx.lookup("jmx/invoker/RMIAdaptor");
        System.out.println("lookup(jmx/invoker/RMIAdaptor): " + data);
    }
```

```
}
```

Example 4.2. A JNDI client that uses HTTPS as the transport

To test the client, first build the chapter 3 example to create the `chap3` configuration fileset.

```
[examples]$ ant -Dchap=naming config
```

Next, start the JBoss server using the `naming` configuration fileset:

```
[bin]$ sh run.sh -c naming
```

And finally, run the `ExClient` using:

```
[examples]$ ant -Dchap=naming -Dex=1 run-example
...
run-example1:

[java] Created InitialContext, env={java.naming. \
provider.url=https://localhost:8443/invoker/JNDIFactorySSL, java.naming. \
factory.initial=org.jboss.naming.HttpNamingContextFactory}
     [java] lookup(jmx/invoker/RMIAdaptor): org.jboss.invocation.jrmp. \
     interfaces.JRMPInvokerP
roxy@cac3fa
```

4.3. Securing Access to JNDI over HTTP

One benefit to accessing JNDI over HTTP is that it is easy to secure access to the JNDI `InitialContext` factory as well as the naming operations using standard web declarative security. This is possible because the server side handling of the JNDI/HTTP transport is implemented with two servlets. These servlets are included in the `http-invoker.sar/invoker.war` directory found in the `default` and `all` configuration deploy directories as shown previously. To enable secured access to JNDI you need to edit the `invoker.war/WEB-INF/web.xml` descriptor and remove all unsecured servlet mappings. For example, the `web.xml` descriptor shown in *Example 4.3, "An example web.xml descriptor for secured access to the JNDI servlets"* only allows access to the `invoker.war` servlets if the user has been authenticated and has a role of `HttpInvoker`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
          "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
          "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
```

```
    <!-- ### Servlets -->
    <servlet>
        <servlet-name>JMXInvokerServlet</servlet-name>
        <servlet-class>
            org.jboss.invocation.http.servlet.InvokerServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>   <servlet>
        <servlet-name>JNDIFactory</servlet-name>
        <servlet-class>
            org.jboss.invocation.http.servlet.NamingFactoryServlet
        </servlet-class>
        <init-param>
            <param-name>namingProxyMBean</param-name>
<param-value>jboss:service=invoker,type=http,target=Naming</param-value>
        </init-param>
        <init-param>
            <param-name>proxyAttribute</param-name>
            <param-value>Proxy</param-value>
        </init-param>
        <load-on-startup>2</load-on-startup>
    </servlet>
    <!-- ### Servlet Mappings -->
    <servlet-mapping>
        <servlet-name>JNDIFactory</servlet-name>
        <url-pattern>/restricted/JNDIFactory/*</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>JMXInvokerServlet</servlet-name>
        <url-pattern>/restricted/JMXInvokerServlet/*</url-pattern>
    </servlet-mapping>   <security-constraint>
        <web-resource-collection>
            <web-resource-name>HttpInvokers</web-resource-name>
            <description>An example security config that only allows users
with
                the role HttpInvoker to access the HTTP invoker servlets
</description>
            <url-pattern>/restricted/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>HttpInvoker</role-name>
        </auth-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>JBoss HTTP Invoker</realm-name>
    </login-config>   <security-role>
        <role-name>HttpInvoker</role-name>
    </security-role>
</web-app>
```

Example 4.3. An example web.xml descriptor for secured access to the JNDI servlets

The `web.xml` descriptor only defines which sevlets are secured, and which roles are allowed to access the secured servlets. You must additionally define the security domain that will handle the authentication and authorization for the war. This is done through the `jboss-web.xml` descriptor, and an example that uses the `http-invoker` security domain is given below.

```
<jboss-web>
    <security-domain>java:/jaas/http-invoker</security-domain>
</jboss-web>
```

The `security-domain` element defines the name of the security domain that will be used for the JAAS login module configuration used for authentication and authorization. See *Section 1.6, "Enabling Declarative Security in JBoss"* for additional details on the meaning and configuration of the security domain name.

4.4. Securing Access to JNDI with a Read-Only Unsecured Context

Another feature available for the JNDI/HTTP naming service is the ability to define a context that can be accessed by unauthenticated users in read-only mode. This can be important for services used by the authentication layer. For example, the `SRPLoginModule` needs to lookup the SRP server interface used to perform authentication. We'll now walk through how read-only JNDI works in JBoss.

First, the `ReadOnlyJNDIFactory` is declared in `invoker.sar/WEB-INF/web.xml`. It will be mapped to `/invoker/ReadOnlyJNDIFactory`.

```
<servlet>
    <servlet-name>ReadOnlyJNDIFactory</servlet-name>
    <description>A servlet that exposes the JBoss JNDI Naming service stub
          through http, but only for a single read-only context. The return
content
          is serialized MarshalledValue containing the
org.jnp.interfaces.Naming
          stub.
    </description>
<servlet-class>org.jboss.invocation.http.servlet.NamingFactoryServlet</servlet-class>
    <init-param>
        <param-name>namingProxyMBean</param-name>
        <param-value>jboss:service=invoker,type=http,target=Naming,readonly=true</param-valu
    </init-param>
    <init-param>
        <param-name>proxyAttribute</param-name>
        <param-value>Proxy</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
```

```
</servlet>

<!-- ... -->

<servlet-mapping>
    <servlet-name>ReadOnlyJNDIFactory</servlet-name>
    <url-pattern>/ReadOnlyJNDIFactory/*</url-pattern>
</servlet-mapping>
```

The factory only provides a JNDI stub which needs to be connected to an invoker. Here the invoker is `jboss:service=invoker,type=http,target=Naming,readonly=true`. This invoker is declared in the `http-invoker.sar/META-INF/jboss-service.xml` file.

```
   <mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
      name="jboss:service=invoker,type=http,target=Naming,readonly=true">
      <attribute name="InvokerName">jboss:service=Naming</attribute>
      <attribute name="InvokerURLPrefix">http://</attribute>
      <attribute
name="InvokerURLSuffix">:8080/invoker/readonly/JMXInvokerServlet</attribute>
      <attribute name="UseHostName">true</attribute>
      <attribute
name="ExportedInterface">org.jnp.interfaces.Naming</attribute>
      <attribute name="JndiName"></attribute>
      <attribute name="ClientInterceptors">
          <interceptors>
<interceptor>org.jboss.proxy.ClientMethodInterceptor</interceptor>
             <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
<interceptor>org.jboss.naming.interceptors.ExceptionInterceptor</interceptor>
<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
          </interceptors>
      </attribute>
   </mbean>
```

The proxy on the client side needs to talk back to a specific invoker servlet on the server side. The configuration here has the actual invocations going to `/invoker/readonly/JMXInvokerServlet`. This is actually the standard `JMXInvokerServlet` with a read-only filter attached.

```
    <filter>
        <filter-name>ReadOnlyAccessFilter</filter-name>
<filter-class>org.jboss.invocation.http.servlet.ReadOnlyAccessFilter</filter-class>
        <init-param>
            <param-name>readOnlyContext</param-name>
            <param-value>readonly</param-value>
            <description>The top level JNDI context the filter will enforce
                read-only access on. If specified only Context.lookup
operations
                will be allowed on this context. Another other operations or
                lookups on any other context will fail. Do not associate
this
                filter with the JMXInvokerServlets if you want unrestricted
                access. </description>
        </init-param>
```

```
        <init-param>
            <param-name>invokerName</param-name>
            <param-value>jboss:service=Naming</param-value>
            <description>The JMX ObjectName of the naming service mbean
</description>
        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>ReadOnlyAccessFilter</filter-name>
        <url-pattern>/readonly/*</url-pattern>
    </filter-mapping>

    <!-- ... -->
    <!-- A mapping for the JMXInvokerServlet that only allows invocations
            of lookups under a read-only context. This is enforced by the
            ReadOnlyAccessFilter
            -->
    <servlet-mapping>
        <servlet-name>JMXInvokerServlet</servlet-name>
        <url-pattern>/readonly/JMXInvokerServlet/*</url-pattern>
    </servlet-mapping>
```

The `readOnlyContext` parameter is set to `readonly` which means that when you access JBoss through the `ReadOnlyJNDIFactory`, you will only be able to access data in the `readonly` context. Here is a code fragment that illustrates the usage:

```
Properties env = new Properties();
env.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                "org.jboss.naming.HttpNamingContextFactory");
env.setProperty(Context.PROVIDER_URL,
                "http://localhost:8080/invoker/ReadOnlyJNDIFactory");

Context ctx2 = new InitialContext(env);
Object data = ctx2.lookup("readonly/data");
```

Attempts to look up any objects outside of the readonly context will fail. Note that JBoss doesn't ship with any data in the `readonly` context, so the readonly context won't be bound usable unless you create it.

5. Additional Naming MBeans

In addition to the `NamingService` MBean that configures an embedded JBossNS server within JBoss, there are several additional MBean services related to naming that ship with JBoss. They are `JndiBindingServiceMgr`, `NamingAlias`, `ExternalContext`, and `JNDIView`.

5.1. JNDI Binding Manager

The JNDI binding manager service allows you to quickly bind objects into JNDI for use by application code. The MBean class for the binding service is

`org.jboss.naming.JNDIBindingServiceMgr`. It has a single attribute, `BindingsConfig`, which accepts an XML document that conforms to the `jndi-binding-service_1_0.xsd` schema. The content of the `BindingsConfig` attribute is unmarshalled using the JBossXB framework. The following is an MBean definition that shows the most basic form usage of the JNDI binding manager service.

```
<mbean code="org.jboss.naming.JNDIBindingServiceMgr"
       name="jboss.tests:name=example1">
    <attribute name="BindingsConfig" serialDataType="jbxb">
        <jndi:bindings xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
                       xmlns:jndi="urn:jboss:jndi-binding-service"
                       xs:schemaLocation="urn:jboss:jndi-binding-service  \
                       resource:jndi-binding-service_1_0.xsd">
            <jndi:binding name="bindexample/message">
                <jndi:value trim="true">
                    Hello, JNDI!
                </jndi:value>
            </jndi:binding>
        </jndi:bindings>
    </attribute>
</mbean>
```

This binds the text string "`Hello, JNDI!`" under the JNDI name `bindexample/message`. An application would look up the value just as it would for any other JNDI value. The `trim` attribute specifies that leading and trailing whitespace should be ignored. The use of the attribute here is purely for illustrative purposes as the default value is true.

```
InitialContext ctx  = new InitialContext();
String         text = (String) ctx.lookup("bindexample/message");
```

String values themselves are not that interesting. If a JavaBeans property editor is available, the desired class name can be specified using the `type` attribute

```
<jndi:binding name="urls/jboss-home">
    <jndi:value type="java.net.URL">http://www.jboss.org</jndi:value>
</jndi:binding>
```

The `editor` attribute can be used to specify a particular property editor to use.

```
<jndi:binding name="hosts/localhost">
    <jndi:value editor="org.jboss.util.propertyeditor.InetAddressEditor">
        127.0.0.1
    </jndi:value>
</jndi:binding>
```

For more complicated structures, any JBossXB-ready schema may be used. The following example shows how a `java.util.Properties` object would be mapped.

```
<jndi:binding name="maps/testProps">
```

```
    <java:properties xmlns:java="urn:jboss:java-properties"
                     xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
                     xs:schemaLocation="urn:jboss:java-properties \
                     resource:java-properties_1_0.xsd">
        <java:property>
            <java:key>key1</java:key>
            <java:value>value1</java:value>
        </java:property>
        <java:property>
            <java:key>key2</java:key>
            <java:value>value2</java:value>
        </java:property>
    </java:properties>
</jndi:binding>
```

For more information on JBossXB, see the *JBossXB wiki page* [http://wiki.jboss.org/wiki/Wiki.jsp?page=JBossXB].

5.2. The org.jboss.naming.NamingAlias MBean

The `NamingAlias` MBean is a simple utility service that allows you to create an alias in the form of a JNDI `javax.naming.LinkRef` from one JNDI name to another. This is similar to a symbolic link in the UNIX file system. To an alias you add a configuration of the `NamingAlias` MBean to the `jboss-service.xml` configuration file. The configurable attributes of the `NamingAlias` service are as follows:

- **FromName**: The location where the `LinkRef` is bound under JNDI.
- **ToName**: The to name of the alias. This is the target name to which the `LinkRef` refers. The name is a URL, or a name to be resolved relative to the `InitialContext`, or if the first character of the name is a dot (`.`), the name is relative to the context in which the link is bound.

The following example provides a mapping of the JNDI name `QueueConnectionFactory` to the name `ConnectionFactory`.

```
<mbean code="org.jboss.naming.NamingAlias"
       name="jboss.mq:service=NamingAlias,fromName=QueueConnectionFactory">
    <attribute name="ToName">ConnectionFactory</attribute>
    <attribute name="FromName">QueueConnectionFactory</attribute>
</mbean>
```

5.3. org.jboss.naming.ExternalContext MBean

The `ExternalContext` MBean allows you to federate external JNDI contexts into the JBoss server JNDI namespace. The term external refers to any naming service external to the JBossNS naming service running inside of the JBoss server VM. You can incorporate LDAP

servers, file systems, DNS servers, and so on, even if the JNDI provider root context is not serializable. The federation can be made available to remote clients if the naming service supports remote access.

To incorporate an external JNDI naming service, you have to add a configuration of the `ExternalContext` MBean service to the `jboss-service.xml` configuration file. The configurable attributes of the `ExternalContext` service are as follows:

- **JndiName**: The JNDI name under which the external context is to be bound.
- **RemoteAccess**: A boolean flag indicating if the external `InitialContext` should be bound using a `Serializable` form that allows a remote client to create the external `InitialContext` . When a remote client looks up the external context via the JBoss JNDI `InitialContext`, they effectively create an instance of the external `InitialContext` using the same env properties passed to the `ExternalContext` MBean. This will only work if the client can do a `new InitialContext(env)` remotely. This requires that the `Context.PROVIDER_URL` value of env is resolvable in the remote VM that is accessing the context. This should work for the LDAP example. For the file system example this most likely won't work unless the file system path refers to a common network path. If this property is not given it defaults to false.
- **CacheContext**: The `cacheContext` flag. When set to true, the external `Context` is only created when the MBean is started and then stored as an in memory object until the MBean is stopped. If cacheContext is set to false, the external `Context` is created on each lookup using the MBean properties and InitialContext class. When the uncached `Context` is looked up by a client, the client should invoke `close()` on the Context to prevent resource leaks.
- **InitialContext**: The fully qualified class name of the `InitialContext` implementation to use. Must be one of: `javax.naming.InitialContext`, `javax.naming.directory.InitialDirContext` or `javax.naming.ldap.InitialLdapContext`. In the case of the `InitialLdapContext` a null `Controls` array is used. The default is `javax.naming.InitialContex`.
- **Properties**: The `Properties` attribute contains the JNDI properties for the external `InitialContext`. The input should be the text equivalent to what would go into a `jndi.properties` file.
- **PropertiesURL**: This set the `jndi.properties` information for the external `InitialContext` from an external properties file. This is either a URL, string or a classpath resource name. Examples are as follows:
  - file:///config/myldap.properties
  - http://config.mycompany.com/myldap.properties
  - /conf/myldap.properties
  - myldap.properties

The MBean definition below shows a binding to an external LDAP context into the JBoss JNDI namespace under the name `external/ldap/jboss`.

```
<!-- Bind a remote LDAP server -->
<mbean code="org.jboss.naming.ExternalContext"
name="jboss.jndi:service=ExternalContext,jndiName=external/ldap/jboss">
    <attribute name="JndiName">external/ldap/jboss</attribute>
    <attribute name="Properties">
        java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
        java.naming.provider.url=ldap://ldaphost.jboss.org:389/o=jboss.org
        java.naming.security.principal=cn=Directory Manager
        java.naming.security.authentication=simple
        java.naming.security.credentials=secret
    </attribute>
    <attribute name="InitialContext"> javax.naming.ldap.InitialLdapContext
</attribute>
    <attribute name="RemoteAccess">true</attribute>
</mbean>
```

With this configuration, you can access the external LDAP context located at `ldap://ldaphost.jboss.org:389/o=jboss.org` from within the JBoss VM using the following code fragment:

```
InitialContext iniCtx = new InitialContext();
LdapContext ldapCtx = iniCtx.lookup("external/ldap/jboss");
```

Using the same code fragment outside of the JBoss server VM will work in this case because the `RemoteAccess` property was set to true. If it were set to false, it would not work because the remote client would receive a `Reference` object with an `ObjectFactory` that would not be able to recreate the external `InitialContext`

```
<!-- Bind the /usr/local file system directory  -->
<mbean code="org.jboss.naming.ExternalContext"
name="jboss.jndi:service=ExternalContext,jndiName=external/fs/usr/local">
    <attribute name="JndiName">external/fs/usr/local</attribute>
    <attribute name="Properties">
java.naming.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory
        java.naming.provider.url=file:///usr/local
    </attribute>
    <attribute name="InitialContext">javax.naming.IntialContext</attribute>
</mbean>
```

This configuration describes binding a local file system directory `/usr/local` into the JBoss JNDI namespace under the name `external/fs/usr/local`.

With this configuration, you can access the external file system context located at `file:///usr/local` from within the JBoss VM using the following code fragment:

```
InitialContext iniCtx = new InitialContext();
                Context ldapCtx = iniCtx.lookup("external/fs/usr/local");
```

Note that the use the Sun JNDI service providers, which must be downloaded from *http://java.sun.com/products/jndi/serviceproviders.html*. The provider JARs should be placed in the server configuration `lib` directory.

5.4. The org.jboss.naming.JNDIView MBean

The JNDIView MBean allows the user to view the JNDI namespace tree as it exists in the JBoss server using the JMX agent view interface. To view the JBoss JNDI namespace using the JNDIView MBean, you connect to the JMX Agent View using the http interface. The default settings put this at `http://localhost:8080/jmx-console/`. On this page you will see a section that lists the registered MBeans sorted by domain. It should look something like that shown in *Figure 4.4, "The JMX Console view of the configured JBoss MBeans"*.

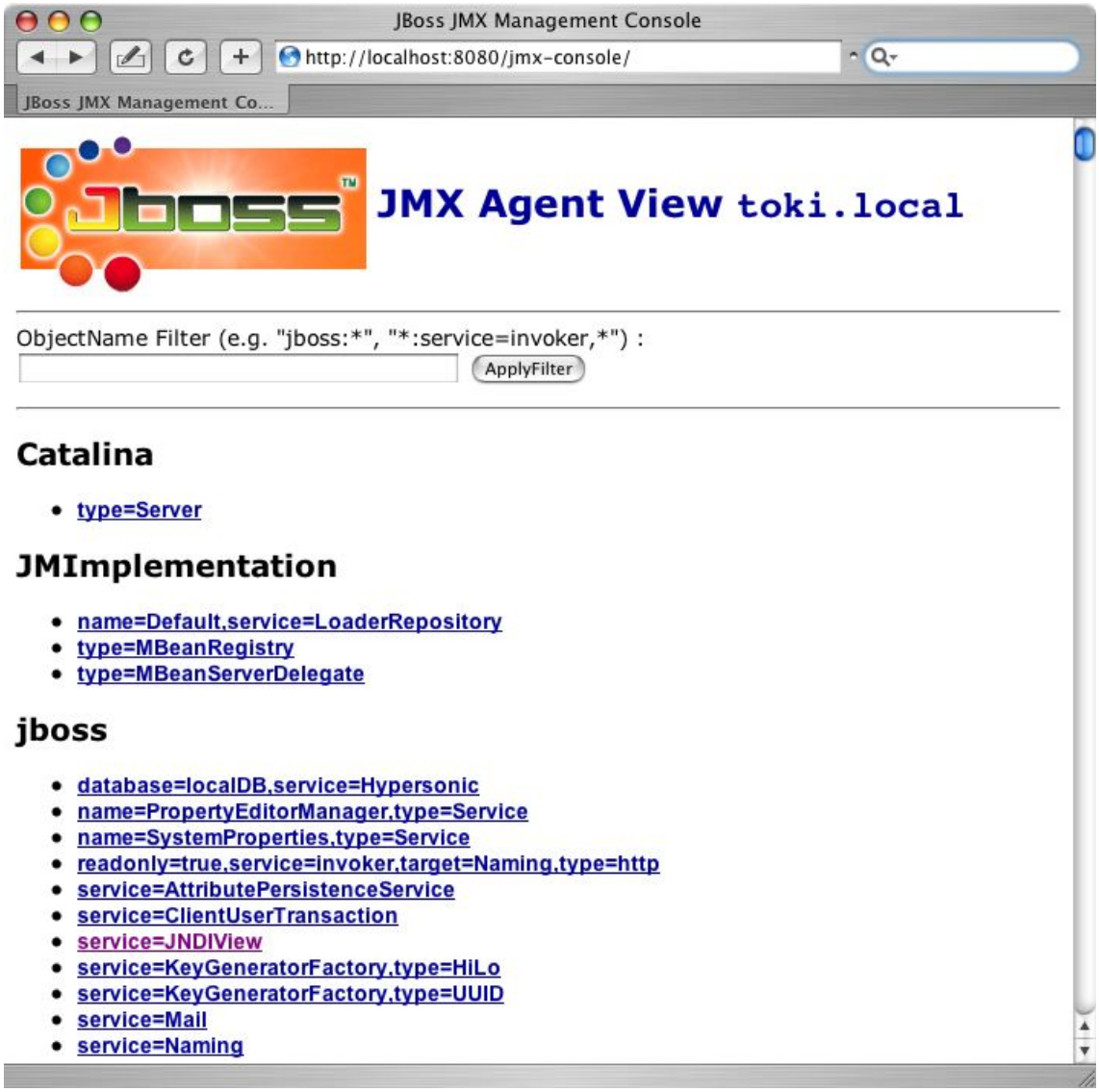


Figure 4.4. The JMX Console view of the configured JBoss MBeans

Selecting the JNDIView link takes you to the JNDIView MBean view, which will have a list of the JNDIView MBean operations. This view should look similar to that shown in *Figure 4.5, "The JMX Console view of the JNDIView MBean"*.

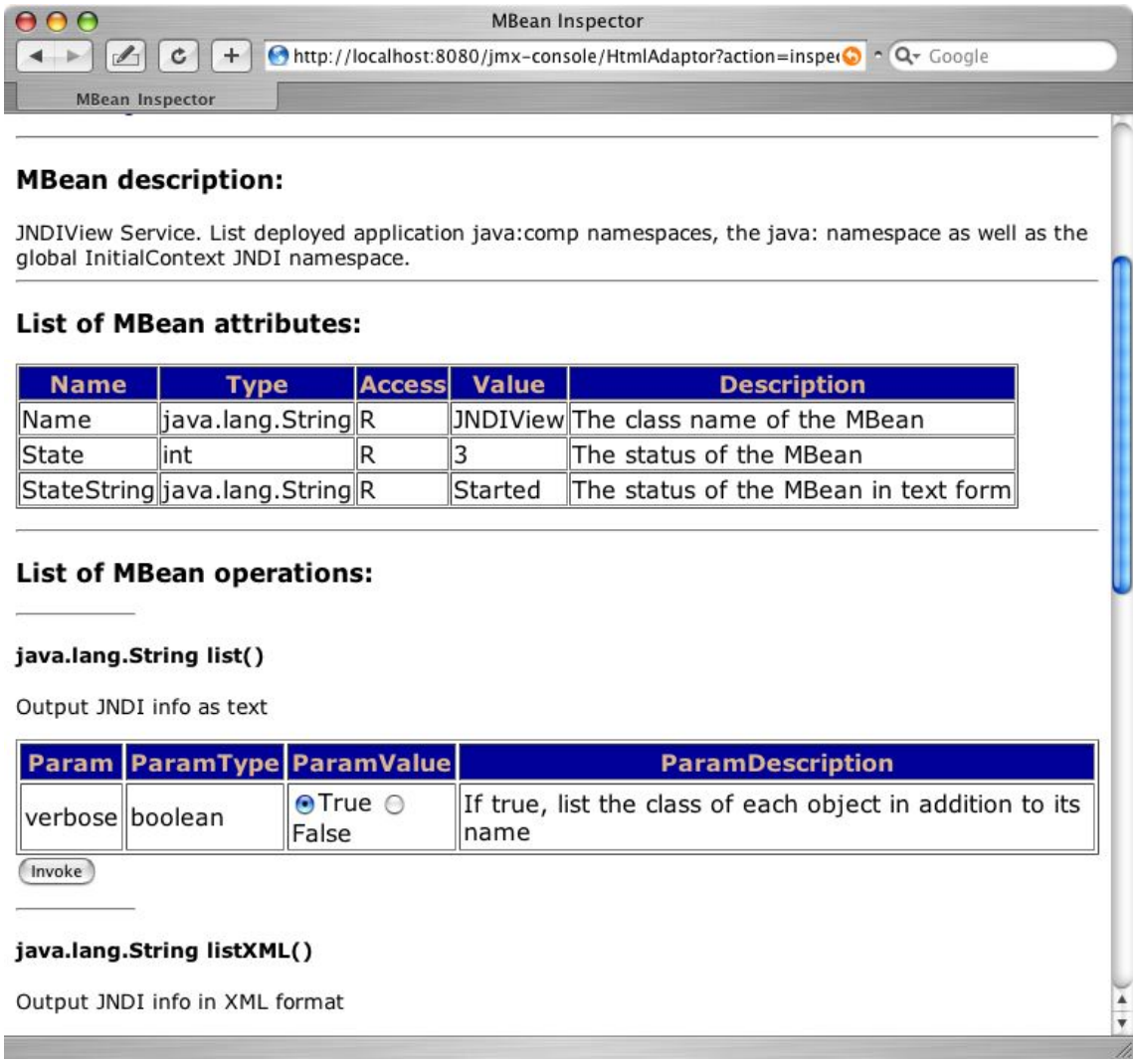


Figure 4.5. The JMX Console view of the JNDIView MBean

The list operation dumps out the JBoss server JNDI namespace as an HTML page using a simple text view. As an example, invoking the list operation produces the view shown in *Figure 4.6, "The JMX Console view of the JNDIView list operation output"*.

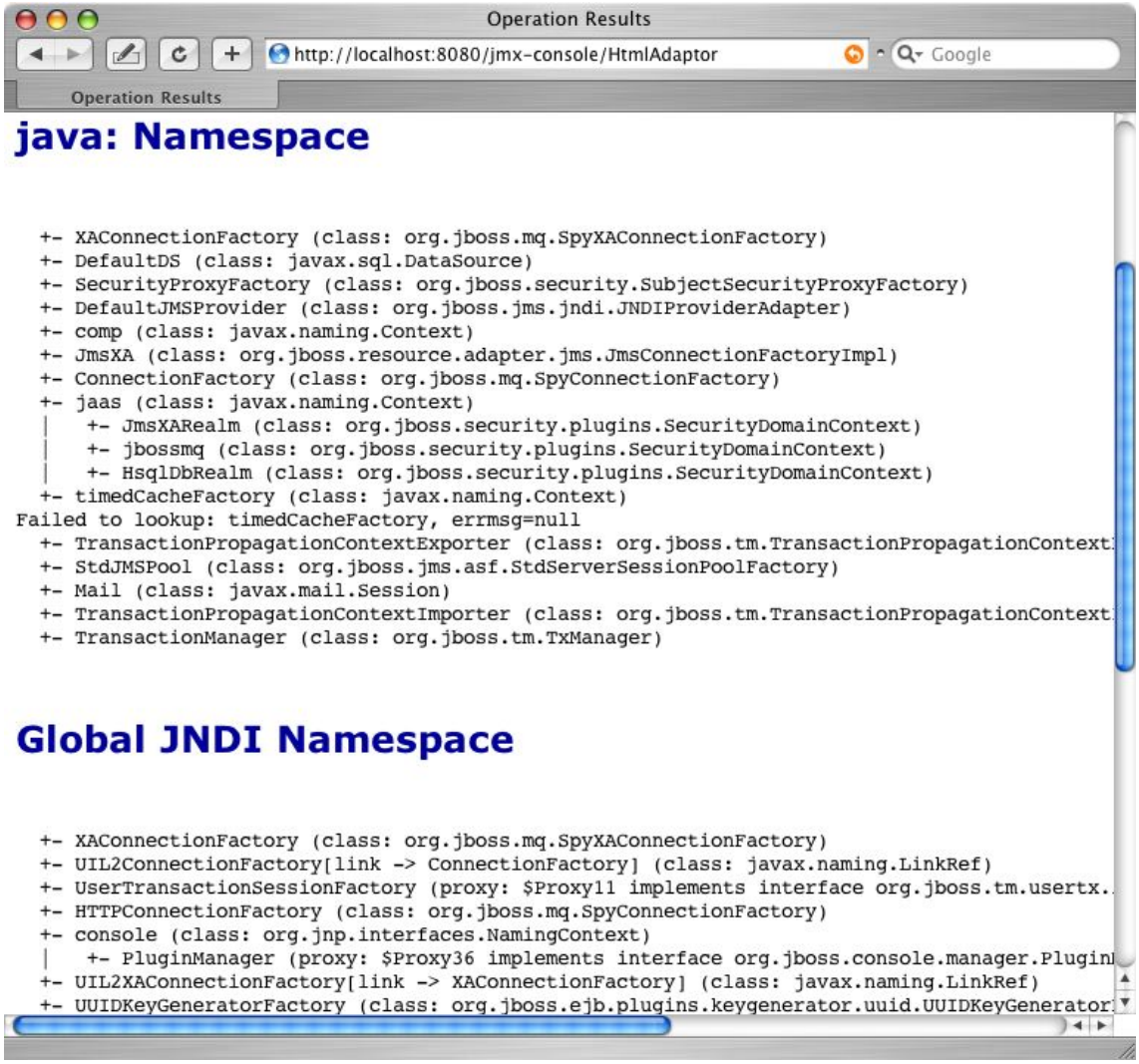


Figure 4.6. The JMX Console view of the JNDIView list operation output

6. J2EE and JNDI - The Application Component Environment

JNDI is a fundamental aspect of the J2EE specifications. One key usage is the isolation of J2EE component code from the environment in which the code is deployed. Use of the application component's environment allows the application component to be customized without the need to access or change the application component's source code. The application component environment is referred to as the ENC, the enterprise naming context. It is the responsibility of the application component container to make an ENC available to the container components in the form of JNDI Context. The ENC is utilized by the participants involved in the life cycle of a J2EE component in the following ways.

- Application component business logic should be coded to access information from its ENC. The component provider uses the standard deployment descriptor for the component to specify the required ENC entries. The entries are declarations of the information and resources the component requires at runtime.
- The container provides tools that allow a deployer of a component to map the ENC references made by the component developer to the deployment environment entity that satisfies the reference.
- The component deployer utilizes the container tools to ready a component for final deployment.
- The component container uses the deployment package information to build the complete component ENC at runtime

The complete specification regarding the use of JNDI in the J2EE platform can be found in section 5 of the J2EE 1.4 specification. The J2EE specification is available at *http://java.sun.com/j2ee/download.html*.

An application component instance locates the ENC using the JNDI API. An application component instance creates a `javax.naming.InitialContext` object by using the no argument constructor and then looks up the naming environment under the name `java:comp/env`. The application component's environment entries are stored directly in the ENC, or in its subcontexts. *Example 4.4, "ENC access sample code"* illustrates the prototypical lines of code a component uses to access its ENC.

```
// Obtain the application component's ENC
Context iniCtx = new InitialContext();
Context compEnv = (Context) iniCtx.lookup("java:comp/env");
```

Example 4.4. ENC access sample code

An application component environment is a local environment that is accessible only by the component when the application server container thread of control is interacting with the application component. This means that an EJB `Bean1` cannot access the ENC elements of EJB `Bean2`, and vice versa. Similarly, Web application `Web1` cannot access the ENC elements of Web application `Web2` or `Bean1` or `Bean2` for that matter. Also, arbitrary client code, whether it is executing inside of the application server VM or externally cannot access a component's `java:comp` JNDI context. The purpose of the ENC is to provide an isolated, read-only namespace that the application component can rely on regardless of the type of environment in which the component is deployed. The ENC must be isolated from other components because each component defines its own ENC content. Components `A` and `B`, for example, may define the same name to refer to different objects. For example, EJB `Bean1` may define an environment entry `java:comp/env/red` to refer to the hexadecimal value for the RGB color for red, while Web application `Web1` may bind the same name to the deployment environment

language locale representation of red.

There are three commonly used levels of naming scope in JBoss: names under `java:comp`, names under `java:`, and any other name. As discussed, the `java:comp` context and its subcontexts are only available to the application component associated with that particular context. Subcontexts and object bindings directly under `java:` are only visible within the JBoss server virtual machine and not to remote clients. Any other context or object binding is available to remote clients, provided the context or object supports serialization. You'll see how the isolation of these naming scopes is achieved in the *Section 2, "The JBossNS Architecture"*.

An example of where the restricting a binding to the `java:` context is useful would be a `javax.sql.DataSource` connection factory that can only be used inside of the JBoss server where the associated database pool resides. On the other hand, an EJB home interface would be bound to a globally visible name that should accessible by remote client.

6.1. ENC Usage Conventions

JNDI is used as the API for externalizing a great deal of information from an application component. The JNDI name that the application component uses to access the information is declared in the standard `ejb-jar.xml` deployment descriptor for EJB components, and the standard `web.xml` deployment descriptor for Web components. Several different types of information may be stored in and retrieved from JNDI including:

- Environment entries as declared by the `env-entry` elements
- EJB references as declared by `ejb-ref` and `ejb-local-ref` elements.
- Resource manager connection factory references as declared by the `resource-ref` elements
- Resource environment references as declared by the `resource-env-ref` elements

Each type of deployment descriptor element has a JNDI usage convention with regard to the name of the JNDI context under which the information is bound. Also, in addition to the standard deploymentdescriptor element, there is a JBoss server specific deployment descriptor element that maps the JNDI name as used by the application component to the deployment environment JNDI name.

6.1.1. Environment Entries

Environment entries are the simplest form of information stored in a component ENC, and are similar to operating system environment variables like those found on UNIX or Windows. Environment entries are a name-to-value binding that allows a component to externalize a value and refer to the value using a name.

An environment entry is declared using an `env-entry` element in the standard deployment descriptors. The `env-entry` element contains the following child elements:

- An optional **description** element that provides a description of the entry
- An **env-entry-name** element giving the name of the entry relative to `java:comp/env`
- An **env-entry-type** element giving the Java type of the entry value that must be one of:
  - `java.lang.Byte`
  - `java.lang.Boolean`
  - `java.lang.Character`
  - `java.lang.Double`
  - `java.lang.Float`
  - `java.lang.Integer`
  - `java.lang.Long`
  - `java.lang.Short`
  - `java.lang.String`
- An **env-entry-value** element giving the value of entry as a string

An example of an `env-entry` fragment from an `ejb-jar.xml` deployment descriptor is given in *Example 4.5, "An example ejb-jar.xml env-entry fragment"*. There is no JBoss specific deployment descriptor element because an `env-entry` is a complete name and value specification. *Example 4.6, "ENC env-entry access code fragment"* shows a sample code fragment for accessing the `maxExemptions` and `taxRateenv-entry` values declared in the deployment descriptor.

```
<!-- ... -->
<session>
    <ejb-name>ASessionBean</ejb-name>
    <!-- ... -->
    <env-entry>
        <description>The maximum number of tax exemptions allowed
</description>
        <env-entry-name>maxExemptions</env-entry-name>
        <env-entry-type>java.lang.Integer</env-entry-type>
        <env-entry-value>15</env-entry-value>
    </env-entry>
    <env-entry>
        <description>The tax rate </description>
        <env-entry-name>taxRate</env-entry-name>
        <env-entry-type>java.lang.Float</env-entry-type>
        <env-entry-value>0.23</env-entry-value>
    </env-entry>
</session>
<!-- ... -->
```

Example 4.5. An example ejb-jar.xml env-entry fragment

```
InitialContext iniCtx = new InitialContext();
```

```
Context envCtx = (Context) iniCtx.lookup("java:comp/env");
Integer maxExemptions = (Integer) envCtx.lookup("maxExemptions");
Float taxRate = (Float) envCtx.lookup("taxRate");
```

Example 4.6. ENC env-entry access code fragment

6.1.2. EJB References

It is common for EJBs and Web components to interact with other EJBs. Because the JNDI name under which an EJB home interface is bound is a deployment time decision, there needs to be a way for a component developer to declare a reference to an EJB that will be linked by the deployer. EJB references satisfy this requirement.

An EJB reference is a link in an application component naming environment that points to a deployed EJB home interface. The name used by the application component is a logical link that isolates the component from the actual name of the EJB home in the deployment environment. The J2EE specification recommends that all references to enterprise beans be organized in the `java:comp/env/ejb` context of the application component's environment.

An EJB reference is declared using an `ejb-ref` element in the deployment descriptor. Each `ejb-ref` element describes the interface requirements that the referencing application component has for the referenced enterprise bean. The `ejb-ref` element contains the following child elements:

- An optional **description** element that provides the purpose of the reference.
- An **ejb-ref-name** element that specifies the name of the reference relative to the `java:comp/env` context. To place the reference under the recommended `java:comp/env/ejb` context, use an `ejb/link-name` form for the `ejb-ref-name` value.
- An **ejb-ref-type** element that specifies the type of the EJB. This must be either `Entity` or `Session`.
- A **home** element that gives the fully qualified class name of the EJB home interface.
- A **remote** element that gives the fully qualified class name of the EJB remote interface.
- An optional **ejb-link** element that links the reference to another enterprise bean in the same EJB JAR or in the same J2EE application unit. The `ejb-link` value is the `ejb-name` of the referenced bean. If there are multiple enterprise beans with the same `ejb-name`, the value uses the path name specifying the location of the `ejb-jar` file that contains the referenced component. The path name is relative to the referencing `ejb-jar` file. The Application Assembler appends the `ejb-name` of the referenced bean to the path name separated by `#`. This allows multiple beans with the same name to be uniquely identified.

An EJB reference is scoped to the application component whose declaration contains the `ejb-ref` element. This means that the EJB reference is not accessible from other application components at runtime, and that other application components may define `ejb-ref` elements with the same `ejb-ref-name` without causing a name conflict. *Example 4.7, "An example ejb-jar.xml ejb-ref descriptor fragment"* provides an `ejb-jar.xml` fragment that illustrates the use of the `ejb-ref` element. A code sample that illustrates accessing the `ShoppingCartHome` reference declared in *Example 4.7, "An example ejb-jar.xml ejb-ref descriptor fragment"* is given in *Example 4.8, "ENC ejb-ref access code fragment"*.

```
<!-- ... -->
<session>
    <ejb-name>ShoppingCartBean</ejb-name>
    <!-- ...-->
</session>

<session>
    <ejb-name>ProductBeanUser</ejb-name>
    <!--...-->
    <ejb-ref>
        <description>This is a reference to the store products entity
</description>
        <ejb-ref-name>ejb/ProductHome</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <home>org.jboss.store.ejb.ProductHome</home>
        <remote> org.jboss.store.ejb.Product</remote>
    </ejb-ref>

</session>

<session>
    <ejb-ref>
        <ejb-name>ShoppingCartUser</ejb-name>
        <!--...-->
        <ejb-ref-name>ejb/ShoppingCartHome</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <home>org.jboss.store.ejb.ShoppingCartHome</home>
        <remote> org.jboss.store.ejb.ShoppingCart</remote>
        <ejb-link>ShoppingCartBean</ejb-link>
    </ejb-ref>
</session>

<entity>
    <description>The Product entity bean </description>
    <ejb-name>ProductBean</ejb-name>
    <!--...-->
</entity>

<!--...-->
```

Example 4.7. An example ejb-jar.xml ejb-ref descriptor fragment

```
InitialContext iniCtx = new InitialContext();
Context ejbCtx = (Context) iniCtx.lookup("java:comp/env/ejb");
ShoppingCartHome home = (ShoppingCartHome)
ejbCtx.lookup("ShoppingCartHome");
```

Example 4.8. ENC ejb-ref access code fragment

6.1.3. EJB References with `jboss.xml` and `jboss-web.xml`

The JBoss specific `jboss.xml` EJB deployment descriptor affects EJB references in two ways. First, the `jndi-name` child element of the `session` and `entity` elements allows the user to specify the deployment JNDI name for the EJB home interface. In the absence of a `jboss.xml` specification of the `jndi-name` for an EJB, the home interface is bound under the `ejb-jar.xmlejb-name` value. For example, the session EJB with the `ejb-name` of `ShoppingCartBean` in *Example 4.7, "An example ejb-jar.xml ejb-ref descriptor fragment"* would have its home interface bound under the JNDI name `ShoppingCartBean` in the absence of a `jboss.xmljndi-name` specification.

The second use of the `jboss.xml` descriptor with respect to `ejb-ref`s is the setting of the destination to which a component's ENC `ejb-ref` refers. The `ejb-link` element cannot be used to refer to EJBs in another enterprise application. If your `ejb-ref` needs to access an external EJB, you can specify the JNDI name of the deployed EJB home using the `jboss.xmlejb-ref/jndi-name` element.

The `jboss-web.xml` descriptor is used only to set the destination to which a Web application ENC `ejb-ref` refers. The content model for the JBoss `ejb-ref` is as follows:

- An **ejb-ref-name** element that corresponds to the **ejb-ref-name** element in the **ejb-jar.xml** or **web.xml** standard descriptor
- A `jndi-name` element that specifies the JNDI name of the EJB home interface in the deployment environment

Example 4.9, "An example jboss.xml ejb-ref fragment" provides an example `jboss.xml` descriptor fragment that illustrates the following usage points:

- The `ProductBeanUserejb-ref` link destination is set to the deployment name of `jboss/store/ProductHome`
- The deployment JNDI name of the `ProductBean` is set to `jboss/store/ProductHome`

```
<!-- ... -->
<session>
```

```
    <ejb-name>ProductBeanUser</ejb-name>
    <ejb-ref>
        <ejb-ref-name>ejb/ProductHome</ejb-ref-name>
        <jndi-name>jboss/store/ProductHome</jndi-name>
    </ejb-ref>
</session>

<entity>
    <ejb-name>ProductBean</ejb-name>
    <jndi-name>jboss/store/ProductHome</jndi-name>
     <!-- ... -->
</entity>
<!-- ... -->
```

Example 4.9. An example jboss.xml ejb-ref fragment

6.1.4. EJB Local References

EJB 2.0 added local interfaces that do not use RMI call by value semantics. These interfaces use a call by reference semantic and therefore do not incur any RMI serialization overhead. An EJB local reference is a link in an application component naming environment that points to a deployed EJB local home interface. The name used by the application component is a logical link that isolates the component from the actual name of the EJB local home in the deployment environment. The J2EE specification recommends that all references to enterprise beans be organized in the `java:comp/env/ejb` context of the application component's environment.

An EJB local reference is declared using an `ejb-local-ref` element in the deployment descriptor. Each `ejb-local-ref` element describes the interface requirements that the referencing application component has for the referenced enterprise bean. The `ejb-local-ref` element contains the following child elements:

- An optional **description** element that provides the purpose of the reference.
- An **ejb-ref-name** element that specifies the name of the reference relative to the `java:comp/env` context. To place the reference under the recommended `java:comp/env/ejb` context, use an `ejb/link-name` form for the `ejb-ref-name` value.
- An **ejb-ref-type** element that specifies the type of the EJB. This must be either `Entity` or `Session`.
- A **local-home** element that gives the fully qualified class name of the EJB local home interface.
- A **local** element that gives the fully qualified class name of the EJB local interface.
- An **ejb-link** element that links the reference to another enterprise bean in the `ejb-jar` file or

in the same J2EE application unit. The `ejb-link` value is the `ejb-name` of the referenced bean. If there are multiple enterprise beans with the same `ejb-name`, the value uses the path name specifying the location of the `ejb-jar` file that contains the referenced component. The path name is relative to the referencing `ejb-jar` file. The Application Assembler appends the `ejb-name` of the referenced bean to the path name separated by `#`. This allows multiple beans with the same name to be uniquely identified. An `ejb-link` element must be specified in JBoss to match the local reference to the corresponding EJB.

An EJB local reference is scoped to the application component whose declaration contains the `ejb-local-ref` element. This means that the EJB local reference is not accessible from other application components at runtime, and that other application components may define `ejb-local-ref` elements with the same `ejb-ref-name` without causing a name conflict. *Example 4.10, "An example ejb-jar.xml ejb-local-ref descriptor fragment"* provides an `ejb-jar.xml` fragment that illustrates the use of the `ejb-local-ref` element. A code sample that illustrates accessing the `ProbeLocalHome` reference declared in *Example 4.10, "An example ejb-jar.xml ejb-local-ref descriptor fragment"* is given in *Example 4.11, "ENC ejb-local-ref access code fragment"*.

```
    <!-- ... -->
    <session>
        <ejb-name>Probe</ejb-name>
        <home>org.jboss.test.perf.interfaces.ProbeHome</home>
        <remote>org.jboss.test.perf.interfaces.Probe</remote>
<local-home>org.jboss.test.perf.interfaces.ProbeLocalHome</local-home>
        <local>org.jboss.test.perf.interfaces.ProbeLocal</local>
        <ejb-class>org.jboss.test.perf.ejb.ProbeBean</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Bean</transaction-type>
    </session>
    <session>
        <ejb-name>PerfTestSession</ejb-name>
        <home>org.jboss.test.perf.interfaces.PerfTestSessionHome</home>
        <remote>org.jboss.test.perf.interfaces.PerfTestSession</remote>
        <ejb-class>org.jboss.test.perf.ejb.PerfTestSessionBean</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
        <ejb-ref>
            <ejb-ref-name>ejb/ProbeHome</ejb-ref-name>
            <ejb-ref-type>Session</ejb-ref-type>
            <home>org.jboss.test.perf.interfaces.SessionHome</home>
            <remote>org.jboss.test.perf.interfaces.Session</remote>
            <ejb-link>Probe</ejb-link>
        </ejb-ref>
        <ejb-local-ref>
            <ejb-ref-name>ejb/ProbeLocalHome</ejb-ref-name>
            <ejb-ref-type>Session</ejb-ref-type>
<local-home>org.jboss.test.perf.interfaces.ProbeLocalHome</local-home>
            <local>org.jboss.test.perf.interfaces.ProbeLocal</local>
            <ejb-link>Probe</ejb-link>
        </ejb-local-ref>
    </session>
    <!-- ... -->
```

Example 4.10. An example ejb-jar.xml ejb-local-ref descriptor fragment

```
InitialContext iniCtx = new InitialContext();
Context ejbCtx = (Context) iniCtx.lookup("java:comp/env/ejb");
ProbeLocalHome home = (ProbeLocalHome) ejbCtx.lookup("ProbeLocalHome");
```

Example 4.11. ENC ejb-local-ref access code fragment

6.1.5. Resource Manager Connection Factory References

Resource manager connection factory references allow application component code to refer to resource factories using logical names called resource manager connection factory references. Resource manager connection factory references are defined by the `resource-ref` elements in the standard deployment descriptors. The `Deployer` binds the resource manager connection factory references to the actual resource manager connection factories that exist in the target operational environment using the `jboss.xml` and `jboss-web.xml` descriptors.

Each `resource-ref` element describes a single resource manager connection factory reference. The `resource-ref` element consists of the following child elements:

- An optional **description** element that provides the purpose of the reference.
- A **res-ref-name** element that specifies the name of the reference relative to the `java:comp/env` context. The resource type based naming convention for which subcontext to place the `res-ref-name` into is discussed in the next paragraph.
- A **res-type** element that specifies the fully qualified class name of the resource manager connection factory.
- A **res-auth** element that indicates whether the application component code performs resource signon programmatically, or whether the container signs on to the resource based on the principal mapping information supplied by the Deployer. It must be one of `Application` or `Container`.
- An optional **res-sharing-scope** element. This currently is not supported by JBoss.

The J2EE specification recommends that all resource manager connection factory references be organized in the subcontexts of the application component's environment, using a different subcontext for each resource manager type. The recommended resource manager type to subcontext name is as follows:

- JDBC `DataSource` references should be declared in the `java:comp/env/jdbc` subcontext.
- JMS connection factories should be declared in the `java:comp/env/jms` subcontext.
- JavaMail connection factories should be declared in the `java:comp/env/mail` subcontext.
- URL connection factories should be declared in the `java:comp/env/url` subcontext.

Example 4.12, "A web.xml resource-ref descriptor fragment" shows an example `web.xml` descriptor fragment that illustrates the `resource-ref` element usage. *Example 4.13, "ENC resource-ref access sample code fragment"* provides a code fragment that an application component would use to access the `DefaultMail` resource declared by the `resource-ref`.

```
<web>
    <!-- ... -->
    <servlet>
        <servlet-name>AServlet</servlet-name>
        <!-- ... -->
    </servlet>
    <!-- ... -->
    <!-- JDBC DataSources (java:comp/env/jdbc) -->
    <resource-ref>
        <description>The default DS</description>
        <res-ref-name>jdbc/DefaultDS</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
    <!-- JavaMail Connection Factories (java:comp/env/mail) -->
    <resource-ref>
        <description>Default Mail</description>
        <res-ref-name>mail/DefaultMail</res-ref-name>
        <res-type>javax.mail.Session</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
    <!-- JMS Connection Factories (java:comp/env/jms) -->
    <resource-ref>
        <description>Default QueueFactory</description>
        <res-ref-name>jms/QueueFactory</res-ref-name>
        <res-type>javax.jms.QueueConnectionFactory</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
<web>
```

Example 4.12. A web.xml resource-ref descriptor fragment

```
Context initCtx = new InitialContext();
javax.mail.Session s = (javax.mail.Session)
initCtx.lookup("java:comp/env/mail/DefaultMail");
```

Example 4.13. ENC resource-ref access sample code fragment

6.1.6. Resource Manager Connection Factory References with jboss.xml and jboss-web.xml

The purpose of the JBoss `jboss.xml` EJB deployment descriptor and `jboss-web.xml` Web application deployment descriptor is to provide the link from the logical name defined by the `res-ref-name` element to the JNDI name of the resource factory as deployed in JBoss. This is accomplished by providing a `resource-ref` element in the `jboss.xml` or `jboss-web.xml` descriptor. The JBoss `resource-ref` element consists of the following child elements:

- A **res-ref-name** element that must match the `res-ref-name` of a corresponding `resource-ref` element from the `ejb-jar.xml` or `web.xml` standard descriptors
- An optional **res-type** element that specifies the fully qualified class name of the resource manager connection factory
- A **jndi-name** element that specifies the JNDI name of the resource factory as deployed in JBoss
- A **res-url** element that specifies the URL string in the case of a `resource-ref` of type `java.net.URL`

Example 4.14, "A sample jboss-web.xml resource-ref descriptor fragment" provides a sample `jboss-web.xml` descriptor fragment that shows sample mappings of the `resource-ref` elements given in *Example 4.12, "A web.xml resource-ref descriptor fragment"*.

```
<jboss-web>
    <!-- ... -->
    <resource-ref>
        <res-ref-name>jdbc/DefaultDS</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <jndi-name>java:/DefaultDS</jndi-name>
    </resource-ref>
    <resource-ref>
        <res-ref-name>mail/DefaultMail</res-ref-name>
        <res-type>javax.mail.Session</res-type>
        <jndi-name>java:/Mail</jndi-name>
    </resource-ref>
    <resource-ref>
        <res-ref-name>jms/QueueFactory</res-ref-name>
        <res-type>javax.jms.QueueConnectionFactory</res-type>
        <jndi-name>QueueConnectionFactory</jndi-name>
    </resource-ref>
    <!-- ... -->
</jboss-web>
```

Example 4.14. A sample jboss-web.xml resource-ref descriptor fragment

6.1.7. Resource Environment References

Resource environment references are elements that refer to administered objects that are associated with a resource (for example, JMS destinations) using logical names. Resource environment references are defined by the `resource-env-ref` elements in the standard deployment descriptors. The `Deployer` binds the resource environment references to the actual administered objects location in the target operational environment using the `jboss.xml` and `jboss-web.xml` descriptors.

Each `resource-env-ref` element describes the requirements that the referencing application component has for the referenced administered object. The `resource-env-ref` element consists of the following child elements:

- An optional **description** element that provides the purpose of the reference.
- A **resource-env-ref-name** element that specifies the name of the reference relative to the `java:comp/env` context. Convention places the name in a subcontext that corresponds to the associated resource factory type. For example, a JMS queue reference named `MyQueue` should have a `resource-env-ref-name` of `jms/MyQueue`.
- A **resource-env-ref-type** element that specifies the fully qualified class name of the referenced object. For example, in the case of a JMS queue, the value would be `javax.jms.Queue`.

Example 4.15, "An example ejb-jar.xml resource-env-ref fragment" provides an example `resource-ref-env` element declaration by a session bean. *Example 4.16, "ENC resource-env-ref access code fragment"* gives a code fragment that illustrates how to look up the `StockInfo` queue declared by the `resource-env-ref`.

```
<session>
    <ejb-name>MyBean</ejb-name>
    <!-- ... -->
    <resource-env-ref>
        <description>This is a reference to a JMS queue used in the
            processing of Stock info
        </description>
        <resource-env-ref-name>jms/StockInfo</resource-env-ref-name>
        <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
    </resource-env-ref>
    <!-- ... -->
</session>
```

Example 4.15. An example ejb-jar.xml resource-env-ref fragment

```
InitialContext iniCtx = new InitialContext();
javax.jms.Queue q = (javax.jms.Queue)
envCtx.lookup("java:comp/env/jms/StockInfo");
```

Example 4.16. ENC resource-env-ref access code fragment

6.1.8. Resource Environment References and jboss.xml, jboss-web.xml

The purpose of the JBoss `jboss.xml` EJB deployment descriptor and `jboss-web.xml` Web application deployment descriptor is to provide the link from the logical name defined by the `resource-env-ref-name` element to the JNDI name of the administered object deployed in JBoss. This is accomplished by providing a `resource-env-ref` element in the `jboss.xml` or `jboss-web.xml` descriptor. The JBoss `resource-env-ref` element consists of the following child elements:

- A `resource-env-ref-name` element that must match the `resource-env-ref-name` of a corresponding `resource-env-ref` element from the `ejb-jar.xml` or `web.xml` standard descriptors
- A `jndi-name` element that specifies the JNDI name of the resource as deployed in JBoss

Example 4.17, "A sample jboss.xml resource-env-ref descriptor fragment" provides a sample `jboss.xml` descriptor fragment that shows a sample mapping for the `StockInforesource-env-ref`.

```
<session>
    <ejb-name>MyBean</ejb-name>
    <!-- ... -->
    <resource-env-ref>
        <resource-env-ref-name>jms/StockInfo</resource-env-ref-name>
        <jndi-name>queue/StockInfoQueue</jndi-name>
    </resource-env-ref>
    <!-- ... -->
</session>
```

Example 4.17. A sample jboss.xml resource-env-ref descriptor fragment

Connectors on JBoss

The JCA Configuration and Architecture

This chapter discusses the JBoss server implementation of the J2EE Connector Architecture (JCA). JCA is a resource manager integration API whose goal is to standardize access to non-relational resources in the same way the JDBC API standardized access to relational data. The purpose of this chapter is to introduce the utility of the JCA APIs and then describe the architecture of JCA in JBoss

1. JCA Overview

J2EE 1.4 contains a connector architecture (JCA) specification that allows for the integration of transacted and secure resource adaptors into a J2EE application server environment. The JCA specification describes the notion of such resource managers as Enterprise Information Systems (EIS). Examples of EIS systems include enterprise resource planning packages, mainframe transaction processing, non-Java legacy applications, etc.

The reason for focusing on EIS is primarily because the notions of transactions, security, and scalability are requirements in enterprise software systems. However, the JCA is applicable to any resource that needs to integrate into JBoss in a secure, scalable and transacted manner. In this introduction we will focus on resource adapters as a generic notion rather than something specific to the EIS environment.

The connector architecture defines a standard SPI (Service Provider Interface) for integrating the transaction, security and connection management facilities of an application server with those of a resource manager. The SPI defines the system level contract between the resource adaptor and the application server.

The connector architecture also defines a Common Client Interface (CCI) for accessing resources. The CCI is targeted at EIS development tools and other sophisticated users of integrated resources. The CCI provides a way to minimize the EIS specific code required by such tools. Typically J2EE developers will access a resource using such a tool, or a resource specific interface rather than using CCI directly. The reason is that the CCI is not a type specific API. To be used effectively it must be used in conjunction with metadata that describes how to map from the generic CCI API to the resource manager specific data types used internally by the resource manager.

The purpose of the connector architecture is to enable a resource vendor to provide a standard adaptor for its product. A resource adaptor is a system-level software driver that is used by a Java application to connect to resource. The resource adaptor plugs into an application server and provides connectivity between the resource manager, the application server, and the enterprise application. A resource vendor need only implement a JCA compliant adaptor once to allow use of the resource manager in any JCA capable application server.

An application server vendor extends its architecture once to support the connector architecture and is then assured of seamless connectivity to multiple resource managers. Likewise, a

resource manager vendor provides one standard resource adaptor and it has the capability to plug in to any application server that supports the connector architecture.

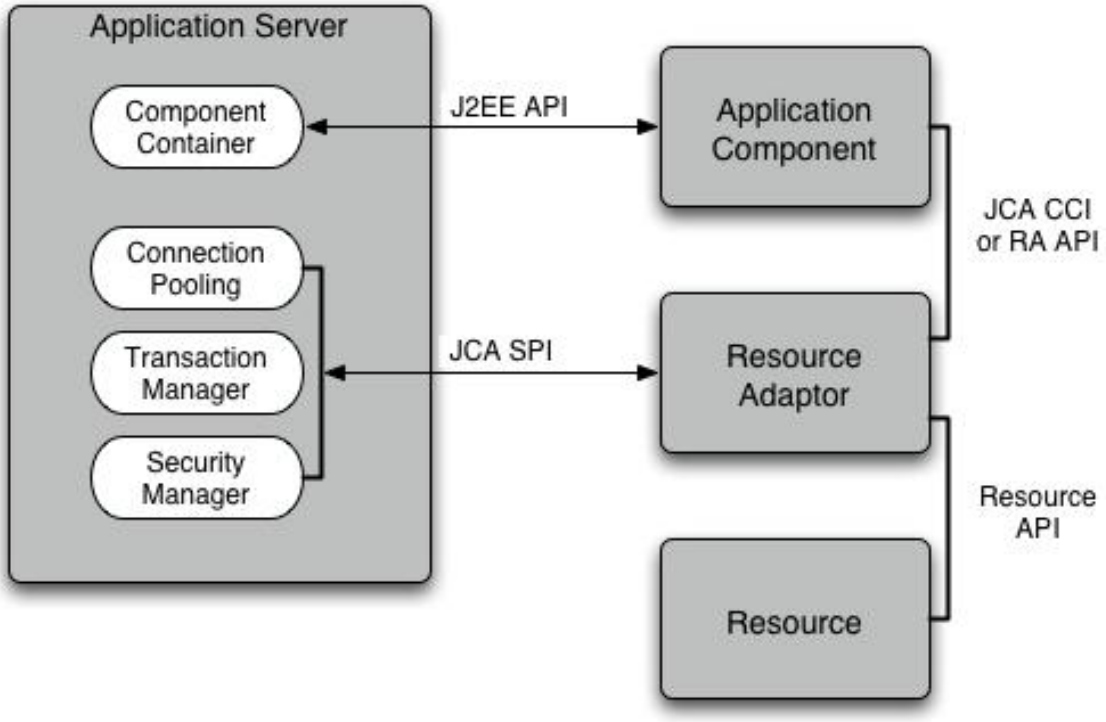


Figure 5.1. The relationship between a J2EE application server and a JCA resource adaptor

Figure 5.1, "The relationship between a J2EE application server and a JCA resource adaptor" illustrates that the application server is extended to provide support for the JCA SPI to allow a resource adaptor to integrate with the server connection pooling, transaction management and security management facilities. This integration API defines a three-part system contract.

- **Connection management**: a contract that allows the application server to pool resource connections. The purpose of the pool management is to allow for scalability. Resource connections are typically expense objects to create and pooling them allows for more effective reuse and management.
- **Transaction Management**: a contract that allows the application server transaction manager to manage transactions that engage resource managers.
- **Security Management**: a contract that enables secured access to resource managers.

The resource adaptor implements the resource manager side of the system contract. This entails using the application server connection pooling, providing transaction resource information and using the security integration information. The resource adaptor also exposes

the resource manager to the application server components. This can be done using the CCI and/or a resource adaptor specific API.

The application component integrates into the application server using a standard J2EE container to component contract. For an EJB component this contract is defined by the EJB specification. The application component interacts with the resource adaptor in the same way as it would with any other standard resource factory, for example, a `javax.sql.DataSource` JDBC resource factory. The only difference with a JCA resource adaptor is that the client has the option of using the resource adaptor independent CCI API if the resource adaptor supports this.

Figure 5.2, "The JCA 1.0 specification class diagram for the connection management architecture." (from the JCA 1.5 specification) illustrates the relationship between the JCA architecture participants in terms of how they relate to the JCA SPI, CCI and JTA packages.

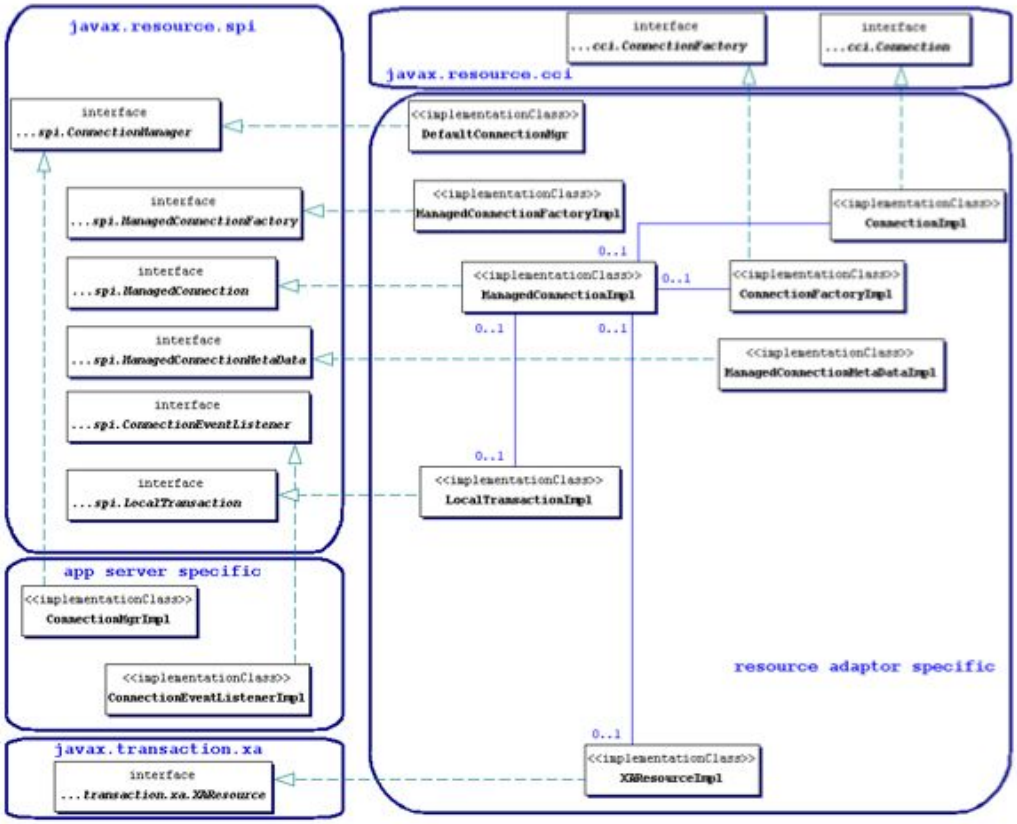


Figure 5.2. The JCA 1.0 specification class diagram for the connection management architecture.

The JBossCX architecture provides the implementation of the application server specific classes. *Figure 5.2, "The JCA 1.0 specification class diagram for the connection management architecture."* shows that this comes down to the implementation of the `javax.resource.spi.ConnectionManager` and

`javax.resource.spi.ConnectionEventListener` interfaces. The key aspects of this implementation are discussed in the following section on the JBossCX architecture.

2. An Overview of the JBossCX Architecture

The JBossCX framework provides the application server architecture extension required for the use of JCA resource adaptors. This is primarily a connection pooling and management extension along with a number of MBeans for loading resource adaptors into the JBoss server.

There are three coupled MBeans that make up a RAR deployment. These are the `org.jboss.resource.deployment.RARDeployment`, `org.jboss.resource.connectionmanager.RARDeployment`, and `org.jboss.resource.connectionmanager.BaseConnectionManager2`. The `org.jboss.resource.deployment.RARDeployment` is simply an encapsulation of the metadata of a RAR `META-INF/ra.xml` descriptor. It exposes this information as a DynamicMBean simply to make it available to the `org.jboss.resource.connectionmanager.RARDeployment` MBean.

The RARDeployer service handles the deployment of archives files containing resource adaptors (RARs). It creates the `org.jboss.resource.deployment.RARDeployment` MBeans when a RAR file is deployed. Deploying the RAR file is the first step in making the resource adaptor available to application components. For each deployed RAR, one or more connection factories must be configured and bound into JNDI. This task performed using a JBoss service descriptor that sets up a `org.jboss.resource.connectionmanager.BaseConnectionManager2` MBean implementation with a `org.jboss.resource.connectionmgr.RARDeployment` dependent.

2.1. BaseConnectionManager2 MBean

The `org.jboss.resource.connectionmanager.BaseConnectionManager2` MBean is a base class for the various types of connection managers required by the JCA spec. Subclasses include `NoTxConnectionManager`, `LocalTxConnectionManager` and `XATxConnectionManager`. These correspond to resource adaptors that support no transactions, local transaction and XA transaction respectively. You choose which subclass to use based on the type of transaction semantics you want, provided the JCA resource adaptor supports the corresponding transaction capability.

The common attributes supported by the BaseConnectionManager2 MBean are:

- **ManagedConnectionPool**: This specifies the ObjectName of the MBean representing the pool for this connection manager. The MBean must have an `ManagedConnectionPool` attribute that is an implementation of the `org.jboss.resource.connectionmanager.ManagedConnectionPool` interface. Normally it will be an embedded MBean in a depends tag rather than an `ObjectName` reference to an existing MBean. The default MBean for use is the `org.jboss.resource.connectionmanager.JBossManagedConnectionPool`. Its configurable attributes are discussed below.

- **CachedConnectionManager**: This specifies the `ObjectName` of the `CachedConnectionManager` MBean implementation used by the connection manager. Normally this is specified using a depends tag with the `ObjectName` of the unique `CachedConnectionManager` for the server. The name `jboss.jca:service=CachedConnectionManager` is the standard setting to use.
- **SecurityDomainJndiName**: This specifies the JNDI name of the security domain to use for authentication and authorization of resource connections. This is typically of the form `java:/jaas/<domain>` where the `<domain>` value is the name of an entry in the `conf/login-config.xml` JAAS login module configuration file. This defines which JAAS login modules execute to perform authentication.
- **JaasSecurityManagerService**: This is the `ObjectName` of the security manager service. This should be set to the security manager MBean name as defined in the `conf/jboss-service.xml` descriptor, and currently this is `jboss.security:service=JaasSecurityManager`. This attribute will likely be removed in the future.

2.2. RARDeployment MBean

The `org.jboss.resource.connectionmanager.RARDeployment` MBean manages configuration and instantiation `ManagedConnectionFactory` instance. It does this using the resource adaptor metadata settings from the RAR `META-INF/ra.xml` descriptor along with the `RARDeployment` attributes. The configurable attributes are:

- **OldRarDeployment**: This is the `ObjectName` of the `org.jboss.resource.RarDeployment` MBean that contains the resource adaptor metadata. The form of this name is `jboss.jca:service=RARDeployment,name=<ra-display-name>` where the `<ra-display-name>` is the `ra.xml` descriptor `display-name` attribute value. The `RARDeployer` creates this when it deploys a RAR file. This attribute will likely be removed in the future.
- **ManagedConnectionFactoryProperties**: This is a collection of (name, type, value) triples that define attributes of the `ManagedConnectionFactory` instance. Therefore, the names of the attributes depend on the resource adaptor `ManagedConnectionFactory` instance. The following example shows the structure of the content of this attribute.

```
<properties>
    <config-property>
        <config-property-name>Attr0Name</config-property-name>
        <config-property-type>Attr0Type</config-property-type>
        <config-property-value>Attr0Value</config-property-value>
    </config-property>
    <config-property>
        <config-property-name>Attr1Name</config-property-name>
        <config-property-type>Attr2Type</config-property-type>
        <config-property-value>Attr2Value</config-property-value>
    </config-property>
```

```
    ...
</properties>
```

`AttrXName` is the Xth attribute name, `AttrXType` is the fully qualified Java type of the attribute, and `AttrXValue` is the string representation of the value. The conversion from string to `AttrXType` is done using the `java.beans.PropertyEditor` class for the `AttrXType`.

- **JndiName**: This is the JNDI name under which the resource adaptor will be made available. Clients of the resource adaptor use this name to obtain either the `javax.resource.cci.ConnectionFactory` or resource adaptor specific connection factory. The full JNDI name will be `java:/<JndiName>` meaning that the `JndiName` attribute value will be prefixed with `java:/`. This prevents use of the connection factory outside of the JBoss server VM. In the future this restriction may be configurable.

2.3. JBossManagedConnectionPool MBean

The `org.jboss.resource.connectionmanager.JBossManagedConnectionPool` MBean is a connection pooling MBean. It is typically used as the embedded MBean value of the `BaseConnectionManager2ManagedConnectionPool` attribute. When you setup a connection manager MBean you typically embed the pool configuration in the connection manager descriptor. The configurable attributes of the `JBossManagedConnectionPool` are:

- **ManagedConnectionFactoryName**: This specifies the `ObjectName` of the MBean that creates `javax.resource.spi.ManagedConnectionFactory` instances. Normally this is configured as an embedded MBean in a depends element rather than a separate MBean reference using the `RARDeployment` MBean. The MBean must provide an appropriate `startManagedConnectionFactory` operation.
- **MinSize**: This attribute indicates the minimum number of connections this pool should hold. These are not created until a `Subject` is known from a request for a connection. `MinSize` connections will be created for each sub-pool.
- **MaxSize**: This attribute indicates the maximum number of connections for a pool. No more than MaxSize connections will be created in each sub-pool.
- **BlockingTimeoutMillis**: This attribute indicates the maximum time to block while waiting for a connection before throwing an exception. Note that this blocks only while waiting for a permit for a connection, and will never throw an exception if creating a new connection takes an inordinately long time.
- **IdleTimeoutMinutes**: This attribute indicates the maximum time a connection may be idle before being closed. The actual maximum time depends also on the idle remover thread scan time, which is 1/2 the smallest idle timeout of any pool.
- **NoTxSeparatePools**: Setting this to true doubles the available pools. One pool is for

connections used outside a transaction the other inside a transaction. The actual pools are lazily constructed on first use. This is only relevant when setting the pool parameters associated with the `LocalTxConnectionManager` and `XATxConnectionManager`. Its use case is for Oracle (and possibly other vendors) XA implementations that don't like using an XA connection with and without a JTA transaction.

- **Criteria**: This attribute indicates if the JAAS `javax.security.auth.Subject` from security domain associated with the connection, or app supplied parameters (such as from `getConnection(user, pw)`) are used to distinguish connections in the pool. The allowed values are:
  - **ByContainer**: use `Subject`
  - **ByApplication**: use application supplied parameters only
  - **ByContainerAndApplication**: use both
  - **ByNothing**: all connections are equivalent, usually if adapter supports reauthentication

2.4. CachedConnectionManager MBean

The `org.jboss.resource.connectionmanager.CachedConnectionManager` MBean manages associations between meta-aware objects (those accessed through interceptor chains) and connection handles, as well as between user transactions and connection handles. Normally there should only be one such MBean, and this is configured in the core `jboss-service.xml` descriptor. It is used by `CachedConnectionInterceptor`, JTA `UserTransaction` implementation and all `BaseConnectionManager2` instances. The configurable attributes of the `CachedConnectionManager` MBean are:

- **SpecCompliant**: Enable this boolean attribute for spec compliant non-shareable connections reconnect processing. This allows a connection to be opened in one call and used in another. Note that specifying this behavior disables connection close processing.
- **Debug**: Enable this boolean property for connection close processing. At the completion of an EJB method invocation, unclosed connections are registered with a transaction synchronization. If the transaction ends without the connection being closed, an error is reported and JBoss closes the connection. This is a development feature that should be turned off in production for optimal performance.
- **TransactionManagerServiceName**: This attribute specifies the JMX `ObjectName` of the JTA transaction manager service. Connection close processing is now synchronized with the transaction manager and this attribute specifies the transaction manager to use.

2.5. A Sample Skeleton JCA Resource Adaptor

To conclude our discussion of the JBoss JCA framework we will create and deploy a single non-transacted resource adaptor that simply provides a skeleton implementation that stubs out the required interfaces and logs all method calls. We will not discuss the details of the requirements of a resource adaptor provider as these are discussed in detail in the JCA specification. The purpose of the adaptor is to demonstrate the steps required to create and

deploy a RAR in JBoss, and to see how JBoss interacts with the adaptor.

The adaptor we will create could be used as the starting point for a non-transacted file system adaptor. The source to the example adaptor can be found in the `src/main/org/jboss/book/jca/ex1` directory of the book examples. A class diagram that shows the mapping from the required `javax.resource.spi` interfaces to the resource adaptor implementation is given in *Figure 5.3, "The file system RAR class diagram"*.

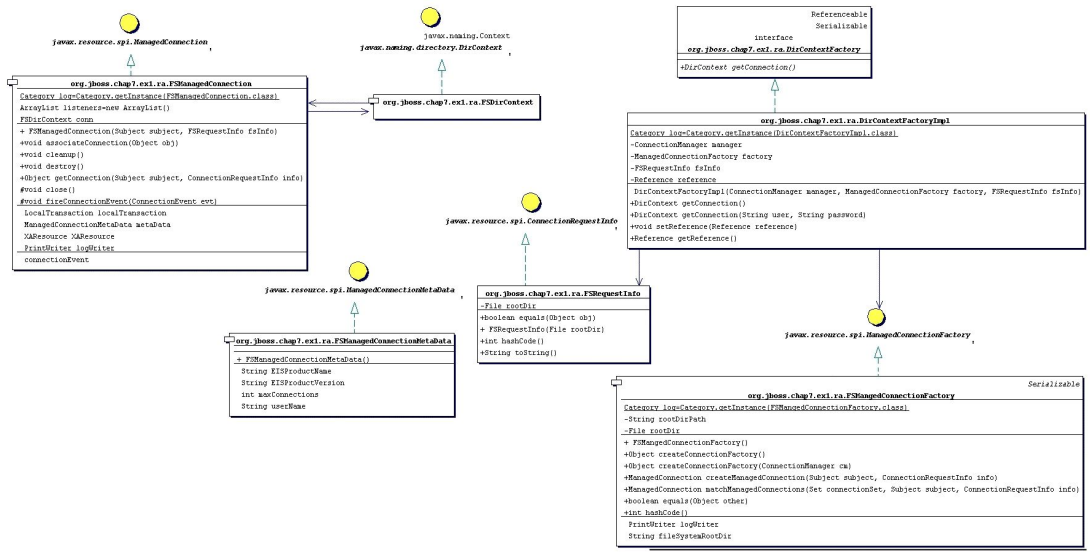


Figure 5.3. The file system RAR class diagram

We will build the adaptor, deploy it to the JBoss server and then run an example client against an EJB that uses the resource adaptor to demonstrate the basic steps in a complete context. We'll then take a look at the JBoss server log to see how the JBoss JCA framework interacts with the resource adaptor to help you better understand the components in the JCA system level contract.

To build the example and deploy the RAR to the JBoss server `deploy/lib` directory, execute the following Ant command in the book examples directory.

```
[examples]$ ant -Dchap=jca build-chap
```

The deployed files include a `jca-ex1.sar` and a `notxfs-service.xml` service descriptor. The example resource adaptor deployment descriptor is shown in *Example 5.1, "The nontransactional file system resource adaptor deployment descriptor."*.

```
<?xml version="1.0" encoding="UTF-8"?>
<connector
xmlns="http://java.sun.com/xml/ns/"Whats_new_in_JBoss_4-J2EE_Certification_and_Standards_Comp
    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/connector_1_5.xsd"
version="1.5">
    <display-name>File System Adapter</display-name>
    <vendor-name>JBoss</vendor-name>
    <eis-type>FileSystem</eis-type>
    <resourceadapter-version>1.0</resourceadapter-version>
    <license>
        <description>LGPL</description>
        <license-required>false</license-required>
    </license>
    <resourceadapter>
        <resourceadapter-class>
            org.jboss.resource.deployment.DummyResourceAdapter
        </resourceadapter-class>
        <outbound-resourceadapter>
            <connection-definition>
                <managedconnectionfactory-class>
org.jboss.book.jca.ex1.ra.FSManagedConnectionFactory
</managedconnectionfactory-class>
                <config-property>
<config-property-name>FileSystemRootDir</config-property-name>
<config-property-type>java.lang.String</config-property-type>
<config-property-value>/tmp/db/fs_store</config-property-value>
                </config-property>
                <config-property>
                    <config-property-name>UserName</config-property-name>
<config-property-type>java.lang.String</config-property-type>
                    <config-property-value/>
                </config-property>
                <config-property>
                    <config-property-name>Password</config-property-name>
<config-property-type>java.lang.String</config-property-type>
                    <config-property-value/>
                </config-property>
                <connectionfactory-interface>
org.jboss.book.jca.ex1.ra.DirContextFactory </connectionfactory-interface>
<connectionfactory-impl-class>
org.jboss.book.jca.ex1.ra.DirContextFactoryImpl
</connectionfactory-impl-class> <connection-interface>
javax.naming.directory.DirContext </connection-interface>
<connection-impl-class> org.jboss.book.jca.ex1.ra.FSDirContext
</connection-impl-class>
            </connection-definition>
            <transaction-support>NoTransaction</transaction-support>
            <authentication-mechanism>
<authentication-mechanism-type>BasicPassword</authentication-mechanism-type>
                <credential-interface>
                    javax.resource.spi.security.PasswordCredential
                </credential-interface>
            </authentication-mechanism>
            <reauthentication-support>true</reauthentication-support>
        </outbound-resourceadapter>
        <security-permission>
            <description> Read/Write access is required to the contents of
the
                FileSystemRootDir </description>
```

```
            <security-permission-spec> permission java.io.FilePermission
                "/tmp/db/fs_store/*", "read,write";
            </security-permission-spec>
        </security-permission>
    </resourceadapter>
</connector>
```

Example 5.1. The nontransactional file system resource adaptor deployment descriptor.

The key items in the resource adaptor deployment descriptor are highlighted in bold. These define the classes of the resource adaptor, and the elements are:

- **managedconnectionfactory-class**: The implementation of the `ManagedConnectionFactory` interface, `org.jboss.book.jca.ex1.ra.FSManagedConnectionFactory`
- **connectionfactory-interface**: This is the interface that clients will obtain when they lookup the connection factory instance from JNDI, here a proprietary resource adaptor value, `org.jboss.book.jca.ex1.ra.DirContextFactory`. This value will be needed when we create the JBoss `ds.xml` to use the resource.
- **connectionfactory-impl-class**: This is the class that provides the implementation of the `connectionfactory-interface`, `org.jboss.book.jca.ex1.ra.DirContextFactoryImpl`.
- **connection-interface**: This is the interface for the connections returned by the resource adaptor connection factory, here the JNDI `javax.naming.directory.DirContext` interface.
- **connection-impl-class**: This is he class that provides the `connection-interface` implementation, `org.jboss.book.jca.ex1.ra.FSDirContext`.
- **transaction-support**: The level of transaction support, here defined as `NoTransaction`, meaning the file system resource adaptor does not do transactional work.

The RAR classes and deployment descriptor only define a resource adaptor. To use the resource adaptor it must be integrated into the JBoss application server using a `ds.xml` descriptor file. An example of this for the file system adaptor is shown in *Example 5.2, “The notxfs-ds.xml resource adaptor MBeans service descriptor.”*.

```
<!DOCTYPE connection-factories PUBLIC
          "-//JBoss//DTD JBOSS JCA Config 1.5//EN"
          "http://www.jboss.org/j2ee/dtd/jboss-ds_1_5.dtd">
<!--
       The non-transaction FileSystem resource adaptor service configuration
-->
<connection-factories>
    <no-tx-connection-factory>
```

```
        <jndi-name>NoTransFS</jndi-name>
        <rar-name>jca-ex1.rar</rar-name>
        <connection-definition>
             org.jboss.book.jca.ex1.ra.DirContextFactory
        </connection-definition>
        <config-property name="FileSystemRootDir"
type="java.lang.String">/tmp/db/fs_store</config-property>
    </no-tx-connection-factory>
</connection-factories>
```

Example 5.2. The notxfs-ds.xml resource adaptor MBeans service descriptor.

The main attributes are:

- **jndi-name**: This specifies where the connection factory will be bound into JNDI. For this deployment that binding will be `java:/NoTransFS`.
- **rar-name**: This is the name of the RAR file that contains the definition for the resource we want to provide. For nested RAR files, the name would look like `myapplication.ear#my.rar`. In this example, it is simply `jca-ex1.rar`.
- **connection-definition**: This is the connection factory interface class. It should match the `connectionfactory-interface` in the `ra.xml` file. Here our connection factory interface is `org.jboss.book.jca.ex1.ra.DirContextFactory`.
- **config-property**: This can be used to provide non-default settings to the resource adaptor connection factory. Here the `FileSystemRootDir` is being set to `/tmp/db/fs_store`. This overrides the default value in the `ra.xml` file.

To deploy the RAR and connection manager configuration to the JBoss server, run the following:

```
[examples]$ ant -Dchap=jca config
```

The server console will display some logging output indicating that the resource adaptor has been deployed.

Now we want to test access of the resource adaptor by a J2EE component. To do this we have created a trivial stateless session bean that has a single method called `echo`. Inside of the `echo` method the EJB accesses the resource adaptor connection factory, creates a connection, and then immediately closes the connection. The `echo` method code is shown below.

```
public String echo(String arg)
```

```
{
    log.info("echo, arg="+arg);
    try {
        InitialContext ctx = new InitialContext();
        Object         ref =
ctx.lookup("java:comp/env/ra/DirContextFactory");
        log.info("echo, ra/DirContextFactory=" + ref);

        DirContextFactory dcf = (DirContextFactory) ref;
        log.info("echo, found dcf=" + dcf);

        DirContext dc = dcf.getConnection();
        log.info("echo, lookup dc=" + dc);

        dc.close();
    } catch(NamingException e) {
        log.error("Failed during JNDI access", e);
    }
    return arg;
}
```

Example 5.3. The stateless session bean echo method code that shows the access of the resource adaptor connection factory.

The EJB is not using the CCI interface to access the resource adaptor. Rather, it is using the resource adaptor specific API based on the proprietary `DirContextFactory` interface that returns a JNDI `DirContext` object as the connection object. The example EJB is simply exercising the system contract layer by looking up the resource adaptor connection factory, creating a connection to the resource and closing the connection. The EJB does not actually do anything with the connection, as this would only exercise the resource adaptor implementation since this is a non-transactional resource.

Run the test client which calls the `EchoBean.echo` method by running Ant as follows from the examples directory:

```
[examples]$ ant -Dchap=jca -Dex=1 run-example
```

You'll see some output from the bean in the system console, but much more detailed logging output can be found in the `server/production/log/server.log` file. Don't worry if you see exceptions. They are just stack traces to highlight the call path into parts of the adaptor. To help understand the interaction between the adaptor and the JBoss JCA layer, we'll summarize the events seen in the log using a sequence diagram. *Figure 5.4, "A sequence diagram illustrating the key interactions between the JBossCX framework and the example resource adaptor that result when the EchoBean accesses the resource adaptor connection factory."* is a sequence diagram that summarizes the events that occur when the `EchoBean` accesses the resource adaptor connection factory from JNDI and creates a connection.

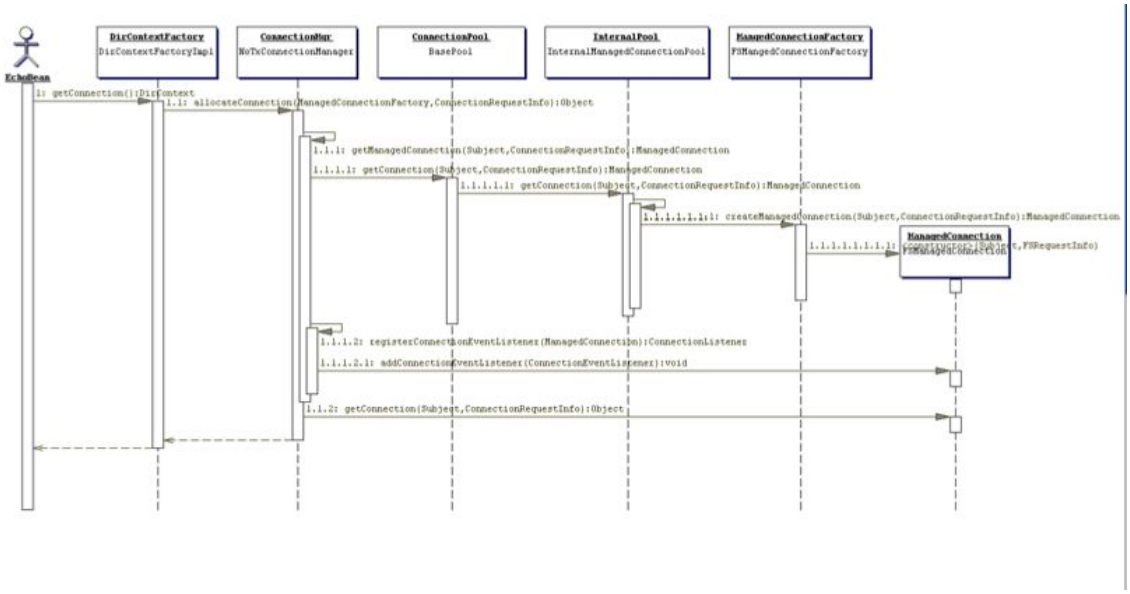


Figure 5.4. A sequence diagram illustrating the key interactions between the JBossCX framework and the example resource adaptor that result when the EchoBean accesses the resource adaptor connection factory.

The starting point is the client's invocation of the `EchoBean.echo` method. For the sake of conciseness of the diagram, the client is shown directly invoking the EchoBean.echo method when in reality the JBoss EJB container handles the invocation. There are three distinct interactions between the `EchoBean` and the resource adaptor; the lookup of the connection factory, the creation of a connection, and the close of the connection.

The lookup of the resource adaptor connection factory is illustrated by the 1.1 sequences of events. The events are:

- 1, the echo method invokes the `getConnection` method on the resource adaptor connection factory obtained from the JNDI lookup on the `java:comp/env/ra/DirContextFactory` name which is a link to the `java:/NoTransFS` location.
- 1.1, the `DirContextFactoryImpl` class asks its associated `ConnectionManager` to allocate a connection. It passes in the `ManagedConnectionFactory` and `FSRequestInfo` that were associated with the `DirContextFactoryImpl` during its construction.
- 1.1.1, the `ConnectionManager` invokes its `getManagedConnection` method with the current `Subject` and `FSRequestInfo`.
- 1.1.1.1, the `ConnectionManager` asks its object pool for a connection object. The `JBossManagedConnectionPool$BasePool` is get the key for the connection and then asks the matching `InternalPool` for a connection.

- 1.1.1.1.1, Since no connections have been created the pool must create a new connection. This is done by requesting a new managed connection from the `ManagedConnectionFactory`. The `Subject` associated with the pool as well as the `FSRequestInfo` data are passed as arguments to the `createManagedConnection` method invocation.
- 1.1.1.1.1.1, the `ConnectionFactory` creates a new `FSManagedConnection` instance and passes in the `Subject` and `FSRequestInfo` data.
- 1.1.1.2, a `javax.resource.spi.ConnectionListener` instance is created. The type of listener created is based on the type of `ConnectionManager`. In this case it is an `org.jboss.resource.connectionmgr.BaseConnectionManager2$NoTransactionListener` instance.
- 1.1.1.2.1, the listener registers as a `javax.resource.spi.ConnectionEventListener` with the `ManagedConnection` instance created in 1.2.1.1.
- 1.1.2, the `ManagedConnection` is asked for the underlying resource manager connection. The `Subject` and `FSRequestInfo` data are passed as arguments to the `getConnection` method invocation.
- The resulting connection object is cast to a `javax.naming.directory.DirContext` instance since this is the public interface defined by the resource adaptor.
- After the `EchoBean` has obtained the `DirContext` for the resource adaptor, it simply closes the connection to indicate its interaction with the resource manager is complete.

This concludes the resource adaptor example. Our investigation into the interaction between the JBossCX layer and a trivial resource adaptor should give you sufficient understanding of the steps required to configure any resource adaptor. The example adaptor can also serve as a starting point for the creation of your own custom resource adaptors if you need to integrate non-JDBC resources into the JBoss server environment.

3. Configuring JDBC DataSources

Rather than configuring the connection manager factory related MBeans discussed in the previous section via a mbean services deployment descriptor, JBoss provides a simplified datasource centric descriptor. This is transformed into the standard `jboss-service.xml` MBean services deployment descriptor using a XSL transform applied by the `org.jboss.deployment.XSLSubDeployer` included in the `jboss-jca.sar` deployment. The simplified configuration descriptor is deployed the same as other deployable components. The descriptor must be named using a `*-ds.xml` pattern in order to be recognized by the `XSLSubDeployer`.

The schema for the top-level datasource elements of the `*-ds.xml` configuration deployment file is shown in *Figure 5.5, "The simplified JCA DataSource configuration descriptor top-level schema elements"*.

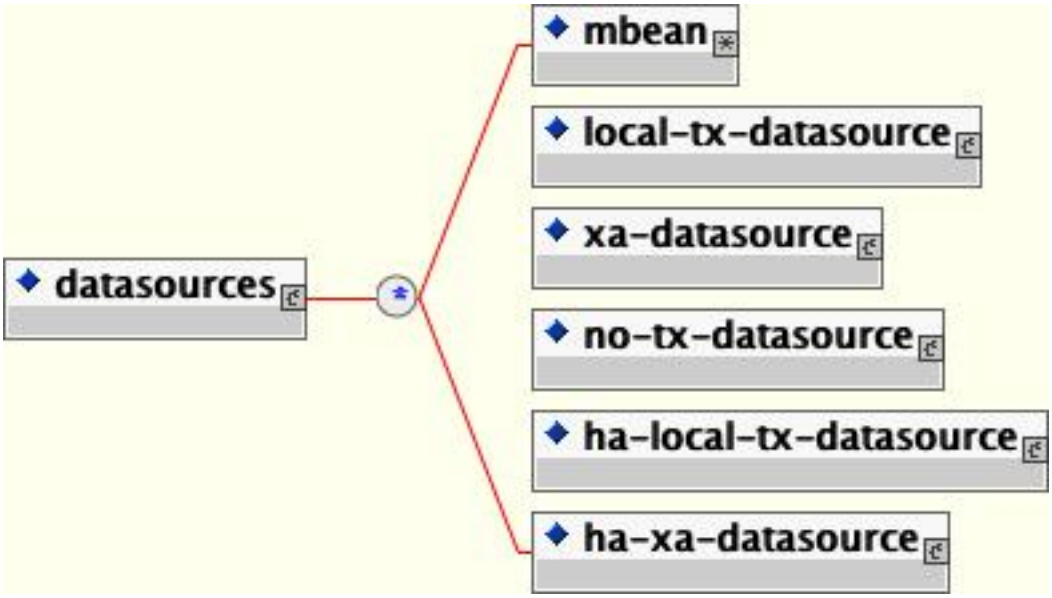


Figure 5.5. The simplified JCA DataSource configuration descriptor top-level schema elements

Multiple datasource configurations may be specified in a configuration deployment file. The child elements of the datasources root are:

- **mbean**: Any number mbean elements may be specified to define MBean services that should be included in the `jboss-service.xml` descriptor that results from the transformation. This may be used to configure services used by the datasources.
- **no-tx-datasource**: This element is used to specify the (`org.jboss.resource.connectionmanager`) `NoTxConnectionManager` service configuration. `NoTxConnectionManager` is a JCA connection manager with no transaction support. The `no-tx-datasource` child element schema is given in *Figure 5.6, "The non-transactional DataSource configuration schema"*.
- **local-tx-datasource**: This element is used to specify the (`org.jboss.resource.connectionmanager`) `LocalTxConnectionManager` service configuration. `LocalTxConnectionManager` implements a `ConnectionEventListener` that implements `XAResource` to manage transactions through the transaction manager. To ensure that all work in a local transaction occurs over the same `ManagedConnection`, it includes a xid to `ManagedConnection` map. When a Connection is requested or a transaction started with a connection handle in use, it checks to see if a `ManagedConnection` already exists enrolled in the global transaction and uses it if found. Otherwise, a free `ManagedConnection` has its `LocalTransaction` started and is used. The `local-tx-datasource` child element schema is given in *Figure 5.7, "The non-XA DataSource configuration schema"*
- **xa-datasource**: This element is used to specify the

`org.jboss.resource.connectionmanager`) `XATxConnectionManager` service configuration. `XATxConnectionManager` implements a `ConnectionEventListener` that obtains the `XAResource` to manage transactions through the transaction manager from the adaptor `ManagedConnection`. To ensure that all work in a local transaction occurs over the same `ManagedConnection`, it includes a xid to `ManagedConnection` map. When a `Connection` is requested or a transaction started with a connection handle in use, it checks to see if a `ManagedConnection` already exists enrolled in the global transaction and uses it if found. Otherwise, a free `ManagedConnection` has its `LocalTransaction` started and is used. The `xa-datasource` child element schema is given in *Figure 5.8, "The XA DataSource configuration schema"*.

- **ha-local-tx-datasource**: This element is identical to `local-tx-datasource`, with the addition of the experimental datasource failover capability allowing JBoss to failover to an alternate database in the event of a database failure.
- **ha-xa-datasource**: This element is identical to `xa-datasource`, with the addition of the experimental datasource failover capability allowing JBoss to failover to an alternate database in the event of a database failure.

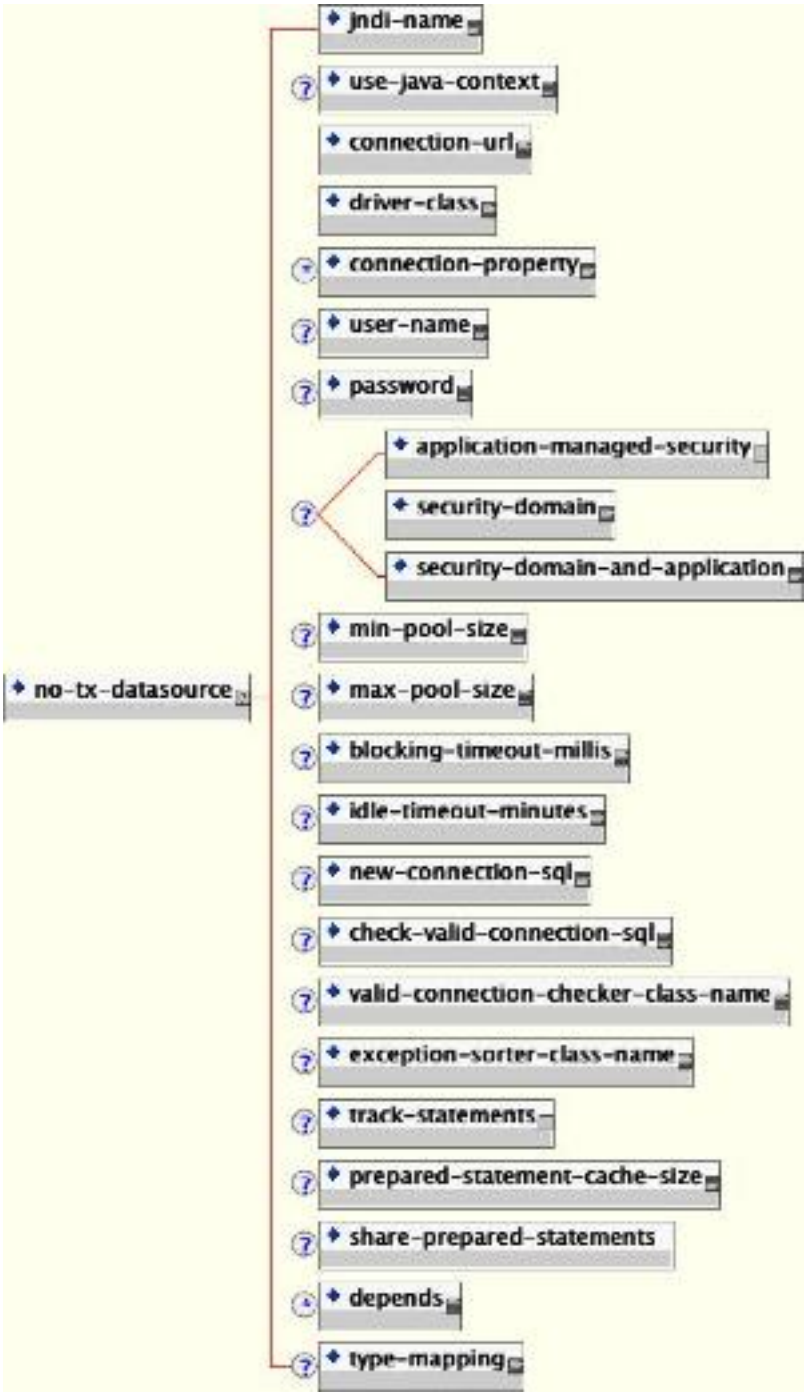


Figure 5.6. The non-transactional DataSource configuration schema

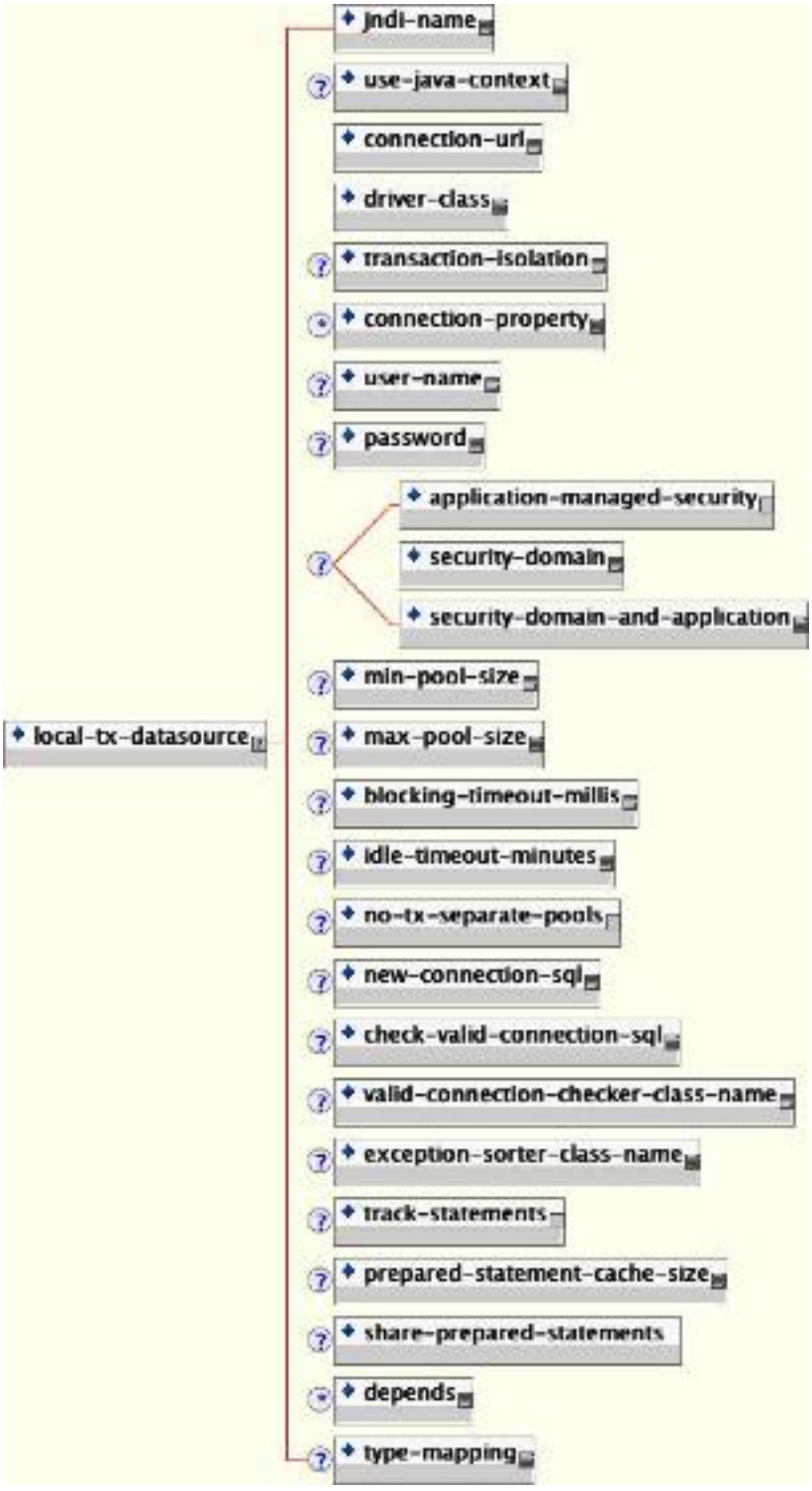


Figure 5.7. The non-XA DataSource configuration schema

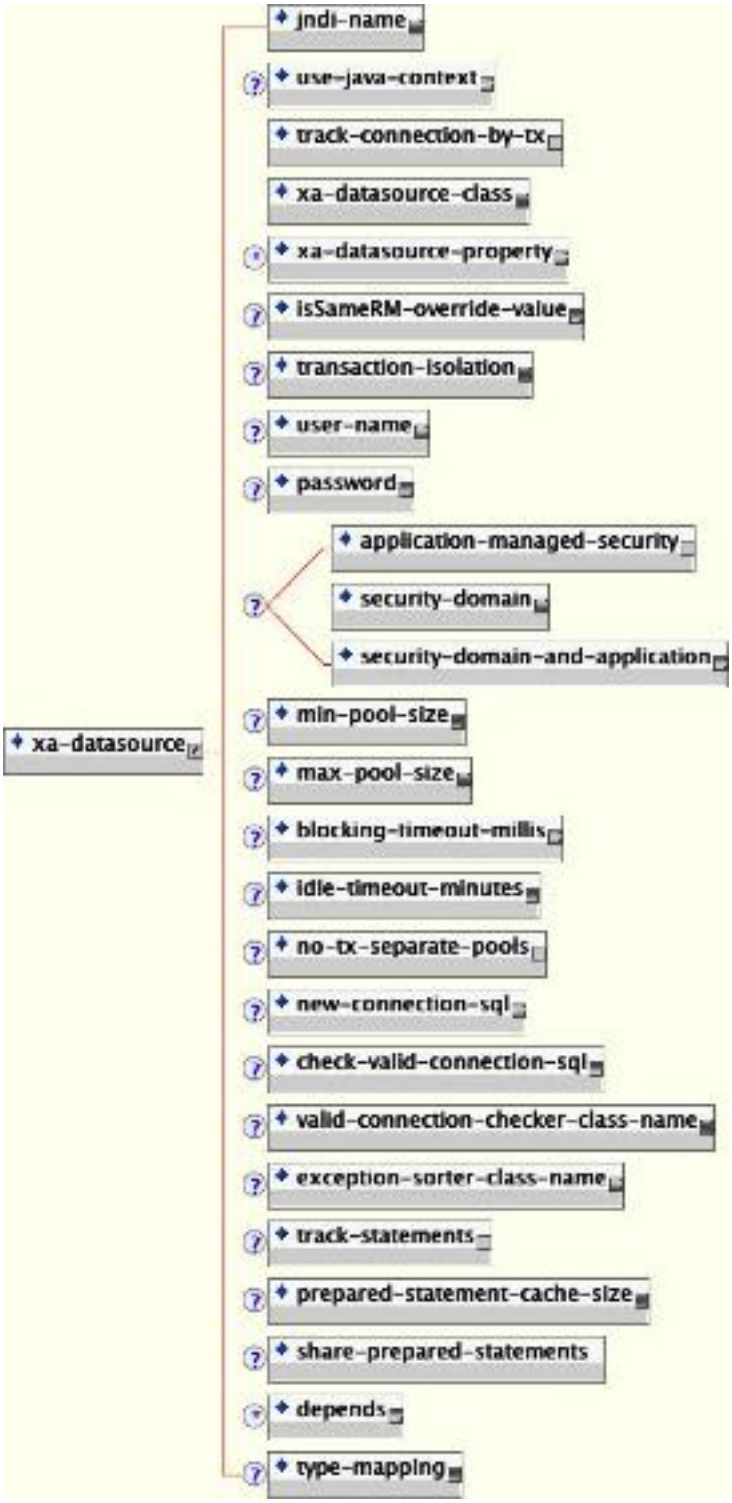


Figure 5.8. The XA DataSource configuration schema

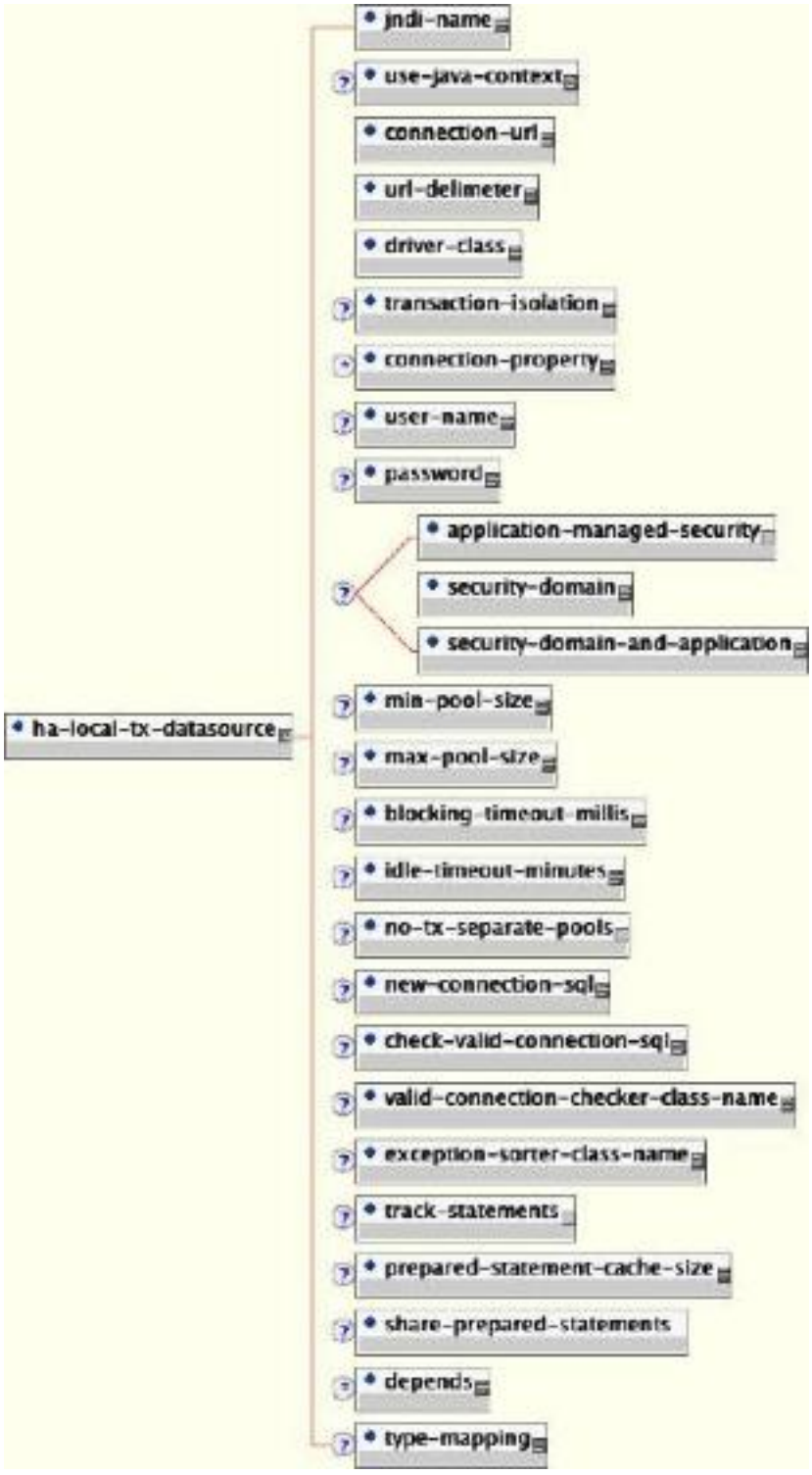


Figure 5.9. The schema for the experimental non-XA DataSource with failover

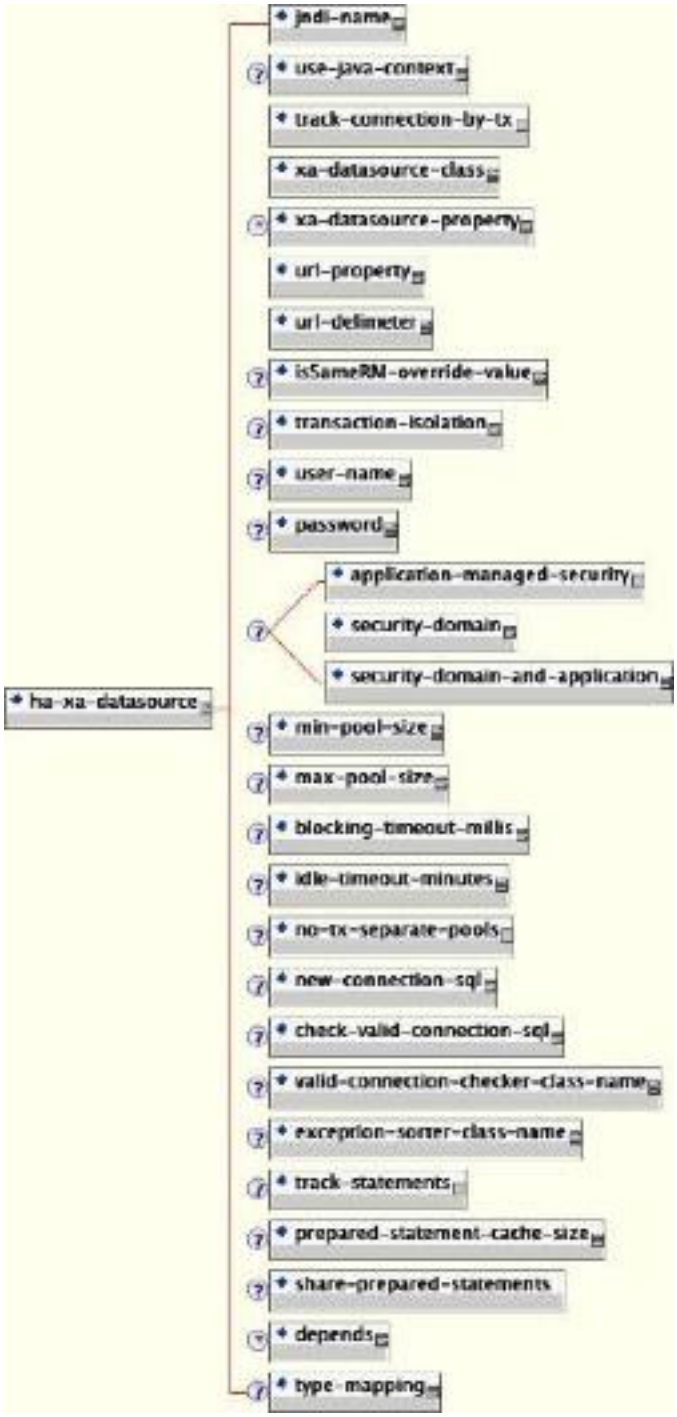


Figure 5.10. The schema for the experimental XA Datasource with failover

Elements that are common to all datasources include:

- **jndi-name**: The JNDI name under which the `DataSource` wrapper will be bound. Note that

this name is relative to the `java:/` context, unless `use-java-context` is set to false. `DataSource` wrappers are not usable outside of the server VM, so they are normally bound under the `java:/`, which isn't shared outside the local VM.

- **use-java-context**: If this is set to false the the datasource will be bound in the global JNDI context rather than the `java:` context.
- **user-name**: This element specifies the default username used when creating a new connection. The actual username may be overridden by the application code `getConnection` parameters or the connection creation context JAAS Subject.
- **password**: This element specifies the default password used when creating a new connection. The actual password may be overridden by the application code `getConnection` parameters or the connection creation context JAAS Subject.
- **application-managed-security**: Specifying this element indicates that connections in the pool should be distinguished by application code supplied parameters, such as from `getConnection(user, pw)`.
- **security-domain**: Specifying this element indicates that connections in the pool should be distinguished by JAAS Subject based information. The content of the `security-domain` is the name of the JAAS security manager that will handle authentication. This name correlates to the JAAS `login-config.xml` descriptor `application-policy/name` attribute.
- **security-domain-and-application**: Specifying this element indicates that connections in the pool should be distinguished both by application code supplied parameters and JAAS Subject based information. The content of the `security-domain` is the name of the JAAS security manager that will handle authentication. This name correlates to the JAAS `login-config.xml` descriptor `application-policy/name` attribute.
- **min-pool-size**: This element specifies the minimum number of connections a pool should hold. These pool instances are not created until an initial request for a connection is made. This default to 0.
- **max-pool-size**: This element specifies the maximum number of connections for a pool. No more than the `max-pool-size` number of connections will be created in a pool. This defaults to 20.
- **blocking-timeout-millis**: This element specifies the maximum time in milliseconds to block while waiting for a connection before throwing an exception. Note that this blocks only while waiting for a permit for a connection, and will never throw an exception if creating a new connection takes an inordinately long time. The default is 5000.
- **idle-timeout-minutes**: This element specifies the maximum time in minutes a connection may be idle before being closed. The actual maximum time depends also on the `IdleRemover` scan time, which is 1/2 the smallest idle-timeout-minutes of any pool.
- **new-connection-sql**: This is a SQL statement that should be executed when a new connection is created. This can be used to configure a connection with database specific

settings not configurable via connection properties.

- **check-valid-connection-sql**: This is a SQL statement that should be run on a connection before it is returned from the pool to test its validity to test for stale pool connections. An example statement could be: `select count(*) from x`.

- **exception-sorter-class-name**: This specifies a class that implements the `org.jboss.resource.adapter.jdbc.ExceptionSorter` interface to examine database exceptions to determine whether or not the exception indicates a connection error. Current implementations include:
  - org.jboss.resource.adapter.jdbc.vendor.OracleExceptionSorter
  - org.jboss.resource.adapter.jdbc.vendor.MySQLExceptionSorter
  - org.jboss.resource.adapter.jdbc.vendor.SybaseExceptionSorter
  - org.jboss.resource.adapter.jdbc.vendor.InformixExceptionSorte

- **valid-connection-checker-class-name**: This specifies a class that implements the `org.jboss.resource.adapter.jdbc.ValidConnectionChecker` interface to provide a `SQLException isValidConnection(Connection e)` method that is called with a connection that is to be returned from the pool to test its validity. This overrides the `check-valid-connection-sql` when present. The only provided implementation is `org.jboss.resource.adapter.jdbc.vendor.OracleValidConnectionChecker`.

- **track-statements**: This boolean element specifies whether to check for unclosed statements when a connection is returned to the pool. If true, a warning message is issued for each unclosed statement. If the log4j category `org.jboss.resource.adapter.jdbc.WrappedConnection` has trace level enabled, a stack trace of the connection close call is logged as well. This is a debug feature that can be turned off in production.

- **prepared-statement-cache-size**: This element specifies the number of prepared statements per connection in an LRU cache, which is keyed by the SQL query. Setting this to zero disables the cache.

- **depends**: The `depends` element specifies the JMX `ObjectName` string of a service that the connection manager services depend on. The connection manager service will not be started until the dependent services have been started.

- **type-mapping**: This element declares a default type mapping for this datasource. The type mapping should match a `type-mapping/name` element from `standardjbosscmp-jdbc.xml`.

Additional common child elements for both `no-tx-datasource` and `local-tx-datasource` include:

- **connection-url**: This is the JDBC driver connection URL string, for example, `jdbc:hsqldb:hsql://localhost:1701`.

- **driver-class**: This is the fully qualified name of the JDBC driver class, for example,

`org.hsqldb.jdbcDriver`.

- **connection-property**: The `connection-property` element allows you to pass in arbitrary connection properties to the `java.sql.Driver.connect(url, props)` method. Each `connection-property` specifies a string name/value pair with the property name coming from the name attribute and the value coming from the element content.

Elements in common to the `local-tx-datasource` and `xa-datasource` are:

- **transaction-isolation**: This element specifies the `java.sql.Connection` transaction isolation level to use. The constants defined in the Connection interface are the possible element content values and include:
  - TRANSACTION_READ_UNCOMMITTED
  - TRANSACTION_READ_COMMITTED
  - TRANSACTION_REPEATABLE_READ
  - TRANSACTION_SERIALIZABLE
  - TRANSACTION_NONE
- **no-tx-separate-pools**: The presence of this element indicates that two connection pools are required to isolate connections used with JTA transaction from those used without a JTA transaction. The pools are lazily constructed on first use. Its use case is for Oracle (and possibly other vendors) XA implementations that don't like using an XA connection with and without a JTA transaction.

The unique `xa-datasource` child elements are:

- **track-connection-by-tx**: Specifying a true value for this element makes the connection manager keep an xid to connection map and only put the connection back in the pool when the transaction completes and all the connection handles are closed or disassociated (by the method calls returning). As a side effect, we never suspend and resume the xid on the connection's `XAResource`. This is the same connection tracking behavior used for local transactions.

  The XA spec implies that any connection may be enrolled in any transaction using any xid for that transaction at any time from any thread (suspending other transactions if necessary). The original JCA implementation assumed this and aggressively delisted connections and put them back in the pool as soon as control left the EJB they were used in or handles were closed. Since some other transaction could be using the connection the next time work needed to be done on the original transaction, there is no way to get the original connection back. It turns out that most `XADataSource` driver vendors do not support this, and require that all work done under a particular xid go through the same connection.
- **xa-datasource-class**: The fully qualified name of the `javax.sql.XADataSource` implementation class, for example, `com.informix.jdbcx.IfxXADataSource`.
- **xa-datasource-property**: The `xa-datasource-property` element allows for specification of

the properties to assign to the `XADataSource` implementation class. Each property is identified by the name attribute and the property value is given by the `xa-datasource-property` element content. The property is mapped onto the `XADataSource` implementation by looking for a JavaBeans style getter method for the property name. If found, the value of the property is set using the JavaBeans setter with the element text translated to the true property type using the `java.beans.PropertyEditor` for the type.

- **isSameRM-override-value**: A boolean flag that allows one to override the behavior of the `javax.transaction.xa.XAResource.isSameRM(XAResource xaRes)` method behavior on the XA managed connection. If specified, this value is used unconditionally as the `isSameRM(xaRes)` return value regardless of the `xaRes` parameter.

The failover options common to `ha-xa-datasource` and `ha-local-tx-datasource` are:

- **url-delimeter**: This element specifies a character used to separate multiple JDBC URLs.
- **url-property**: In the case of XA datasources, this property specifies the name of the `xa-datasource-property` that contains the list of JDBC URLs to use.

Example configurations for many third-party JDBC drivers are included in the `JBOSS_DIST/docs/examples/jca` directory. Current example configurations include:

- asapxcess-jb3.2-ds.xml
- cicsr9s-service.xml
- db2-ds.xml
- db2-xa-ds.xml
- facets-ds.xml
- fast-objects-jboss32-ds.xml
- firebird-ds.xml
- firstsql-ds.xml
- firstsql-xa-ds.xml
- generic-ds.xml
- hsqldb-ds.xml
- informix-ds.xml
- informix-xa-ds.xml
- jdatastore-ds.xml
- jms-ds.xml
- jsql-ds.xml
- lido-versant-service.xml
- mimer-ds.xml
- mimer-xa-ds.xml
- msaccess-ds.xml
- mssql-ds.xml
- mssql-xa-ds.xml
- mysql-ds.xml

- oracle-ds.xml
- oracle-xa-ds.xml
- postgres-ds.xml
- sapdb-ds.xml
- sapr3-ds.xml
- solid-ds.xml
- sybase-ds.xml

4. Configuring Generic JCA Adaptors

The XSLSubDeployer also supports the deployment of arbitrary non-JDBC JCA resource adaptors. The schema for the top-level connection factory elements of the `*-ds.xml` configuration deployment file is shown in *Figure 5.11, "The simplified JCA adaptor connection factory configuration descriptor top-level schema elements"*.

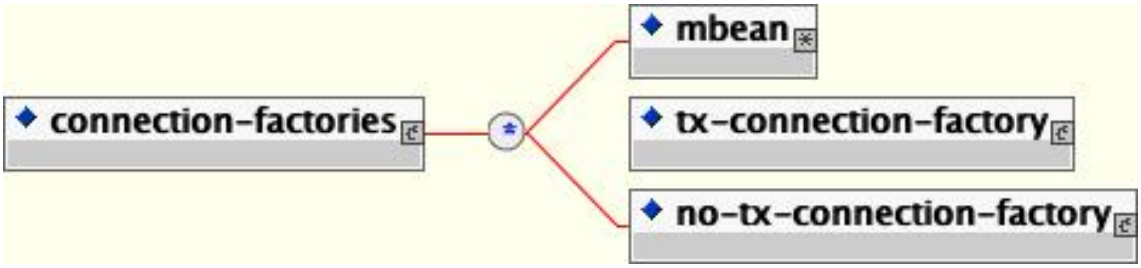


Figure 5.11. The simplified JCA adaptor connection factory configuration descriptor top-level schema elements

Multiple connection factory configurations may be specified in a configuration deployment file. The child elements of the `connection-factories` root are:

- **mbean**: Any number mbean elements may be specified to define MBean services that should be included in the `jboss-service.xml` descriptor that results from the transformation. This may be used to configure additional services used by the adaptor.
- **no-tx-connection-factory**: this element is used to specify the (`org.jboss.resource.connectionmanager`) `NoTxConnectionManager` service configuration. `NoTxConnectionManager` is a JCA connection manager with no transaction support. The `no-tx-connection-factory` child element schema is given in *Figure 5.12, "The no-tx-connection-factory element schema"*.
- **tx-connection-factory**: this element is used to specify the (`org.jboss.resource.connectionmanager`) `TxConnectionManager` service configuration. The `tx-connection-factory` child element schema is given in *Figure 5.13, "The tx-connection-factory element schema"*.

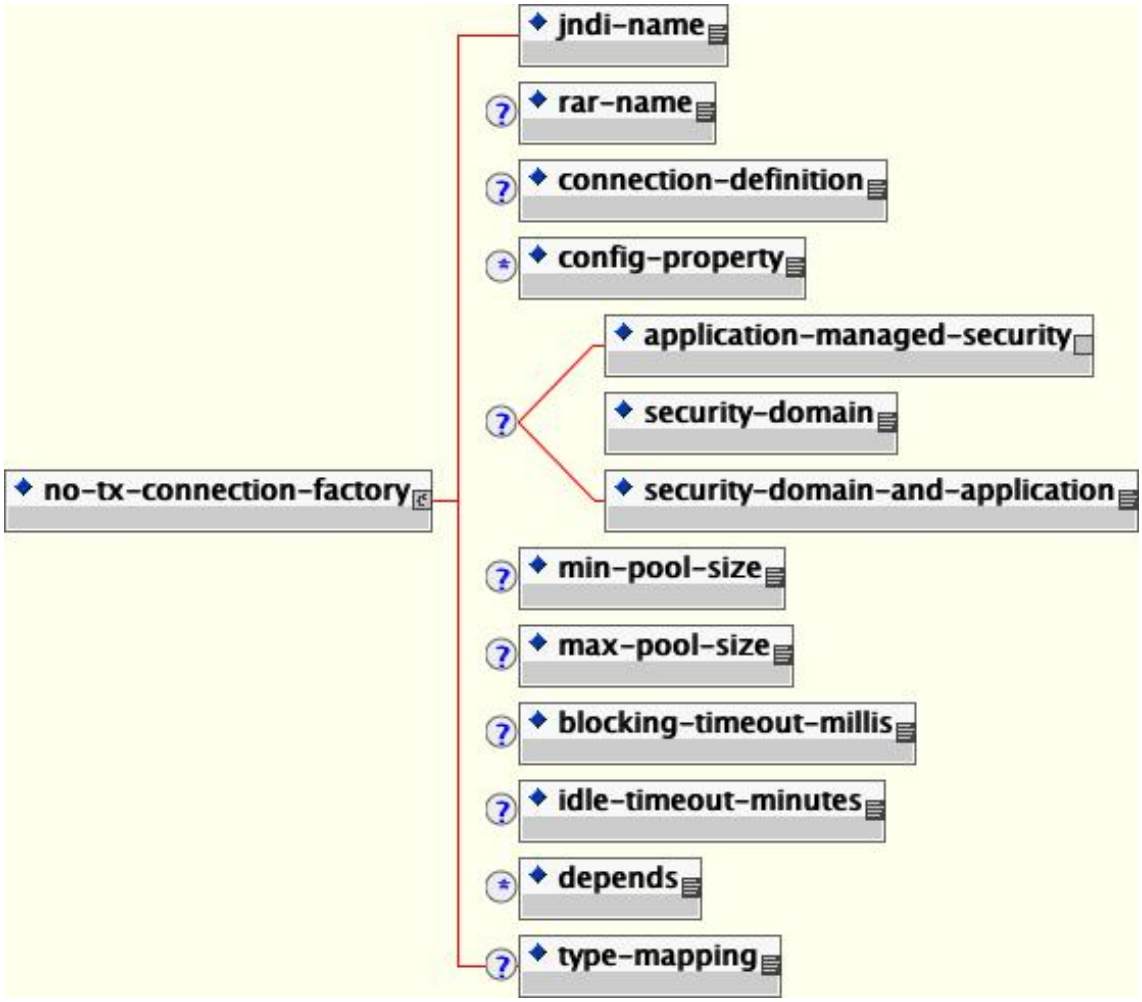


Figure 5.12. The no-tx-connection-factory element schema

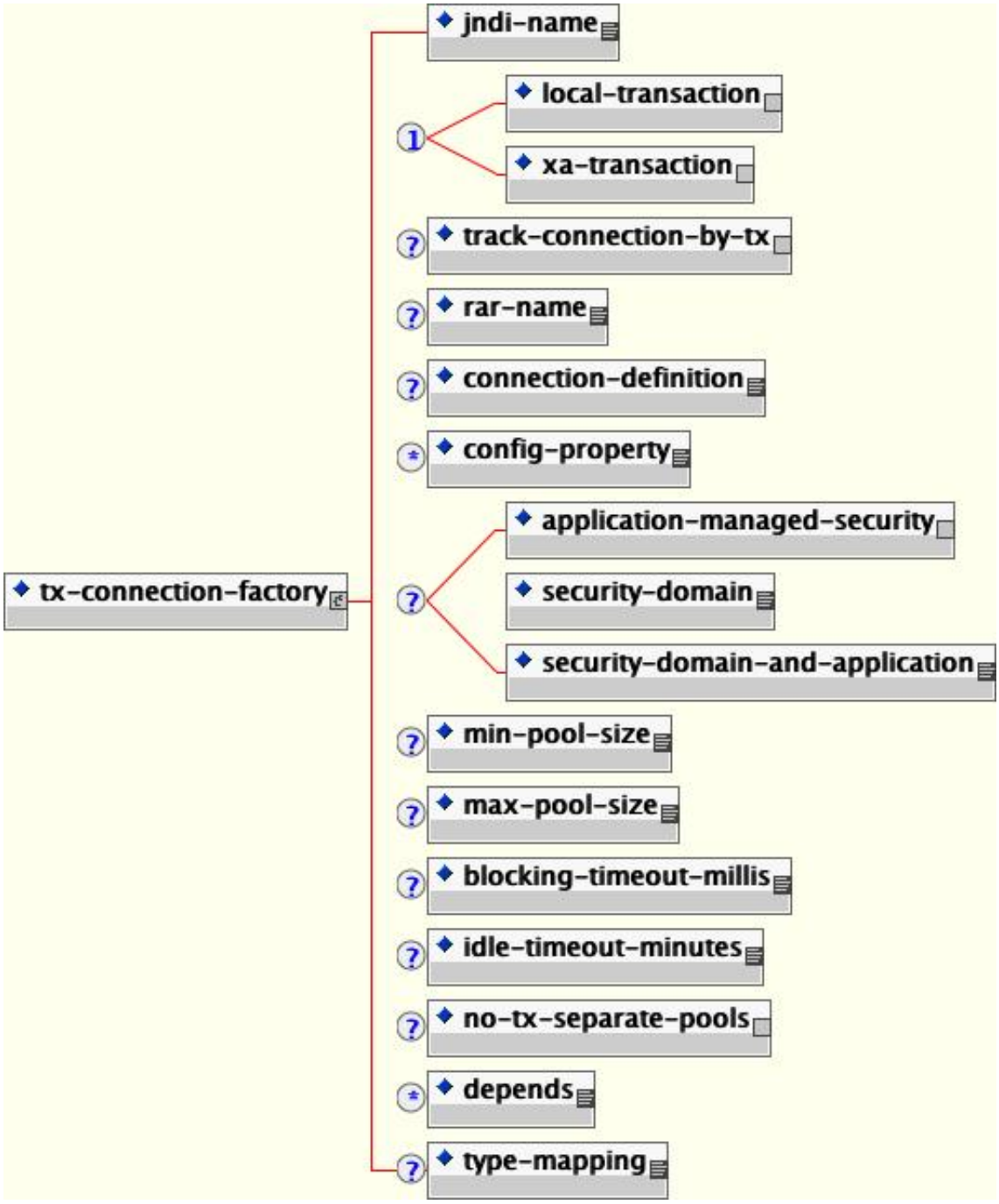


Figure 5.13. The tx-connection-factory element schema

The majority of the elements are the same as those of the datasources configuration. The element unique to the connection factory configuration include:

- **adaptor-display-name**: A human readable display name to assign to the connection manager MBean.

- **local-transaction**: This element specifies that the `tx-connection-factory` supports local transactions.
- **xa-transaction**: This element specifies that the `tx-connection-factory` supports XA transactions.
- **track-connection-by-tx**: This element specifies that a connection should be used only on a single transaction and that a transaction should only be associated with one connection.
- **rar-name**: This is the name of the RAR file that contains the definition for the resource we want to provide. For nested RAR files, the name would look like `myapplication.ear#my.rar`.
- **connection-definition**: This is the connection factory interface class. It should match the `connectionfactory-interface` in the `ra.xml` file.
- **config-property**: Any number of properties to supply to the `ManagedConnectionFactory` (MCF) MBean service configuration. Each `config-property` element specifies the value of a MCF property. The `config-property` element has two required attributes:
  - **name**: The name of the property
  - **type**: The fully qualified type of the property

  The content of the `config-property` element provides the string representation of the property value. This will be converted to the true property type using the associated type `PropertyEditor`.

Transactions on JBoss

The JTA Transaction Service

This chapter discusses transaction management in JBoss and the JBossTX architecture. The JBossTX architecture allows for any Java Transaction API (JTA) transaction manager implementation to be used. JBossTX includes a fast in-VM implementation of a JTA compatible transaction manager that is used as the default transaction manager. We will first provide an overview of the key transaction concepts and notions in the JTA to provide sufficient background for the JBossTX architecture discussion. We will then discuss the interfaces that make up the JBossTX architecture and conclude with a discussion of the MBeans available for integration of alternate transaction managers.

1. Transaction/JTA Overview

For the purpose of this discussion, we can define a transaction as a unit of work containing one or more operations involving one or more shared resources having ACID properties. ACID is an acronym for atomicity, consistency, isolation and durability, the four important properties of transactions. The meanings of these terms is:

- **Atomicity**: A transaction must be atomic. This means that either all the work done in the transaction must be performed, or none of it must be performed. Doing part of a transaction is not allowed.
- **Consistency**: When a transaction is completed, the system must be in a stable and consistent condition.
- **Isolation**: Different transactions must be isolated from each other. This means that the partial work done in one transaction is not visible to other transactions until the transaction is committed, and that each process in a multi-user system can be programmed as if it was the only process accessing the system.
- **Durability**: The changes made during a transaction are made persistent when it is committed. When a transaction is committed, its changes will not be lost, even if the server crashes afterwards.

To illustrate these concepts, consider a simple banking account application. The banking application has a database with a number of accounts. The sum of the amounts of all accounts must always be 0. An amount of money M is moved from account A to account B by subtracting M from account A and adding M to account B. This operation must be done in a transaction, and all four ACID properties are important.

The atomicity property means that both the withdrawal and deposit is performed as an indivisible unit. If, for some reason, both cannot be done nothing will be done.

The consistency property means that after the transaction, the sum of the amounts of all

accounts must still be 0.

The isolation property is important when more than one bank clerk uses the system at the same time. A withdrawal or deposit could be implemented as a three-step process: First the amount of the account is read from the database; then something is subtracted from or added to the amount read from the database; and at last the new amount is written to the database. Without transaction isolation several bad things could happen. For example, if two processes read the amount of account A at the same time, and each independently added or subtracted something before writing the new amount to the database, the first change would be incorrectly overwritten by the last.

The durability property is also important. If a money transfer transaction is committed, the bank must trust that some subsequent failure cannot undo the money transfer.

1.1. Pessimistic and optimistic locking

Transactional isolation is usually implemented by locking whatever is accessed in a transaction. There are two different approaches to transactional locking: Pessimistic locking and optimistic locking.

The disadvantage of pessimistic locking is that a resource is locked from the time it is first accessed in a transaction until the transaction is finished, making it inaccessible to other transactions during that time. If most transactions simply look at the resource and never change it, an exclusive lock may be overkill as it may cause lock contention, and optimistic locking may be a better approach. With pessimistic locking, locks are applied in a fail-safe way. In the banking application example, an account is locked as soon as it is accessed in a transaction. Attempts to use the account in other transactions while it is locked will either result in the other process being delayed until the account lock is released, or that the process transaction will be rolled back. The lock exists until the transaction has either been committed or rolled back.

With optimistic locking, a resource is not actually locked when it is first is accessed by a transaction. Instead, the state of the resource at the time when it would have been locked with the pessimistic locking approach is saved. Other transactions are able to concurrently access to the resource and the possibility of conflicting changes is possible. At commit time, when the resource is about to be updated in persistent storage, the state of the resource is read from storage again and compared to the state that was saved when the resource was first accessed in the transaction. If the two states differ, a conflicting update was made, and the transaction will be rolled back.

In the banking application example, the amount of an account is saved when the account is first accessed in a transaction. If the transaction changes the account amount, the amount is read from the store again just before the amount is about to be updated. If the amount has changed since the transaction began, the transaction will fail itself, otherwise the new amount is written to persistent storage.

1.2. The components of a distributed transaction

There are a number of participants in a distributed transaction. These include:

- **Transaction Manager**: This component is distributed across the transactional system. It manages and coordinates the work involved in the transaction. The transaction manager is exposed by the `javax.transaction.TransactionManager` interface in JTA.
- **Transaction Context**: A transaction context identifies a particular transaction. In JTA the corresponding interface is `javax.transaction.Transaction`.
- **Transactional Client**: A transactional client can invoke operations on one or more transactional objects in a single transaction. The transactional client that started the transaction is called the transaction originator. A transaction client is either an explicit or implicit user of JTA interfaces and has no interface representation in the JTA.
- **Transactional Object**: A transactional object is an object whose behavior is affected by operations performed on it within a transactional context. A transactional object can also be a transactional client. Most Enterprise Java Beans are transactional objects.
- **Recoverable Resource**: A recoverable resource is a transactional object whose state is saved to stable storage if the transaction is committed, and whose state can be reset to what it was at the beginning of the transaction if the transaction is rolled back. At commit time, the transaction manager uses the two-phase XA protocol when communicating with the recoverable resource to ensure transactional integrity when more than one recoverable resource is involved in the transaction being committed. Transactional databases and message brokers like JBossMQ are examples of recoverable resources. A recoverable resource is represented using the `javax.transaction.xa.XAResource` interface in JTA.

1.3. The two-phase XA protocol

When a transaction is about to be committed, it is the responsibility of the transaction manager to ensure that either all of it is committed, or that all of is rolled back. If only a single recoverable resource is involved in the transaction, the task of the transaction manager is simple: It just has to tell the resource to commit the changes to stable storage.

When more than one recoverable resource is involved in the transaction, management of the commit gets more complicated. Simply asking each of the recoverable resources to commit changes to stable storage is not enough to maintain the atomic property of the transaction. The reason for this is that if one recoverable resource has committed and another fails to commit, part of the transaction would be committed and the other part rolled back.

To get around this problem, the two-phase XA protocol is used. The XA protocol involves an extra prepare phase before the actual commit phase. Before asking any of the recoverable resources to commit the changes, the transaction manager asks all the recoverable resources to prepare to commit. When a recoverable resource indicates it is prepared to commit the transaction, it has ensured that it can commit the transaction. The resource is still able to rollback the transaction if necessary as well.

So the first phase consists of the transaction manager asking all the recoverable resources to prepare to commit. If any of the recoverable resources fails to prepare, the transaction will be rolled back. But if all recoverable resources indicate they were able to prepare to commit, the

second phase of the XA protocol begins. This consists of the transaction manager asking all the recoverable resources to commit the transaction. Because all the recoverable resources have indicated they are prepared, this step cannot fail.

1.4. Heuristic exceptions

In a distributed environment communications failures can happen. If communication between the transaction manager and a recoverable resource is not possible for an extended period of time, the recoverable resource may decide to unilaterally commit or rollback changes done in the context of a transaction. Such a decision is called a heuristic decision. It is one of the worst errors that may happen in a transaction system, as it can lead to parts of the transaction being committed while other parts are rolled back, thus violating the atomicity property of transaction and possibly leading to data integrity corruption.

Because of the dangers of heuristic exceptions, a recoverable resource that makes a heuristic decision is required to maintain all information about the decision in stable storage until the transaction manager tells it to forget about the heuristic decision. The actual data about the heuristic decision that is saved in stable storage depends on the type of recoverable resource and is not standardized. The idea is that a system manager can look at the data, and possibly edit the resource to correct any data integrity problems.

There are several different kinds of heuristic exceptions defined by the JTA. The `javax.transaction.HeuristicCommitException` is thrown when a recoverable resource is asked to rollback to report that a heuristic decision was made and that all relevant updates have been committed. On the opposite end is the `javax.transaction.HeuristicRollbackException`, which is thrown by a recoverable resource when it is asked to commit to indicate that a heuristic decision was made and that all relevant updates have been rolled back.

The `javax.transaction.HeuristicMixedException` is the worst heuristic exception. It is thrown to indicate that parts of the transaction were committed, while other parts were rolled back. The transaction manager throws this exception when some recoverable resources did a heuristic commit, while other recoverable resources did a heuristic rollback.

1.5. Transaction IDs and branches

In JTA, the identity of transactions is encapsulated in objects implementing the `javax.transaction.xa.Xid` interface. The transaction ID is an aggregate of three parts:

- The format identifier indicates the transaction family and tells how the other two parts should be interpreted.
- The global transaction id identified the global transaction within the transaction family.
- The branch qualifier denotes a particular branch of the global transaction.

Transaction branches are used to identify different parts of the same global transaction.

Whenever the transaction manager involves a new recoverable resource in a transaction it creates a new transaction branch.

2. JTS support

JBoss Transactions is a 100% Java implementation of a distributed transaction management system based on the Sun Microsystems J2EE Java Transaction Service (JTS) standard. Our implementation of the JTS utilizes the Object Management Group's (OMG) Object Transaction Service (OTS) model for transaction interoperability as recommended in the J2EE and EJB standards and leads the market in providing many advanced features such as fully distributed transactions and ORB portability with POA support.

3. Web Services Transactions

In traditional ACID transaction systems, transactions are short lived, resources (such as databases) are locked for the duration of the transaction and participants have a high degree of trust with each other. With the advent of the Internet and Web services, the scenario that is now emerging requires involvement of participants unknown to each other in distributed transactions. These transactions have the following characteristics:

- Transactions may be of a long duration, sometimes lasting hours, days, or more.
- Participants may not allow their resources to be locked for long durations.
- The communication infrastructure between participants may not be reliable.
- Some of the ACID properties of traditional transactions are not mandatory.
- A transaction may succeed even if only some of the participants choose to confirm and others cancel.
- All participants may choose to have their own coordinator (Transaction Manager), because of lack of trust.
- All activities are logged.
- Transactions that have to be rolled back have the concept of compensation.

JBoss Transactions adds native support for Web services transactions by providing all of the components necessary to build interoperable, reliable, multi-party, Web services-based applications with the minimum of effort. The programming interfaces are based on the Java API for XML Transactioning (JAXTX) and the product includes protocol support for the WS-AtomicTransaction and WS-BusinessActivity specifications. JBossTS 4.2 is designed to support multiple coordination protocols and therefore helps to future-proof transactional applications.

4. Configuring JBoss Transactions

JBossTS is configured through the jbossjts-properties.xml property file. You should consult the JBossTS documentation for all of the configurable options it supports.

5. Local versus distributed transactions

JBossTS supports both local and distributed transactions. A transaction is considered to be distributed if it spans multiple process instances, i.e., VMs. Typically a distributed transaction will contain participants (e.g., XAResources) that are located within multiple VMs but the transaction is coordinated in a separate VM (or co-located with one of the participants). If your architecture requires distributed transactions then you should consider using either the JTS implementation from JBossTS, which uses CORBA for communication, or the Web Services transactions component, which uses SOAP/HTTP.

Messaging on JBoss

JMS Configuration and Architecture

The JMS API stands for Java Message Service Application Programming Interface, and it is used by applications to send asynchronous *business-quality* messages to other applications. In the messaging world, messages are not sent directly to other applications. Instead, messages are sent to destinations, known as queues or topics. Applications sending messages do not need to worry if the receiving applications are up and running, and conversely, receiving applications do not need to worry about the sending application's status. Both senders, and receivers only interact with the destinations.

The JMS API is the standardized interface to a JMS provider, sometimes called a Message Oriented Middleware (MOM) system. JBoss comes with a JMS 1.1 compliant JMS provider called JBoss Messaging or JBossMQ. When you use the JMS API with JBoss, you are using the JBoss Messaging engine transparently. JBoss Messaging fully implements the JMS specification; therefore, the best JBoss Messaging user guide is the JMS specification. For more information about the JMS API please visit the JMS Tutorial or JMS Downloads & Specifications.

This chapter focuses on the JBoss specific aspects of using JMS and message driven beans as well as the JBoss Messaging configuration and MBeans.

1. JMS Examples

In this section we discuss the basics needed to use the JBoss JMS implementation. JMS leaves the details of accessing JMS connection factories and destinations as provider specific details. What you need to know to use the JBoss Messaging layer is:

- The location of the queue and topic connect factories: In JBoss both connection factory implementations are located under the JNDI name `ConnectionFactory`.
- How to lookup JMS destinations (queues and topics): Destinations are configured via MBeans as we will see when we discuss the messaging MBeans. JBoss comes with a few queues and topics preconfigured. You can find them under the `jboss.mq.destination` domain in the JMX Console..
- Which JARS JMS requires: These include `concurrent.jar`, `jbossmq-client.jar`, `jboss-common-client.jar`, `jboss-system-client.jar`, `jnp-client.jar` and `log4j.jar`.

In the following sections we will look at examples of the various JMS messaging models and message driven beans. The chapter example source is located under the `src/main/org/jboss/book/jms` directory of the book examples.

1.1. A Point-To-Point Example

Let's start out with a point-to-point (P2P) example. In the P2P model, a sender delivers messages to a queue and a single receiver pulls the message off of the queue. The receiver does not need to be listening to the queue at the time the message is sent. *Example 7.1, "A P2P JMS client example"* shows a complete P2P example that sends a `javax.jms.TextMessage` to the queue `queue/testQueue` and asynchronously receives the message from the same queue.

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

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueReceiver;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import EDU.oswego.cs.dl.util.concurrent.CountDown;
import org.apache.log4j.Logger;
import org.jboss.util.ChapterExRepository;

/**
 * A complete JMS client example program that sends a
 * TextMessage to a Queue and asynchronously receives the
 * message from the same Queue.
 *
 * @author  Scott.Stark@jboss.org
 * @version $Revision: 1.9 $
 */
public class SendRecvClient
{
    static Logger log;
    static CountDown done = new CountDown(1);

    QueueConnection conn;
    QueueSession session;
    Queue que;

    public static class ExListener
        implements MessageListener
    {
        public void onMessage(Message msg)
        {
            done.release();
            TextMessage tm = (TextMessage) msg;
            try {
                log.info("onMessage, recv text=" + tm.getText());
            } catch(Throwable t) {
                t.printStackTrace();
```

```
            }
        }
    }

    public void setupPTP()
        throws JMSException,
               NamingException
    {
        InitialContext iniCtx = new InitialContext();
        Object tmp = iniCtx.lookup("ConnectionFactory");
        QueueConnectionFactory qcf = (QueueConnectionFactory) tmp;
        conn = qcf.createQueueConnection();
        que = (Queue) iniCtx.lookup("queue/testQueue");
        session = conn.createQueueSession(false,
                                          QueueSession.AUTO_ACKNOWLEDGE);
        conn.start();
    }

    public void sendRecvAsync(String text)
        throws JMSException,
               NamingException
    {
        log.info("Begin sendRecvAsync");
        // Setup the PTP connection, session
        setupPTP();

        // Set the async listener
        QueueReceiver recv = session.createReceiver(que);
        recv.setMessageListener(new ExListener());

        // Send a text msg
        QueueSender send = session.createSender(que);
        TextMessage tm = session.createTextMessage(text);
        send.send(tm);
        log.info("sendRecvAsync, sent text=" + tm.getText());
        send.close();
        log.info("End sendRecvAsync");
    }

    public void stop()
        throws JMSException
    {
        conn.stop();
        session.close();
        conn.close();
    }

    public static void main(String args[])
        throws Exception
    {
        ChapterExRepository.init(SendRecvClient.class);
        log = Logger.getLogger("SendRecvClient");

        log.info("Begin SendRecvClient, now=" + System.currentTimeMillis());
        SendRecvClient client = new SendRecvClient();
        client.sendRecvAsync("A text msg");
```

```
        client.done.acquire();
        client.stop();
        log.info("End SendRecvClient");
        System.exit(0);
    }
}
```

Example 7.1. A P2P JMS client example

The client may be run using the following command line:

```
[examples]$ ant -Dchap=jms -Dex=1p2p run-example
...
run-example1p2p:
     [java] [INFO,SendRecvClient] Begin SendRecvClient, now=1102808673386
     [java] [INFO,SendRecvClient] Begin sendRecvAsync
     [java] [INFO,SendRecvClient] onMessage, recv text=A text msg
     [java] [INFO,SendRecvClient] sendRecvAsync, sent text=A text msg
     [java] [INFO,SendRecvClient] End sendRecvAsync
     [java] [INFO,SendRecvClient] End SendRecvClient
```

1.2. A Pub-Sub Example

The JMS publish/subscribe (Pub-Sub) message model is a one-to-many model. A publisher sends a message to a topic and all active subscribers of the topic receive the message. Subscribers that are not actively listening to the topic will miss the published message. shows a complete JMS client that sends a `javax.jms.TextMessage` to a topic and asynchronously receives the message from the same topic.

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

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicPublisher;
import javax.jms.TopicSubscriber;
import javax.jms.TopicSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import EDU.oswego.cs.dl.util.concurrent.CountDown;

/**
 *  A complete JMS client example program that sends a TextMessage to
```

```
 *  a Topic and asynchronously receives the message from the same
 *  Topic.
 *
 *  @author Scott.Stark@jboss.org
 *  @version $Revision: 1.9 $
 */

public class TopicSendRecvClient
{
    static CountDown done = new CountDown(1);
    TopicConnection conn = null;
    TopicSession session = null;
    Topic topic = null;

    public static class ExListener implements MessageListener
    {
        public void onMessage(Message msg)
        {
            done.release();
            TextMessage tm = (TextMessage) msg;
            try {
                System.out.println("onMessage, recv text=" + tm.getText());
            } catch(Throwable t) {
                t.printStackTrace();
            }
        }
    }

    public void setupPubSub()
        throws JMSException, NamingException
    {
        InitialContext iniCtx = new InitialContext();
        Object tmp = iniCtx.lookup("ConnectionFactory");
        TopicConnectionFactory tcf = (TopicConnectionFactory) tmp;
        conn = tcf.createTopicConnection();
        topic = (Topic) iniCtx.lookup("topic/testTopic");
        session = conn.createTopicSession(false,
                                          TopicSession.AUTO_ACKNOWLEDGE);
        conn.start();
    }

    public void sendRecvAsync(String text)
        throws JMSException, NamingException
    {
        System.out.println("Begin sendRecvAsync");
        // Setup the PubSub connection, session
        setupPubSub();
        // Set the async listener

        TopicSubscriber recv = session.createSubscriber(topic);
        recv.setMessageListener(new ExListener());
        // Send a text msg
        TopicPublisher send = session.createPublisher(topic);
        TextMessage tm = session.createTextMessage(text);
        send.publish(tm);
        System.out.println("sendRecvAsync, sent text=" + tm.getText());
```

```
        send.close();
        System.out.println("End sendRecvAsync");
    }

    public void stop() throws JMSException
    {
        conn.stop();
        session.close();
        conn.close();
    }

    public static void main(String args[]) throws Exception
    {
        System.out.println("Begin TopicSendRecvClient, now=" +
                           System.currentTimeMillis());
        TopicSendRecvClient client = new TopicSendRecvClient();
        client.sendRecvAsync("A text msg, now="+System.currentTimeMillis());
        client.done.acquire();
        client.stop();
        System.out.println("End TopicSendRecvClient");
        System.exit(0);
    }

}
```

Example 7.2. A Pub-Sub JMS client example

The client may be run using the following command line:

```
[examples]$ ant -Dchap=jms -Dex=1ps run-example
...
run-example1ps:
     [java] Begin TopicSendRecvClient, now=1102809427043
     [java] Begin sendRecvAsync
     [java] onMessage, recv text=A text msg, now=1102809427071
     [java] sendRecvAsync, sent text=A text msg, now=1102809427071
     [java] End sendRecvAsync
     [java] End TopicSendRecvClient
```

Now let's break the publisher and subscribers into separate programs to demonstrate that subscribers only receive messages while they are listening to a topic. *Example 7.3, "A JMS publisher client"* shows a variation of the previous pub-sub client that only publishes messages to the `topic/testTopic` topic. The subscriber only client is shown in *Example 7.4, "A JMS subscriber client"*.

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

import javax.jms.JMSException;
import javax.jms.Message;
```

```
import javax.jms.MessageListener;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicPublisher;
import javax.jms.TopicSlistubscriber;
import javax.jms.TopicSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 *  A JMS client example program that sends a TextMessage to a Topic
 *
 *  @author Scott.Stark@jboss.org
 *  @version $Revision: 1.9 $
 */
public class TopicSendClient
{
    TopicConnection conn = null;
    TopicSession session = null;
    Topic topic = null;

    public void setupPubSub()
        throws JMSException, NamingException
    {
        InitialContext iniCtx = new InitialContext();
        Object tmp = iniCtx.lookup("ConnectionFactory");
        TopicConnectionFactory tcf = (TopicConnectionFactory) tmp;
        conn = tcf.createTopicConnection();
        topic = (Topic) iniCtx.lookup("topic/testTopic");
        session = conn.createTopicSession(false,
                                          TopicSession.AUTO_ACKNOWLEDGE);
        conn.start();
    }

    public void sendAsync(String text)
        throws JMSException, NamingException
    {
        System.out.println("Begin sendAsync");
        // Setup the pub/sub connection, session
        setupPubSub();
        // Send a text msg
        TopicPublisher send = session.createPublisher(topic);
        TextMessage tm = session.createTextMessage(text);
        send.publish(tm);
        System.out.println("sendAsync, sent text=" +  tm.getText());
        send.close();
        System.out.println("End sendAsync");
    }

    public void stop()
        throws JMSException
    {
        conn.stop();
        session.close();
```

```
        conn.close();
    }

    public static void main(String args[])
        throws Exception
    {
        System.out.println("Begin TopicSendClient, now=" +
                                System.currentTimeMillis());
        TopicSendClient client = new TopicSendClient();
            client.sendAsync("A text msg, now="+System.currentTimeMillis());
        client.stop();
        System.out.println("End TopicSendClient");
        System.exit(0);
    }

}
```

Example 7.3. A JMS publisher client

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

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicPublisher;
import javax.jms.TopicSubscriber;
import javax.jms.TopicSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 * A JMS client example program that synchronously receives a message a
Topic
 *
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.9 $
 */
public class TopicRecvClient
{
    TopicConnection conn = null;
    TopicSession session = null;
    Topic topic = null;

    public void setupPubSub()
        throws JMSException, NamingException
    {
        InitialContext iniCtx = new InitialContext();
```

```
        Object tmp = iniCtx.lookup("ConnectionFactory");
        TopicConnectionFactory tcf = (TopicConnectionFactory) tmp;
        conn = tcf.createTopicConnection();
        topic = (Topic) iniCtx.lookup("topic/testTopic");
        session = conn.createTopicSession(false,
                                          TopicSession.AUTO_ACKNOWLEDGE);
        conn.start();
    }

    public void recvSync()
        throws JMSException, NamingException
    {
        System.out.println("Begin recvSync");
        // Setup the pub/sub connection, session
        setupPubSub();

        // Wait upto 5 seconds for the message
        TopicSubscriber recv = session.createSubscriber(topic);
        Message msg = recv.receive(5000);
        if (msg == null) {
            System.out.println("Timed out waiting for msg");
        } else {
            System.out.println("TopicSubscriber.recv, msgt="+msg);
        }
    }

    public void stop()
        throws JMSException
    {
        conn.stop();
        session.close();
        conn.close();
    }

    public static void main(String args[])
        throws Exception
    {
        System.out.println("Begin TopicRecvClient, now=" +
                           System.currentTimeMillis());
        TopicRecvClient client = new TopicRecvClient();
        client.recvSync();
        client.stop();
        System.out.println("End TopicRecvClient");
        System.exit(0);
    }

}
```

Example 7.4. A JMS subscriber client

Run the `TopicSendClient` followed by the `TopicRecvClient` as follows:

```
[examples]$ ant -Dchap=jms -Dex=1ps2 run-example
...
run-example1ps2:
     [java] Begin TopicSendClient, now=1102810007899
     [java] Begin sendAsync
     [java] sendAsync, sent text=A text msg, now=1102810007909
     [java] End sendAsync
     [java] End TopicSendClient
     [java] Begin TopicRecvClient, now=1102810011524
     [java] Begin recvSync
     [java] Timed out waiting for msg
     [java] End TopicRecvClient
```

The output shows that the topic subscriber client (`TopicRecvClient`) fails to receive the message sent by the publisher due to a timeout.

1.3. A Pub-Sub With Durable Topic Example

JMS supports a messaging model that is a cross between the P2P and pub-sub models. When a pub-sub client wants to receive all messages posted to the topic it subscribes to even when it is not actively listening to the topic, the client may achieve this behavior using a durable topic. Let's look at a variation of the preceding subscriber client that uses a durable topic to ensure that it receives all messages, include those published when the client is not listening to the topic. *Example 7.5, "A durable topic JMS client example"* shows the durable topic client with the key differences between the *Example 7.4, "A JMS subscriber client"* client highlighted in bold.

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

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicPublisher;
import javax.jms.TopicSubscriber;
import javax.jms.TopicSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 *  A JMS client example program that synchronously receives a message a
Topic
 *
 *  @author Scott.Stark@jboss.org
 *  @version $Revision: 1.9 $
 */
public class DurableTopicRecvClient
{
    TopicConnection conn = null;
    TopicSession session = null;
```

```
    Topic topic = null;

    public void setupPubSub()
        throws JMSException, NamingException
    {
        InitialContext iniCtx = new InitialContext();
        Object tmp = iniCtx.lookup("ConnectionFactory");

        TopicConnectionFactory tcf = (TopicConnectionFactory) tmp;
        conn = tcf.createTopicConnection("john", "needle");
        topic = (Topic) iniCtx.lookup("topic/testTopic");

        session = conn.createTopicSession(false,
                                          TopicSession.AUTO_ACKNOWLEDGE);
        conn.start();
    }

    public void recvSync()
        throws JMSException, NamingException
    {
        System.out.println("Begin recvSync");
        // Setup the pub/sub connection, session
        setupPubSub();
        // Wait upto 5 seconds for the message
        TopicSubscriber recv = session.createDurableSubscriber(topic,
"jms-ex1dtps");
        Message msg = recv.receive(5000);
        if (msg == null) {
            System.out.println("Timed out waiting for msg");
        } else {
            System.out.println("DurableTopicRecvClient.recv, msgt=" + msg);
        }
    }

    public void stop()
        throws JMSException
    {
        conn.stop();
        session.close();
        conn.close();
    }

    public static void main(String args[])
        throws Exception
    {
        System.out.println("Begin DurableTopicRecvClient, now=" +
                           System.currentTimeMillis());
        DurableTopicRecvClient client = new DurableTopicRecvClient();
        client.recvSync();
        client.stop();
        System.out.println("End DurableTopicRecvClient");
        System.exit(0);
    }

}
```

Example 7.5. A durable topic JMS client example

Now run the previous topic publisher with the durable topic subscriber as follows:

```
[examples]$ ant -Dchap=jms -Dex=1psdt run-example
...
run-example1psdt:
     [java] Begin DurableTopicSetup
     [java] End DurableTopicSetup
     [java] Begin TopicSendClient, now=1102899834273
     [java] Begin sendAsync
     [java] sendAsync, sent text=A text msg, now=1102899834345
     [java] End sendAsync
     [java] End TopicSendClient
     [java] Begin DurableTopicRecvClient, now=1102899840043
     [java] Begin recvSync
     [java] DurableTopicRecvClient.recv, msgt=SpyTextMessage {
     [java] Header {
     [java]    jmsDestination  : TOPIC.testTopic.DurableSubscription[
               clientId=DurableSubscriberExample name=jms-ex1dtps
selector=null]
     [java]    jmsDeliveryMode : 2
     [java]    jmsExpiration   : 0
     [java]    jmsPriority     : 4
     [java]    jmsMessageID    : ID:3-11028998375501
     [java]    jmsTimeStamp    : 1102899837550
     [java]    jmsCorrelationID: null
     [java]    jmsReplyTo      : null
     [java]    jmsType         : null
     [java]    jmsRedelivered  : false
     [java]    jmsProperties   : {}
     [java]    jmsPropReadWrite: false
     [java]    msgReadOnly     : true
     [java]    producerClientId: ID:3
     [java] }
     [java] Body {
     [java]    text            :A text msg, now=1102899834345
     [java] }
     [java] }
     [java] End DurableTopicRecvClient
```

Items of note for the durable topic example include:

- The `TopicConnectionFactory` creation in the durable topic client used a username and password, and the `TopicSubscriber` creation was done using the `createDurableSubscriber(Topic, String)` method. This is a requirement of durable topic subscribers. The messaging server needs to know what client is requesting the durable topic and what the name of the durable topic subscription is. We will discuss the details of durable topic setup in the configuration section.

- An `org.jboss.book.jms.DurableTopicSetup` client was run prior to the `TopicSendClient`. The reason for this is a durable topic subscriber must have registered a subscription at some point in the past in order for the messaging server to save messages. JBoss supports dynamic durable topic subscribers and the `DurableTopicSetup` client simply creates a durable subscription receiver and the exits. This leaves an active durable topic subscriber on the `topic/testTopic` and the messaging server knows that any messages posted to this topic must be saved for latter delivery.
- The `TopicSendClient` does not change for the durable topic. The notion of a durable topic is a subscriber only notion.
- The `DurableTopicRecvClient` sees the message published to the `topic/testTopic` even though it was not listening to the topic at the time the message was published.

1.4. A Point-To-Point With MDB Example

Example 7.6, "A TextMessage processing MDB" shows an message driven bean (MDB) that transforms the `TextMessages` it receives and sends the transformed messages to the queue found in the incoming message `JMSReplyTo` header.

```
package org.jboss.book.jms.ex2;

import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.ejb.EJBException;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 * An MDB that transforms the TextMessages it receives and send the
 * transformed messages to the Queue found in the incoming message
 * JMSReplyTo header.
 *
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.9 $
 */
public class TextMDB
    implements MessageDrivenBean, MessageListener
{
    private MessageDrivenContext ctx = null;
    private QueueConnection conn;
    private QueueSession session;
```

```
public TextMDB()
{
    System.out.println("TextMDB.ctor, this="+hashCode());
}

public void setMessageDrivenContext(MessageDrivenContext ctx)
{
    this.ctx = ctx;
    System.out.println("TextMDB.setMessageDrivenContext, this=" +
                       hashCode());
}

public void ejbCreate()
{
    System.out.println("TextMDB.ejbCreate, this="+hashCode());
    try {
        setupPTP();
    } catch (Exception e) {
        throw new EJBException("Failed to init TextMDB", e);
    }
}

public void ejbRemove()
{
    System.out.println("TextMDB.ejbRemove, this="+hashCode());
    ctx = null;
    try {
        if (session != null) {
            session.close();
        }
        if (conn != null) {
            conn.close();
        }
    } catch(JMSException e) {
        e.printStackTrace();
    }
}

public void onMessage(Message msg)
{
    System.out.println("TextMDB.onMessage, this="+hashCode());
    try {
        TextMessage tm = (TextMessage) msg;
        String text = tm.getText() + "processed by: "+hashCode();
        Queue dest = (Queue) msg.getJMSReplyTo();
        sendReply(text, dest);
    } catch(Throwable t) {
        t.printStackTrace();
    }
}

private void setupPTP()
    throws JMSException, NamingException
{
    InitialContext iniCtx = new InitialContext();
    Object tmp = iniCtx.lookup("java:comp/env/jms/QCF");
```

```
        QueueConnectionFactory qcf = (QueueConnectionFactory) tmp;
        conn = qcf.createQueueConnection();
        session = conn.createQueueSession(false,
                                          QueueSession.AUTO_ACKNOWLEDGE);
        conn.start();
    }

    private void sendReply(String text, Queue dest)
        throws JMSException
    {
        System.out.println("TextMDB.sendReply, this=" +
                           hashCode() + ", dest="+dest);
        QueueSender sender = session.createSender(dest);
        TextMessage tm = session.createTextMessage(text);
        sender.send(tm);
        sender.close();
    }
}
```

Example 7.6. A TextMessage processing MDB

The MDB `ejb-jar.xml` and `jboss.xml` deployment descriptors are shown in *Example 7.7, "The MDB ejb-jar.xml descriptor"* and *Example 7.8, "The MDB jboss.xml descriptor"*.

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC
          "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
          "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
    <enterprise-beans>
        <message-driven>
            <ejb-name>TextMDB</ejb-name>
            <ejb-class>org.jboss.book.jms.ex2.TextMDB</ejb-class>
            <transaction-type>Container</transaction-type>
            <acknowledge-mode>AUTO_ACKNOWLEDGE</acknowledge-mode>
            <message-driven-destination>
                <destination-type>javax.jms.Queue</destination-type>
            </message-driven-destination>
            <res-ref-name>jms/QCF</res-ref-name>
            <resource-ref>
                <res-type>javax.jms.QueueConnectionFactory</res-type>
                <res-auth>Container</res-auth>
            </resource-ref>
        </message-driven>
    </enterprise-beans>
</ejb-jar>
```

Example 7.7. The MDB ejb-jar.xml descriptor

```
<?xml version="1.0"?>
<jboss>
    <enterprise-beans>
        <message-driven>
            <ejb-name>TextMDB</ejb-name>
            <destination-jndi-name>queue/B</destination-jndi-name>
            <resource-ref>
                <res-ref-name>jms/QCF</res-ref-name>
                <jndi-name>ConnectionFactory</jndi-name>
            </resource-ref>
        </message-driven>
    </enterprise-beans>
</jboss>
```

Example 7.8. The MDB jboss.xml descriptor

Example 7.9, "A JMS client that interacts with the TextMDB" shows a variation of the P2P client that sends several messages to the `queue/B` destination and asynchronously receives the messages as modified by `TextMDB` from queue `A`.

```
package org.jboss.book.jms.ex2;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueReceiver;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import EDU.oswego.cs.dl.util.concurrent.CountDown;

/**
 *  A complete JMS client example program that sends N TextMessages to
 *  a Queue B and asynchronously receives the messages as modified by
 *  TextMDB from Queue A.
 *
 *  @author Scott.Stark@jboss.org
 *  @version $Revision: 1.9 $
 */
public class SendRecvClient
{
    static final int N = 10;
    static CountDown done = new CountDown(N);

    QueueConnection conn;
```

```
QueueSession session;
Queue queA;
Queue queB;

public static class ExListener
    implements MessageListener
{
    public void onMessage(Message msg)
    {
        done.release();
        TextMessage tm = (TextMessage) msg;
        try {
            System.out.println("onMessage, recv text="+tm.getText());
        } catch(Throwable t) {
            t.printStackTrace();
        }
    }
}

public void setupPTP()
    throws JMSException, NamingException
{
    InitialContext iniCtx = new InitialContext();
    Object tmp = iniCtx.lookup("ConnectionFactory");
    QueueConnectionFactory qcf = (QueueConnectionFactory) tmp;
    conn = qcf.createQueueConnection();
    queA = (Queue) iniCtx.lookup("queue/A");
    queB = (Queue) iniCtx.lookup("queue/B");
    session = conn.createQueueSession(false,
                                      QueueSession.AUTO_ACKNOWLEDGE);
    conn.start();
}

public void sendRecvAsync(String textBase)
    throws JMSException, NamingException, InterruptedException
{
    System.out.println("Begin sendRecvAsync");

    // Setup the PTP connection, session
    setupPTP();

    // Set the async listener for queA
    QueueReceiver recv = session.createReceiver(queA);
    recv.setMessageListener(new ExListener());

    // Send a few text msgs to queB
    QueueSender send = session.createSender(queB);

    for(int m = 0; m < 10; m ++) {
        TextMessage tm = session.createTextMessage(textBase+"#"+m);
        tm.setJMSReplyTo(queA);
        send.send(tm);
        System.out.println("sendRecvAsync, sent text=" + tm.getText());
    }
    System.out.println("End sendRecvAsync");
}
```

```
    public void stop()
        throws JMSException
    {
        conn.stop();
        session.close();
        conn.close();
    }

    public static void main(String args[])
        throws Exception
    {
        System.out.println("Begin SendRecvClient,now=" +
                           System.currentTimeMillis());
        SendRecvClient client = new SendRecvClient();
        client.sendRecvAsync("A text msg");
        client.done.acquire();
        client.stop();
        System.exit(0);
        System.out.println("End SendRecvClient");
    }

}
```

Example 7.9. A JMS client that interacts with the TextMDB

Run the client as follows:

```
[examples]$ ant -Dchap=jms -Dex=2 run-example
...
run-example2:
...
     [java] Begin SendRecvClient, now=1102900541558
     [java] Begin sendRecvAsync
     [java] sendRecvAsync, sent text=A text msg#0
     [java] sendRecvAsync, sent text=A text msg#1
     [java] sendRecvAsync, sent text=A text msg#2
     [java] sendRecvAsync, sent text=A text msg#3
     [java] sendRecvAsync, sent text=A text msg#4
     [java] sendRecvAsync, sent text=A text msg#5
     [java] sendRecvAsync, sent text=A text msg#6
     [java] sendRecvAsync, sent text=A text msg#7
     [java] sendRecvAsync, sent text=A text msg#8
     [java] sendRecvAsync, sent text=A text msg#9
     [java] End sendRecvAsync
     [java] onMessage, recv text=A text msg#0processed by: 12855623
     [java] onMessage, recv text=A text msg#5processed by: 9399816
     [java] onMessage, recv text=A text msg#9processed by: 6598158
     [java] onMessage, recv text=A text msg#3processed by: 8153998
     [java] onMessage, recv text=A text msg#4processed by: 10118602
     [java] onMessage, recv text=A text msg#2processed by: 1792333
     [java] onMessage, recv text=A text msg#7processed by: 14251014
```

```
     [java] onMessage, recv text=A text msg#1processed by: 10775981
     [java] onMessage, recv text=A text msg#8processed by: 6056676
     [java] onMessage, recv text=A text msg#6processed by: 15679078
```

The corresponding JBoss server console output is:

```
19:15:40,232 INFO  [EjbModule] Deploying TextMDB
        19:15:41,498 INFO  [EJBDeployer] Deployed:
file:/jboss-4.2.0.GA/server/production/deploy/
  jms-ex2.jar
19:15:45,606 INFO  [TextMDB] TextMDB.ctor, this=10775981
19:15:45,620 INFO  [TextMDB] TextMDB.ctor, this=1792333
19:15:45,627 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=10775981
19:15:45,638 INFO  [TextMDB] TextMDB.ejbCreate, this=10775981
19:15:45,640 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=1792333
19:15:45,640 INFO  [TextMDB] TextMDB.ejbCreate, this=1792333
19:15:45,649 INFO  [TextMDB] TextMDB.ctor, this=12855623
19:15:45,658 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=12855623
19:15:45,661 INFO  [TextMDB] TextMDB.ejbCreate, this=12855623
19:15:45,742 INFO  [TextMDB] TextMDB.ctor, this=8153998
19:15:45,744 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=8153998
19:15:45,744 INFO  [TextMDB] TextMDB.ejbCreate, this=8153998
19:15:45,763 INFO  [TextMDB] TextMDB.ctor, this=10118602
19:15:45,764 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=10118602
19:15:45,764 INFO  [TextMDB] TextMDB.ejbCreate, this=10118602
19:15:45,777 INFO  [TextMDB] TextMDB.ctor, this=9399816
19:15:45,779 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=9399816
19:15:45,779 INFO  [TextMDB] TextMDB.ejbCreate, this=9399816
19:15:45,792 INFO  [TextMDB] TextMDB.ctor, this=15679078
19:15:45,798 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=15679078
19:15:45,799 INFO  [TextMDB] TextMDB.ejbCreate, this=15679078
19:15:45,815 INFO  [TextMDB] TextMDB.ctor, this=14251014
19:15:45,816 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=14251014
19:15:45,817 INFO  [TextMDB] TextMDB.ejbCreate, this=14251014
19:15:45,829 INFO  [TextMDB] TextMDB.ctor, this=6056676
19:15:45,831 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=6056676
19:15:45,864 INFO  [TextMDB] TextMDB.ctor, this=6598158
19:15:45,903 INFO  [TextMDB] TextMDB.ejbCreate, this=6056676
19:15:45,906 INFO  [TextMDB] TextMDB.setMessageDrivenContext, this=6598158
19:15:45,906 INFO  [TextMDB] TextMDB.ejbCreate, this=6598158
19:15:46,236 INFO  [TextMDB] TextMDB.onMessage, this=12855623
19:15:46,238 INFO  [TextMDB] TextMDB.sendReply, this=12855623, dest=QUEUE.A
19:15:46,734 INFO  [TextMDB] TextMDB.onMessage, this=9399816
19:15:46,736 INFO  [TextMDB] TextMDB.onMessage, this=8153998
19:15:46,737 INFO  [TextMDB] TextMDB.onMessage, this=6598158
19:15:46,768 INFO  [TextMDB] TextMDB.sendReply, this=9399816, dest=QUEUE.A
19:15:46,768 INFO  [TextMDB] TextMDB.sendReply, this=6598158, dest=QUEUE.A
19:15:46,774 INFO  [TextMDB] TextMDB.sendReply, this=8153998, dest=QUEUE.A
19:15:46,903 INFO  [TextMDB] TextMDB.onMessage, this=10118602
19:15:46,904 INFO  [TextMDB] TextMDB.sendReply, this=10118602, dest=QUEUE.A
19:15:46,927 INFO  [TextMDB] TextMDB.onMessage, this=1792333
19:15:46,928 INFO  [TextMDB] TextMDB.sendReply, this=1792333, dest=QUEUE.A
19:15:47,002 INFO  [TextMDB] TextMDB.onMessage, this=14251014
19:15:47,007 INFO  [TextMDB] TextMDB.sendReply, this=14251014, dest=QUEUE.A
19:15:47,051 INFO  [TextMDB] TextMDB.onMessage, this=10775981
```

```
19:15:47,051 INFO  [TextMDB] TextMDB.sendReply, this=10775981, dest=QUEUE.A
19:15:47,060 INFO  [TextMDB] TextMDB.onMessage, this=6056676
19:15:47,061 INFO  [TextMDB] TextMDB.sendReply, this=6056676, dest=QUEUE.A
19:15:47,064 INFO  [TextMDB] TextMDB.onMessage, this=15679078
19:15:47,065 INFO  [TextMDB] TextMDB.sendReply, this=15679078, dest=QUEUE.A
```

Items of note in this example include:

- The JMS client has no explicit knowledge that it is dealing with an MDB. The client simply uses the standard JMS APIs to send messages to a queue and receive messages from another queue.
- The MDB declares whether it will listen to a queue or topic in the `ejb-jar.xml` descriptor. The name of the queue or topic must be specified using a `jboss.xml` descriptor. In this example the MDB also sends messages to a JMS queue. MDBs may act as queue senders or topic publishers within their `onMessage` callback.
- The messages received by the client include a "processed by: NNN" suffix, where NNN is the `hashCode` value of the MDB instance that processed the message. This shows that many MDBs may actively process messages posted to a destination. Concurrent processing is one of the benefits of MDBs.

2. JBoss MQ Overview

JBossMQ is composed of several services working together to provide JMS API level services to client applications. The services that make up the JBossMQ JMS implementation are introduced in this section.

2.1. Invocation Layer

The Invocation Layer (IL) services are responsible for handling the communication protocols that clients use to send and receive messages. JBossMQ can support running different types of Invocation Layers concurrently. All Invocation Layers support bidirectional communication which allows clients to send and receive messages concurrently. ILs only handle the transport details of messaging. They delegate messages to the JMS server JMX gateway service known as the invoker. This is similar to how the detached invokers expose the EJB container via different transports.

Each IL service binds a JMS connection factory to a specific location in the JNDI tree. Clients choose the protocol they wish to use by the JNDI location used to obtain the JMS connection factory. JBossMQ currently has several different invocation layers.

- **UIL2 IL**: The Unified Invocation Layer version 2(UIL2) is the preferred invocation layer for remote messaging. A multiplexing layer is used to provide bidirectional communication. The multiplexing layer creates two virtual sockets over one physical socket. This allows

communication with clients that cannot have a connection created from the server back to the client due to firewall or other restrictions. Unlike the older UIL invocation layer which used a blocking round-trip message at the socket level, the UIL2 protocol uses true asynchronous send and receive messaging at the transport level, providing for improved throughput and utilization.

- **JVM IL**: The Java Virtual Machine (JVM) Invocation Layer was developed to cut out the TCP/IP overhead when the JMS client is running in the same JVM as the server. This IL uses direct method calls for the server to service the client requests. This increases efficiency since no sockets are created and there is no need for the associated worker threads. This is the IL that should be used by Message Driven Beans (MDB) or any other component that runs in the same virtual machine as the server such as servlets, MBeans, or EJBs.
- **HTTP IL**: The HTTP Invocation Layer (HTTPIL) allows for accessing the JBossMQ service over the HTTP or HTTPS protocols. This IL relies on the servlet deployed in the `deploy/jms/jbossmq-httpil.sar` to handle the http traffic. This IL is useful for access to JMS through a firewall when the only port allowed requires HTTP.

2.2. Security Manager

The JBossMQ `SecurityManager` is the service that enforces an access control list to guard access to your destinations. This subsystem works closely with the `StateManager` service.

2.3. Destination Manager

The `DestinationManager` can be thought as being the central service in JBossMQ. It keeps track of all the destinations that have been created on the server. It also keeps track of the other key services such as the `MessageCache`, `StateManager`, and `PersistenceManager`.

2.4. Message Cache

Messages created in the server are passed to the `MessageCache` for memory management. JVM memory usage goes up as messages are added to a destination that does not have any receivers. These messages are held in the main memory until the receiver picks them up. If the `MessageCache` notices that the JVM memory usage starts passing the defined limits, the `MessageCache` starts moving those messages from memory to persistent storage on disk. The `MessageCache` uses a least recently used (LRU) algorithm to determine which messages should go to disk.

2.5. State Manager

The `StateManager` (SM) is in charge of keeping track of who is allowed to log into the server and what their durable subscriptions are.

2.6. Persistence Manager

The `PersistenceManager` (PM) is used by a destination to store messages marked as being

persistent. JBossMQ has several different implementations of the persistent manager, but only one can be enabled per server instance. You should enable the persistence manager that best matches your requirements.

- **JDBC2 persistence manager**: The JDBC2 persistence manager allows you to store persistent messages to a relational database using JDBC. The performance of this PM is directly related to the performance that can be obtained from the database. This PM has a very low memory overhead compared to the other persistence managers. Furthermore it is also highly integrated with the `MessageCache` to provide efficient persistence on a system that has a very active `MessageCache`.
- **Null Persistence Manager**: A wrapper persistence manager that can delegate to a real persistence manager. Configuration on the destinations decide whether persistence and caching is actually performed. The example configuration can be found in `docs/examples/jms`. To use the null persistence manager backed by a real persistence manager, you need to change the `ObjectName` of the real persistence manager and link the new name to the null persistence manager.

2.7. Destinations

A destination is the object on the JBossMQ server that clients use to send and receive messages. There are two types of destination objects, `Queues` and `Topics`. References to the destinations created by JBossMQ are stored in JNDI.

2.7.1. Queues

Clients that are in the point-to-point paradigm typically use queues. They expect that message sent to a queue will be receive by only one other client once and only once. If multiple clients are receiving messages from a single queue, the messages will be load balanced across the receivers. Queue objects, by default, will be stored under the JNDI `queue/` sub context.

2.7.2. Topics

Topics are used in the publish-subscribe paradigm. When a client publishes a message to a topic, he expects that a copy of the message will be delivered to each client that has subscribed to the topic. Topic messages are delivered in the same manner a television show is delivered. Unless you have the TV on and are watching the show, you will miss it. Similarly, if the client is not up, running and receiving messages from the topics, it will miss messages published to the topic. To get around this problem of missing messages, clients can start a durable subscription. This is like having a VCR record a show you cannot watch at its scheduled time so that you can see what you missed when you turn your TV back on.

3. JBoss MQ Configuration and MBeans

This section defines the MBean services that correspond to the components introduced in the previous section along with their MBean attributes. The configuration and service files that make up the JBossMQ system include:

- **deploy/hsqldb-jdbc-state-service.xml**: This configures the JDBC state service for storing state in the embedded Hypersonic database.
- **deploy/jms/hsqldb-jdbc2-service.xml**: This service descriptor configures the `DestinationManager`, `MessageCache`, and jdbc2 `PersistenceManager` for the embedded Hypersonic database.
- **deploy/jms/jbossmq-destinations-service.xml**: This service describes defines default JMS queue and topic destination configurations used by the testsuite unit tests. You can add/remove destinations to this file, or deploy another `*-service.xml` descriptor with the destination configurations.
- **jbossmq-httpil.sar**: This SAR file configures the HTTP invocation layer.
- **deploy/jms/jbossmq-service.xml**: This service descriptor configures the core JBossMQ MBeans like the `Invoker`, `SecurityManager`, `DynamicStateManager`, and core interceptor stack. It also defines the MDB default dead letter queue `DLQ`.
- **deploy/jms/jms-ds.xml**: This is a JCA connection factory and JMS provider MDB integration services configuration which sets JBossMQ as the JMS provider.
- **deploy/jms/jms-ra.rar**: This is a JCA resource adaptor for JMS providers.
- **deploy/jms/jvm-il-service.xml**: This service descriptor configures the `JVMServerILService` which provides the JVM IL transport.
- **deploy/jms/rmi-il-service.xml**: This service descriptor configures the `RMIServerILService` which provides the RMI IL. The queue and topic connection factory for this IL is bound under the name `RMIConnectionFactory`.
- **deploy/jms/uil2-service.xml**: This service descriptor configures the `UILServerILService` which provides the UIL2 transport. The queue and topic connection factory for this IL is bound under the name `UIL2ConnectionFactory` as well as `UILConnectionFactory` to replace the deprecated version 1 UIL service.

We will discuss the associated MBeans in the following subsections.

3.1. org.jboss.mq.il.jvm.JVMServerILService

The `org.jboss.mq.il.jvm.JVMServerILService` MBean is used to configure the JVM IL. The configurable attributes are as follows:

- **Invoker**: This attribute specifies JMX ObjectName of the JMS entry point service that is used to pass incoming requests to the JMS server. This is not something you would typically change from the `jboss.mq:service=Invoker` setting unless you change the entry point service.
- **ConnectionFactoryJNDIRef**: The JNDI location that this IL will bind a `ConnectionFactory`

setup to use this IL.

- **XAConnectionFactoryJNDIRef**: The JNDI location that this IL will bind a `XAConnectionFactory` setup to use this IL.
- **PingPeriod**: How often, in milliseconds, the client should send a ping message to the server to validate that the connection is still valid. If this is set to zero, then no ping message will be sent. Since it is impossible for JVM IL connection to go bad, it is recommended that you keep this set to 0.

3.2. org.jboss.mq.il.uil2.UILServerILService

The `org.jboss.mq.il.uil2.UILServerILService` is used to configure the UIL2 IL. The configurable attributes are as follows:

- **Invoker**: This attribute specifies JMX `ObjectName` of the JMS entry point service that is used to pass incoming requests to the JMS server. This is not something you would typically change from the `jboss.mq:service=Invoker` setting unless you change the entry point service.
- **ConnectionFactoryJNDIRef**: The JNDI location that this IL will bind a `ConnectionFactory` setup to use this IL.
- **XAConnectionFactoryJNDIRef**: The JNDI location that this IL will bind a `XAConnectionFactory` setup to use this IL.
- **PingPeriod**: How often, in milliseconds, the client should send a ping message to the server to validate that the connection is still valid. If this is set to zero, then no ping message will be sent.
- **ReadTimeout**: The period in milliseconds is passed onto as the `SoTimeout` value of the UIL2 socket. This allows detection of dead sockets that are not responsive and are not capable of receiving ping messages. Note that this setting should be longer in duration than the `PingPeriod` setting.
- **BufferSize**: The size in bytes used as the buffer over the basic socket streams. This corresponds to the `java.io.BufferedOutputStream` buffer size.
- **ChunkSize**: The size in bytes between stream listener notifications. The UIL2 layer uses the `org.jboss.util.stream.NotifyingBufferedOutputStream` and `NotifyingBufferedInputStream` implementations that support the notion of a heartbeat that is triggered based on data read/written to the stream. Whenever `ChunkSize` bytes are read/written to a stream. This allows serves as a ping or keepalive notification when large reads or writes require a duration greater than the `PingPeriod`.
- **ServerBindPort**: The protocol listening port for this IL. If not specified default is 0, which means that a random port will be chosen.

- **BindAddress**: The specific address this IL listens on. This can be used on a multi-homed host for a `java.net.ServerSocket` that will only accept connection requests on one of its addresses.
- **EnableTcpNoDelay**: `TcpNoDelay` causes TCP/IP packets to be sent as soon as the request is flushed. This may improve request response times. Otherwise request packets may be buffered by the operating system to create larger IP packets.
- **ServerSocketFactory**: The `javax.net.ServerSocketFactory` implementation class name to use to create the service `java.net.ServerSocket`. If not specified the default factory will be obtained from `javax.net.ServerSocketFactory.getDefault()`.
- **ClientAddress**: The address passed to the client as the address that should be used to connect to the server.
- **ClientSocketFactory**: The `javax.net.SocketFactory` implementation class name to use on the client. If not specified the default factory will be obtained from `javax.net.SocketFactory.getDefault()`.
- **SecurityDomain**: Specify the security domain name to use with JBoss SSL aware socket factories. This is the JNDI name of the security manager implementation as described for the `security-domain` element of the `jboss.xml` and `jboss-web.xml` descriptors.

3.2.1. Configuring UIL2 for SSL

The UIL2 service support the use of SSL through custom socket factories that integrate JSSE using the security domain associated with the IL service. An example UIL2 service descriptor fragment that illustrates the use of the custom JBoss SSL socket factories is shown in *Example 7.10, "An example UIL2 config fragment for using SSL"*.

```
<mbean code="org.jboss.mq.il.uil2.UILServerILService"
    name="jboss.mq:service=InvocationLayer,type=HTTPSUIL2">
    <depends
optional-attribute-name="Invoker">jboss.mq:service=Invoker</depends>
    <attribute
name="ConnectionFactoryJNDIRef">SSLConnectionFactory</attribute>
    <attribute
name="XAConnectionFactoryJNDIRef">SSLXAConnectionFactory</attribute>

    <!-- ... -->

    <!-- SSL Socket Factories -->
    <attribute name="ClientSocketFactory">
        org.jboss.security.ssl.ClientSocketFactory
    </attribute>
    <attribute name="ServerSocketFactory">
        org.jboss.security.ssl.DomainServerSocketFactory
    </attribute>
    <!-- Security domain - see below -->
    <attribute name="SecurityDomain">java:/jaas/SSL</attribute>
</mbean>
```

```
<!-- Configures the keystore on the "SSL" security domain
     This mbean is better placed in conf/jboss-service.xml where it
     can be used by other services, but it will work from anywhere.
     Use keytool from the sdk to create the keystore. -->

<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
       name="jboss.security:service=JaasSecurityDomain,domain=SSL">
    <!-- This must correlate with the java:/jaas/SSL above -->
    <constructor>
        <arg type="java.lang.String" value="SSL"/>
    </constructor>
    <!-- The location of the keystore resource: loads from the
         classpath and the server conf dir is a good default -->
    <attribute name="KeyStoreURL">resource:uil2.keystore</attribute>
    <attribute name="KeyStorePass">changeme</attribute>
</mbean>
```

Example 7.10. An example UIL2 config fragment for using SSL

3.2.2. JMS client properties for the UIL2 transport

There are several system properties that a JMS client using the UIL2 transport can set to control the client connection back to the server

- **org.jboss.mq.il.uil2.useServerHost**: This system property allows a client to connect to the server `InetAddress.getHostName` rather than the`InetAddress.getHostAddress` value. This will only make a difference if name resolution differs between the server and client environments.
- **org.jboss.mq.il.uil2.localAddr**: This system property allows a client to define the local interface to which its sockets should be bound.
- **org.jboss.mq.il.uil2.localPort**: This system property allows a client to define the local port to which its sockets should be bound
- **org.jboss.mq.il.uil2.serverAddr**: This system property allows a client to override the address to which it attempts to connect to. This is useful for networks where NAT is occurring between the client and JMS server.
- **org.jboss.mq.il.uil2.serverPort**: This system property allows a client to override the port to which it attempts to connect. This is useful for networks where port forwarding is occurring between the client and jms server.
- **org.jboss.mq.il.uil2.retryCount**: This system property controls the number of attempts to retry connecting to the JMS server. Retries are only made for `java.net.ConnectException`

failures. A value <= 0 means no retry attempts will be made.

- **org.jboss.mq.il.uil2.retryDelay**: This system property controls the delay in milliseconds between retries due to `ConnectException` failures.

3.3. org.jboss.mq.il.http.HTTPServerILService

The `org.jboss.mq.il.http.HTTPServerILService` is used to manage the HTTP/S IL. This IL allows for the use of the JMS service over HTTP or HTTPS connections. The relies on the servlet deployed in the `deploy/jms/jbossmq-httpil.sar` to handle the HTTP traffic. The configurable attributes are as follows:

- **TimeOut**: The default timeout in seconds that the client HTTP requests will wait for messages. This can be overridden on the client by setting the system property `org.jboss.mq.il.http.timeout` to the number of seconds.
- **RestInterval**: The number of seconds the client will sleep after each request. The default is 0, but you can set this value in conjunction with the `TimeOut` value to implement a pure timed based polling mechanism. For example, you could simply do a short lived request by setting the `TimeOut` value to 0 and then setting the `RestInterval` to 60. This would cause the client to send a single non-blocking request to the server, return any messages if available, then sleep for 60 seconds, before issuing another request. Like the `TimeOut` value, this can be explicitly overridden on a given client by specifying the `org.jboss.mq.il.http.restinterval` with the number of seconds you wish to wait between requests.
- **URL**: Set the servlet URL. This value takes precedence over any individual values set (i.e. the `URLPrefix`, `URLSuffix`, `URLPort`, etc.) It my be a actual URL or a property name which will be used on the client side to resolve the proper URL by calling `System.getProperty(propertyname)`. If not specified the URL will be formed from `URLPrefix + URLHostName + ":" + URLPort + "/" + URLSuffix`.
- **URLPrefix**: The prefix portion of the servlet URL.
- **URLHostName**: The hostname portion of the servlet URL.
- **URLPort**: The port portion of the URL.
- **URLSuffix**: The trailing path portion of the URL.
- **UseHostName**: A flag that if set to true the default setting for the `URLHostName` attribute will be taken from `InetAddress.getLocalHost().getHostName()`. If false the default setting for the `URLHostName` attribute will be taken from `InetAddress.getLocalHost().getHostAddress()`.

3.4. org.jboss.mq.server.jmx.Invoker

The `org.jboss.mq.server.jmx.Invoker` is used to pass IL requests down to the destination manager service through an interceptor stack. The configurable attributes are as follows:

- **NextInterceptor**: The JMX `ObjectName` of the next request interceptor. This attribute is used by all the interceptors to create the interceptor stack. The last interceptor in the chain should be the `DestinationManager`.

3.5. org.jboss.mq.server.jmx.InterceptorLoader

The `org.jboss.mq.server.jmx.InterceptorLoader` is used to load a generic interceptor and make it part of the interceptor stack. This MBean is typically used to load custom interceptors like `org.jboss.mq.server.TracingInterceptor`, which is can be used to efficiently log all client requests via trace level log messages. The configurable attributes are as follows:

- **NextInterceptor**: The JMX `ObjectName` of the next request interceptor. This attribute is used by all the interceptors to create the interceptor stack. The last interceptor in the chain should be the `DestinationManager`. This attribute should be setup via a `<depends optional-attribute-name="NextInterceptor">` XML tag.
- **InterceptorClass**: The class name of the interceptor that will be loaded and made part of the interceptor stack. This class specified here must extend the `org.jboss.mq.server.JMSServerInterceptor` class.

3.6. org.jboss.mq.sm.jdbc.JDBCStateManager

The `JDBCStateManager` MBean is used as the default state manager assigned to the DestinationManager service. It stores user and durable subscriber information in the database. The configurable attributes are as follows:

- **ConnectionManager**: This is the `ObjectName` of the datasource that the JDBC state manager will write to. For Hypersonic, it is `jboss.jca:service=DataSourceBinding,name=DefaultDS`.
- **SqlProperties**: The `SqlProperties` define the SQL statements to be used to persist JMS state data. If the underlying database is changed, the SQL statements used may need to change.

3.7. org.jboss.mq.security.SecurityManager

If the `org.jboss.mq.security.SecurityManager` is part of the interceptor stack, then it will enforce the access control lists assigned to the destinations. The `SecurityManager` uses JAAS, and as such requires that at application policy be setup for in the JBoss `login-config.xml` file. The default configuration is shown below.

```
<application-policy name="jbossmq">
    <authentication>
        <login-module
code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
                      flag="required">
            <module-option
name="unauthenticatedIdentity">guest</module-option>
            <module-option name="dsJndiName">java:/DefaultDS</module-option>
            <module-option name="principalsQuery">SELECT PASSWD FROM
JMS_USERS
                WHERE USERID=?</module-option>
            <module-option name="rolesQuery">SELECT ROLEID, 'Roles' FROM
                JMS_ROLES WHERE USERID=?</module-option>
        </login-module>
    </authentication>
</application-policy>
```

The configurable attributes of the SecurityManager are as follows:

- **NextInterceptor**: The JMX `ObjectName` of the next request interceptor. This attribute is used by all the interceptors to create the interceptor stack. The last interceptor in the chain should be the `DestinationManager`.
- **SecurityDomain**: Specify the security domain name to use for authentication and role based authorization. This is the JNDI name of the JAAS domain to be used to perform authentication and authorization against.
- **DefaultSecurityConfig**: This element specifies the default security configuration settings for destinations. This applies to temporary queues and topics as well as queues and topics that do not specifically specify a security configuration. The `DefaultSecurityConfig` should declare some number of `role` elements which represent each role that is allowed access to a destination. Each `role` should have the following attributes:
  - **name**: The `name` attribute defines the name of the role.
  - **create**: The `create` attribute is a true/false value that indicates whether the role has the ability to create durable subscriptions on the topic.
  - **read**: The `read` attribute is a true/false value that indicates whether the role can receive messages from the destination.
  - **write**: The `write` attribute is a true/false value that indicates whether the role can send messages to the destination.

3.8. org.jboss.mq.server.jmx.DestinationManager

The `org.jboss.mq.server.jmx.DestinationManager` must be the last interceptor in the interceptor stack. The configurable attributes are as follows:

- **PersistenceManager**: The JMX `ObjectName` of the persistence manager service the server should use.
- **StateManager**: The JMX `ObjectName` of the state manager service the server should use.
- **MessageCache**: The JMX `ObjectName` of the message cache service the server should use.

Additional read-only attributes and operations that support monitoring include:

- **ClientCount**: The number of clients connected to the server.
- **Clients**: A `java.util.Map<org.jboss.mq.ConnectionToken, org.jboss.mq.server.ClientConsumer>` instances for the clients connected to the server.
- **MessageCounter**: An array of org.jboss.mq.server.MessageCounter instances that provide statistics for a JMS destination.
- **listMessageCounter()**: This operation generates an HTML table that contains:
  - **Type**: Either `Queue` or `Topic` indicating the destination type.
  - **Name**: The name of the destination.
  - **Subscription**: The subscription ID for a topic.
  - **Durable**: A boolean indicating if the topic subscription is durable.
  - **Count**: The number of message delivered to the destination.
  - **CountDelta**: The change in message count since the previous access of count.
  - **Depth**: The number of messages in the destination.
  - **DepthDelta**: The change in the number of messages in the destination since the previous access of depth.
  - **Last Add**: The date/time string in `DateFormat.SHORT`/`DateFormat.MEDIUM` format of the last time a message was added to the destination.
- **resetMessageCounter()**: This zeros all destination counts and last added times.

Queues and topics can be created and destroyed at runtime through the `DestinationManager` MBean. The `DestinationManager` provides `createQueue` and `createTopic` operations for this. Both methods have a one argument version which takes the destination name and a two argument version which takes the destination and the JNDI name of the destination. Queues and topics can be removed using the `destroyQueue` and `destroyTopic` operations, both of which take a destination name is input.

3.9. org.jboss.mq.server.MessageCache

The server determines when to move messages to secondary storage by using the `org.jboss.mq.server.MessageCache` MBean. The configurable attributes are as follows:

- **CacheStore**: The JMX `ObjectName` of the service that will act as the cache store. The cache store is used by the `MessageCache` to move messages to persistent storage. The value you set here typically depends on the type of persistence manager you are using.
- **HighMemoryMark**: The amount of JVM heap memory in megabytes that must be reached before the `MessageCache` starts to move messages to secondary storage.
- **MaxMemoryMark**: The maximum amount of JVM heap memory in megabytes that the `MessageCache` considers to be the max memory mark. As memory usage approaches the max memory mark, the `MessageCache` will move messages to persistent storage so that the number of messages kept in memory approaches zero.
- **MakeSoftReferences**: This controls whether or not the message cache will keep soft references to messages that need to be removed. The default is true.
- **MinimumHard**: The minimum number of the in memory cache. JBoss won't try to go below this number of messages in the cache. The default value is 1.
- **MaximumHard**: The upper bound on the number of hard references to messages in the cache. JBoss will soften messages to reduce the number of hard references to this level. A value of 0 means that there is no size based upper bound. The default is 0.
- **SoftenWaitMillis**: The maximum wait time before checking whether messages need softening. The default is 1000 milliseconds (1 second).
- **SoftenNoMoreOftenThanMillis**: The minimum amount of time between checks to soften messages. A value of 0 means that this check should be skipped. The default is 0 milliseconds.
- **SoftenAtLeastEveryMillis**: The maximum amount of time between checks to soften messages. A value of 0 means that this check should be skipped. The default is 0.

Additional read-only cache attribute that provide statistics include:

- **CacheHits**: The number of times a hard referenced message was accessed
- **CacheMisses**: The number of times a softened message was accessed.
- **HardRefCacheSize**: The number of messages in the cache that are not softened.
- **SoftRefCacheSize**: The number of messages that are currently softened.
- **SoftenedSize**: The total number of messages softened since the last boot.
- **TotalCacheSize**: The total number of messages that are being managed by the cache.

3.10. org.jboss.mq.pm.jdbc2.PersistenceManager

The `org.jboss.mq.pm.jdbc.PersistenceManager` should be used as the persistence manager assigned to the `DestinationManager` if you wish to store messages in a database. This PM has been tested against the HypersonSQL, MS SQL, Oracle, MySQL and Postgres databases. The configurable attributes are as follows:

- **MessageCache**: The JMX `ObjectName` of the `MessageCache` that has been assigned to the `DestinationManager`..
- **ConnectionManager**: The JMX `ObjectName` of the JCA data source that will be used to obtain JDBC connections.
- **ConnectionRetryAttempts**: An integer count used to allow the PM to retry attempts at getting a connection to the JDBC store. There is a 1500 millisecond delay between each connection failed connection attempt and the next attempt. This must be greater than or equal to 1 and defaults to 5.
- **SqlProperties**: A property list is used to define the SQL Queries and other JDBC2 Persistence Manager options. You will need to adjust these properties if you which to run against another database other than Hypersonic. *Example 7.11, "Default JDBC2 PersistenceManager SqlProperties"* shows default setting for this attribute for the Hypersonic database.

```
<attribute name="SqlProperties">
      CREATE_TABLES_ON_STARTUP = TRUE
      CREATE_USER_TABLE = CREATE TABLE JMS_USERS \
           (USERID VARCHAR(32) NOT NULL, PASSWD VARCHAR(32) NOT NULL, \
           CLIENTID VARCHAR(128), PRIMARY KEY(USERID))
      CREATE_ROLE_TABLE = CREATE TABLE JMS_ROLES \
           (ROLEID VARCHAR(32) NOT NULL, USERID VARCHAR(32) NOT NULL, \
                    PRIMARY KEY(USERID, ROLEID))
      CREATE_SUBSCRIPTION_TABLE = CREATE TABLE JMS_SUBSCRIPTIONS \
           (CLIENTID VARCHAR(128) NOT NULL, \
           SUBNAME VARCHAR(128) NOT NULL, TOPIC VARCHAR(255) NOT NULL, \
           SELECTOR VARCHAR(255), PRIMARY KEY(CLIENTID, SUBNAME))
      GET_SUBSCRIPTION = SELECT TOPIC, SELECTOR FROM JMS_SUBSCRIPTIONS \
           WHERE CLIENTID=? AND SUBNAME=?
      LOCK_SUBSCRIPTION = SELECT TOPIC, SELECTOR FROM JMS_SUBSCRIPTIONS \
           WHERE CLIENTID=? AND SUBNAME=?
      GET_SUBSCRIPTIONS_FOR_TOPIC =
           SELECT CLIENTID, SUBNAME, SELECTOR FROM JMS_SUBSCRIPTIONS WHERE
TOPIC=?
      INSERT_SUBSCRIPTION = \
           INSERT INTO JMS_SUBSCRIPTIONS (CLIENTID, SUBNAME, TOPIC,
SELECTOR) VALUES(?,?,?,?)
      UPDATE_SUBSCRIPTION = \
           UPDATE JMS_SUBSCRIPTIONS SET TOPIC=?, SELECTOR=? WHERE CLIENTID=?
AND SUBNAME=?
      REMOVE_SUBSCRIPTION = DELETE FROM JMS_SUBSCRIPTIONS WHERE CLIENTID=?
AND SUBNAME=?
```

```
      GET_USER_BY_CLIENTID = SELECT USERID, PASSWD, CLIENTID FROM JMS_USERS
WHERE CLIENTID=?
      GET_USER = SELECT PASSWD, CLIENTID FROM JMS_USERS WHERE USERID=?
      POPULATE.TABLES.01 = INSERT INTO JMS_USERS (USERID, PASSWD) \
                    VALUES ('guest', 'guest')
      POPULATE.TABLES.02 = INSERT INTO JMS_USERS (USERID, PASSWD) \
                    VALUES ('j2ee', 'j2ee')
      POPULATE.TABLES.03 = INSERT INTO JMS_USERS (USERID, PASSWD, CLIENTID)
\
                    VALUES ('john', 'needle', 'DurableSubscriberExample')
      POPULATE.TABLES.04 = INSERT INTO JMS_USERS (USERID, PASSWD) \
                    VALUES ('nobody', 'nobody')
      POPULATE.TABLES.05 = INSERT INTO JMS_USERS (USERID, PASSWD) \
                    VALUES ('dynsub', 'dynsub')
      POPULATE.TABLES.06 = INSERT INTO JMS_ROLES (ROLEID, USERID) \
                    VALUES ('guest','guest')
      POPULATE.TABLES.07 = INSERT INTO JMS_ROLES (ROLEID, USERID) \
                    VALUES ('j2ee','guest')
      POPULATE.TABLES.08 = INSERT INTO JMS_ROLES (ROLEID, USERID) \
                    VALUES ('john','guest')
      POPULATE.TABLES.09 = INSERT INTO JMS_ROLES (ROLEID, USERID) \
                    VALUES ('subscriber','john')
      POPULATE.TABLES.10 = INSERT INTO JMS_ROLES (ROLEID, USERID) \
                    VALUES ('publisher','john')
      POPULATE.TABLES.11 = INSERT INTO JMS_ROLES (ROLEID, USERID) \
                    VALUES ('publisher','dynsub')
      POPULATE.TABLES.12 = INSERT INTO JMS_ROLES (ROLEID, USERID) \
                    VALUES ('durpublisher','john')
      POPULATE.TABLES.13 = INSERT INTO JMS_ROLES (ROLEID, USERID) \
                    VALUES ('durpublisher','dynsub')
      POPULATE.TABLES.14 = INSERT INTO JMS_ROLES (ROLEID, USERID) \
                    VALUES ('noacc','nobody')
</attribute>
```

Example 7.11. Default JDBC2 PersistenceManager SqlProperties

Example 7.12, "A sample JDBC2 PersistenceManager SqlProperties for Oracle" shows an alternate setting for Oracle.

```
<attribute name="SqlProperties">
      BLOB_TYPE=BINARYSTREAM_BLOB
      INSERT_TX = INSERT INTO JMS_TRANSACTIONS (TXID) values(?)
      INSERT_MESSAGE = \
          INSERT INTO JMS_MESSAGES (MESSAGEID, DESTINATION, MESSAGEBLOB,
TXID, TXOP) \
          VALUES(?,?,?,?,?)
      SELECT_ALL_UNCOMMITED_TXS = SELECT TXID FROM JMS_TRANSACTIONS
      SELECT_MAX_TX = SELECT MAX(TXID) FROM JMS_MESSAGES
      SELECT_MESSAGES_IN_DEST = \
          SELECT MESSAGEID, MESSAGEBLOB FROM JMS_MESSAGES WHERE
DESTINATION=?
```

```
      SELECT_MESSAGE = \
          SELECT MESSAGEID, MESSAGEBLOB FROM JMS_MESSAGES WHERE MESSAGEID=?
AND DESTINATION=?
      MARK_MESSAGE = \
          UPDATE JMS_MESSAGES SET TXID=?, TXOP=? WHERE MESSAGEID=? AND
DESTINATION=?
      UPDATE_MESSAGE = \
          UPDATE JMS_MESSAGES SET MESSAGEBLOB=? WHERE MESSAGEID=? AND
DESTINATION=?
      UPDATE_MARKED_MESSAGES = UPDATE JMS_MESSAGES SET TXID=?, TXOP=? WHERE
TXOP=?
      UPDATE_MARKED_MESSAGES_WITH_TX = \
          UPDATE JMS_MESSAGES SET TXID=?, TXOP=? WHERE TXOP=? AND TXID=?
      DELETE_MARKED_MESSAGES_WITH_TX = \
          DELETE FROM JMS_MESSAGES MESS WHERE TXOP=:1 AND EXISTS \
          (SELECT TXID FROM JMS_TRANSACTIONS TX WHERE TX.TXID = MESS.TXID)
      DELETE_TX = DELETE FROM JMS_TRANSACTIONS WHERE TXID = ?
      DELETE_MARKED_MESSAGES = DELETE FROM JMS_MESSAGES WHERE TXID=? AND
TXOP=?
      DELETE_TEMPORARY_MESSAGES = DELETE FROM JMS_MESSAGES WHERE TXOP='T'
      DELETE_MESSAGE = DELETE FROM JMS_MESSAGES WHERE MESSAGEID=? AND
DESTINATION=?
      CREATE_MESSAGE_TABLE = CREATE TABLE JMS_MESSAGES ( MESSAGEID INTEGER
NOT NULL, \
         DESTINATION VARCHAR(255) NOT NULL, TXID INTEGER, TXOP CHAR(1), \
         MESSAGEBLOB BLOB, PRIMARY KEY (MESSAGEID, DESTINATION) )
      CREATE_IDX_MESSAGE_TXOP_TXID = \
         CREATE INDEX JMS_MESSAGES_TXOP_TXID ON JMS_MESSAGES (TXOP, TXID)
      CREATE_IDX_MESSAGE_DESTINATION = \
         CREATE INDEX JMS_MESSAGES_DESTINATION ON JMS_MESSAGES (DESTINATION)
      CREATE_TX_TABLE = CREATE TABLE JMS_TRANSACTIONS ( TXID INTEGER,
PRIMARY KEY (TXID) )
      CREATE_TABLES_ON_STARTUP = TRUE
</attribute>
```

Example 7.12. A sample JDBC2 PersistenceManager SqlProperties for Oracle

Additional examples can be found in the `docs/examples/jms` directory of the distribution.

3.11. Destination MBeans

This section describes the destination MBeans used in the `jbossmq-destinations-service.xml` and `jbossmq-service.xml` descriptors.

3.11.1. org.jboss.mq.server.jmx.Queue

The `Queue` is used to define a queue destination in JBoss. The following shows the configuration of one of the default JBoss queues.

```
<mbean code="org.jboss.mq.server.jmx.Queue"
       name="jboss.mq.destination:service=Queue,name=testQueue">
    <depends optional-attribute-name="DestinationManager">
        jboss.mq:service=DestinationManager
    </depends>
    <depends optional-attribute-name="SecurityManager">
        jboss.mq:service=SecurityManager
    </depends>
    <attribute name="MessageCounterHistoryDayLimit">-1</attribute>
    <attribute name="SecurityConf">
        <security>
            <role name="guest"     read="true"  write="true"/>
            <role name="publisher" read="true"  write="true"
create="false"/>
            <role name="noacc"     read="false" write="false"
create="false"/>
        </security>
    </attribute>
</mbean>
```

The `name` attribute of the JMX object name of this MBean is used to determine the destination name. For example. In the case of the queue we just looked at, the name of the queue is `testQueue`. The configurable attributes are as follows:

- **DestinationManager**: The JMX ObjectName of the destination manager service for the server. This attribute should be set via a `<depends optional-attribute-name="DestinationManager">` XML tag.
- **SecurityManager**: The JMX `ObjectName` of the security manager service that is being used to validate client requests.
- **SecurityConf**: This element specifies a XML fragment which describes the access control list to be used by the `SecurityManager` to authorize client operations against the destination. The content model is the same as for the `SecurityManagerSecurityConf` attribute.
- **JNDIName**: The location in JNDI to which the queue object will be bound. If this is not set it will be bound under the `queue` context using the name of the queue. For the `testQueue` shown above, the JNDI name would be `queue/testQueue`.
- **MaxDepth**: The `MaxDepth` is an upper limit to the backlog of messages that can exist for a destination. If exceeded, attempts to add new messages will result in a `org.jboss.mq.DestinationFullException`. The `MaxDepth` can still be exceeded in a number of situations, e.g. when a message is placed back into the queue. Also transactions performing read committed processing, look at the current size of queue, ignoring any messages that may be added as a result of the current transaction or other transactions. This is because we don't want the transaction to fail during the commit phase when the message is physically added to the queue.
- **MessageCounterHistoryDayLimit**: Sets the destination message counter history day limit

with a value less than 0 indicating unlimited history, a 0 value disabling history and a value greater than 0 giving the history days count.

Additional read-only attributes that provide statistics information include:

- **MessageCounter**: An array of `org.jboss.mq.server.MessageCounter` instances that provide statistics for this destination.
- **QueueDepth**: The current backlog of waiting messages.
- **ReceiversCount**: The number of receivers currently associated with the queue.
- **ScheduledMessageCount**: The number of messages waiting in the queue for their scheduled delivery time to arrive.

The following are some of the operations available on queues.

- **listMessageCounter()**: This operation generates an HTML table that contains the same data we as the `listMessageCounter` operation on the `DestinationManager`, but only for this one queue.
- **resetMessageCounter()**: This zeros all destination counts and last added times.
- **listMessageCounterHistory()**: This operation display an HTML table showing the hourly message counts per hour for each day in the history.
- **resetMessageCounterHistory()**: This operation resets the day history message counts.
- **removeAllMessages()**: This method removes all the messages on the queue.

3.11.2. org.jboss.mq.server.jmx.Topic

The `org.jboss.mq.server.jmx.Topic` is used to define a topic destination in JBoss. The following shows the configuration of one of the default JBoss topics.

```
<mbean code="org.jboss.mq.server.jmx.Topic"
       name="jboss.mq.destination:service=Topic,name=testTopic">
    <depends optional-attribute-name="DestinationManager">
        jboss.mq:service=DestinationManager
    </depends>
    <depends optional-attribute-name="SecurityManager">
        jboss.mq:service=SecurityManager
    </depends>
    <attribute name="SecurityConf">
        <security>
            <role name="guest"        read="true" write="true" />
            <role name="publisher"    read="true" write="true"
create="false" />
            <role name="durpublisher" read="true" write="true" create="true"
/>
```

```
        </security>
    </attribute>
</mbean>
```

The `name` attribute of the JMX object name of this MBean is used to determine the destination name. For example, in the case of the topic we just looked at, the name of the topic is `testTopic`. The configurable attributes are as follows:

- **DestinationManager**: The JMX object name of the destination manager configured for the server.
- **SecurityManager**: The JMX object name of the security manager that is being used to validate client requests.
- **SecurityConf**: This element specifies a XML fragment which describes the access control list to be used by the `SecurityManager` to authorize client operations against the destination. The content model is the same as that for the `SecurityManagerSecurityConf` attribute.
- **JNDIName**: The location in JNDI to which the topic object will be bound. If this is not set it will be bound under the `topic` context using the name of the queue. For the `testTopic` shown above, the JNDI name would be `topic/testTopic`.
- **MaxDepth**: The `MaxDepth` is an upper limit to the backlog of messages that can exist for a destination, and if exceeded, attempts to add new messages will result in a `org.jboss.mq.DestinationFullException`. The `MaxDepth` can still be exceeded in a number of situations, e.g. when a message is knacked back into the queue. Also transactions performing read committed processing, look at the current size of queue, ignoring any messages that may be added as a result of the current transaction or other transactions. This is because we don't want the transaction to fail during the commit phase when the message is physically added to the topic.
- **MessageCounterHistoryDayLimit**: Sets the destination message counter history day limit with a value < 0 indicating unlimited history, a 0 value disabling history, and a value > 0 giving the history days count.

Additional read-only attributes that provide statistics information include:

- **AllMessageCount**: The message count across all queue types associated with the topic.
- **AllSubscriptionsCount**: The count of durable and non-durable subscriptions.
- **DurableMessageCount**: The count of messages in durable subscription queues.
- d **DurableSubscriptionsCount**: The count of durable subscribers.
- **MessageCounter**: An array of `org.jboss.mq.server.MessageCounter` instances that

provide statistics for this destination.

- **NonDurableMessageCount**: The count on messages in non-durable subscription queues.
- **NonDurableSubscriptionsCount**: The count of non-durable subscribers.

The following are some of the operations available on topics.

- **listMessageCounter()**: This operation generates an HTML table that contains the same data we as the `listMessageCounter` operation on the `DestinationManager`, but only for this one topic. Message counters are only maintained for each active subscription, durable or otherwise.
- **resetMessageCounter()**: This zeros all destination counts and last added times.
- **listMessageCounterHistory()**: This operation display an HTML table showing the hourly message counts per hour for each day of history.
- **resetMessageCounterHistory()**: This operation resets the day history message counts.

4. Specifying the MDB JMS Provider

Up to this point we have looked at the standard JMS client/server architecture. The JMS specification defines an advanced set of interfaces that allow for concurrent processing of a destination's messages, and collectively this functionality is referred to as application server facilities (ASF). Two of the interfaces that support concurrent message processing, `javax.jms.ServerSessionPool` and `javax.jms.ServerSession`, must be provided by the application server in which the processing will occur. Thus, the set of components that make up the JBossMQ ASF involves both JBossMQ components as well as JBoss server components. The JBoss server MDB container utilizes the JMS service's ASF to concurrently process messages sent to MDBs.

The responsibilities of the ASF domains are well defined by the JMS specification and so we won't go into a discussion of how the ASF components are implemented. Rather, we want to discuss how ASF components used by the JBoss MDB layer are integrated using MBeans that allow either the application server interfaces, or the JMS provider interfaces to be replaced with alternate implementations.

Let's start with the `org.jboss.jms.jndi.JMSProviderLoader` MBean. This MBean is responsible for loading an instance of the `org.jboss.jms.jndi.JMSProviderAdaptor` interface into the JBoss server and binding it into JNDI. The `JMSProviderAdaptor` interface is an abstraction that defines how to get the root JNDI context for the JMS provider, and an interface for getting and setting the JNDI names for the `Context.PROVIDER_URL` for the root `InitialContext`, and the `QueueConnectionFactory` and `TopicConnectionFactory` locations in the root context. This is all that is really necessary to bootstrap use of a JMS provider. By abstracting this information into an interface, alternate JMS ASF provider implementations can be used with the JBoss MDB container. The `org.jboss.jms.jndi.JBossMQProvider` is the

default implementation of `JMSProviderAdaptor` interface, and provides the adaptor for the JBossMQ JMS provider. To replace the JBossMQ provider with an alternate JMS ASF implementation, simply create an implementation of the `JMSProviderAdaptor` interface and configure the JMSProviderLoader with the class name of the implementation. We'll see an example of this in the configuration section.

In addition to being able to replace the JMS provider used for MDBs, you can also replace the javax.jms.ServerSessionPool interface implementation. This is possible by configuring the class name of the `org.jboss.jms.asf.ServerSessionPoolFactory` implementation using the `org.jboss.jms.asf.ServerSessionPoolLoader` MBean `PoolFactoryClass` attribute. The default `ServerSessionPoolFactory` factory implementation is the JBoss `org.jboss.jms.asf.StdServerSessionPoolFactory` class.

4.1. org.jboss.jms.jndi.JMSProviderLoader MBean

The `JMSProviderLoader` MBean service creates a JMS provider adaptor and binds it into JNDI. A JMS provider adaptor is a class that implements the `org.jboss.jms.jndi.JMSProviderAdapter` interface. It is used by the message driven bean container to access a JMS service provider in a provider independent manner. The configurable attributes of the `JMSProviderLoader` service are:

- **ProviderName**: A unique name for the JMS provider. This will be used to bind the `JMSProviderAdapter` instance into JNDI under `java:/<ProviderName>` unless overridden by the `AdapterJNDIName` attribute.
- **ProviderAdapterClass**: The fully qualified class name of the org.jboss.jms.jndi.JMSProviderAdapter interface to create an instance of.
- **FactoryRef**: The JNDI name under which the provider `javax.jms.ConnectionFactory` will be bound.
- **QueueFactoryRef**: The JNDI name under which the provider `javax.jms.QueueConnectionFactory` will be bound.
- **TopicFactoryRef**: The JNDI name under which the `javax.jms.TopicConnectionFactory` will be bound.
- **Properties**: The JNDI properties of the initial context used to look up the factories.

```
<mbean code="org.jboss.jms.jndi.JMSProviderLoader"
       name="jboss.mq:service=JMSProviderLoader,name=RemoteJBossMQProvider">
    <attribute name="ProviderName">RemoteJMSProvider</attribute>
    <attribute name="ProviderUrl"></attribute>
    <attribute name="ProviderAdapterClass">
        org.jboss.jms.jndi.JBossMQProvider
    </attribute>
    <attribute name="FactoryRef">XAConnectionFactory</attribute>
    <attribute name="QueueFactoryRef">XAConnectionFactory</attribute>
    <attribute name="TopicFactoryRef">XAConnectionFactory</attribute>
```

```
    <attribute name="Properties>
        java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
        java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
        java.naming.provider.url=jnp://remotehost:1099
    </attribute>
</mbean>
```

Example 7.13. A JMSProviderLoader for accessing a remote JBossMQ server

The RemoteJMSProvider can be referenced on the MDB invoker config as shown in the `jboss.xml` fragment given in *Example 7.14, " A jboss.xml fragment for specifying the MDB JMS provider adaptor"*.

```
<proxy-factory-config>
    <JMSProviderAdapterJNDI>RemoteJMSProvider</JMSProviderAdapterJNDI>
    <ServerSessionPoolFactoryJNDI>StdJMSPool</ServerSessionPoolFactoryJNDI>
    <MaximumSize>15</MaximumSize>
    <MaxMessages>1</MaxMessages>
    <MDBConfig>
        <ReconnectIntervalSec>10</ReconnectIntervalSec>
        <DLQConfig>
            <DestinationQueue>queue/DLQ</DestinationQueue>
            <MaxTimesRedelivered>10</MaxTimesRedelivered>
            <TimeToLive>0</TimeToLive>
        </DLQConfig>
    </MDBConfig>
</proxy-factory-config>
```

Example 7.14. A jboss.xml fragment for specifying the MDB JMS provider adaptor

Incidentally, because one can specify multiple `invoker-proxy-binding` elements, this allows an MDB to listen to the same queue/topic on multiple servers by configuring multiple bindings with different `JMSProviderAdapterJNDI` settings.

Alternatively, one can integrate the JMS provider using JCA configuration like that shown in *Example 7.15, "A jms-ds.xml descriptor for integrating a JMS provider adaptor via JCA"*.

```
<tx-connection-factory>
    <jndi-name>RemoteJmsXA</jndi-name>
    <xa-transaction/>
    <adapter-display-name>JMS Adapter</adapter-display-name>
    <config-property name="JMSProviderAdapterJNDI"
```

```
type="java.lang.String">RemoteJMSProvider</config-property>
    <config-property name="SessionDefaultType"
type="java.lang.String">javax.jms.Topic</config-property>

<security-domain-and-application>JmsXARealm</security-domain-and-application>
</tx-connection-factory>
```

Example 7.15. A jms-ds.xml descriptor for integrating a JMS provider adaptor via JCA

4.2. org.jboss.jms.asf.ServerSessionPoolLoader MBean

The `ServerSessionPoolLoader` MBean service manages a factory for `javax.jms.ServerSessionPool` objects used by the message driven bean container. The configurable attributes of the `ServerSessionPoolLoader` service are:

- **PoolName**: A unique name for the session pool. This will be used to bind the `ServerSessionPoolFactory` instance into JNDI under `java:/PoolName`.
- **PoolFactoryClass**: The fully qualified class name of the `org.jboss.jms.asf.ServerSessionPoolFactory` interface to create an instance of.
- **XidFactory**: The JMX `ObjectName` of the service to use for generating `javax.transaction.xa.Xid` values for local transactions when two phase commit is not required. The `XidFactory` MBean must provide an `Instance` operation which returns a `org.jboss.tm.XidFactoryMBean` instance.

4.3. Integrating non-JBoss JMS Providers

We have mentioned that one can replace the JBossMQ JMS implementation with a foreign implementation. Here we summarize the various approaches one can take to do the replacement:

- Replace the JMSProviderLoader JBossMQProvider class with one that instantiates the correct JNDI context for communicating with the foreign JMS providers managed objects.
- Use the `ExternalContext` MBean to federate the foreign JMS providers managed objects into the JBoss JNDI tree.
- Use MBeans to instantiate the foreign JMS objects into the JBoss JNDI tree. An example of this approach can be found for Websphere MQ at *http://wiki.jboss.org/wiki/Wiki.jsp?page=IntegrationWithWebSphereMQSeries*.

Security on JBoss

J2EE Security Configuration and Architecture

Security is a fundamental part of any enterprise application. You need to be able to restrict who is allowed to access your applications and control what operations application users may perform. The J2EE specifications define a simple role-based security model for EJBs and web components. The JBoss component framework that handles security is the JBossSX extension framework. The JBossSX security extension provides support for both the role-based declarative J2EE security model and integration of custom security via a security proxy layer. The default implementation of the declarative security model is based on Java Authentication and Authorization Service (JAAS) login modules and subjects. The security proxy layer allows custom security that cannot be described using the declarative model to be added to an EJB in a way that is independent of the EJB business object. Before getting into the JBoss security implementation details, we will review EJB and servlet specification security models, as well as JAAS to establish the foundation for these details.

1. J2EE Declarative Security Overview

The J2EE security model declarative in that you describe the security roles and permissions in a standard XML descriptor rather than embedding security into your business component. This isolates security from business-level code because security tends to be more a function of where the component is deployed than an inherent aspect of the component's business logic. For example, consider an ATM component that is to be used to access a bank account. The security requirements, roles and permissions will vary independently of how you access the bank account, based on what bank is managing the account, where the ATM is located, and so on.

Securing a J2EE application is based on the specification of the application security requirements via the standard J2EE deployment descriptors. You secure access to EJBs and web components in an enterprise application by using the `ejb-jar.xml` and `web.xml` deployment descriptors. The following sections look at the purpose and usage of the various security elements.

1.1. Security References

Both EJBs and servlets can declare one or more `security-role-ref` elements as shown in *Figure 8.1, "The security-role-ref element"*. This element declares that a component is using the `role-name` value as an argument to the `isCallerInRole(String)` method. By using the `isCallerInRole` method, a component can verify whether the caller is in a role that has been declared with a `security-role-ref/role-name` element. The `role-name` element value must link to a `security-role` element through the `role-link` element. The typical use of `isCallerInRole` is to perform a security check that cannot be defined by using the role-based `method-permissions` elements.

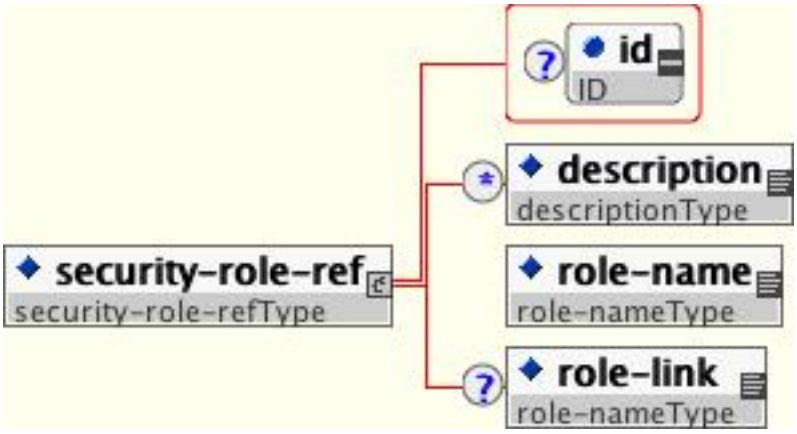


Figure 8.1. The security-role-ref element

Example 8.1, "An ejb-jar.xml descriptor fragment that illustrates the security-role-ref element usage." shows the use of `security-role-ref` in an `ejb-jar.xml`.

```
<!-- A sample ejb-jar.xml fragment -->
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>ASessionBean</ejb-name>
      ...
      <security-role-ref>
          <role-name>TheRoleICheck</role-name>
          <role-link>TheApplicationRole</role-link>
      </security-role-ref>
    </session>
  </enterprise-beans>
  ...
</ejb-jar>
```

Example 8.1. An ejb-jar.xml descriptor fragment that illustrates the security-role-ref element usage.

Example 8.2, "An example web.xml descriptor fragment that illustrates the security-role-ref element usage." shows the use of `security-role-ref` in a `web.xml`.

```
<web-app>
    <servlet>
        <servlet-name>AServlet</servlet-name>
        ...
        <security-role-ref>
            <role-name>TheServletRole</role-name>
            <role-link>TheApplicationRole</role-link>
```

```
        </security-role-ref>
    </servlet>
    ...
</web-app>
```

Example 8.2. An example web.xml descriptor fragment that illustrates the security-role-ref element usage.

1.2. Security Identity

An EJB has the capability to specify what identity an EJB should use when it invokes methods on other components using the `security-identity` element, shown in *Figure 8.2, "The security-identity element"*

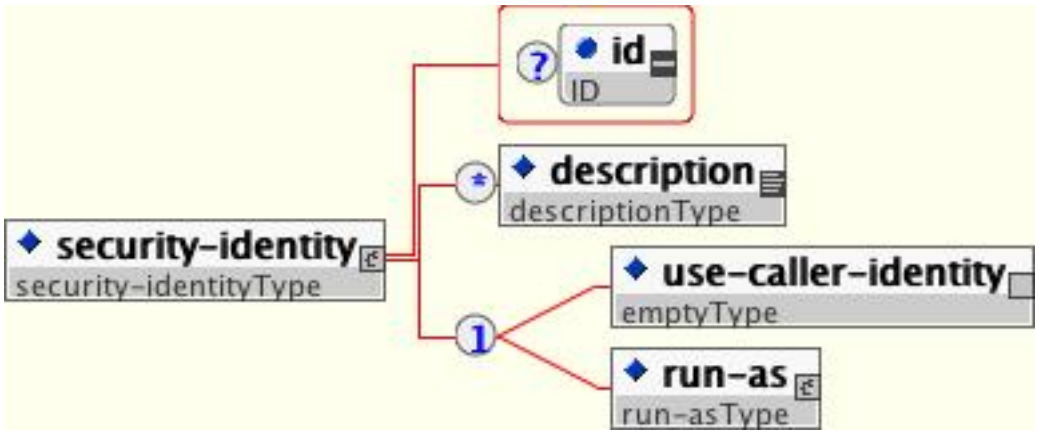


Figure 8.2. The security-identity element

The invocation identity can be that of the current caller, or it can be a specific role. The application assembler uses the `security-identity` element with a `use-caller-identity` child element to indicate that the current caller's identity should be propagated as the security identity for method invocations made by the EJB. Propagation of the caller's identity is the default used in the absence of an explicit `security-identity` element declaration.

Alternatively, the application assembler can use the `run-as/role-name` child element to specify that a specific security role given by the `role-name` value should be used as the security identity for method invocations made by the EJB. Note that this does not change the caller's identity as seen by the `EJBContext.getCallerPrincipal()` method. Rather, the caller's security roles are set to the single role specified by the `run-as/role-name` element value. One use case for the `run-as` element is to prevent external clients from accessing internal EJBs. You accomplish this by assigning the internal EJB `method-permission` elements that restrict access to a role never assigned to an external client. EJBs that need to use internal EJB are then configured

with a `run-as/role-name` equal to the restricted role. The following descriptor fragment that illustrates `security-identity` element usage.

```
<!-- A sample ejb-jar.xml fragment -->
<ejb-jar>
    <enterprise-beans>
        <session>
            <ejb-name>ASessionBean</ejb-name>
            <!-- ... -->
            <security-identity>
                <use-caller-identity/>
            </security-identity>
        </session>
        <session>
            <ejb-name>RunAsBean</ejb-name>
            <!-- ... -->
            <security-identity>
                <run-as>
                    <description>A private internal role</description>
                    <role-name>InternalRole</role-name>
                </run-as>
            </security-identity>
        </session>
    </enterprise-beans>
    <!-- ... -->
</ejb-jar>
```

When you use `run-as` to assign a specific role to outgoing calls, JBoss associates a principal named `anonymous`. If you want another principal to be associated with the call, you need to associate a `run-as-principal` with the bean in the `jboss.xml` file. The following fragment associates a principal named `internal` with `RunAsBean` from the prior example.

```
<session>
    <ejb-name>RunAsBean</ejb-name>
    <security-identity>
        <run-as-principal>internal</run-as-principal>
    </security-identity>
</session>
```

The `run-as` element is also available in servlet definitions in a `web.xml` file. The following example shows how to assign the role `InternalRole` to a servlet:

```
<servlet>
    <servlet-name>AServlet</servlet-name>
    <!-- ... -->
    <run-as>
        <role-name>InternalRole</role-name>
    </run-as>
</servlet>
```

Calls from this servlet will be associated with the anonymous `principal`. The

`run-as-principal` element is available in the `jboss-web.xml` file to assign a specific principal to go along with the `run-as` role. The following fragment shows how to associate a principal named `internal` to the servlet in the prior example.

```
<servlet>
    <servlet-name>AServlet</servlet-name>
    <run-as-principal>internal</run-as-principal>
</servlet>
```

1.3. Security roles

The security role name referenced by either the `security-role-ref` or `security-identity` element needs to map to one of the application's declared roles. An application assembler defines logical security roles by declaring `security-role` elements. The `role-name` value is a logical application role name like Administrator, Architect, SalesManager, etc.

The J2EE specifications note that it is important to keep in mind that the security roles in the deployment descriptor are used to define the logical security view of an application. Roles defined in the J2EE deployment descriptors should not be confused with the user groups, users, principals, and other concepts that exist in the target enterprise's operational environment. The deployment descriptor roles are application constructs with application domain-specific names. For example, a banking application might use role names such as BankManager, Teller, or Customer.

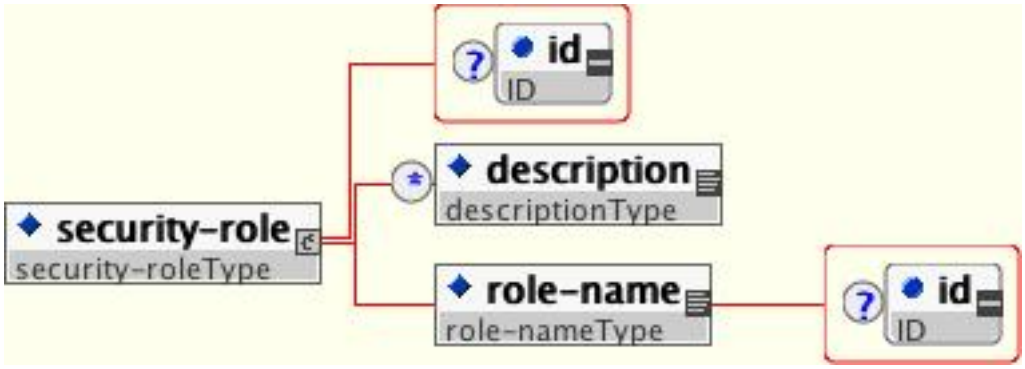


Figure 8.3. The security-role element

In JBoss, a `security-role` element is only used to map `security-role-ref/role-name` values to the logical role that the component role references. The user's assigned roles are a dynamic function of the application's security manager, as you will see when we discuss the JBossSX implementation details. JBoss does not require the definition of `security-role` elements in order to declare method permissions. However, the specification of `security-role` elements is still a recommended practice to ensure portability across application servers and for deployment descriptor maintenance. *Example 8.3, "An ejb-jar.xml descriptor fragment that illustrates the security-role element usage."* shows the usage of the `security-role` in an `ejb-jar.xml` file.

```
<!-- A sample ejb-jar.xml fragment -->
<ejb-jar>
    <!-- ... -->
    <assembly-descriptor>
        <security-role>
            <description>The single application role</description>
            <role-name>TheApplicationRole</role-name>
        </security-role>
    </assembly-descriptor>
</ejb-jar>
```

Example 8.3. An ejb-jar.xml descriptor fragment that illustrates the security-role element usage.

Example 8.4, "An example web.xml descriptor fragment that illustrates the security-role element usage." shows the usage of the `security-role` in an `web.xml` file.

```
<!-- A sample web.xml fragment -->
<web-app>
    <!-- ... -->
    <security-role>
        <description>The single application role</description>
        <role-name>TheApplicationRole</role-name>
    </security-role>
</web-app>
```

Example 8.4. An example web.xml descriptor fragment that illustrates the security-role element usage.

1.4. EJB method permissions

An application assembler can set the roles that are allowed to invoke an EJB's home and remote interface methods through method-permission element declarations.

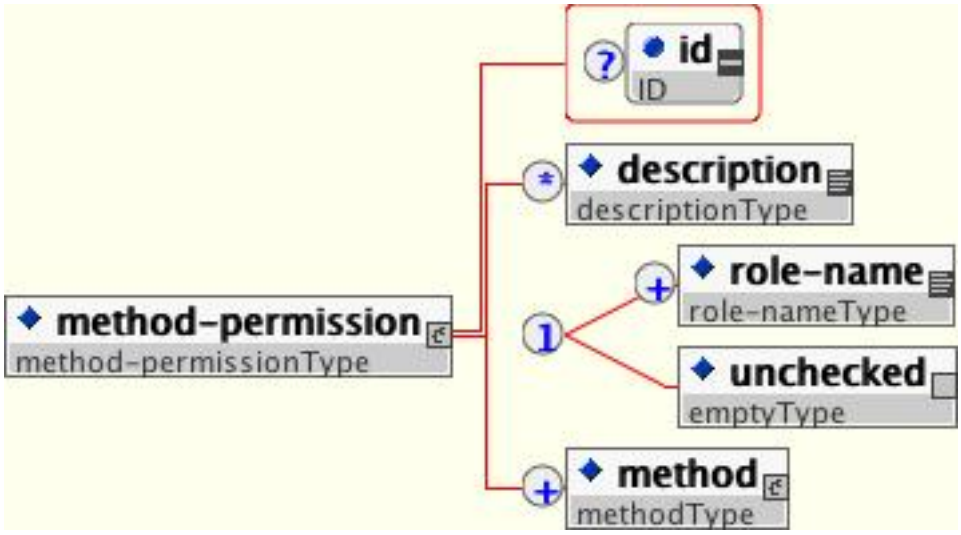


Figure 8.4. The method-permissions element

Each `method-permission` element contains one or more role-name child elements that define the logical roles that are allowed to access the EJB methods as identified by method child elements. You can also specify an `unchecked` element instead of the `role-name` element to declare that any authenticated user can access the methods identified by method child elements. In addition, you can declare that no one should have access to a method that has the `exclude-list` element. If an EJB has methods that have not been declared as accessible by a role using a `method-permission` element, the EJB methods default to being excluded from use. This is equivalent to defaulting the methods into the `exclude-list`.

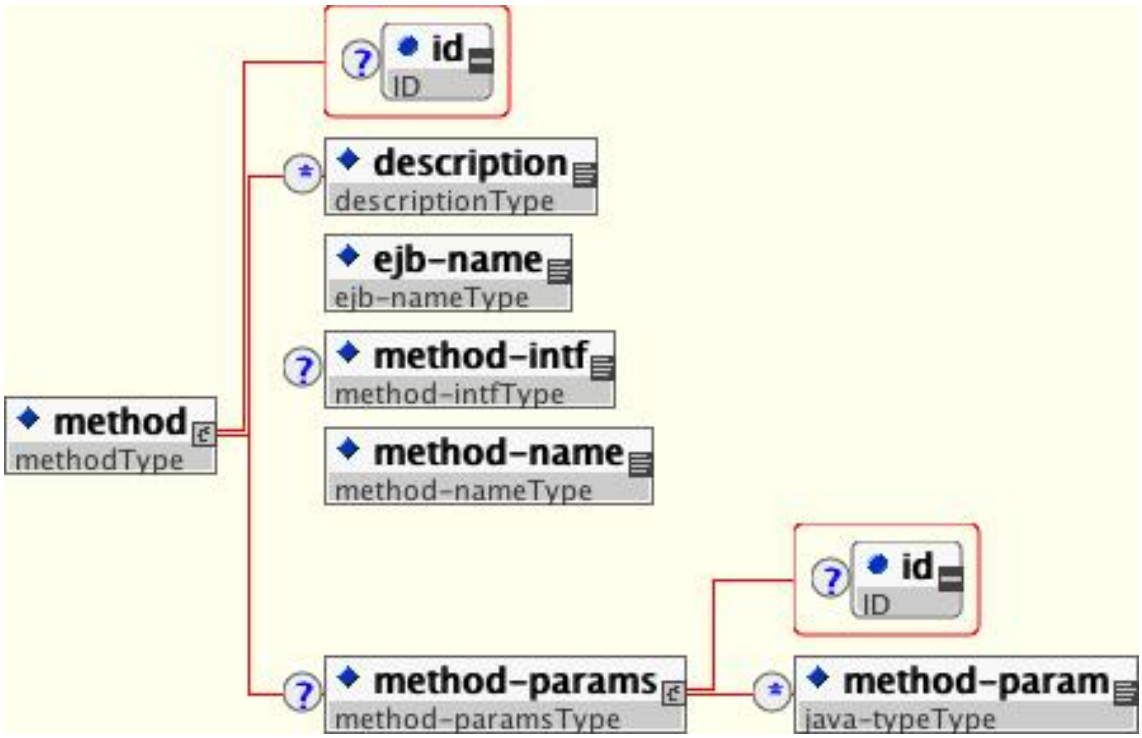


Figure 8.5. The method element

There are three supported styles of method element declarations.

The first is used for referring to all the home and component interface methods of the named enterprise bean:

```
<method>
    <ejb-name>EJBNAME</ejb-name>
    <method-name>*</method-name>
</method>
```

The second style is used for referring to a specified method of the home or component interface of the named enterprise bean:

```
<method>
    <ejb-name>EJBNAME</ejb-name>
    <method-name>METHOD</method-name>
                </method>
```

If there are multiple methods with the same overloaded name, this style refers to all of the overloaded methods.

The third style is used to refer to a specified method within a set of methods with an overloaded name:

```
<method>
    <ejb-name>EJBNAME</ejb-name>
    <method-name>METHOD</method-name>
    <method-params>
        <method-param>PARAMETER_1</method-param>
        <!-- ... -->
        <method-param>PARAMETER_N</method-param>
    </method-params>
</method>
```

The method must be defined in the specified enterprise bean's home or remote interface. The method-param element values are the fully qualified name of the corresponding method parameter type. If there are multiple methods with the same overloaded signature, the permission applies to all of the matching overloaded methods.

The optional `method-intf` element can be used to differentiate methods with the same name and signature that are defined in both the home and remote interfaces of an enterprise bean.

Example 8.5, “An ejb-jar.xml descriptor fragment that illustrates the method-permission element usage.” provides complete examples of the `method-permission` element usage.

```
<ejb-jar>
    <assembly-descriptor>
        <method-permission>
            <description>The employee and temp-employee roles may access any
                method of the EmployeeService bean </description>
            <role-name>employee</role-name>
            <role-name>temp-employee</role-name>
            <method>
                <ejb-name>EmployeeService</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <method-permission>
            <description>The employee role may access the findByPrimaryKey,
                getEmployeeInfo, and the updateEmployeeInfo(String) method
of
                the AardvarkPayroll bean </description>
            <role-name>employee</role-name>
            <method>
                <ejb-name>AardvarkPayroll</ejb-name>
                <method-name>findByPrimaryKey</method-name>
            </method>
            <method>
                <ejb-name>AardvarkPayroll</ejb-name>
                <method-name>getEmployeeInfo</method-name>
            </method>
            <method>
                <ejb-name>AardvarkPayroll</ejb-name>
                <method-name>updateEmployeeInfo</method-name>
                <method-params>
                    <method-param>java.lang.String</method-param>
                </method-params>
            </method>
        </method-permission>
        <method-permission>
            <description>The admin role may access any method of the
                EmployeeServiceAdmin bean </description>
            <role-name>admin</role-name>
            <method>
                <ejb-name>EmployeeServiceAdmin</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <method-permission>
            <description>Any authenticated user may access any method of the
                EmployeeServiceHelp bean</description>
            <unchecked/>
            <method>
                <ejb-name>EmployeeServiceHelp</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <exclude-list>
            <description>No fireTheCTO methods of the EmployeeFiring bean
may be
                used in this deployment</description>
```

```
            <method>
                <ejb-name>EmployeeFiring</ejb-name>
                <method-name>fireTheCTO</method-name>
            </method>
        </exclude-list>
    </assembly-descriptor>
</ejb-jar>
```

Example 8.5. An ejb-jar.xml descriptor fragment that illustrates the method-permission element usage.

1.5. Web Content Security Constraints

In a web application, security is defined by the roles that are allowed access to content by a URL pattern that identifies the protected content. This set of information is declared by using the `web.xml` `security-constraint` element.

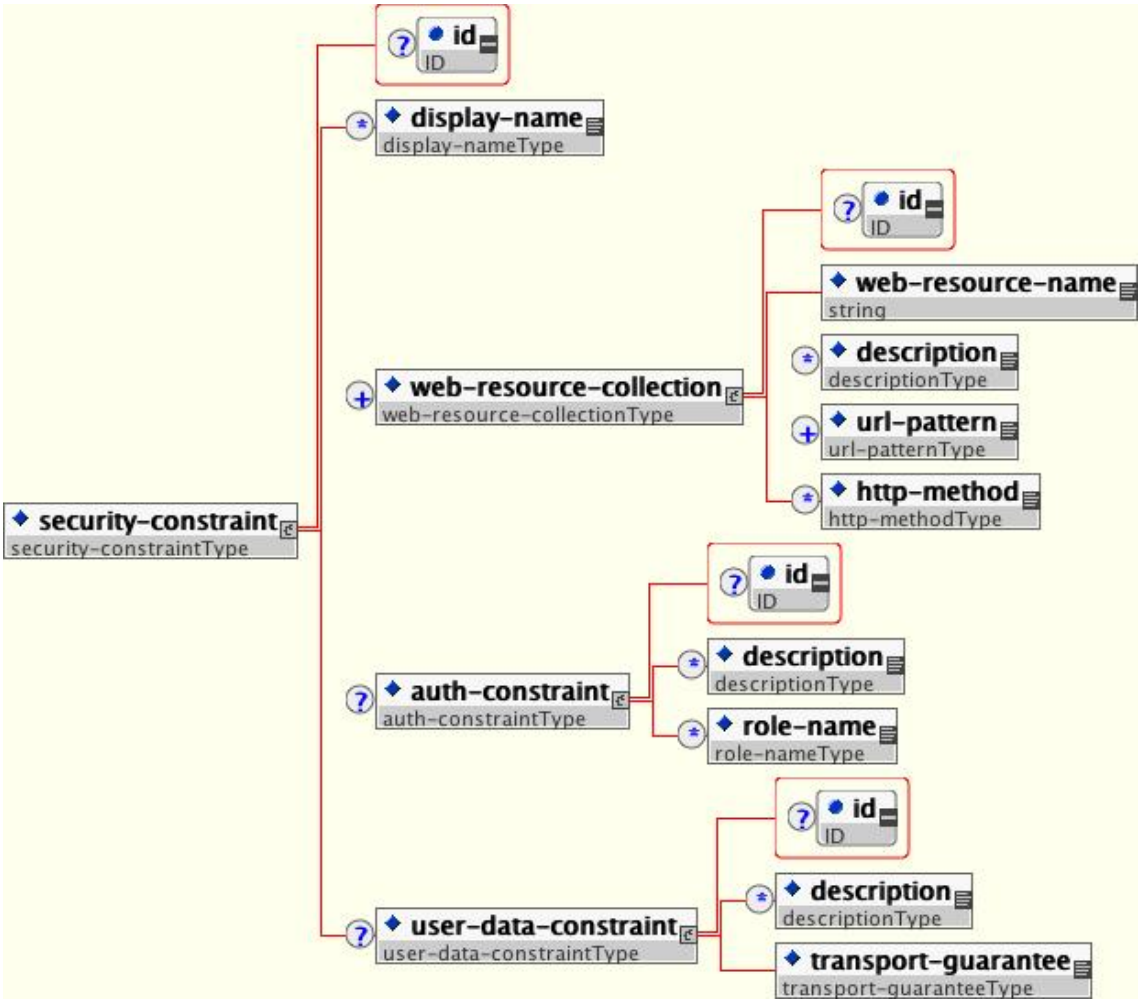


Figure 8.6. The security-constraint element

The content to be secured is declared using one or more `web-resource-collection` elements. Each `web-resource-collection` element contains an optional series of `url-pattern` elements followed by an optional series of `http-method` elements. The `url-pattern` element value specifies a URL pattern against which a request URL must match for the request to correspond to an attempt to access secured content. The `http-method` element value specifies a type of HTTP request to allow.

The optional `user-data-constraint` element specifies the requirements for the transport layer of the client to server connection. The requirement may be for content integrity (preventing data tampering in the communication process) or for confidentiality (preventing reading while in transit). The transport-guarantee element value specifies the degree to which communication between the client and server should be protected. Its values are `NONE`, `INTEGRAL`, and `CONFIDENTIAL`. A value of `NONE` means that the application does not require any transport guarantees. A value of `INTEGRAL` means that the application requires the data sent between the client and server to be sent in such a way that it can't be changed in transit. A value of `CONFIDENTIAL` means that the application requires the data to be transmitted in a fashion that prevents other entities from observing the contents of the transmission. In most cases, the presence of the `INTEGRAL` or `CONFIDENTIAL` flag indicates that the use of SSL is required.

The optional `login-config` element is used to configure the authentication method that should be used, the realm name that should be used for rhw application, and the attributes that are needed by the form login mechanism.

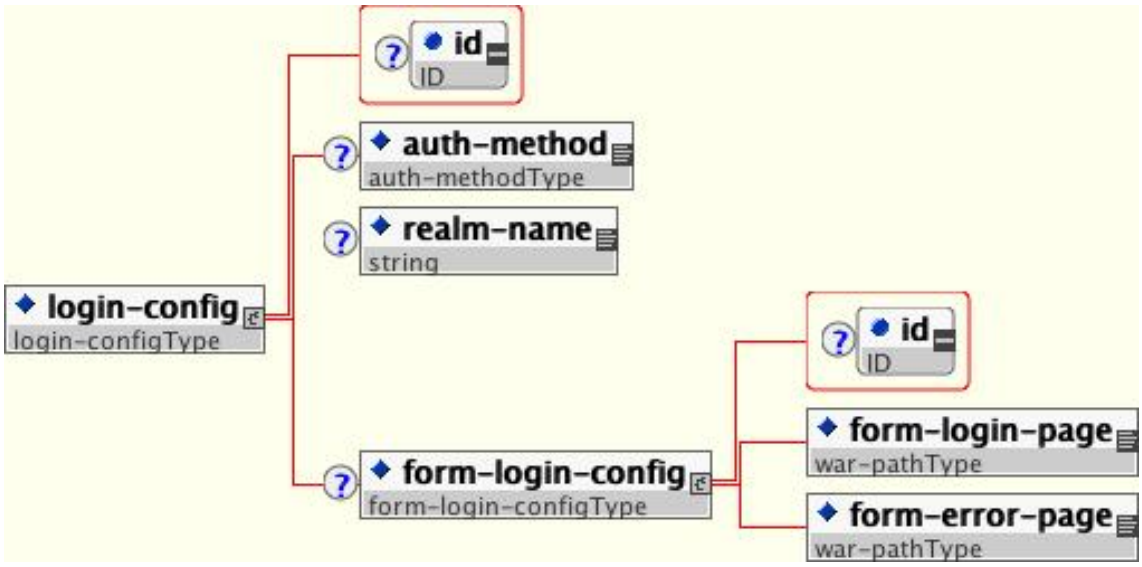


Figure 8.7. The login-config element

The `auth-method` child element specifies the authentication mechanism for the web application. As a prerequisite to gaining access to any web resources that are protected by an authorization

constraint, a user must have authenticated using the configured mechanism. Legal `auth-method` values are `BASIC`, `DIGEST`, `FORM`, and `CLIENT-CERT`. The `realm-name` child element specifies the realm name to use in HTTP basic and digest authorization. The `form-login-config` child element specifies the log in as well as error pages that should be used in form-based login. If the `auth-method` value is not `FORM`, then `form-login-config` and its child elements are ignored.

As an example, the `web.xml` descriptor fragment given in *Example 8.6, " A web.xml descriptor fragment which illustrates the use of the security-constraint and related elements."* indicates that any URL lying under the web application's `/restricted` path requires an `AuthorizedUser` role. There is no required transport guarantee and the authentication method used for obtaining the user identity is BASIC HTTP authentication.

```
<web-app>
    <!-- ... -->
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Secure Content</web-resource-name>
            <url-pattern>/restricted/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>AuthorizedUser</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>NONE</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
    <!-- ... -->
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>The Restricted Zone</realm-name>
    </login-config>
    <!-- ... -->
    <security-role>
        <description>The role required to access restricted content
</description>
        <role-name>AuthorizedUser</role-name>
    </security-role>
</web-app>
```

Example 8.6. A web.xml descriptor fragment which illustrates the use of the security-constraint and related elements.

1.6. Enabling Declarative Security in JBoss

The J2EE security elements that have been covered so far describe the security requirements only from the application's perspective. Because J2EE security elements declare logical roles, the application deployer maps the roles from the application domain onto the deployment environment. The J2EE specifications omit these application server-specific details. In JBoss,

mapping the application roles onto the deployment environment entails specifying a security manager that implements the J2EE security model using JBoss server specific deployment descriptors. The details behind the security configuration are discussed in *Section 3, “The JBoss Security Model”*.

2. An Introduction to JAAS

The JBossSX framework is based on the JAAS API. It is important that you understand the basic elements of the JAAS API to understand the implementation details of JBossSX. The following sections provide an introduction to JAAS to prepare you for the JBossSX architecture discussion later in this chapter.

2.1. What is JAAS?

The JAAS 1.0 API consists of a set of Java packages designed for user authentication and authorization. It implements a Java version of the standard Pluggable Authentication Module (PAM) framework and compatibly extends the Java 2 Platform's access control architecture to support user-based authorization. JAAS was first released as an extension package for JDK 1.3 and is bundled with JDK 1.4+. Because the JBossSX framework uses only the authentication capabilities of JAAS to implement the declarative role-based J2EE security model, this introduction focuses on only that topic.

JAAS authentication is performed in a pluggable fashion. This permits Java applications to remain independent from underlying authentication technologies and allows the JBossSX security manager to work in different security infrastructures. Integration with a security infrastructure can be achieved without changing the JBossSX security manager implementation. All that needs to change is the configuration of the authentication stack that JAAS uses.

2.1.1. The JAAS Core Classes

The JAAS core classes can be broken down into three categories: common, authentication, and authorization. The following list presents only the common and authentication classes because these are the specific classes used to implement the functionality of JBossSX covered in this chapter.

The are the common classes:

- `Subject` (`javax.security.auth.Subject`)
- `Principal` (`java.security.Principal`)

These are the authentication classes:

- `Callback` (`javax.security.auth.callback.Callback`)
- `CallbackHandler` (`javax.security.auth.callback.CallbackHandler`)
- `Configuration` (`javax.security.auth.login.Configuration`)
- `LoginContext` (`javax.security.auth.login.LoginContext`)

- `LoginModule` (`javax.security.auth.spi.LoginModule`)

2.1.1.1. The Subject and Principal Classes

To authorize access to resources, applications first need to authenticate the request's source. The JAAS framework defines the term subject to represent a request's source. The `Subject` class is the central class in JAAS. A `Subject` represents information for a single entity, such as a person or service. It encompasses the entity's principals, public credentials, and private credentials. The JAAS APIs use the existing Java 2 `java.security.Principal` interface to represent a principal, which is essentially just a typed name.

During the authentication process, a subject is populated with associated identities, or principals. A subject may have many principals. For example, a person may have a name principal (John Doe), a social security number principal (123-45-6789), and a username principal (johnd), all of which help distinguish the subject from other subjects. To retrieve the principals associated with a subject, two methods are available:

```
public Set getPrincipals() {...}
public Set getPrincipals(Class c) {...}
```

The first method returns all principals contained in the subject. The second method returns only those principals that are instances of class `c` or one of its subclasses. An empty set is returned if the subject has no matching principals. Note that the `java.security.acl.Group` interface is a subinterface of `java.security.Principal`, so an instance in the principals set may represent a logical grouping of other principals or groups of principals.

2.1.1.2. Authentication of a Subject

Authentication of a subject requires a JAAS login. The login procedure consists of the following steps:

1. An application instantiates a `LoginContext` and passes in the name of the login configuration and a `CallbackHandler` to populate the `Callback` objects, as required by the configuration `LoginModules`.
2. The `LoginContext` consults a `Configuration` to load all the `LoginModules` included in the named login configuration. If no such named configuration exists the `other` configuration is used as a default.
3. The application invokes the `LoginContext.login` method.
4. The login method invokes all the loaded `LoginModules`. As each `LoginModule` attempts to authenticate the subject, it invokes the handle method on the associated `CallbackHandler` to obtain the information required for the authentication process. The required information is passed to the handle method in the form of an array of `Callback` objects. Upon success, the `LoginModules` associate relevant principals and credentials with the subject.

5. The `LoginContext` returns the authentication status to the application. Success is represented by a return from the login method. Failure is represented through a LoginException being thrown by the login method.

6. If authentication succeeds, the application retrieves the authenticated subject using the `LoginContext.getSubject` method.

7. After the scope of the subject authentication is complete, all principals and related information associated with the subject by the login method can be removed by invoking the `LoginContext.logout` method.

The `LoginContext` class provides the basic methods for authenticating subjects and offers a way to develop an application that is independent of the underlying authentication technology. The `LoginContext` consults a `Configuration` to determine the authentication services configured for a particular application. `LoginModule` classes represent the authentication services. Therefore, you can plug different login modules into an application without changing the application itself. The following code shows the steps required by an application to authenticate a subject.

```
CallbackHandler handler = new MyHandler();
LoginContext lc = new LoginContext("some-config", handler);

try {
    lc.login();
    Subject subject = lc.getSubject();
} catch(LoginException e) {
    System.out.println("authentication failed");
    e.printStackTrace();
}

// Perform work as authenticated Subject
// ...

// Scope of work complete, logout to remove authentication info
try {
    lc.logout();
} catch(LoginException e) {
    System.out.println("logout failed");
    e.printStackTrace();
}

// A sample MyHandler class
class MyHandler
    implements CallbackHandler
{
    public void handle(Callback[] callbacks) throws
        IOException, UnsupportedCallbackException
    {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof NameCallback) {
                NameCallback nc = (NameCallback)callbacks[i];
                nc.setName(username);
            } else if (callbacks[i] instanceof PasswordCallback) {
                PasswordCallback pc = (PasswordCallback)callbacks[i];
```

```
                pc.setPassword(password);
            } else {
                throw new UnsupportedCallbackException(callbacks[i],
                                                       "Unrecognized
Callback");
            }
        }
    }
}
```

Developers integrate with an authentication technology by creating an implementation of the `LoginModule` interface. This allows an administrator to plug different authentication technologies into an application. You can chain together multiple `LoginModule`s to allow for more than one authentication technology to participate in the authentication process. For example, one `LoginModule` may perform username/password-based authentication, while another may interface to hardware devices such as smart card readers or biometric authenticators.

The life cycle of a `LoginModule` is driven by the `LoginContext` object against which the client creates and issues the login method. The process consists of two phases. The steps of the process are as follows:

- The `LoginContext` creates each configured `LoginModule` using its public no-arg constructor.
- Each `LoginModule` is initialized with a call to its initialize method. The `Subject` argument is guaranteed to be non-null. The signature of the initialize method is: `public void initialize(Subject subject, CallbackHandler callbackHandler, Map sharedState, Map options)`.
- The `login` method is called to start the authentication process. For example, a method implementation might prompt the user for a username and password and then verify the information against data stored in a naming service such as NIS or LDAP. Alternative implementations might interface to smart cards and biometric devices, or simply extract user information from the underlying operating system. The validation of user identity by each `LoginModule` is considered phase 1 of JAAS authentication. The signature of the `login` method is `boolean login() throws LoginException`. A `LoginException` indicates failure. A return value of true indicates that the method succeeded, whereas a return valueof false indicates that the login module should be ignored.
- If the `LoginContext`'s overall authentication succeeds, `commit` is invoked on each `LoginModule`. If phase 1 succeeds for a `LoginModule`, then the commit method continues with phase 2 and associates the relevant principals, public credentials, and/or private credentials with the subject. If phase 1 fails for a `LoginModule`, then `commit` removes any previously stored authentication state, such as usernames or passwords. The signature of the `commit` method is: `boolean commit() throws LoginException`. Failure to complete the commit phase is indicated by throwing a `LoginException`. A return of true indicates that the method succeeded, whereas a return of false indicates that the login module should be

ignored.

- If the `LoginContext`'s overall authentication fails, then the `abort` method is invoked on each `LoginModule`. The `abort` method removes or destroys any authentication state created by the login or initialize methods. The signature of the `abort` method is `boolean abort() throws LoginException`. Failure to complete the `abort` phase is indicated by throwing a `LoginException`. A return of true indicates that the method succeeded, whereas a return of false indicates that the login module should be ignored.
- To remove the authentication state after a successful login, the application invokes `logout` on the `LoginContext`. This in turn results in a `logout` method invocation on each `LoginModule`. The `logout` method removes the principals and credentials originally associated with the subject during the `commit` operation. Credentials should be destroyed upon removal. The signature of the `logout` method is: `boolean logout() throws LoginException`. Failure to complete the logout process is indicated by throwing a `LoginException`. A return of true indicates that the method succeeded, whereas a return of false indicates that the login module should be ignored.

When a `LoginModule` must communicate with the user to obtain authentication information, it uses a `CallbackHandler` object. Applications implement the `CallbackHandler` interface and pass it to the LoginContext, which forwards it directly to the underlying login modules. Login modules use the `CallbackHandler` both to gather input from users, such as a password or smart card PIN, and to supply information to users, such as status information. By allowing the application to specify the `CallbackHandler`, underlying `LoginModule`s remain independent from the different ways applications interact with users. For example, a `CallbackHandler`'s implementation for a GUI application might display a window to solicit user input. On the other hand, a `callbackhandler`'s implementation for a non-GUI environment, such as an application server, might simply obtain credential information by using an application server API. The `callbackhandler` interface has one method to implement:

```
void handle(Callback[] callbacks)
    throws java.io.IOException,
           UnsupportedCallbackException;
```

The `Callback` interface is the last authentication class we will look at. This is a tagging interface for which several default implementations are provided, including the `NameCallback` and `PasswordCallback` used in an earlier example. A `LoginModule` uses a `Callback` to request information required by the authentication mechanism. `LoginModule`s pass an array of `Callback`s directly to the `CallbackHandler.handle` method during the authentication's login phase. If a `callbackhandler` does not understand how to use a `Callback` object passed into the handle method, it throws an `UnsupportedCallbackException` to abort the login call.

3. The JBoss Security Model

Similar to the rest of the JBoss architecture, security at the lowest level is defined as a set of interfaces for which alternate implementations may be provided. Three basic interfaces define

the JBoss server security layer: `org.jboss.security.AuthenticationManager`, `org.jboss.security.RealmMapping`, and `org.jboss.security.SecurityProxy`. *Figure 8.8, "The key security model interfaces and their relationship to the JBoss server EJB container elements."* shows a class diagram of the security interfaces and their relationship to the EJB container architecture.

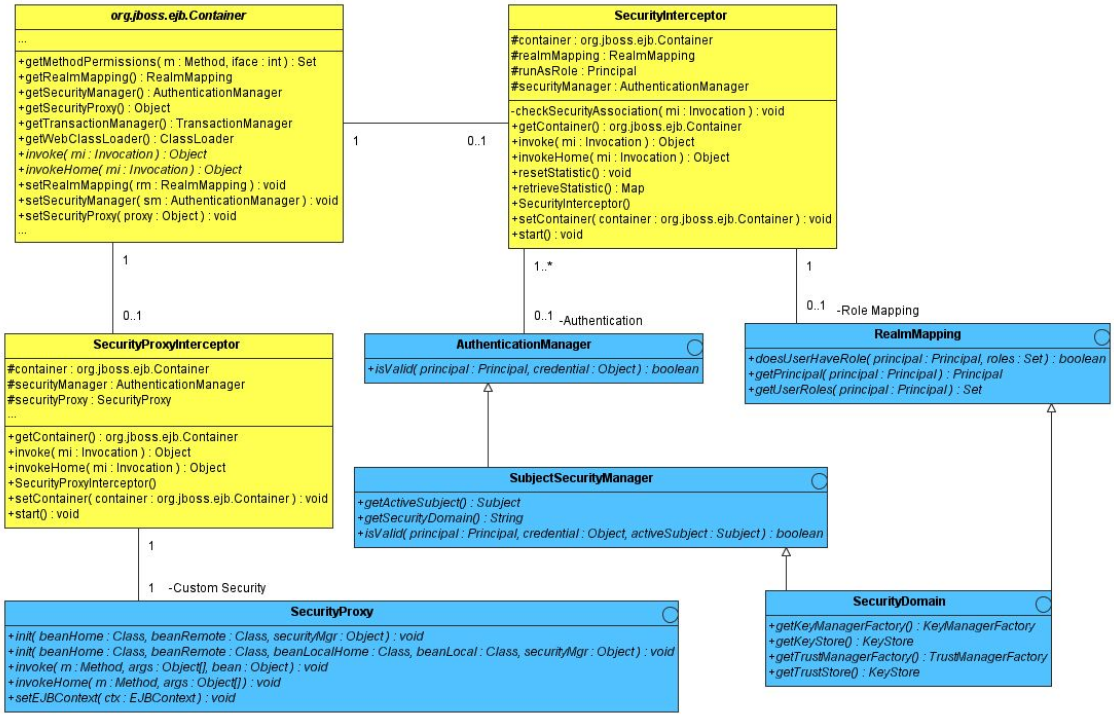


Figure 8.8. The key security model interfaces and their relationship to the JBoss server EJB container elements.

The light blue classes represent the security interfaces while the yellow classes represent the EJB container layer. The two interfaces required for the implementation of the J2EE security model are `org.jboss.security.AuthenticationManager` and `org.jboss.security.RealmMapping`. The roles of the security interfaces presented in *Figure 8.8, "The key security model interfaces and their relationship to the JBoss server EJB container elements."* are summarized in the following list.

- **AuthenticationManager**: This interface is responsible for validating credentials associated with principals. Principals are identities, such as usernames, employee numbers, and social security numbers. Credentials are proof of the identity, such as passwords, session keys, and digital signatures. The `isValid` method is invoked to determine whether a user identity and associated credentials as known in the operational environment are valid proof of the user's identity.

- **RealmMapping**: This interface is responsible for principal mapping and role mapping. The `getPrincipal` method takes a user identity as known in the operational environment and returns the application domain identity. The `doesUserHaveRole` method validates that the user identity in the operation environment has been assigned the indicated role from the application domain.
- **SecurityProxy**: This interface describes the requirements for a custom `SecurityProxyInterceptor` plugin. A `SecurityProxy` allows for the externalization of custom security checks on a per-method basis for both the EJB home and remote interface methods.
- **SubjectSecurityManager**: This is a subinterface of `AuthenticationManager` that adds accessor methods for obtaining the security domain name of the security manager and the current thread's authenticated `Subject`.
- **SecurityDomain**: This is an extension of the `AuthenticationManager`, `RealmMapping`, and `SubjectSecurityManager` interfaces. It is a move to a comprehensive security interface based on the JAAS Subject, a `java.security.KeyStore`, and the JSSE `com.sun.net.ssl.KeyManagerFactory` and `com.sun.net.ssl.TrustManagerFactory` interfaces. This interface is a work in progress that will be the basis of a multi-domain security architecture that will better support ASP style deployments of applications and resources.

Note that the `AuthenticationManager`, `RealmMapping` and `SecurityProxy` interfaces have no association to JAAS related classes. Although the JBossSX framework is heavily dependent on JAAS, the basic security interfaces required for implementation of the J2EE security model are not. The JBossSX framework is simply an implementation of the basic security plug-in interfaces that are based on JAAS. The component diagram presented in *Figure 8.9, “The relationship between the JBossSX framework implementation classes and the JBoss server EJB container layer.”* illustrates this fact. The implication of this plug-in architecture is that you are free to replace the JAAS-based JBossSX implementation classes with your own custom security manager implementation that does not make use of JAAS, if you so desire. You'll see how to do this when you look at the JBossSX MBeans available for the configuration of JBossSX in *Figure 8.9, “The relationship between the JBossSX framework implementation classes and the JBoss server EJB container layer.”*.

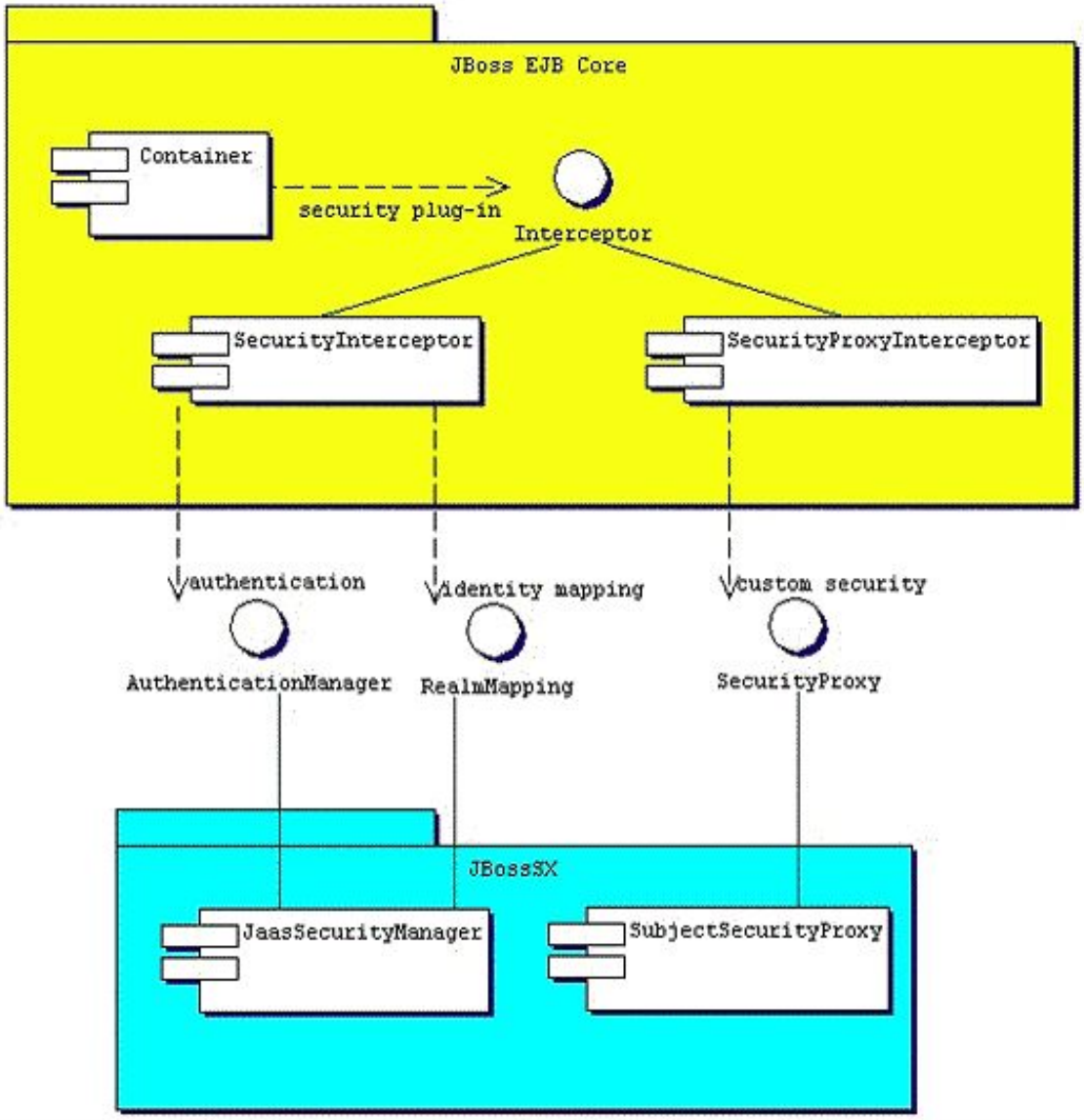


Figure 8.9. The relationship between the JBossSX framework implementation classes and the JBoss server EJB container layer.

3.1. Enabling Declarative Security in JBoss Revisited

Earlier in this chapter, the discussion of the J2EE standard security model ended with a requirement for the use of JBoss server-specific deployment descriptor to enable security. The details of this configuration are presented here. *Figure 8.10, "The security element subsets of the JBoss server jboss.xml and jboss-web.xml deployment descriptors."* shows the JBoss-specific EJB and web application deployment descriptor's security-related elements.

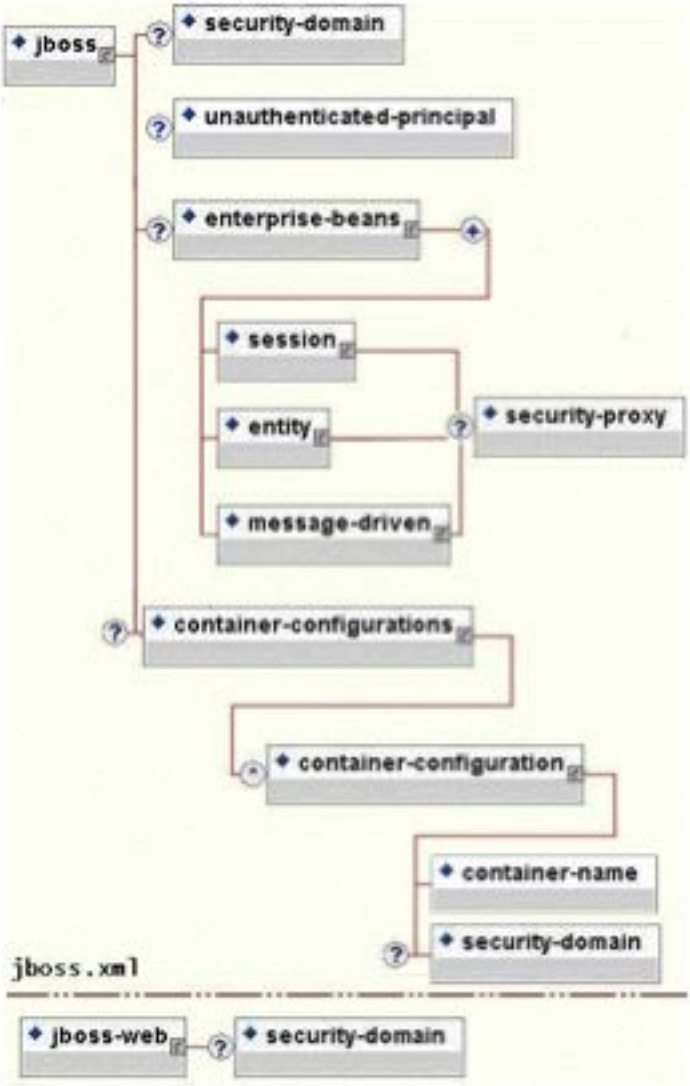


Figure 8.10. The security element subsets of the JBoss server jboss.xml and jboss-web.xml deployment descriptors.

The value of a `security-domain` element specifies the JNDI name of the security manager interface implementation that JBoss uses for the EJB and web containers. This is an object that implements both of the `AuthenticationManager` and `RealmMapping` interfaces. When specified as a top-level element it defines what security domain in effect for all EJBs in the deployment unit. This is the typical usage because mixing security managers within a deployment unit complicates inter-component operation and administration.

To specify the security domain for an individual EJB, you specify the `security-domain` at the container configuration level. This will override any top-level security-domain element.

The `unauthenticated-principal` element specifies the name to use for the `Principal` object

returned by the `EJBContext.getUserPrincipal` method when an unauthenticated user invokes an EJB. Note that this conveys no special permissions to an unauthenticated caller. Its primary purpose is to allow unsecured servlets and JSP pages to invoke unsecured EJBs and allow the target EJB to obtain a non-null `Principal` for the caller using the `getUserPrincipal` method. This is a J2EE specification requirement.

The `security-proxy` element identifies a custom security proxy implementation that allows per-request security checks outside the scope of the EJB declarative security model without embedding security logic into the EJB implementation. This may be an implementation of the `org.jboss.security.SecurityProxy` interface, or just an object that implements methods in the home, remote, local home or local interfaces of the EJB to secure without implementing any common interface. If the given class does not implement the `SecurityProxy` interface, the instance must be wrapped in a `SecurityProxy` implementation that delegates the method invocations to the object. The `org.jboss.security.SubjectSecurityProxy` is an example `SecurityProxy` implementation used by the default JBossSX installation.

Take a look at a simple example of a custom `SecurityProxy` in the context of a trivial stateless session bean. The custom `SecurityProxy` validates that no one invokes the bean's `echo` method with a four-letter word as its argument. This is a check that is not possible with role-based security; you cannot define a `FourLetterEchoInvoker` role because the security context is the method argument, not a property of the caller. The code for the custom `SecurityProxy` is given in *Example 8.7, "The example 1 custom EchoSecurityProxy implementation that enforces the echo argument-based security constraint."*, and the full source code is available in the `src/main/org/jboss/book/security/ex1` directory of the book examples.

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

import java.lang.reflect.Method;
import javax.ejb.EJBContext;

import org.apache.log4j.Category;

import org.jboss.security.SecurityProxy;

/** A simple example of a custom SecurityProxy implementation
 *  that demonstrates method argument based security checks.
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.12 $
 */
public class EchoSecurityProxy implements SecurityProxy
{
    Category log = Category.getInstance(EchoSecurityProxy.class);
    Method echo;

    public void init(Class beanHome, Class beanRemote,
                     Object securityMgr)
        throws InstantiationException
    {
        log.debug("init, beanHome="+beanHome
                  + ", beanRemote="+beanRemote
                  + ", securityMgr="+securityMgr);
```

```
        // Get the echo method for equality testing in invoke
        try {
            Class[] params = {String.class};
            echo = beanRemote.getDeclaredMethod("echo", params);
        } catch(Exception e) {
            String msg = "Failed to finde an echo(String) method";
            log.error(msg, e);
            throw new InstantiationException(msg);
        }
    }

    public void setEJBContext(EJBContext ctx)
    {
        log.debug("setEJBContext, ctx="+ctx);
    }

    public void invokeHome(Method m, Object[] args)
        throws SecurityException
    {
        // We don't validate access to home methods
    }

    public void invoke(Method m, Object[] args, Object bean)
        throws SecurityException
    {
        log.debug("invoke, m="+m);
        // Check for the echo method
        if (m.equals(echo)) {
            // Validate that the msg arg is not 4 letter word
            String arg = (String) args[0];
            if (arg == null || arg.length() == 4)
                throw new SecurityException("No 4 letter words");
        }
        // We are not responsible for doing the invoke
    }
}
```

Example 8.7. The example 1 custom EchoSecurityProxy implementation that enforces the echo argument-based security constraint.

The `EchoSecurityProxy` checks that the method to be invoked on the bean instance corresponds to the `echo(String)` method loaded the init method. If there is a match, the method argument is obtained and its length compared against 4 or null. Either case results in a `SecurityException` being thrown. Certainly this is a contrived example, but only in its application. It is a common requirement that applications must perform security checks based on the value of method arguments. The point of the example is to demonstrate how custom security beyond the scope of the standard declarative security model can be introduced independent of the bean implementation. This allows the specification and coding of the security requirements to be delegated to security experts. Since the security proxy layer can be done

independent of the bean implementation, security can be changed to match the deployment environment requirements.

The associated `jboss.xml` descriptor that installs the `EchoSecurityProxy` as the custom proxy for the `EchoBean` is given in *Example 8.8, "The jboss.xml descriptor, which configures the EchoSecurityProxy as the custom security proxy for the EchoBean."*.

```
<jboss>
    <security-domain>java:/jaas/other</security-domain>

    <enterprise-beans>
        <session>
            <ejb-name>EchoBean</ejb-name>
<security-proxy>org.jboss.book.security.ex1.EchoSecurityProxy</security-proxy>
        </session>
    </enterprise-beans>
</jboss>
```

Example 8.8. The jboss.xml descriptor, which configures the EchoSecurityProxy as the custom security proxy for the EchoBean.

Now test the custom proxy by running a client that attempts to invoke the `EchoBean.echo` method with the arguments `Hello` and `Four` as illustrated in this fragment:

```
public class ExClient
{
    public static void main(String args[])
        throws Exception
    {
        Logger log = Logger.getLogger("ExClient");
        log.info("Looking up EchoBean");

        InitialContext iniCtx = new InitialContext();
        Object ref = iniCtx.lookup("EchoBean");
        EchoHome home = (EchoHome) ref;
        Echo echo = home.create();

        log.info("Created Echo");
        log.info("Echo.echo('Hello') = "+echo.echo("Hello"));
        log.info("Echo.echo('Four') = "+echo.echo("Four"));
    }
}
```

The first call should succeed, while the second should fail due to the fact that `Four` is a four-letter word. Run the client as follows using Ant from the examples directory:

```
[examples]$ ant -Dchap=security -Dex=1 run-example
run-example1:
...
```

```
     [echo] Waiting for 5 seconds for deploy...
     [java] [INFO,ExClient] Looking up EchoBean
     [java] [INFO,ExClient] Created Echo
     [java] [INFO,ExClient] Echo.echo('Hello') = Hello
     [java] Exception in thread "main" java.rmi.AccessException:
SecurityException; nested exception is:
     [java]     java.lang.SecurityException: No 4 letter words
...
     [java] Caused by: java.lang.SecurityException: No 4 letter words
...
```

The result is that the `echo('Hello')` method call succeeds as expected and the `echo('Four')` method call results in a rather messy looking exception, which is also expected. The above output has been truncated to fit in the book. The key part to the exception is that the `SecurityException("No 4 letter words")` generated by the `EchoSecurityProxy` was thrown to abort the attempted method invocation as desired.

4. The JBoss Security Extension Architecture

The preceding discussion of the general JBoss security layer has stated that the JBossSX security extension framework is an implementation of the security layer interfaces. This is the primary purpose of the JBossSX framework. The details of the implementation are interesting in that it offers a great deal of customization for integration into existing security infrastructures. A security infrastructure can be anything from a database or LDAP server to a sophisticated security software suite. The integration flexibility is achieved using the pluggable authentication model available in the JAAS framework.

The heart of the JBossSX framework is `org.jboss.security.plugins.JaasSecurityManager`. This is the default implementation of the `AuthenticationManager` and `RealmMapping` interfaces. *Figure 8.11, “The relationship between the security-domain component deployment descriptor value, the component container and the JaasSecurityManager.”* shows how the `JaasSecurityManager` integrates into the EJB and web container layers based on the `security-domain` element of the corresponding component deployment descriptor.

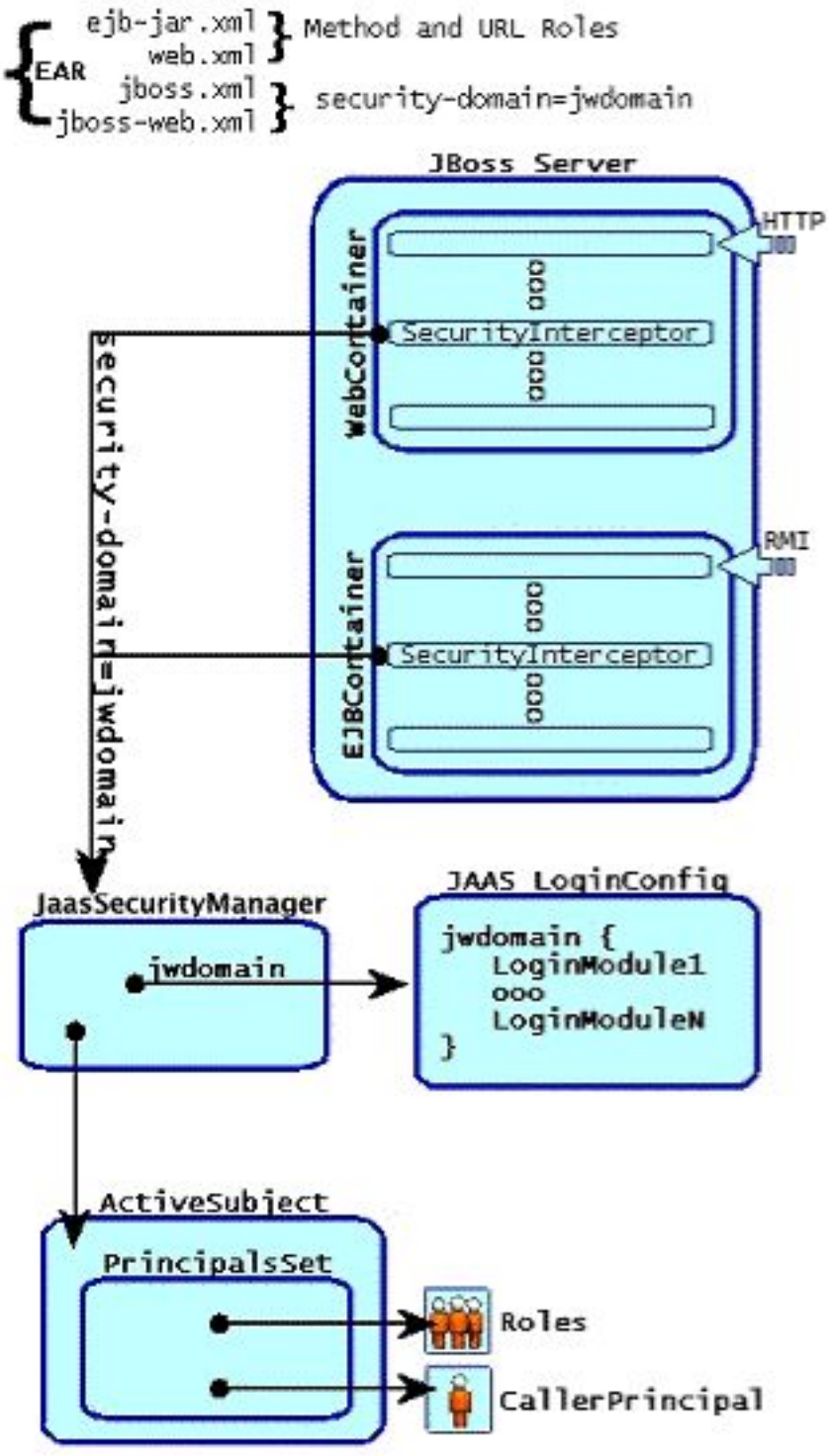


Figure 8.11. The relationship between the security-domain component deployment descriptor value, the component container and the JaasSecurityManager.

Figure 8.11, "The relationship between the security-domain component deployment descriptor value, the component container and the JaasSecurityManager." depicts an enterprise application that contains both EJBs and web content secured under the security domain `jwdomain`. The EJB and web containers have a request interceptor architecture that includes a security interceptor, which enforces the container security model. At deployment time, the `security-domain` element value in the `jboss.xml` and `jboss-web.xml` descriptors is used to obtain the security manager instance associated with the container. The security interceptor then uses the security manager to perform its role. When a secured component is requested, the security interceptor delegates security checks to the security manager instance associated with the container.

The JBossSX `JaasSecurityManager` implementation performs security checks based on the information associated with the `Subject` instance that results from executing the JAAS login modules configured under the name matching the `security-domain` element value. We will drill into the `JaasSecurityManager` implementation and its use of JAAS in the following section.

4.1. How the JaasSecurityManager Uses JAAS

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

To illustrate the details of the `JaasSecurityManager`'s usage of the JAAS authentication process, you will walk through a client invocation of an EJB home method invocation. The prerequisite setting is that the EJB has been deployed in the JBoss server and its home interface methods have been secured using `method-permission` elements in the `ejb-jar.xml` descriptor, and it has been assigned a security domain named `jwdomain` using the `jboss.xml` descriptor `security-domain` element.

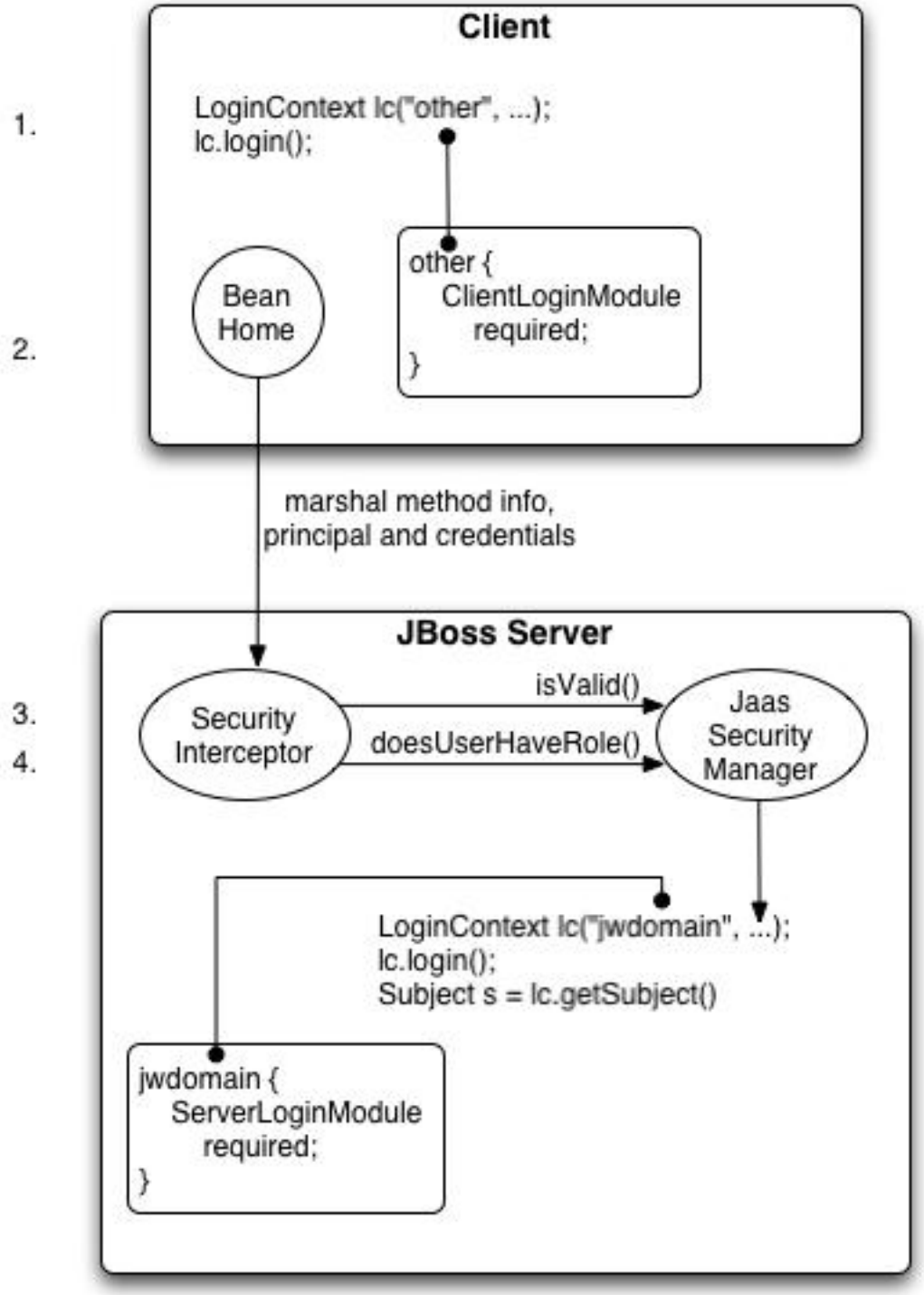


Figure 8.12. An illustration of the steps involved in the authentication and authorization of a secured EJB home method invocation.

Figure 8.12, "An illustration of the steps involved in the authentication and authorization of a secured EJB home method invocation." provides a view of the client to server communication

we will discuss. The numbered steps shown are:

1. The client first has to perform a JAAS login to establish the principal and credentials for authentication, and this is labeled *Client Side Login* in the figure. This is how clients establish their login identities in JBoss. Support for presenting the login information via JNDI `InitialContext` properties is provided via an alternate configuration. A JAAS login entails creating a `LoginContext` instance and passing the name of the configuration to use. The configuration name is `other`. This one-time login associates the login principal and credentials with all subsequent EJB method invocations. Note that the process might not authenticate the user. The nature of the client-side login depends on the login module configuration that the client uses. In this example, the `other` client-side login configuration entry is set up to use the `ClientLoginModule` module (an `org.jboss.security.ClientLoginModule`). This is the default client side module that simply binds the username and password to the JBoss EJB invocation layer for later authentication on the server. The identity of the client is not authenticated on the client.

2. Later, the client obtains the EJB home interface and attempts to create a bean. This event is labeled as *Home Method Invocation*. This results in a home interface method invocation being sent to the JBoss server. The invocation includes the method arguments passed by the client along with the user identity and credentials from the client-side JAAS login performed in step 1.

3. On the server side, the security interceptor first requires authentication of the user invoking the call, which, as on the client side, involves a JAAS login.

4. The security domain under which the EJB is secured determines the choice of login modules. The security domain name is used as the login configuration entry name passed to the `LoginContext` constructor. The EJB security domain is `jwdomain`. If the JAAS login authenticates the user, a JAAS `Subject` is created that contains the following in its `PrincipalsSet`:

   - A `java.security.Principal` that corresponds to the client identity as known in the deployment security environment.

   - A `java.security.acl.Group` named `Roles` that contains the role names from the application domain to which the user has been assigned. `org.jboss.security.SimplePrincipal` objects are used to represent the role names; `SimplePrincipal` is a simple string-based implementation of `Principal`. These roles are used to validate the roles assigned to methods in `ejb-jar.xml` and the `EJBContext.isCallerInRole(String)` method implementation.

   - An optional `java.security.acl.Group` named `CallerPrincipal`, which contains a single `org.jboss.security.SimplePrincipal` that corresponds to the identity of the application domain's caller. The `CallerPrincipal` sole group member will be the value returned by the `EJBContext.getCallerPrincipal()` method. The purpose of this mapping is to allow a `Principal` as known in the operational security environment to map to a `Principal` with a name known to the application. In the absence of a `CallerPrincipal` mapping the

deployment security environment principal is used as the `getCallerPrincipal` method value. That is, the operational principal is the same as the application domain principal.

5. The final step of the security interceptor check is to verify that the authenticated user has permission to invoke the requested method This is labeled as *Server Side Authorization* in *Figure 8.12, "An illustration of the steps involved in the authentication and authorization of a secured EJB home method invocation."*. Performing the authorization this entails the following steps:

   - Obtain the names of the roles allowed to access the EJB method from the EJB container. The role names are determined by `ejb-jar.xml` descriptor role-name elements of all `method-permission` elements containing the invoked method.
   - If no roles have been assigned, or the method is specified in an `exclude-list` element, then access to the method is denied. Otherwise, the `doesUserHaveRole` method is invoked on the security manager by the security interceptor to see if the caller has one of the assigned role names. This method iterates through the role names and checks if the authenticated user's Subject `Roles` group contains a `SimplePrincipal` with the assigned role name. Access is allowed if any role name is a member of the `Roles` group. Access is denied if none of the role names are members.
   - If the EJB was configured with a custom security proxy, the method invocation is delegated to it. If the security proxy wants to deny access to the caller, it will throw a `java.lang.SecurityException`. If no `SecurityException` is thrown, access to the EJB method is allowed and the method invocation passes to the next container interceptor. Note that the `SecurityProxyInterceptor` handles this check and this interceptor is not shown.

Every secured EJB method invocation, or secured web content access, requires the authentication and authorization of the caller because security information is handled as a stateless attribute of the request that must be presented and validated on each request. This can be an expensive operation if the JAAS login involves client-to-server communication. Because of this, the `JaasSecurityManager` supports the notion of an authentication cache that is used to store principal and credential information from previous successful logins. You can specify the authentication cache instance to use as part of the `JaasSecurityManager` configuration as you will see when the associated MBean service is discussed in following section. In the absence of any user-defined cache, a default cache that maintains credential information for a configurable period of time is used.

4.2. The JaasSecurityManagerService MBean

The `JaasSecurityManagerService` MBean service manages security managers. Although its name begins with *Jaas*, the security managers it handles need not use JAAS in their implementation. The name arose from the fact that the default security manager implementation is the `JaasSecurityManager`. The primary role of the `JaasSecurityManagerService` is to externalize the security manager implementation. You can change the security manager implementation by providing an alternate implementation of the `AuthenticationManager` and

`RealmMapping` interfaces.

The second fundamental role of the `JaasSecurityManagerService` is to provide a JNDI `javax.naming.spi.ObjectFactory` implementation to allow for simple code-free management of the JNDI name to security manager implementation mapping. It has been mentioned that security is enabled by specifying the JNDI name of the security manager implementation via the `security-domain` deployment descriptor element. When you specify a JNDI name, there has to be an object-binding there to use. To simplify the setup of the JNDI name to security manager bindings, the `JaasSecurityManagerService` manages the association of security manager instances to names by binding a next naming system reference with itself as the JNDI ObjectFactory under the name `java:/jaas`. This allows one to use a naming convention of the form `java:/jaas/XYZ` as the value for the `security-domain` element, and the security manager instance for the `XYZ` security domain will be created as needed for you. The security manager for the domain `XYZ` is created on the first lookup against the `java:/jaas/XYZ` binding by creating an instance of the class specified by the `SecurityManagerClassName` attribute using a constructor that takes the name of the security domain. For example, consider the following container security configuration snippet:

```
<jboss>
    <!-- Configure all containers to be secured under the "hades" security
domain -->
    <security-domain>java:/jaas/hades</security-domain>
    <!-- ... -->
</jboss>
```

Any lookup of the name `java:/jaas/hades` will return a security manager instance that has been associated with the security domain named `hades`. This security manager will implement the AuthenticationManager and RealmMapping security interfaces and will be of the type specified by the `JaasSecurityManagerServiceSecurityManagerClassName` attribute.

The `JaasSecurityManagerService` MBean is configured by default for use in the standard JBoss distribution, and you can often use the default configuration as is. The configurable attributes of the `JaasSecurityManagerService` include:

- **SecurityManagerClassName**: The name of the class that provides the security manager implementation. The implementation must support both the `org.jboss.security.AuthenticationManager` and `org.jboss.security.RealmMapping` interfaces. If not specified this defaults to the JAAS-based `org.jboss.security.plugins.JaasSecurityManager`.
- **CallbackHandlerClassName**: The name of the class that provides the `javax.security.auth.callback.CallbackHandler` implementation used by the `JaasSecurityManager`. You can override the handler used by the `JaasSecurityManager` if the default implementation (`org.jboss.security.auth.callback.SecurityAssociationHandler`) does not meet your needs. This is a rather deep configuration that generally should not be set unless you know what you are doing.

- **SecurityProxyFactoryClassName**: The name of the class that provides the `org.jboss.security.SecurityProxyFactory` implementation. If not specified this defaults to `org.jboss.security.SubjectSecurityProxyFactory`.
- **AuthenticationCacheJndiName**: Specifies the location of the security credential cache policy. This is first treated as an `ObjectFactory` location capable of returning `CachePolicy` instances on a per-security-domain basis. This is done by appending the name of the security domain to this name when looking up the `CachePolicy` for a domain. If this fails, the location is treated as a single `CachePolicy` for all security domains. As a default, a timed cache policy is used.
- **DefaultCacheTimeout**: Specifies the default timed cache policy timeout in seconds. The default value is 1800 seconds (30 minutes). The value you use for the timeout is a tradeoff between frequent authentication operations and how long credential information may be out of sync with respect to the security information store. If you want to disable caching of security credentials, set this to 0 to force authentication to occur every time. This has no affect if the `AuthenticationCacheJndiName` has been changed from the default value.
- **DefaultCacheResolution**: Specifies the default timed cache policy resolution in seconds. This controls the interval at which the cache current timestamp is updated and should be less than the `DefaultCacheTimeout` in order for the timeout to be meaningful. The default resolution is 60 seconds(1 minute). This has no affect if the `AuthenticationCacheJndiName` has been changed from the default value.
- **DefaultUnauthenticatedPrincipal**: Specifies the principal to use for unauthenticated users. This setting makes it possible to set default permissions for users who have not been authenticated.

The `JaasSecurityManagerService` also supports a number of useful operations. These include flushing any security domain authentication cache at runtime, getting the list of active users in a security domain authentication cache, and any of the security manager interface methods.

Flushing a security domain authentication cache can be used to drop all cached credentials when the underlying store has been updated and you want the store state to be used immediately. The MBean operation signature is: `public void flushAuthenticationCache(String securityDomain)`.

This can be invoked programmatically using the following code snippet:

```
MBeanServer server = ...;
String jaasMgrName = "jboss.security:service=JaasSecurityManager";
ObjectName jaasMgr = new ObjectName(jaasMgrName);
Object[] params = {domainName};
String[] signature = {"java.lang.String"};
server.invoke(jaasMgr, "flushAuthenticationCache", params, signature);
```

Getting the list of active users provides a snapshot of the `Principals` keys in a security domain

authentication cache that are not expired. The MBean operation signature is: `public List getAuthenticationCachePrincipals(String securityDomain)`.

This can be invoked programmatically using the following code snippet:

```
MBeanServer server = ...;
String jaasMgrName = "jboss.security:service=JaasSecurityManager";
ObjectName jaasMgr = new ObjectName(jaasMgrName);
Object[] params = {domainName};
String[] signature = {"java.lang.String"};
List users = (List) server.invoke(jaasMgr,
"getAuthenticationCachePrincipals",
                                  params, signature);
```

The security manager has a few additional access methods.

```
public boolean isValid(String securityDomain, Principal principal, Object
credential);
public Principal getPrincipal(String securityDomain, Principal principal);
public boolean doesUserHaveRole(String securityDomain, Principal principal,
                                Object credential, Set roles);
public Set getUserRoles(String securityDomain, Principal principal, Object
credential);
```

They provide access to the corresponding `AuthenticationManager` and `RealmMapping` interface method of the associated security domain named by the `securityDomain` argument.

4.3. The JaasSecurityDomain MBean

The `org.jboss.security.plugins.JaasSecurityDomain` is an extension of `JaasSecurityManager` that adds the notion of a `KeyStore`, a JSSE `KeyManagerFactory` and a `TrustManagerFactory` for supporting SSL and other cryptographic use cases. The additional configurable attributes of the `JaasSecurityDomain` include:

- **KeyStoreType**: The type of the `KeyStore` implementation. This is the type argument passed to the `java.security.KeyStore.getInstance(String type)` factory method. The default is `JKS`.
- **KeyStoreURL**: A URL to the location of the `KeyStore` database. This is used to obtain an `InputStream` to initialize the `KeyStore`. If the string is not a value URL, it is treated as a file.
- **KeyStorePass**: The password associated with the `KeyStore` database contents. The `KeyStorePass` is also used in combination with the `Salt` and `IterationCount` attributes to create a PBE secret key used with the encode/decode operations. The `KeyStorePass` attribute value format is one of the following:
  - The plaintext password for the `KeyStore` The `toCharArray()` value of the string is used without any manipulation.

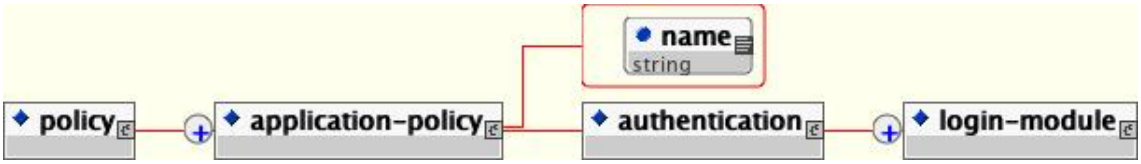
- A command to execute to obtain the plaintext password. The format is `{EXT}...` where the `...` is the exact command line that will be passed to the `Runtime.exec(String)` method to execute a platform-specific command. The first line of the command output is used as the password.
- A class to create to obtain the plaintext password. The format is `{CLASS}classname[:ctorarg]` where the `[:ctorarg]` is an optional string that will be passed to the constructor when instantiating the `classname`. The password is obtained from classname by invoking a `toCharArray()` method if found, otherwise, the `toString()` method is used.

- **Salt**: The `PBEParameterSpec` salt value.
- **IterationCount**: The `PBEParameterSpec` iteration count value.
- **TrustStoreType**: The type of the `TrustStore` implementation. This is the type argument passed to the `java.security.KeyStore.getInstance(String type)` factory method. The default is `JKS`.
- **TrustStoreURL**: A URL to the location of the `TrustStore` database. This is used to obtain an `InputStream` to initialize the `KeyStore`. If the string is not a value URL, it is treated as a file.
- **TrustStorePass**: The password associated with the trust store database contents. The `TrustStorePass` is a simple password and doesn't have the same configuration options as the `KeyStorePass`.
- **ManagerServiceName**: Sets the JMX object name string of the security manager service MBean. This is used to register the defaults to register the `JaasSecurityDomain` as a the security manager under `java:/jaas/<domain>` where `<domain>` is the name passed to the MBean constructor. The name defaults to `jboss.security:service=JaasSecurityManager`.

5. Defining Security Domains

The standard way of configuring security domains for authentication and authorization in JBoss is to use the XML login configuration file. The login configuration policy defines a set of named security domains that each define a stack of login modules that will be called upon to authenticate and authorize users.

The XML configuration file conforms to the DTD given by *Figure 8.13, "The XMLLoginConfig DTD"*. This DTD can be found in `docs/dtd/security_config.dtd`.



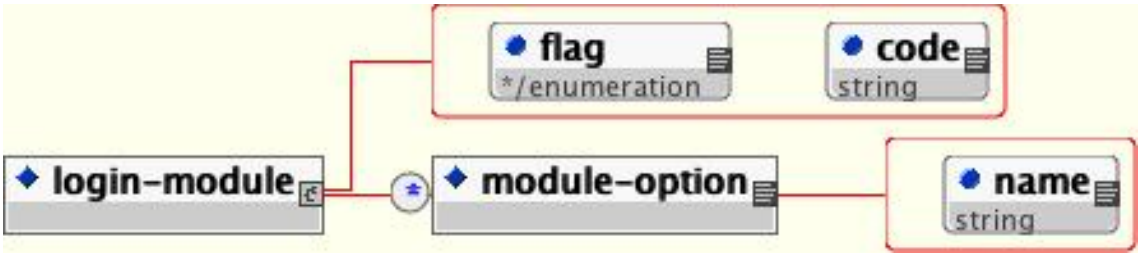


Figure 8.13. The XMLLoginConfig DTD

The following example shows a simple configuration named jmx-console that is backed by a single login module. The login module is configured by a simple set of name/value configuration pairs that have meaning to the login module in question. We'll see what these options mean later, for now we'll just be concerned with the structure of the configuration file.

```
<application-policy name="jmx-console">
    <authentication>
        <login-module
code="org.jboss.security.auth.spi.UsersRolesLoginModule" flag="required">
            <module-option
name="usersProperties">props/jmx-console-users.properties</module-option>
            <module-option
name="rolesProperties">props/jmx-console-roles.properties</module-option>
        </login-module>
    </authentication>
</application-policy>
```

The `name` attribute of the `application-policy` is the login configuration name. Applications policy elements will be bound by that name in JNDI under the the `java:/jaas` context. Applications will link to security domains through this JNDI name in their deployment descriptors. (See the `security-domain` elements in `jboss.xml`, `jboss-web.xml` and `jboss-service.xml` files for examples)

The `code` attribute of the `login-module` element specifies the class name of the login module implementation. The `required` flag attribute controls the overall behavior of the authentication stack. The allowed values and meanings are:

- **required**: The login module is required to succeed for the authentication to be successful. If any required module fails, the authentication will fail. The remaining login modules in the stack will be called regardless of the outcome of the authentication.
- **requisite**: The login module is required to succeed. If it succeeds, authentication continues down the login stack. If it fails, control immediately returns to the application.
- **sufficient**: The login module is not required to succeed. If it does succeed, control immediately returns to the application. If it fails, authentication continues down the login stack.

- **optional**: The login module is not required to succeed. Authentication still continues to proceed down the login stack regardless of whether the login module succeeds or fails.

The following example shows the definition of a security domain that uses multiple login modules. Since both modules are marked as sufficient, only one of them need to succeed for login to proceed.

```
<application-policy name="todo">
    <authentication>
        <login-module code="org.jboss.security.auth.spi.LdapLoginModule"
                      flag="sufficient">
            <!-- LDAP configuration -->
        </login-module>
        <login-module
code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
                      flag="sufficient">
            <!-- database configuration -->
        </login-module>
    </authentication>
</application-policy>
```

Each login module has its own set of configuration options. These are set as name/value pairs using the `module-option` elements. We'll cover module options in more depth when we look at the individual login modules available in JBoss AS.

5.1. Loading Security Domains

Authentication security domains are configured statically in the `conf/login-config.xml` file. The `XMLLoginConfig` MBean is responsible for loading security configurations from this configurations from a local configuration file. The MBean is defined as shown below.

```
<mbean code="org.jboss.security.auth.login.XMLLoginConfig"
       name="jboss.security:service=XMLLoginConfig">
    <attribute name="ConfigResource">login-config.xml</attribute>
</mbean>
```

The MBean supports the following attributes:

- **ConfigURL**: specifies the URL of the XML login configuration file that should be loaded by this MBean on startup. This must be a valid URL string representation.
- **ConfigResource**: specifies the resource name of the XML login configuration file that should be loaded by this MBean on startup. The name is treated as a classpath resource for which a URL is located using the thread context class loader.
- **ValidateDTD**: a flag indicating if the XML configuration should be validated against its DTD. This defaults to true.

The MBean also supports the following operations that allow one to dynamically extend the login configurations at runtime. Note that any operation that attempts to alter login configuration requires a `javax.security.auth.AuthPermission("refreshLoginConfiguration")` when running with a security manager. The `org.jboss.book.security.service.SecurityConfig` service demonstrates how this can be used to add/remove a deployment specific security configuration dynamically.

- `void addAppConfig(String appName, AppConfigurationEntry[] entries)`: this adds the given login module configuration stack to the current configuration under the given `appName`. This replaces any existing entry under that name.
- `void removeAppConfig(String appName)`: this removes the login module configuration registered under the given `appName`.
- `String[] loadConfig(URL configURL) throws Exception`: this loads one or more login configurations from a URL representing either an XML or legacy Sun login configuration file. Note that all login configurations must be added or none will be added. It returns the names of the login configurations that were added.
- `void removeConfigs(String[] appNames)`: this removes the login configurations specified `appNames` array.
- `String displayAppConfig(String appName)`: this operation displays a simple string format of the named configuration if it exists.

The `SecurityConfig` MBean is responsible for selecting the `javax.security.auth.login.Configuration` to be used. The default configuration simply references the `XMLLoginConfig` MBean.

```
   <mbean code="org.jboss.security.plugins.SecurityConfig"
        name="jboss.security:service=SecurityConfig">
     <attribute
name="LoginConfig">jboss.security:service=XMLLoginConfig</attribute>
                 </mbean>
```

There is one configurable attribute:

- **LoginConfig**: Specifies the JMX `ObjectName` string of the MBean that provides the default JAAS login configuration. When the `SecurityConfig` is started, this MBean is queried for its `javax.security.auth.login.Configuration` by calling its `getConfiguration(Configuration currentConfig)` operation. If the `LoginConfig` attribute is not specified then the default Sun `Configuration` implementation described in the `Configuration` class JavaDocs is used.

In addition to allowing for a custom JAAS login configuration implementation, this service allows

configurations to be chained together in a stack at runtime. This allows one to push a login configuration onto the stack and latter pop it. This is a feature used by the security unit tests to install custom login configurations into a default JBoss installation. Pushing a new configuration is done using:

```
public void pushLoginConfig(String objectName) throws
                JMException, MalformedObjectNameException;
```

The `objectName` parameters specifies an MBean similar to the `LoginConfig` attribute. The current login configuration may be removed using:

```
public void popLoginConfig() throws JMException;
```

5.2. The DynamicLoginConfig service

Security domains defined in the `login-config.xml` file are essentially static. They are read when JBoss starts up, but there is no easy way to add a new security domain or change the definition for an existing one. The `DynamicLoginConfig` service allows you to dynamically deploy security domains. This allows you to specify JAAS login configuration as part of a deployment (or just as a standalone service) rather than having to edit the static `login-config.xml` file.

The service supports the following attributes:

- **AuthConfig**: The resource path to the JAAS login configuration file to use. This defaults to `login-config.xml`
- **LoginConfigService**: the `XMLLoginConfig` service name to use for loading. This service must support a `String loadConfig(URL)` operation to load the configurations.
- **SecurityManagerService**: The `SecurityManagerService` name used to flush the registered security domains. This service must support a `flushAuthenticationCache(String)` operation to flush the case for the argument security domain. Setting this triggers the flush of the authentication caches when the service is stopped.

Here is an example MBean definition using the `DynamicLoginConfig` service.

```
<server>
    <mbean code="org.jboss.security.auth.login.DynamicLoginConfig"
name="...">
        <attribute name="AuthConfig">login-config.xml</attribute>

        <!-- The service which supports dynamic processing of
login-config.xml
         configurations.
        -->
        <depends optional-attribute-name="LoginConfigService">
```

```
            jboss.security:service=XMLLoginConfig </depends>

        <!-- Optionally specify the security mgr service to use when
         this service is stopped to flush the auth caches of the domains
         registered by this service.
        -->
        <depends optional-attribute-name="SecurityManagerService">
            jboss.security:service=JaasSecurityManager </depends>
    </mbean>
</server>
```

This will load the specified `AuthConfig` resource using the specified `LoginConfigService` MBean by invoking `loadConfig` with the appropriate resource URL. When the service is stopped the configurations are removed. The resource specified may be either an XML file, or a Sun JAAS login configuration.

5.3. Using JBoss Login Modules

JBoss includes several bundled login modules suitable for most user management needs. JBoss can read user information from a relational database, an LDAP server or flat files. In addition to these core login modules, JBoss provides several other login modules that provide user information for very customized needs in JBoss. Before we explore the individual login modules, let's take a look at a few login module configuration options that are common to multiple modules.

5.3.1. Password Stacking

Multiple login modules can be chained together in a stack, with each login module providing both the authentication and authorization components. This works for many use cases, but sometimes authentication and authorization are split across multiple user management stores. A previous example showed how to combine LDAP and a relational database, allowing a user to be authenticated by either system. However, consider the case where users are managed in a central LDAP server but application-specific roles are stored in the application's relational database. The password-stacking module option captures this relationship.

- **password-stacking**: When `password-stacking` option is set to `useFirstPass`, this module first looks for a shared username and password under the property names `javax.security.auth.login.name` and `javax.security.auth.login.password` respectively in the login module shared state map. If found these are used as the principal name and password. If not found the principal name and password are set by this login module and stored under the property names `javax.security.auth.login.name` and `javax.security.auth.login.password` respectively.

To use password stacking, each login module should set `password-stacking` to `useFirstPass`. If a previous module configured for password stacking has authenticated the user, all the other stacking modules will consider the user authenticated and only attempt to

provide a set of roles for the authorization step.

The following listing shows how password stacking could be used:

```
<application-policy name="todo">
    <authentication>
        <login-module code="org.jboss.security.auth.spi.LdapLoginModule"
                      flag="required">
            <!-- LDAP configuration -->
            <module-option
name="password-stacking">useFirstPass</module-option>
        </login-module>
        <login-module
code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
                      flag="required">
            <!-- database configuration -->
            <module-option
name="password-stacking">useFirstPass</module-option>
        </login-module>
    </authentication>
</application-policy>
```

When using password stacking, it is usually appropriate to set all modules to be required to make sure that all modules are considered and have chance to contribute roles to the authorization process.

5.3.2. Password Hashing

Most of the login modules need to compare a client-supplied password to a password stored in a user management system. These modules generally work with plain text passwords, but can also be configured to support hashed passwords to prevent plain text passwords from being stored on the server side.

- **hashAlgorithm**: The name of the `java.security.MessageDigest` algorithm to use to hash the password. There is no default so this option must be specified to enable hashing. Typical values are `MD5` and `SHA`.
- **hashEncoding**: The string format for the hashed pass and must be either `base64`, `hex` or `rfc2617`. The default is `base64`.
- **hashCharset**: The encoding used to convert the clear text password to a byte array. The platform default encoding is the default.
- **hashUserPassword**: This indicates that the hashing algorithm should be applied to the password the user submits. The hashed user password will be compared against the value in the login module, which is expected to be a hash of the password. The default is true.
- **hashStorePassword**: This indicates that the hashing algorithm should be applied to the password stored on the server side. This is used for digest authentication where the user submits a hash of the user password along with a request-specific tokens from the server to

be comare. JBoss uses the hash algorithm (for digest, this would be `rfc2617`) to compute a server-side hash that should match the hashed value sent from the client.

The following is an login module configuration that assigns unauthenticated users the principal name `nobody` and contains based64-encoded, MD5 hashes of the passwords in a `usersb64.properties` file.

```
<policy>
    <application-policy name="testUsersRoles">
        <authentication>
            <login-module
code="org.jboss.security.auth.spi.UsersRolesLoginModule"
                          flag="required">
                <module-option name="hashAlgorithm">MD5</module-option>
                <module-option name="hashEncoding">base64</module-option>
            </login-module>
        </authentication>
    </application-policy>
</policy>
```

If you need to generate passwords in code, `the org.jboss.security.Util` class provides a static helper method that will hash a password using a given encoding.

```
String hashedPassword = Util.createPasswordHash("MD5",
                                                Util.BASE64_ENCODING,
                                                null,
                                                null,
                                                "password");
```

OpenSSL provides an alternative way to quickly generate hashed passwords.

```
echo -n password | openssl dgst -md5 -binary | openssl base64
```

In both cases, the text password should hash to "X03MO1qnZdYdgyfeuILPmQ==". This is the value that would need to be stored in the user store.

5.3.3. Unauthenticated Identity

Not all requests come in authenticated. The unauthenticated identity is a login module configuration option that assigns a specific identity (guest, for example) to requests that are made with no associated authentication information. This can be used to allow unprotected servlets to invoke methods on EJBs that do not require a specific role. Such a principal has no associated roles and so can only access either unsecured EJBs or EJB methods that are associated with the unchecked permission constraint.

- **unauthenticatedIdentity**: This defines the principal name that should be assigned to

requests that contain no authentication information.

5.3.4. UsersRolesLoginModule

The `UsersRolesLoginModule` is a simple login module that supports multiple users and user roles loaded from Java properties files. The username-to-password mapping file is called `users.properties` and the username-to-roles mapping file is called `roles.properties`. The properties files are loaded during initialization using the initialize method thread context class loader. This means that these files can be placed into the J2EE deployment JAR, the JBoss configuration directory, or any directory on the JBoss server or system classpath. The primary purpose of this login module is to easily test the security settings of multiple users and roles using properties files deployed with the application.

The `users.properties` file uses a `username=password` format with each user entry on a separate line as show here:

```
username1=password1
username2=password2
...
```

The `roles.properties` file uses as `username=role1,role2,...` format with an optional group name value. For example:

```
username1=role1,role2,...
username1.RoleGroup1=role3,role4,...
username2=role1,role3,...
```

The `username.XXX` form of property name is used to assign the username roles to a particular named group of roles where the `XXX` portion of the property name is the group name. The `username=...` form is an abbreviation for `username.Roles=...`, where the `Roles` group name is the standard name the `JaasSecurityManager` expects to contain the roles which define the users permissions.

The following would be equivalent definitions for the `jduke` username:

```
jduke=TheDuke,AnimatedCharacter
jduke.Roles=TheDuke,AnimatedCharacter
```

The supported login module configuration options include the following:

- **usersProperties**: The name of the properties resource containing the username to password mappings. This defaults to `users.properties`.
- **rolesProperties**: The name of the properties resource containing the username to roles mappings. This defaults to `roles.properties`.

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

5.3.5. LdapLoginModule

The `LdapLoginModule` is a `LoginModule` implementation that authenticates against an LDAP server. You would use the `LdapLoginModule` if your username and credentials are stored in an LDAP server that is accessible using a JNDI LDAP provider.

The LDAP connectivity information is provided as configuration options that are passed through to the environment object used to create JNDI initial context. The standard LDAP JNDI properties used include the following:

- **java.naming.factory.initial**: The classname of the `InitialContextFactory` implementation. This defaults to the Sun LDAP provider implementation `com.sun.jndi.ldap.LdapCtxFactory`.
- **java.naming.provider.url**: The LDAP URL for the LDAP server
- **java.naming.security.authentication**: The security level to use. This defaults to `simple`.
- **java.naming.security.protocol**: The transport protocol to use for secure access, such as, SSL.
- **java.naming.security.principal**: The principal for authenticating the caller to the service. This is built from other properties as described below.
- **java.naming.security.credentials**: The value of the property depends on the authentication scheme. For example, it could be a hashed password, clear-text password, key, certificate, and so on.

The supported login module configuration options include the following:

- **principalDNPrefix**: A prefix to add to the username to form the user distinguished name. See `principalDNSuffix` for more info.
- **principalDNSuffix**: A suffix to add to the username when forming the user distinguished name. This is useful if you prompt a user for a username and you don't want the user to have to enter the fully distinguished name. Using this property and `principalDNSuffix` the `userDN` will be formed as `principalDNPrefix + username + principalDNSuffix`
- **useObjectCredential**: A true/false value that indicates that the credential should be obtained as an opaque `Object` using the `org.jboss.security.auth.callback.ObjectCallback` type of `Callback` rather than as a `char[]` password using a JAAS `PasswordCallback`. This allows for passing non-`char[]` credential information to the LDAP server.
- **rolesCtxDN**: The fixed distinguished name to the context to search for user roles.

- **userRolesCtxDNAttributeName**: The name of an attribute in the user object that contains the distinguished name to the context to search for user roles. This differs from `rolesCtxDN` in that the context to search for a user's roles can be unique for each user.
- **roleAttributeID**: The name of the attribute that contains the user roles. If not specified this defaults to `roles`.
- **roleAttributeIsDN**: A flag indicating whether the `roleAttributeID` contains the fully distinguished name of a role object, or the role name. If false, the role name is taken from the value of `roleAttributeID`. If true, the role attribute represents the distinguished name of a role object. The role name is taken from the value of the `roleNameAttributeId` attribute of the context name by the distinguished name. In certain directory schemas (e.g., MS ActiveDirectory), role attributes in the user object are stored as DNs to role objects instead of as simple names, in which case, this property should be set to true. The default is false.
- **roleNameAttributeID**: The name of the attribute of the context pointed to by the `roleCtxDN` distinguished name value which contains the role name. If the `roleAttributeIsDN` property is set to true, this property is used to find the role object's name attribute. The default is `group`.
- **uidAttributeID**: The name of the attribute in the object containing the user roles that corresponds to the userid. This is used to locate the user roles. If not specified this defaults to `uid`.
- **matchOnUserDN**: A true/false flag indicating if the search for user roles should match on the user's fully distinguished name. If false, just the username is used as the match value against the `uidAttributeName` attribute. If true, the full `userDN` is used as the match value.
- **unauthenticatedIdentity**: The principal name that should be assigned to requests that contain no authentication information. This behavior is inherited from the `UsernamePasswordLoginModule` superclass.
- **allowEmptyPasswords**: A flag indicating if empty (length 0) passwords should be passed to the LDAP server. An empty password is treated as an anonymous login by some LDAP servers and this may not be a desirable feature. Set this to false to reject empty passwords or true to have the LDAP server validate the empty password. The default is true.

The authentication of a user is performed by connecting to the LDAP server based on the login module configuration options. Connecting to the LDAP server is done by creating an `InitialLdapContext` with an environment composed of the LDAP JNDI properties described previously in this section. The `Context.SECURITY_PRINCIPAL` is set to the distinguished name of the user as obtained by the callback handler in combination with the `principalDNPrefix` and `principalDNSuffix` option values, and the `Context.SECURITY_CREDENTIALS` property is either set to the `String` password or the `Object` credential depending on the `useObjectCredential` option.

Once authentication has succeeded by virtue of being able to create an `InitialLdapContext` instance, the user's roles are queried by performing a search on the `rolesCtxDN` location with search attributes set to the `roleAttributeName` and `uidAttributeName` option values. The

roles names are obtaining by invoking the `toString` method on the role attributes in the search result set.

The following is a sample `login-config.xml` entry.

```
    <application-policy name="testLDAP">
        <authentication>
            <login-module code="org.jboss.security.auth.spi.LdapLoginModule"
                          flag="required">
                <module-option name="java.naming.factory.initial">
                    com.sun.jndi.ldap.LdapCtxFactory
                    </module-option>
                <module-option name="java.naming.provider.url">
                    ldap://ldaphost.jboss.org:1389/
                </module-option>
                <module-option name="java.naming.security.authentication">
                    simple
                </module-option>
                <module-option name="principalDNPrefix">uid=</module-option>
                <module-option name="principalDNSuffix">
                    ,ou=People,dc=jboss,dc=org
                </module-option>

                <module-option name="rolesCtxDN">
                    ou=Roles,dc=jboss,dc=org
                </module-option>
                <module-option name="uidAttributeID">member</module-option>
                <module-option name="matchOnUserDN">true</module-option>

                <module-option name="roleAttributeID">cn</module-option>
                <module-option name="roleAttributeIsDN">false
</module-option>
            </login-module>
        </authentication>
    </application-policy>
```

An LDIF file representing the structure of the directory this data operates against is shown below.

```
dn: dc=jboss,dc=org
objectclass: top
objectclass: dcObject
objectclass: organization
dc: jboss
o: JBoss

dn: ou=People,dc=jboss,dc=org
objectclass: top
objectclass: organizationalUnit
ou: People

dn: uid=jduke,ou=People,dc=jboss,dc=org
objectclass: top
objectclass: uidObject
```

```
objectclass: person
uid: jduke
cn: Java Duke
sn: Duke
userPassword: theduke

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

dn: cn=JBossAdmin,ou=Roles,dc=jboss,dc=org
objectclass: top
objectclass: groupOfNames
cn: JBossAdmin
member: uid=jduke,ou=People,dc=jboss,dc=org
description: the JBossAdmin group
```

Looking back at the `testLDAP` login module configuration, the `java.naming.factory.initial`, `java.naming.factory.url` and `java.naming.security` options indicate the Sun LDAP JNDI provider implementation will be used, the LDAP server is located on host `ldaphost.jboss.org` on port 1389, and that the LDAP simple authentication method will be use to connect to the LDAP server.

The login module attempts to connect to the LDAP server using a DN representing the user it is trying to authenticate. This DN is constructed from the `principalDNPrefix`, passed in, the username of the user and the `principalDNSuffix` as described above. In this example, the username `jduke` would map to `uid=jduke,ou=People,dc=jboss,dc=org`. We've assumed the LDAP server authenticates users using the `userPassword` attribute of the user's entry (`theduke` in this example). This is the way most LDAP servers work, however, if your LDAP server handles authentication differently you will need to set the authentication credentials in a way that makes sense for your server.

Once authentication succeeds, the roles on which authorization will be based are retrieved by performing a subtree search of the `rolesCtxDN` for entries whose `uidAttributeID` match the user. If `matchOnUserDN` is true the search will be based on the full DN of the user. Otherwise the search will be based on the actual user name entered. In this example, the search is under `ou=Roles,dc=jboss,dc=org` for any entries that have a `member` attribute equal to `uid=jduke,ou=People,dc=jboss,dc=org`. The search would locate `cn=JBossAdmin` under the roles entry.

The search returns the attribute specified in the `roleAttributeID` option. In this example, the attribute is `cn`. The value returned would be `JBossAdmin`, so the jduke user is assigned to the `JBossAdmin` role.

It's often the case that a local LDAP server provides identity and authentication services but is unable to use the authorization services. This is because application roles don't always map well onto LDAP groups, and LDAP administrators are often hesitant to allow external application-specific data in central LDAP servers. For this reason, the LDAP authentication

module is often paired with another login module, such as the database login module, that can provide roles more suitable to the application being developed.

This login module also supports unauthenticated identity and password stacking.

5.3.6. DatabaseServerLoginModule

The `DatabaseServerLoginModule` is a JDBC based login module that supports authentication and role mapping. You would use this login module if you have your username, password and role information relational database. The `DatabaseServerLoginModule` is based on two logical tables:

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

The `Principals` table associates the user `PrincipalID` with the valid password and the `Roles` table associates the user `PrincipalID` with its role sets. The roles used for user permissions must be contained in rows with a `RoleGroup` column value of `Roles`. The tables are logical in that you can specify the SQL query that the login module uses. All that is required is that the `java.sql.ResultSet` has the same logical structure as the `Principals` and `Roles` tables described previously. The actual names of the tables and columns are not relevant as the results are accessed based on the column index. To clarify this notion, consider a database with two tables, `Principals` and `Roles`, as already declared. The following statements build the tables to contain a `PrincipalIDjava` with a `Password` of `echoman` in the `Principals` table, a `PrincipalIDjava` with a role named `Echo` in the `RolesRoleGroup` in the `Roles` table, and a `PrincipalIDjava` with a role named `caller_java` in the `CallerPrincipalRoleGroup` in the `Roles` table:

```
INSERT INTO Principals VALUES('java', 'echoman')
INSERT INTO Roles VALUES('java', 'Echo', 'Roles')
INSERT INTO Roles VALUES('java', 'caller_java', 'CallerPrincipal')
```

The supported login module configuration options include the following:

- **dsJndiName**: The JNDI name for the `DataSource` of the database containing the logical `Principals` and `Roles` tables. If not specified this defaults to `java:/DefaultDS`.
- **principalsQuery**: The prepared statement query equivalent to: `select Password from Principals where PrincipalID=?`. If not specified this is the exact prepared statement that will be used.
- **rolesQuery**: The prepared statement query equivalent to: `select Role, RoleGroup from Roles where PrincipalID=?`. If not specified this is the exact prepared statement that will be used.
- **ignorePasswordCase**: A boolean flag indicating if the password comparison should ignore

case. This can be useful for hashed password encoding where the case of the hashed password is not significant.

- **principalClass**: An option that specifies a `Principal` implementation class. This must support a constructor taking a string argument for the principal name.

As an example `DatabaseServerLoginModule` configuration, consider a custom table schema like the following:

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

A corresponding `login-config.xml` entry would be:

```
<policy>
    <application-policy name="testDB">
        <authentication>
            <login-module
code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
                          flag="required">
                <module-option
name="dsJndiName">java:/MyDatabaseDS</module-option>
                <module-option name="principalsQuery">
                    select passwd from Users username where
username=?</module-option>
                <module-option name="rolesQuery">
                    select userRoles, 'Roles' from UserRoles where
username=?</module-option>
            </login-module>
        </authentication>
    </application-policy>
</policy>
```

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

5.3.7. BaseCertLoginModule

This is a login module which authenticates users based on X509 certificates. A typical use case for this login module is `CLIENT-CERT` authentication in the web tier. This login module only performs authentication. You need to combine it with another login module capable of acquiring the authorization roles to completely define access to a secured web or EJB component. Two subclasses of this login module, `CertRolesLoginModule` and `DatabaseCertLoginModule` extend the behavior to obtain the authorization roles from either a properties file or database.

The `BaseCertLoginModule` needs a `KeyStore` to perform user validation. This is obtained through a `org.jboss.security.SecurityDomain` implementation. Typically, the `SecurityDomain` implementation is configured using the `org.jboss.security.plugins.JaasSecurityDomain` MBean as shown in this `jboss-service.xml` configuration fragment:

```
<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
       name="jboss.ch8:service=SecurityDomain">
    <constructor>
        <arg type="java.lang.String" value="jmx-console"/>
    </constructor>
    <attribute name="KeyStoreURL">resource:localhost.keystore</attribute>
    <attribute name="KeyStorePass">unit-tests-server</attribute>
</mbean>
```

This creates a security domain with the name `jmx-console` whose `SecurityDomain` implementation is available via JNDI under the name `java:/jaas/jmx-console` following the JBossSX security domain naming pattern. To secure a web application such as the `jmx-console.war` using client certs and role based authorization, one would first modify the `web.xml` to declare the resources to be secured, along with the allowed roles and security domain to be used for authentication and authorization.

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC
                  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
                  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    ...
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>HtmlAdaptor</web-resource-name>
            <description>An example security config that only allows users
with
                the role JBossAdmin to access the HTML JMX console web
                application </description>
            <url-pattern>/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>JBossAdmin</role-name>
        </auth-constraint>
    </security-constraint>
    <login-config>
        <auth-method>CLIENT-CERT</auth-method>
        <realm-name>JBoss JMX Console</realm-name>
    </login-config>
    <security-role>
        <role-name>JBossAdmin</role-name>
    </security-role>
</web-app>
```

Next we, need to specify the JBoss security domain in `jboss-web.xml`:

```
<jboss-web>
    <security-domain>java:/jaas/jmx-console</security-domain>
</jboss-web>
```

Finally, you need to define the login module configuration for the jmx-console security domain you just specified. This is done in the `conf/login-config.xml` file.

```
<application-policy name="jmx-console">
    <authentication>
        <login-module code="org.jboss.security.auth.spi.BaseCertLoginModule"
                      flag="required">
            <module-option
name="password-stacking">useFirstPass</module-option>
            <module-option
name="securityDomain">java:/jaas/jmx-console</module-option>
        </login-module>
        <login-module
code="org.jboss.security.auth.spi.UsersRolesLoginModule"
                      flag="required">
            <module-option
name="password-stacking">useFirstPass</module-option>
            <module-option
name="usersProperties">jmx-console-users.properties</module-option>
            <module-option
name="rolesProperties">jmx-console-roles.properties</module-option>
        </login-module>
    </authentication>
</application-policy>
```

Here the `BaseCertLoginModule` is used for authentication of the client cert, and the `UsersRolesLoginModule` is only used for authorization due to the `password-stacking=useFirstPass` option. Both the `localhost.keystore` and the `jmx-console-roles.properties` need an entry that maps to the principal associated with the client cert. By default, the principal is created using the client certificate distinguished name. Consider the following certificate:

```
[starksm@banshee9100 conf]$ keytool -printcert -file
unit-tests-client.export
Owner: CN=unit-tests-client, OU=JBoss Inc., O=JBoss Inc., ST=Washington,
C=US
Issuer: CN=jboss.com, C=US, ST=Washington, L=Snoqualmie Pass,
EMAILADDRESS=admin
@jboss.com, OU=QA, O=JBoss Inc.
Serial number: 100103
Valid from: Wed May 26 07:34:34 PDT 2004 until: Thu May 26 07:34:34 PDT 2005
Certificate fingerprints:
         MD5:  4A:9C:2B:CD:1B:50:AA:85:DD:89:F6:1D:F5:AF:9E:AB
         SHA1: DE:DE:86:59:05:6C:00:E8:CC:C0:16:D3:C2:68:BF:95:B8:83:E9:58
```

The `localhost.keystore` would need this cert stored with an alias of `CN=unit-tests-client, OU=JBoss Inc., O=JBoss Inc., ST=Washington, C=US` and the `jmx-console-roles.properties` would also need an entry for the same entry. Since the DN contains many characters that are normally treated as delimiters, you will need to escape the problem characters using a backslash ('\') as shown here:

```
# A sample roles.properties file for use with the UsersRolesLoginModule
CN\=unit-tests-client,\ OU\=JBoss\ Inc.,\ O\=JBoss\ Inc.,\ ST\=Washington,\
C\=US=JBossAdmin
admin=JBossAdmin
```

5.3.8. IdentityLoginModule

The `IdentityLoginModule` is a simple login module that associates a hard-coded user name a to any subject authenticated against the module. It creates a `SimplePrincipal` instance using the name specified by the `principal` option. This login module is useful when you need to provide a fixed identity to a service and in development environments when you want to test the security associated with a given principal and associated roles.

The supported login module configuration options include:

- **principal**: This is the name to use for the `SimplePrincipal` all users are authenticated as. The principal name defaults to `guest` if no principal option is specified.
- **roles**: This is a comma-delimited list of roles that will be assigned to the user.

A sample XMLLoginConfig configuration entry that would authenticate all users as the principal named `jduke` and assign role names of `TheDuke`, and `AnimatedCharacter` is:

```
<policy>
    <application-policy name="testIdentity">
        <authentication>
            <login-module
code="org.jboss.security.auth.spi.IdentityLoginModule"
                          flag="required">
                <module-option name="principal">jduke</module-option>
                <module-option
name="roles">TheDuke,AnimatedCharater</module-option>
            </login-module>
        </authentication>
    </application-policy>
</policy>
```

This module supports password stacking.

5.3.9. RunAsLoginModule

JBoss has a helper login module called `RunAsLoginModule` that pushes a run as role for the duration of the login phase of authentication, and pops the run as role in either the commit or abort phase. The purpose of this login module is to provide a role for other login modules that need to access secured resources in order to perform their authentication. An example would be a login module that accesses an secured EJB. This login module must be configured ahead of the login module(s) that need a run as role established.

The only login module configuration option is:

- **roleName**: the name of the role to use as the run as role during login phase. If not specified a default of `nobody` is used.

5.3.10. ClientLoginModule

The `ClientLoginModule` is an implementation of `LoginModule` for use by JBoss clients for the establishment of the caller identity and credentials. This simply sets the `org.jboss.security.SecurityAssociation.principal` to the value of the `NameCallback` filled in by the `callbackhandler`, and the `org.jboss.security.SecurityAssociation.credential` to the value of the `PasswordCallback` filled in by the `callbackhandler`. This is the only supported mechanism for a client to establish the current thread's caller. Both stand-alone client applications and server environments, acting as JBoss EJB clients where the security environment has not been configured to use JBossSX transparently, need to use the `ClientLoginModule`. Of course, you could always set the `org.jboss.security.SecurityAssociation` information directly, but this is considered an internal API that is subject to change without notice.

Note that this login module does not perform any authentication. It merely copies the login information provided to it into the JBoss server EJB invocation layer for subsequent authentication on the server. If you need to perform client-side authentication of users you would need to configure another login module in addition to the `ClientLoginModule`.

The supported login module configuration options include the following:

- **multi-threaded**: When the multi-threaded option is set to true, each login thread has its own principal and credential storage. This is useful in client environments where multiple user identities are active in separate threads. When true, each separate thread must perform its own login. When set to false the login identity and credentials are global variables that apply to all threads in the VM. The default for this option is false.
- **password-stacking**: When `password-stacking` option is set to `useFirstPass`, this module first looks for a shared username and password using `javax.security.auth.login.name` and `javax.security.auth.login.password` respectively in the login module shared state map. This allows a module configured prior to this one to establish a valid username and password that should be passed to JBoss. You would use this option if you want to perform client-side authentication of clients using some other login module such as the `LdapLoginModule`.
- **restore-login-identity**: When `restore-login-identity` is true, the `SecurityAssociation` principal and credential seen on entry to the `login()` method are saved and restored on either abort or logout. When false (the default), the abort and logout simply clear the `SecurityAssociation`. A `restore-login-identity` of true is needed if one need to change identities and then restore the original caller identity.

A sample login configuration for `ClientLoginModule` is the default configuration entry found in the JBoss distribution `client/auth.conf` file. The configuration is:

```
other {
    // Put your login modules that work without jBoss here

    // jBoss LoginModule
    org.jboss.security.ClientLoginModule required;

    // Put your login modules that need jBoss here
};
```

5.4. Writing Custom Login Modules

If the login modules bundled with the JBossSX framework do not work with your security environment, you can write your own custom login module implementation that does. Recall from the section on the `JaasSecurityManager` architecture that the `JaasSecurityManager` expected a particular usage pattern of the `Subject` principals set. You need to understand the JAAS Subject class's information storage features and the expected usage of these features to be able to write a login module that works with the `JaasSecurityManager`. This section examines this requirement and introduces two abstract base `LoginModule` implementations that can help you implement your own custom login modules.

You can obtain security information associated with a `Subject` in six ways in JBoss using the following methods:

```
java.util.Set getPrincipals()
java.util.Set getPrincipals(java.lang.Class c)
java.util.Set getPrivateCredentials()
java.util.Set getPrivateCredentials(java.lang.Class c)
java.util.Set getPublicCredentials()
java.util.Set getPublicCredentials(java.lang.Class c)
```

For `Subject` identities and roles, JBossSX has selected the most natural choice: the principals sets obtained via `getPrincipals()` and `getPrincipals(java.lang.Class)`. The usage pattern is as follows:

- User identities (username, social security number, employee ID, and so on) are stored as `java.security.Principal` objects in the `SubjectPrincipals` set. The `Principal` implementation that represents the user identity must base comparisons and equality on the name of the principal. A suitable implementation is available as the `org.jboss.security.SimplePrincipal` class. Other `Principal` instances may be added to the `SubjectPrincipals` set as needed.
- The assigned user roles are also stored in the `Principals` set, but they are grouped in named role sets using `java.security.acl.Group` instances. The `Group` interface defines a

collection of `Principals` and/or `Groups`, and is a subinterface of `java.security.Principal`. Any number of role sets can be assigned to a `Subject`. Currently, the JBossSX framework uses two well-known role sets with the names `Roles` and `CallerPrincipal`. The `Roles` Group is the collection of `Principals` for the named roles as known in the application domain under which the `Subject` has been authenticated. This role set is used by methods like the `EJBContext.isCallerInRole(String)`, which EJBs can use to see if the current caller belongs to the named application domain role. The security interceptor logic that performs method permission checks also uses this role set. The `CallerPrincipalGroup` consists of the single `Principal` identity assigned to the user in the application domain. The `EJBContext.getCallerPrincipal()` method uses the `CallerPrincipal` to allow the application domain to map from the operation environment identity to a user identity suitable for the application. If a `Subject` does not have a `CallerPrincipalGroup`, the application identity is the same as operational environment identity.

5.4.1. Support for the Subject Usage Pattern

To simplify correct implementation of the `Subject` usage patterns described in the preceding section, JBossSX includes two abstract login modules that handle the population of the authenticated `Subject` with a template pattern that enforces correct `Subject` usage. The most generic of the two is the `org.jboss.security.auth.spi.AbstractServerLoginModule` class. It provides a concrete implementation of the `javax.security.auth.spi.LoginModule` interface and offers abstract methods for the key tasks specific to an operation environment security infrastructure. The key details of the class are highlighted in the following class fragment. The JavaDoc comments detail the responsibilities of subclasses.

```
package org.jboss.security.auth.spi;
/**
 *  This class implements the common functionality required for a JAAS
 *  server-side LoginModule and implements the JBossSX standard
 *  Subject usage pattern of storing identities and roles. Subclass
 *  this module to create your own custom LoginModule and override the
 *  login(), getRoleSets(), and getIdentity() methods.
 */
public abstract class AbstractServerLoginModule
    implements javax.security.auth.spi.LoginModule
{
    protected Subject subject;
    protected CallbackHandler callbackHandler;
    protected Map sharedState;
    protected Map options;
    protected Logger log;

    /** Flag indicating if the shared credential should be used */
    protected boolean useFirstPass;
    /**
     * Flag indicating if the login phase succeeded. Subclasses that
     * override the login method must set this to true on successful
     * completion of login
     */
    protected boolean loginOk;
```

```
    // ...
    /**
     * Initialize the login module. This stores the subject,
     * callbackHandler and sharedState and options for the login
     * session. Subclasses should override if they need to process
     * their own options. A call to super.initialize(...)  must be
     * made in the case of an override.
     *
     * <p>
     * The options are checked for the  <em>password-stacking</em>
parameter.
     * If this is set to "useFirstPass", the login identity will be taken
from the
     * <code>javax.security.auth.login.name</code> value of the sharedState
map,
     * and the proof of identity from the
     * <code>javax.security.auth.login.password</code> value of the
sharedState map.
     *
     * @param subject the Subject to update after a successful login.
     * @param callbackHandler the CallbackHandler that will be used to
obtain the
     * the user identity and credentials.
     * @param sharedState a Map shared between all configured login module
instances
     * @param options the parameters passed to the login module.
     */
    public void initialize(Subject subject,
                           CallbackHandler callbackHandler,
                           Map sharedState,
                           Map options)
    {
        // ...
    }


    /**
     *  Looks for javax.security.auth.login.name and
     *  javax.security.auth.login.password values in the sharedState
     *  map if the useFirstPass option was true and returns true if
     *  they exist. If they do not or are null this method returns
     *  false.
     *  Note that subclasses that override the login method
     *  must set the loginOk var to true if the login succeeds in
     *  order for the commit phase to populate the Subject. This
     *  implementation sets loginOk to true if the login() method
     *  returns true, otherwise, it sets loginOk to false.
     */
    public boolean login()
        throws LoginException
    {
        // ...
    }

    /**
     *  Overridden by subclasses to return the Principal that
     *  corresponds to the user primary identity.
```

```
     */
    abstract protected Principal getIdentity();

    /**
     *  Overridden by subclasses to return the Groups that correspond
     *  to the role sets assigned to the user. Subclasses should
     *  create at least a Group named "Roles" that contains the roles
     *  assigned to the user.  A second common group is
     *  "CallerPrincipal," which provides the application identity of
     *  the user rather than the security domain identity.
     *
     *  @return Group[] containing the sets of roles
     */
    abstract protected Group[] getRoleSets() throws LoginException;
}
```

You'll need to pay attention to the `loginOk` instance variable. This must be set to true if the login succeeds, false otherwise by any subclasses that override the login method. Failure to set this variable correctly will result in the commit method either not updating the subject when it should, or updating the subject when it should not. Tracking the outcome of the login phase was added to allow login modules to be chained together with control flags that do not require that the login module succeed in order for the overall login to succeed.

The second abstract base login module suitable for custom login modules is the `org.jboss.security.auth.spi.UsernamePasswordLoginModule`. This login module further simplifies custom login module implementation by enforcing a string-based username as the user identity and a `char[]` password as the authentication credentials. It also supports the mapping of anonymous users (indicated by a null username and password) to a principal with no roles. The key details of the class are highlighted in the following class fragment. The JavaDoc comments detail the responsibilities of subclasses.

```
package org.jboss.security.auth.spi;

/**
 *  An abstract subclass of AbstractServerLoginModule that imposes a
 *  an identity == String username, credentials == String password
 *  view on the login process. Subclasses override the
 *  getUsersPassword() and getUsersRoles() methods to return the
 *  expected password and roles for the user.
 */
public abstract class UsernamePasswordLoginModule
    extends AbstractServerLoginModule
{
    /** The login identity */
    private Principal identity;
    /** The proof of login identity */
    private char[] credential;
    /** The principal to use when a null username and password are seen */
    private Principal unauthenticatedIdentity;

    /**
     * The message digest algorithm used to hash passwords. If null then
```

```
     * plain passwords will be used. */
    private String hashAlgorithm = null;

    /**
     *  The name of the charset/encoding to use when converting the
     * password String to a byte array. Default is the platform's
     * default encoding.
     */
     private String hashCharset = null;

    /** The string encoding format to use. Defaults to base64. */
    private String hashEncoding = null;

    // ...

    /**
     *  Override the superclass method to look for an
     *  unauthenticatedIdentity property. This method first invokes
     *  the super version.
     *
     *  @param options,
     *  @option unauthenticatedIdentity: the name of the principal to
     *  assign and authenticate when a null username and password are
     *  seen.
     */
    public void initialize(Subject subject,
                           CallbackHandler callbackHandler,
                           Map sharedState,
                           Map options)
    {
        super.initialize(subject, callbackHandler, sharedState,
                         options);
        // Check for unauthenticatedIdentity option.
        Object option = options.get("unauthenticatedIdentity");
        String name = (String) option;
        if (name != null) {
            unauthenticatedIdentity = new SimplePrincipal(name);
        }
    }

    // ...

    /**
     *  A hook that allows subclasses to change the validation of the
     *  input password against the expected password. This version
     *  checks that neither inputPassword or expectedPassword are null
     *  and that inputPassword.equals(expectedPassword) is true;
     *
     *  @return true if the inputPassword is valid, false otherwise.
     */
    protected boolean validatePassword(String inputPassword,
                                       String expectedPassword)
    {
        if (inputPassword == null || expectedPassword == null) {
            return false;
        }
        return inputPassword.equals(expectedPassword);
```

```
    }

    /**
     *  Get the expected password for the current username available
     * via the getUsername() method. This is called from within the
     * login() method after the CallbackHandler has returned the
     * username and candidate password.
     *
     * @return the valid password String
     */
    abstract protected String getUsersPassword()
        throws LoginException;
}
```

The choice of subclassing the `AbstractServerLoginModule` versus `UsernamePasswordLoginModule` is simply based on whether a string-based username and credentials are usable for the authentication technology you are writing the login module for. If the string-based semantic is valid, then subclass `UsernamePasswordLoginModule`, otherwise subclass `AbstractServerLoginModule`.

The steps you are required to perform when writing a custom login module are summarized in the following depending on which base login module class you choose. When writing a custom login module that integrates with your security infrastructure, you should start by subclassing `AbstractServerLoginModule` or `UsernamePasswordLoginModule` to ensure that your login module provides the authenticated `Principal` information in the form expected by the JBossSX security manager.

When subclassing the `AbstractServerLoginModule`, you need to override the following:

- `void initialize(Subject, CallbackHandler, Map, Map)`: if you have custom options to parse.
- `boolean login()`: to perform the authentication activity. Be sure to set the `loginOk` instance variable to true if login succeeds, false if it fails.
- `Principal getIdentity()`: to return the `Principal` object for the user authenticated by the `log()` step.
- `Group[] getRoleSets()`: to return at least one `Group` named `Roles` that contains the roles assigned to the `Principal` authenticated during `login()`. A second common `Group` is named `CallerPrincipal` and provides the user's application identity rather than the security domain identity.

When subclassing the `UsernamePasswordLoginModule`, you need to override the following:

- `void initialize(Subject, CallbackHandler, Map, Map)`: if you have custom options to parse.

- `Group[] getRoleSets()`: to return at least one `Group` named `Roles` that contains the roles assigned to the `Principal` authenticated during `login()`. A second common `Group` is named `CallerPrincipal` and provides the user's application identity rather than the security domain identity.
- `String getUsersPassword()`: to return the expected password for the current username available via the `getUsername()` method. The `getUsersPassword()` method is called from within `login()` after the `callbackhandler` returns the username and candidate password.

5.4.2. A Custom LoginModule Example

In this section we will develop a custom login module example. It will extend the `UsernamePasswordLoginModule` and obtains a user's password and role names from a JNDI lookup. The idea is that there is a JNDI context that will return a user's password if you perform a lookup on the context using a name of the form `password/<username>` where `<username>` is the current user being authenticated. Similarly, a lookup of the form `roles/<username>` returns the requested user's roles.

The source code for the example is located in the `src/main/org/jboss/book/security/ex2` directory of the book examples. *Example 8.9, " A JndiUserAndPass custom login module"* shows the source code for the `JndiUserAndPass` custom login module. Note that because this extends the JBoss `UsernamePasswordLoginModule`, all the `JndiUserAndPass` does is obtain the user's password and roles from the JNDI store. The `JndiUserAndPass` does not concern itself with the JAAS `LoginModule` operations.

```
package org.jboss.book.security.ex2;

import java.security.acl.Group;
import java.util.Map;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginException;

import org.jboss.security.SimpleGroup;
import org.jboss.security.SimplePrincipal;
import org.jboss.security.auth.spi.UsernamePasswordLoginModule;

/**
 *  An example custom login module that obtains passwords and roles
 *  for a user from a JNDI lookup.
 *
 *  @author Scott.Stark@jboss.org
 *  @version $Revision: 1.12 $
*/
public class JndiUserAndPass
    extends UsernamePasswordLoginModule
{
    /** The JNDI name to the context that handles the password/username
lookup */
```

```
    private String userPathPrefix;
    /** The JNDI name to the context that handles the roles/ username lookup
*/
    private String rolesPathPrefix;

    /**
     * Override to obtain the userPathPrefix and rolesPathPrefix options.
     */
    public void initialize(Subject subject, CallbackHandler callbackHandler,
                           Map sharedState, Map options)
    {
        super.initialize(subject, callbackHandler, sharedState, options);
        userPathPrefix = (String) options.get("userPathPrefix");
        rolesPathPrefix = (String) options.get("rolesPathPrefix");
    }

    /**
     *  Get the roles the current user belongs to by querying the
     * rolesPathPrefix + '/' + super.getUsername() JNDI location.
     */
    protected Group[] getRoleSets() throws LoginException
    {
        try {
            InitialContext ctx = new InitialContext();
            String rolesPath = rolesPathPrefix + '/' + super.getUsername();

            String[] roles = (String[]) ctx.lookup(rolesPath);
            Group[] groups = {new SimpleGroup("Roles")};
            log.info("Getting roles for user="+super.getUsername());
            for(int r = 0; r < roles.length; r ++) {
                SimplePrincipal role = new SimplePrincipal(roles[r]);
                log.info("Found role="+roles[r]);
                groups[0].addMember(role);
            }
            return groups;
        } catch(NamingException e) {
            log.error("Failed to obtain groups for
                        user="+super.getUsername(), e);
            throw new LoginException(e.toString(true));
        }
    }

    /**
     * Get the password of the current user by querying the
     * userPathPrefix + '/' + super.getUsername() JNDI location.
     */
    protected String getUsersPassword()
        throws LoginException
    {
        try {
            InitialContext ctx = new InitialContext();
            String userPath = userPathPrefix + '/' + super.getUsername();
            log.info("Getting password for user="+super.getUsername());
            String passwd = (String) ctx.lookup(userPath);
            log.info("Found password="+passwd);
            return passwd;
```

```
        } catch(NamingException e) {
            log.error("Failed to obtain password for
                        user="+super.getUsername(), e);
            throw new LoginException(e.toString(true));
        }
    }
}
```

Example 8.9. A JndiUserAndPass custom login module

The details of the JNDI store are found in the `org.jboss.book.security.ex2.service.JndiStore` MBean. This service binds an `ObjectFactory` that returns a `javax.naming.Context` proxy into JNDI. The proxy handles lookup operations done against it by checking the prefix of the lookup name against `password` and `roles`. When the name begins with `password`, a user's password is being requested. When the name begins with `roles` the user's roles are being requested. The example implementation always returns a password of `theduke` and an array of roles names equal to `{"TheDuke", "Echo"}` regardless of what the username is. You can experiment with other implementations as you wish.

The example code includes a simple session bean for testing the custom login module. To build, deploy and run the example, execute the following command in the examples directory.

```
[examples]$ ant -Dchap=security -Dex=2 run-example
...
run-example2:
     [echo] Waiting for 5 seconds for deploy...
     [java] [INFO,ExClient] Login with username=jduke, password=theduke
     [java] [INFO,ExClient] Looking up EchoBean2
     [java] [INFO,ExClient] Created Echo
     [java] [INFO,ExClient] Echo.echo('Hello') = Hello
```

The choice of using the `JndiUserAndPass` custom login module for the server side authentication of the user is determined by the login configuration for the example security domain. The EJB JAR `META-INF/jboss.xml` descriptor sets the security domain

```
<?xml version="1.0"?>
<jboss>
    <security-domain>java:/jaas/security-ex2</security-domain>
</jboss>
```

The SAR `META-INF/login-config.xml` descriptor defines the login module configuration.

```
<application-policy name = "security-ex2">
    <authentication>
        <login-module code="org.jboss.book.security.ex2.JndiUserAndPass"
```

```
                        flag="required">
            <module-option name =
"userPathPrefix">/security/store/password</module-option>
            <module-option name =
"rolesPathPrefix">/security/store/roles</module-option>
        </login-module>
    </authentication>
</application-policy>
```

6. The Secure Remote Password (SRP) Protocol

The SRP protocol is an implementation of a public key exchange handshake described in the Internet standards working group request for comments 2945(RFC2945). The RFC2945 abstract states:

This document describes a cryptographically strong network authentication mechanism known as the Secure Remote Password (SRP) protocol. This mechanism is suitable for negotiating secure connections using a user-supplied password, while eliminating the security problems traditionally associated with reusable passwords. This system also performs a secure key exchange in the process of authentication, allowing security layers (privacy and/or integrity protection) to be enabled during the session. Trusted key servers and certificate infrastructures are not required, and clients are not required to store or manage any long-term keys. SRP offers both security and deployment advantages over existing challenge-response techniques, making it an ideal drop-in replacement where secure password authentication is needed.

Note: The complete RFC2945 specification can be obtained from *http://www.rfc-editor.org/rfc.html*. Additional information on the SRP algorithm and its history can be found at *http://www-cs-students.stanford.edu/~tjw/srp/*.

SRP is similar in concept and security to other public key exchange algorithms, such as Diffie-Hellman and RSA. SRP is based on simple string passwords in a way that does not require a clear text password to exist on the server. This is in contrast to other public key-based algorithms that require client certificates and the corresponding certificate management infrastructure.

Algorithms like Diffie-Hellman and RSA are known as public key exchange algorithms. The concept of public key algorithms is that you have two keys, one public that is available to everyone, and one that is private and known only to you. When someone wants to send encrypted information to you, then encrpyt the information using your public key. Only you are able to decrypt the information using your private key. Contrast this with the more traditional shared password based encryption schemes that require the sender and receiver to know the shared password. Public key algorithms eliminate the need to share passwords.

The JBossSX framework includes an implementation of SRP that consists of the following elements:

- An implementation of the SRP handshake protocol that is independent of any particular client/server protocol
- An RMI implementation of the handshake protocol as the default client/server SRP implementation
- A client side JAAS `LoginModule` implementation that uses the RMI implementation for use in authenticating clients in a secure fashion
- A JMX MBean for managing the RMI server implementation. The MBean allows the RMI server implementation to be plugged into a JMX framework and externalizes the configuration of the verification information store. It also establishes an authentication cache that is bound into the JBoss server JNDI namespace.
- A server side JAAS `LoginModule` implementation that uses the authentication cache managed by the SRP JMX MBean.

Figure 8.14, "The JBossSX components of the SRP client-server framework." gives a diagram of the key components involved in the JBossSX implementation of the SRP client/server framework.

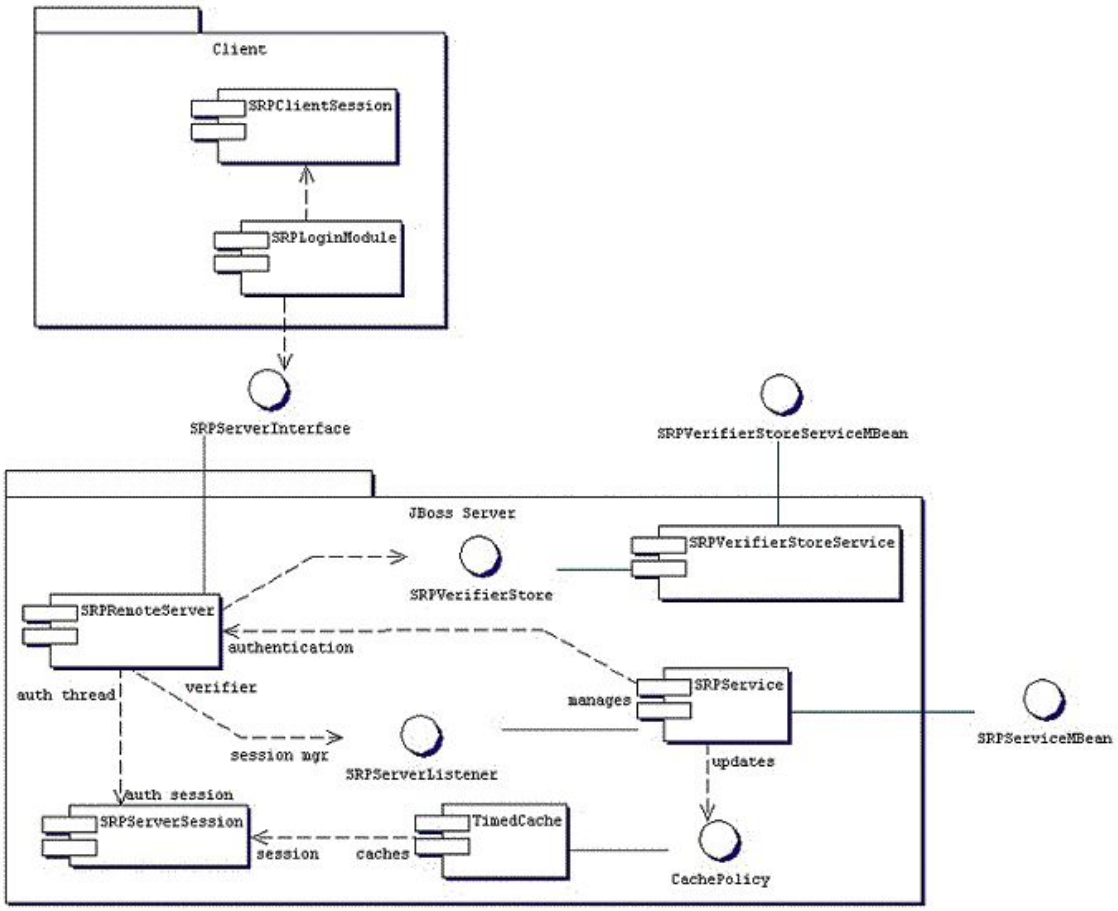


Figure 8.14. The JBossSX components of the SRP client-server

framework.

On the client side, SRP shows up as a custom JAAS `LoginModule` implementation that communicates to the authentication server through an `org.jboss.security.srp.SRPServerInterface` proxy. A client enables authentication using SRP by creating a login configuration entry that includes the `org.jboss.security.srp.jaas.SRPLoginModule`. This module supports the following configuration options:

- **principalClassName**: This option is no longer supported. The principal class is now always `org.jboss.security.srp.jaas.SRPPrincipal`.
- **srpServerJndiName**: The JNDI name of the `SRPServerInterface` object to use for communicating with the SRP authentication server. If both `srpServerJndiName` and `srpServerRmiUrl` options are specified, the `srpServerJndiName` is tried before `srpServerRmiUrl`.
- **srpServerRmiUrl**: The RMI protocol URL string for the location of the `SRPServerInterface` proxy to use for communicating with the SRP authentication server.
- **externalRandomA**: A true/false flag indicating if the random component of the client public key A should come from the user callback. This can be used to input a strong cryptographic random number coming from a hardware token for example.
- **hasAuxChallenge**: A true/false flag indicating that a string will be sent to the server as an additional challenge for the server to validate. If the client session supports an encryption cipher then a temporary cipher will be created using the session private key and the challenge object sent as a `javax.crypto.SealedObject`.
- **multipleSessions**: a true/false flag indicating if a given client may have multiple SRP login sessions active simultaneously.

Any other options passed in that do not match one of the previous named options is treated as a JNDI property to use for the environment passed to the `InitialContext` constructor. This is useful if the SRP server interface is not available from the default `InitialContext`.

The `SRPLoginModule` needs to be configured along with the standard `ClientLoginModule` to allow the SRP authentication credentials to be used for validation of access to security J2EE components. An example login configuration entry that demonstrates such a setup is:

```
srp {
    org.jboss.security.srp.jaas.SRPLoginModule required
    srpServerJndiName="SRPServerInterface"
    ;

    org.jboss.security.ClientLoginModule required
    password-stacking="useFirstPass"
```

```
    ;
};
```

On the JBoss server side, there are two MBeans that manage the objects that collectively make up the SRP server. The primary service is the `org.jboss.security.srp.SRPService` MBean, and it is responsible for exposing an RMI accessible version of the SRPServerInterface as well as updating the SRP authentication session cache. The configurable SRPService MBean attributes include the following:

- **JndiName**: The JNDI name from which the SRPServerInterface proxy should be available. This is the location where the `SRPService` binds the serializable dynamic proxy to the `SRPServerInterface`. If not specified it defaults to `srp/SRPServerInterface`.
- **VerifierSourceJndiName**: The JNDI name of the `SRPVerifierSource` implementation that should be used by the `SRPService`. If not set it defaults to `srp/DefaultVerifierSource`.
- **AuthenticationCacheJndiName**: The JNDI name under which the authentication `org.jboss.util.CachePolicy` implementation to be used for caching authentication information is bound. The SRP session cache is made available for use through this binding. If not specified it defaults to `srp/AuthenticationCache`.
- **ServerPort**: RMI port for the `SRPRemoteServerInterface`. If not specified it defaults to 10099.
- **ClientSocketFactory**: An optional custom `java.rmi.server.RMIClientSocketFactory` implementation class name used during the export of the `SRPServerInterface`. If not specified the default `RMIClientSocketFactory` is used.
- **ServerSocketFactory**: An optional custom `java.rmi.server.RMIServerSocketFactory` implementation class name used during the export of the `SRPServerInterface`. If not specified the default `RMIServerSocketFactory` is used.
- **AuthenticationCacheTimeout**: Specifies the timed cache policy timeout in seconds. If not specified this defaults to 1800 seconds(30 minutes).
- **AuthenticationCacheResolution**: Specifies the timed cache policy resolution in seconds. This controls the interval between checks for timeouts. If not specified this defaults to 60 seconds(1 minute).
- **RequireAuxChallenge**: Set if the client must supply an auxiliary challenge as part of the verify phase. This gives control over whether the `SRPLoginModule` configuration used by the client must have the `useAuxChallenge` option enabled.
- **OverwriteSessions**: A flag indicating if a successful user auth for an existing session should overwrite the current session. This controls the behavior of the server SRP session cache when clients have not enabled the multiple session per user mode. The default is false meaning that the second attempt by a user to authentication will succeed, but the resulting

SRP session will not overwrite the previous SRP session state.

The one input setting is the `VerifierSourceJndiName` attribute. This is the location of the SRP password information store implementation that must be provided and made available through JNDI. The `org.jboss.security.srp SRPVerifierStoreService` is an example MBean service that binds an implementation of the `SRPVerifierStore` interface that uses a file of serialized objects as the persistent store. Although not realistic for a production environment, it does allow for testing of the SRP protocol and provides an example of the requirements for an `SRPVerifierStore` service. The configurable `SRPVerifierStoreService` MBean attributes include the following:

- **JndiName**: The JNDI name from which the `SRPVerifierStore` implementation should be available. If not specified it defaults to `srp/DefaultVerifierSource`.
- **StoreFile**: The location of the user password verifier serialized object store file. This can be either a URL or a resource name to be found in the classpath. If not specified it defaults to `SRPVerifierStore.ser`.

The `SRPVerifierStoreService` MBean also supports `addUser` and `delUser` operations for addition and deletion of users. The signatures are:

```
public void addUser(String username, String password) throws IOException;
public void delUser(String username) throws IOException;
```

An example configuration of these services is presented in *Example 8.10, "The SRPVerifierStore interface"*.

6.1. Providing Password Information for SRP

The default implementation of the `SRPVerifierStore` interface is not likely to be suitable for your production security environment as it requires all password hash information to be available as a file of serialized objects. You need to provide an MBean service that provides an implementation of the `SRPVerifierStore` interface that integrates with your existing security information stores. The `SRPVerifierStore` interface is shown in.

```
package org.jboss.security.srp;

import java.io.IOException;
import java.io.Serializable;
import java.security.KeyException;

public interface SRPVerifierStore
{
    public static class VerifierInfo implements Serializable
    {
        /**
         * The username the information applies to. Perhaps redundant
```

```
          * but it makes the object self contained.
          */
         public String username;

         /** The SRP password verifier hash */
         public byte[] verifier;
         /** The random password salt originally used to verify the password
*/
         public byte[] salt;
         /** The SRP algorithm primitive generator */
         public byte[] g;
         /** The algorithm safe-prime modulus */
         public byte[] N;
     }

     /**
      *  Get the indicated user's password verifier information.
      */
     public VerifierInfo getUserVerifier(String username)
         throws KeyException, IOException;
     /**
      *  Set the indicated users' password verifier information. This
      *  is equivalent to changing a user's password and should
      *  generally invalidate any existing SRP sessions and caches.
      */
     public void setUserVerifier(String username, VerifierInfo info)
         throws IOException;

     /**
      * Verify an optional auxiliary challenge sent from the client to
      * the server.  The auxChallenge object will have been decrypted
      * if it was sent encrypted from the client. An example of a
      * auxiliary challenge would be the validation of a hardware token
      * (SafeWord, SecureID, iButton) that the server validates to
      * further strengthen the SRP password exchange.
      */
      public void verifyUserChallenge(String username, Object auxChallenge)
          throws SecurityException;
}
```

The primary function of a `SRPVerifierStore` implementation is to provide access to the `SRPVerifierStore.VerifierInfo` object for a given username. The `getUserVerifier(String)` method is called by the `SRPService` at that start of a user SRP session to obtain the parameters needed by the SRP algorithm. The elements of the `VerifierInfo` objects are:

- **username**: The user's name or id used to login.
- **verifier**: This is the one-way hash of the password or PIN the user enters as proof of their identity. The `org.jboss.security.Util` class has a `calculateVerifier` method that performs that password hashing algorithm. The output password `H(salt | H(username |`

`':' | password))` as defined by RFC2945. Here `H` is the SHA secure hash function. The username is converted from a string to a `byte[]` using the UTF-8 encoding.

- **salt**: This is a random number used to increase the difficulty of a brute force dictionary attack on the verifier password database in the event that the database is compromised. It is a value that should be generated from a cryptographically strong random number algorithm when the user's existing clear-text password is hashed.
- **g**: The SRP algorithm primitive generator. In general this can be a well known fixed parameter rather than a per-user setting. The `org.jboss.security.srp.SRPConf` utility class provides several settings for g including a good default which can obtained via `SRPConf.getDefaultParams().g()`.
- **N**: The SRP algorithm safe-prime modulus. In general this can be a well known fixed parameter rather than a per-user setting. The `org.jboss.security.srp.SRPConf` utility class provides several settings for `N` including a good default which can obtained via `SRPConf.getDefaultParams().N()`.

So, step 1 of integrating your existing password store is the creation of a hashed version of the password information. If your passwords are already store in an irreversible hashed form, then this can only be done on a per-user basis as part of an upgrade procedure for example. Note that the `setUserVerifier(String, VerifierInfo)` method is not used by the current SRPSerivce and may be implemented as no-op method, or even one that throws an exception stating that the store is read-only.

Step 2 is the creation of the custom `SRPVerifierStore` interface implementation that knows how to obtain the `VerifierInfo` from the store you created in step 1. The `verifyUserChallenge(String, Object)` method of the interface is only called if the client `SRPLoginModule` configuration specifies the `hasAuxChallenge` option. This can be used to integrate existing hardware token based schemes like SafeWord or Radius into the SRP algorithm.

Step 3 is the creation of an MBean that makes the step 2 implementation of the `SRPVerifierStore` interface available via JNDI, and exposes any configurable parameters you need. In addition to the default `org.jboss.security.srp.SRPVerifierStoreService` example, the SRP example presented later in this chapter provides a Java properties file based `SRPVerifierStore` implementation. Between the two examples you should have enough to integrate your security store.

Example 8.10. The SRPVerifierStore interface

6.2. Inside of the SRP algorithm

The appeal of the SRP algorithm is that is allows for mutual authentication of client and server using simple text passwords without a secure communication channel. You might be wondering how this is done. If you want the complete details and theory behind the algorithm, refer to the SRP references mentioned in a note earlier. There are six steps that are performed to complete

authentication:

1. The client side `SRPLoginModule` retrieves the SRPServerInterface instance for the remote authentication server from the naming service.

2. The client side `SRPLoginModule` next requests the SRP parameters associated with the username attempting the login. There are a number of parameters involved in the SRP algorithm that must be chosen when the user password is first transformed into the verifier form used by the SRP algorithm. Rather than hard-coding the parameters (which could be done with minimal security risk), the JBossSX implementation allows a user to retrieve this information as part of the exchange protocol. The `getSRPParameters(username)` call retrieves the SRP parameters for the given username.

3. The client side `SRPLoginModule` begins an SRP session by creating an `SRPClientSession` object using the login username, clear-text password, and SRP parameters obtained from step 2. The client then creates a random number A that will be used to build the private SRP session key. The client then initializes the server side of the SRP session by invoking the `SRPServerInterface.init` method and passes in the username and client generated random number `A`. The server returns its own random number `B`. This step corresponds to the exchange of public keys.

4. The client side `SRPLoginModule` obtains the private SRP session key that has been generated as a result of the previous messages exchanges. This is saved as a private credential in the login `Subject`. The server challenge response `M2` from step 4 is verified by invoking the `SRPClientSession.verify` method. If this succeeds, mutual authentication of the client to server, and server to client have been completed. The client side `SRPLoginModule` next creates a challenge `M1` to the server by invoking `SRPClientSession.response` method passing the server random number `B` as an argument. This challenge is sent to the server via the `SRPServerInterface.verify` method and server's response is saved as `M2`. This step corresponds to an exchange of challenges. At this point the server has verified that the user is who they say they are.

5. The client side `SRPLoginModule` saves the login username and `M1` challenge into the `LoginModule` sharedState map. This is used as the Principal name and credentials by the standard JBoss `ClientLoginModule`. The `M1` challenge is used in place of the password as proof of identity on any method invocations on J2EE components. The `M1` challenge is a cryptographically strong hash associated with the SRP session. Its interception via a third partly cannot be used to obtain the user's password.

6. At the end of this authentication protocol, the SRPServerSession has been placed into the SRPService authentication cache for subsequent use by the `SRPCacheLoginModule`.

Although SRP has many interesting properties, it is still an evolving component in the JBossSX framework and has some limitations of which you should be aware. Issues of note include the following:

- Because of how JBoss detaches the method transport protocol from the component container where authentication is performed, an unauthorized user could snoop the SRP `M1` challenge and effectively use the challenge to make requests as the associated username. Custom interceptors that encrypt the challenge using the SRP session key can be used to prevent this issue.
- The SRPService maintains a cache of SRP sessions that time out after a configurable period. Once they time out, any subsequent J2EE component access will fail because there is currently no mechanism for transparently renegotiating the SRP authentication credentials. You must either set the authentication cache timeout very long (up to 2,147,483,647 seconds, or approximately 68 years), or handle re-authentication in your code on failure.
- By default there can only be one SRP session for a given username. Because the negotiated SRP session produces a private session key that can be used for encryption/decryption between the client and server, the session is effectively a stateful one. JBoss supports for multiple SRP sessions per user, but you cannot encrypt data with one session key and then decrypt it with another.

To use end-to-end SRP authentication for J2EE component calls, you need to configure the security domain under which the components are secured to use the `org.jboss.security.srp.jaas.SRPCacheLoginModule`. The `SRPCacheLoginModule` has a single configuration option named `cacheJndiName` that sets the JNDI location of the SRP authentication `CachePolicy` instance. This must correspond to the `AuthenticationCacheJndiName` attribute value of the `SRPService` MBean. The `SRPCacheLoginModule` authenticates user credentials by obtaining the client challenge from the `SRPServerSession` object in the authentication cache and comparing this to the challenge passed as the user credentials. *Figure 8.15, “A sequence diagram illustrating the interaction of the SRPCacheLoginModule with the SRP session cache.”* illustrates the operation of the SRPCacheLoginModule.login method implementation.

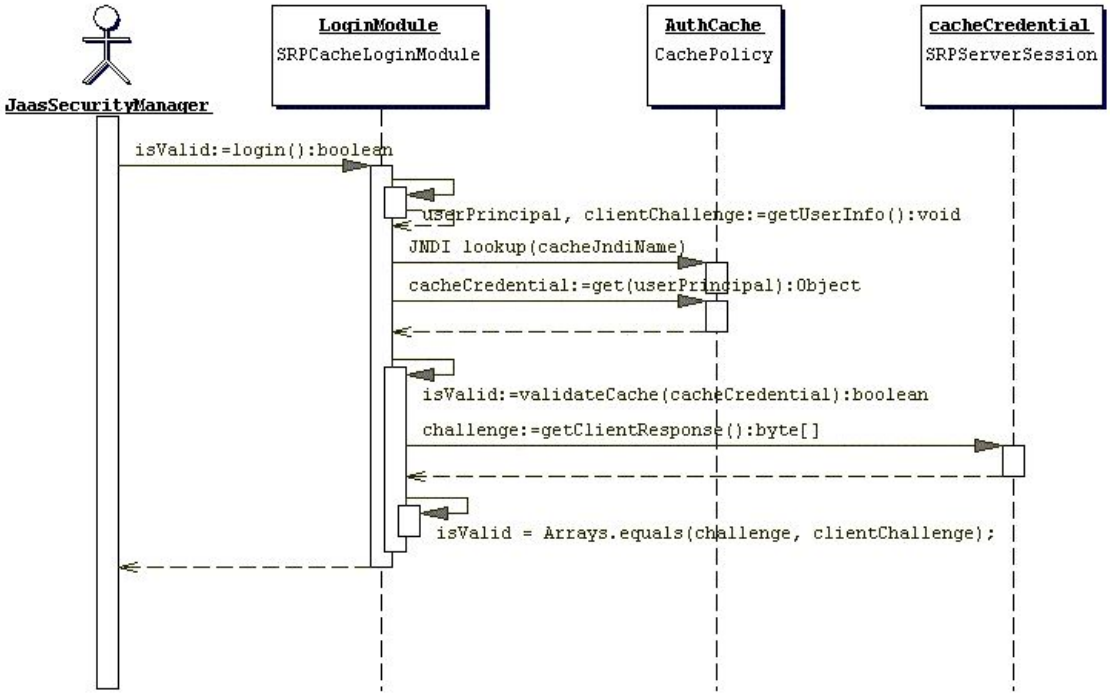


Figure 8.15. A sequence diagram illustrating the interaction of the SRPCacheLoginModule with the SRP session cache.

6.2.1. An SRP example

We have covered quite a bit of material on SRP and now its time to demonstrate SRP in practice with an example. The example demonstrates client side authentication of the user via SRP as well as subsequent secured access to a simple EJB using the SRP session challenge as the user credential. The test code deploys an EJB JAR that includes a SAR for the configuration of the server side login module configuration and SRP services. As in the previous examples we will dynamically install the server side login module configuration using the `SecurityConfig` MBean. In this example we also use a custom implementation of the `SRPVerifierStore` interface that uses an in memory store that is seeded from a Java properties file rather than a serialized object store as used by the `SRPVerifierStoreService`. This custom service is `org.jboss.book.security.ex3.service.PropertiesVerifierStore`. The following shows the contents of the JAR that contains the example EJB and SRP services.

```
[examples]$ jar tf output/security/security-ex3.jar
META-INF/MANIFEST.MF
META-INF/ejb-jar.xml
META-INF/jboss.xml
org/jboss/book/security/ex3/Echo.class
org/jboss/book/security/ex3/EchoBean.class
org/jboss/book/security/ex3/EchoHome.class
roles.properties
users.properties
security-ex3.sar
```

The key SRP related items in this example are the SRP MBean services configuration, and the SRP login module configurations. The `jboss-service.xml` descriptor of the `security-ex3.sar` is given in *Example 8.11, "The security-ex3.sar jboss-service.xml descriptor for the SRP services"*, while *Example 8.12, "The client side standard JAAS configuration"* and *Example 8.13, "The server side XMLLoginConfig configuration"* give the example client side and server side login module configurations.

```
<server>
    <!-- The custom JAAS login configuration that installs
         a Configuration capable of dynamically updating the
         config settings -->

    <mbean code="org.jboss.book.security.service.SecurityConfig"
           name="jboss.docs.security:service=LoginConfig-EX3">
        <attribute name="AuthConfig">META-INF/login-config.xml</attribute>
        <attribute
name="SecurityConfigName">jboss.security:name=SecurityConfig</attribute>
    </mbean>

    <!-- The SRP service that provides the SRP RMI server and server side
         authentication cache -->
    <mbean code="org.jboss.security.srp.SRPService"
           name="jboss.docs.security:service=SRPService">
        <attribute
name="VerifierSourceJndiName">srp-test/security-ex3</attribute>
        <attribute name="JndiName">srp-test/SRPServerInterface</attribute>
        <attribute
name="AuthenticationCacheJndiName">srp-test/AuthenticationCache</attribute>
        <attribute name="ServerPort">0</attribute>
<depends>jboss.docs.security:service=PropertiesVerifierStore</depends>
    </mbean>

    <!-- The SRP store handler service that provides the user password
verifier
         information -->
    <mbean code="org.jboss.security.ex3.service.PropertiesVerifierStore"
           name="jboss.docs.security:service=PropertiesVerifierStore">
        <attribute name="JndiName">srp-test/security-ex3</attribute>
    </mbean>
</server>
```

Example 8.11. The security-ex3.sar jboss-service.xml descriptor for the SRP services

```
srp {
    org.jboss.security.srp.jaas.SRPLoginModule required
    srpServerJndiName="srp-test/SRPServerInterface"
```

```
    ;

    org.jboss.security.ClientLoginModule required
    password-stacking="useFirstPass"
    ;
};
```

Example 8.12. The client side standard JAAS configuration

```
<application-policy name="security-ex3">
    <authentication>
        <login-module code="org.jboss.security.srp.jaas.SRPCacheLoginModule"
                      flag = "required">
            <module-option
name="cacheJndiName">srp-test/AuthenticationCache</module-option>
        </login-module>
        <login-module
code="org.jboss.security.auth.spi.UsersRolesLoginModule"
                      flag = "required">
            <module-option
name="password-stacking">useFirstPass</module-option>
        </login-module>
    </authentication>
</application-policy>
```

Example 8.13. The server side XMLLoginConfig configuration

The example services are the `ServiceConfig` and the `PropertiesVerifierStore` and `SRPService` MBeans. Note that the `JndiName` attribute of the `PropertiesVerifierStore` is equal to the `VerifierSourceJndiName` attribute of the `SRPService`, and that the `SRPService` depends on the `PropertiesVerifierStore`. This is required because the `SRPService` needs an implementation of the `SRPVerifierStore` interface for accessing user password verification information.

The client side login module configuration makes use of the `SRPLoginModule` with a `srpServerJndiName` option value that corresponds to the JBoss server component `SRPService` JndiName attribute value(`srp-test/SRPServerInterface`). Also needed is the `ClientLoginModule` configured with the `password-stacking="useFirstPass"` value to propagate the user authentication credentials generated by the `SRPLoginModule` to the EJB invocation layer.

There are two issues to note about the server side login module configuration. First, note the `cacheJndiName=srp-test/AuthenticationCache` configuration option tells the `SRPCacheLoginModule` the location of the `CachePolicy` that contains the `SRPServerSession`

for users who have authenticated against the `SRPService`. This value corresponds to the `SRPServiceAuthenticationCacheJndiName` attribute value. Second, the configuration includes a `UsersRolesLoginModule` with the `password-stacking=useFirstPass` configuration option. It is required to use a second login module with the `SRPCacheLoginModule` because SRP is only an authentication technology. A second login module needs to be configured that accepts the authentication credentials validated by the `SRPCacheLoginModule` to set the principal's roles that determines the principal's permissions. The `UsersRolesLoginModule` is augmenting the SRP authentication with properties file based authorization. The user's roles are coming the `roles.properties` file included in the EJB JAR.

Now, run the example 3 client by executing the following command from the book examples directory:

```
[examples]$ ant -Dchap=security -Dex=3 run-example
...
run-example3:
     [echo] Waiting for 5 seconds for deploy...
     [java] Logging in using the 'srp' configuration
     [java] Created Echo
     [java] Echo.echo()#1 = This is call 1
     [java] Echo.echo()#2 = This is call 2
```

In the `examples/logs` directory you will find a file called `ex3-trace.log`. This is a detailed trace of the client side of the SRP algorithm. The traces show step-by-step the construction of the public keys, challenges, session key and verification.

Note that the client has taken a long time to run relative to the other simple examples. The reason for this is the construction of the client's public key. This involves the creation of a cryptographically strong random number, and this process takes quite a bit of time the first time it occurs. If you were to log out and log in again within the same VM, the process would be much faster. Also note that `Echo.echo()#2` fails with an authentication exception. The client code sleeps for 15 seconds after making the first call to demonstrate the behavior of the `SRPService` cache expiration. The `SRPService` cache policy timeout has been set to a mere 10 seconds to force this issue. As stated earlier, you need to make the cache timeout very long, or handle re-authentication on failure.

7. Running JBoss with a Java 2 security manager

By default the JBoss server does not start with a Java 2 security manager. If you want to restrict privileges of code using Java 2 permissions you need to configure the JBoss server to run under a security manager. This is done by configuring the Java VM options in the `run.bat` or `run.sh` scripts in the JBoss server distribution bin directory. The two required VM options are as follows:

- **java.security.manager**: This is used without any value to specify that the default security manager should be used. This is the preferred security manager. You can also pass a value to the `java.security.manager` option to specify a custom security manager implementation.

The value must be the fully qualified class name of a subclass of `java.lang.SecurityManager`. This form specifies that the policy file should augment the default security policy as configured by the VM installation.

- **java.security.policy**: This is used to specify the policy file that will augment the default security policy information for the VM. This option takes two forms: `java.security.policy=policyFileURL` and `java.security.policy==policyFileURL`. The first form specifies that the policy file should augment the default security policy as configured by the VM installation. The second form specifies that only the indicated policy file should be used. The `policyFileURL` value can be any URL for which a protocol handler exists, or a file path specification.

Both the `run.bat` and `run.sh` start scripts reference an JAVA_OPTS variable which you can use to set the security manager properties.

Enabling Java 2 security is the easy part. The difficult part of Java 2 security is establishing the allowed permissions. If you look at the `server.policy` file that is contained in the default configuration file set, you'll see that it contains the following permission grant statement:

```
grant {
    // Allow everything for now
    permission java.security.AllPermission;
};
```

This effectively disables security permission checking for all code as it says any code can do anything, which is not a reasonable default. What is a reasonable set of permissions is entirely up to you.

The current set of JBoss specific `java.lang.RuntimePermissions` that are required include:

TargetName	What the permission allows	Risks
org.jboss.security.SecurityAssociation.getPrincipalInfo	Access to the org.jboss.security.SecurityAssociation getPrincipal() and getCredentials() methods.	The ability to see the current thread caller and credentials.
org.jboss.security.SecurityAssociation.setPrincipalInfo	Access to the org.jboss.security.SecurityAssociation setPrincipal() and setCredentials() methods.	The ability to set the current thread caller and credentials.
org.jboss.security.SecurityAssociation.setServer	Access to the org.jboss.security.SecurityAssociation setServer method.	The ability to enable or disable multithread storage of the caller principal and credential.
org.jboss.security.SecurityAssociation.setRunAsRole	Access to the org.jboss.security.SecurityAssociation pushRunAsRole and	The ability to change the current caller run-as role principal.

TargetName	What the permission allows	Risks
	popRunAsRole methods.	

To conclude this discussion, here is a little-known tidbit on debugging security policy settings. There are various debugging flag that you can set to determine how the security manager is using your security policy file as well as what policy files are contributing permissions. Running the VM as follows shows the possible debugging flag settings:

```
[bin]$ java -Djava.security.debug=help

all           turn on all debugging
access        print all checkPermission results
combiner      SubjectDomainCombiner debugging
jar           jar verification
logincontext  login context results
policy        loading and granting
provider      security provider debugging
scl           permissions SecureClassLoader assigns

The following can be used with access:

stack     include stack trace
domain    dumps all domains in context
failure   before throwing exception, dump stack
          and domain that didn't have permission

Note: Separate multiple options with a comma
```

Running with `-Djava.security.debug=all` provides the most output, but the output volume is torrential. This might be a good place to start if you don't understand a given security failure at all. A less verbose setting that helps debug permission failures is to use `-Djava.security.debug=access,failure`. This is still relatively verbose, but not nearly as bad as the all mode as the security domain information is only displayed on access failures.

8. Using SSL with JBoss using JSSE

JBoss uses JSEE, the Java Secure Socket Extension (JSSE), for SSL. JSSE is bundled with JDK 1.4. To get started with JSSE you need a public key/private key pair in the form of an X509 certificate for use by the SSL server sockets. For the purpose of this example we have created a self-signed certificate using the JDK keytool and included the resulting keystore file, `example.keystore`. It was created using the following command and input:

```
 keytool -genkey -keystore example.keystore -storepass rmi+ssl -keypass
rmi+ssl -keyalg RSA -alias example -validity 3650 -dname
"cn=example,ou=admin book,dc=jboss,dc=org"
```

This produces a keystore file called `example.keystore`. A keystore is a database of security

keys. There are two different types of entries in a keystore:

- **key entries**: each entry holds very sensitive cryptographic key information, which is stored in a protected format to prevent unauthorized access. Typically, a key stored in this type of entry is a secret key, or a private key accompanied by the certificate chain for the corresponding public key. The `keytool` and `jarsigner` tools only handle the later type of entry, that is private keys and their associated certificate chains.
- **trusted certificate entries**: each entry contains a single public key certificate belonging to another party. It is called a trusted certificate because the keystore owner trusts that the public key in the certificate indeed belongs to the identity identified by the subject (owner) of the certificate. The issuer of the certificate vouches for this, by signing the certificate.

Listing the `src/main/org/jboss/book/security/example.keystore` examples file contents using the keytool shows one self-signed certificate:

```
[examples]$ keytool -list -v -keystore
src/main/org/jboss/book/security/example.keystore
Enter keystore password:  rmi+ssl

Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: example
Creation date: Oct 31, 2006
Entry type: keyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=example, OU=admin book, DC=jboss, DC=org
Issuer: CN=example, OU=admin book, DC=jboss, DC=org
Serial number: 45481c1b
Valid from: Tue Oct 31 22:01:31 CST 2006 until: Fri Oct 28 23:01:31 CDT 2016
Certificate fingerprints:
         MD5:  C2:CA:CA:D3:00:71:3E:32:CB:B3:C8:A8:4E:68:9A:BB
         SHA1: A6:44:EF:66:2A:49:14:B0:A4:14:74:8B:64:61:E4:E6:AF:E3:70:41


*******************************************
*******************************************
```

With JSSE working and a keystore with the certificate you will use for the JBoss server, your are ready to configure JBoss to use SSL for EJB access. This is done by configuring the EJB invoker RMI socket factories. The JBossSX framework includes implementations of the `java.rmi.server.RMIServerSocketFactory` and `java.rmi.server.RMIClientSocketFactory` interfaces that enable the use of RMI over SSL encrypted sockets. The implementation classes are `org.jboss.security.ssl.RMISSLServerSocketFactory` and `org.jboss.security.ssl.RMISSLClientSocketFactory` respectively. There are two steps to

enable the use of SSL for RMI access to EJBs. The first is to enable the use of a keystore as the database for the SSL server certificate, which is done by configuring an `org.jboss.security.plugins.JaasSecurityDomain` MBean. The `jboss-service.xml` descriptor in the `book/security/ex4` directory includes the `JaasSecurityDomain` definition shown in *Example 8.14, "A sample JaasSecurityDomain config for RMI/SSL"*.

```
<!-- The SSL domain setup -->
<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
       name="jboss.security:service=JaasSecurityDomain,domain=RMI+SSL">
    <constructor>
        <arg type="java.lang.String" value="RMI+SSL"/>
    </constructor>
    <attribute name="KeyStoreURL">example.keystore</attribute>
    <attribute name="KeyStorePass">rmi+ssl</attribute>
</mbean>
```

Example 8.14. A sample JaasSecurityDomain config for RMI/SSL

The `JaasSecurityDomain` is a subclass of the standard `JaasSecurityManager` class that adds the notions of a keystore as well JSSE `KeyManagerFactory` and `TrustManagerFactory` access. It extends the basic security manager to allow support for SSL and other cryptographic operations that require security keys. This configuration simply loads the example.keystore from the example 4 MBean SAR using the indicated password.

The second step is to define an EJB invoker configuration that uses the JBossSX RMI socket factories that support SSL. To do this you need to define a custom configuration for the `JRMPInvoker` we saw in *Chapter 14, EJBs on JBoss* as well as an EJB setup that makes use of this invoker. The top of the listing shows the `jboss-service.xml` descriptor that defines the custom `JRMPInovker`

```
<mbean code="org.jboss.invocation.jrmp.server.JRMPInvoker"
       name="jboss:service=invoker,type=jrmp,socketType=SSL">
    <attribute name="RMIObjectPort">14445</attribute>
    <attribute name="RMIClientSocketFactory">
        org.jboss.security.ssl.RMISSLClientSocketFactory
    </attribute>
    <attribute name="RMIServerSocketFactory">
        org.jboss.security.ssl.RMISSLServerSocketFactory
    </attribute>
    <attribute name="SecurityDomain">java:/jaas/RMI+SSL</attribute>
<depends>jboss.security:service=JaasSecurityDomain,domain=RMI+SSL</depends>
</mbean>
```

To set up an SSL invoker, we will create an invoker binding named `stateless-ssl-invoker` that uses our custom JRMPInvoker. We can declare the invoker binding and connect it to `EchoBean4` as shown in the following `jboss.xml` file.

```
<?xml version="1.0"?>
```

```
<jboss>
    <enterprise-beans>
        <session>
            <ejb-name>EchoBean4</ejb-name>
            <configuration-name>Standard Stateless
SessionBean</configuration-name>
            <invoker-bindings>
                <invoker>
                    <invoker-proxy-binding-name>
                        stateless-ssl-invoker
                    </invoker-proxy-binding-name>
                </invoker>
            </invoker-bindings>
        </session>
    </enterprise-beans>

    <invoker-proxy-bindings>
        <invoker-proxy-binding>
            <name>stateless-ssl-invoker</name>
<invoker-mbean>jboss:service=invoker,type=jrmp,socketType=SSL</invoker-mbean>
            <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-factory>
            <proxy-factory-config>
            <client-interceptors>
                <home>
<interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
                </home>
                <bean>
<interceptor>org.jboss.proxy.ejb.StatelessSessionInterceptor</interceptor>
<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
                </bean>
            </client-interceptors>
            </proxy-factory-config>
        </invoker-proxy-binding>
    </invoker-proxy-bindings>
</jboss>
```

The example 4 code is located under the `src/main/org/jboss/book/security/ex4` directory of the book examples. This is another simple stateless session bean with an echo method that returns its input argument. It is hard to tell when SSL is in use unless it fails, so we'll run the example 4 client in two different ways to demonstrate that the EJB deployment is in fact using SSL. Start the JBoss server using the default configuration and then run example 4b as follows:

```
[examples]$ ant -Dchap=security -Dex=4b run-example
...
run-example4b:
...
     [java] Exception in thread "main" java.rmi.ConnectIOException: error
during JRMP connect
ion establishment; nested exception is:
```

```
     [java]     javax.net.ssl.SSLHandshakeException:
sun.security.validator.ValidatorExceptio
n: No trusted certificate found
...
```

The resulting exception is expected, and is the purpose of the 4b version of the example. Note that the exception stack trace has been edited to fit into the book format, so expect some difference. The key item to notice about the exception is it clearly shows you are using the Sun JSSE classes to communicate with the JBoss EJB container. The exception is saying that the self-signed certificate you are using as the JBoss server certificate cannot be validated as signed by any of the default certificate authorities. This is expected because the default certificate authority keystore that ships with the JSSE package only includes well known certificate authorities such as VeriSign, Thawte, and RSA Data Security. To get the EJB client to accept your self-signed certificate as valid, you need to tell the JSSE classes to use your `example.keystore` as its truststore. A truststore is just a keystore that contains public key certificates used to sign other certificates. To do this, run example 4 using `-Dex=4` rather than `-Dex=4b` to pass the location of the correct truststore using the `javax.net.ssl.trustStore` system property:

```
[examples]$ ant -Dchap=security -Dex=4 run-example
...
run-example4:
        [copy] Copying 1 file to /tmp/jboss-4.2.0/server/production/deploy
        [echo] Waiting for 5 seconds for deploy...
...
        [java] Created Echo
        [java] Echo.echo()#1 = This is call 1
```

This time the only indication that an SSL socket is involved is because of the `SSL handshakeCompleted` message. This is coming from the `RMISSLClientSocketFactory` class as a debug level log message. If you did not have the client configured to print out log4j debug level messages, there would be no direct indication that SSL was involved. If you note the run times and the load on your system CPU, there definitely is a difference. SSL, like SRP, involves the use of cryptographically strong random numbers that take time to seed the first time they are used. This shows up as high CPU utilization and start up times.

One consequence of this is that if you are running on a system that is slower than the one used to run the examples for the book, such as when running example 4b, you may seen an exception similar to the following:

```
javax.naming.NameNotFoundException: EchoBean4 not bound
   at sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer
...
```

The problem is that the JBoss server has not finished deploying the example EJB in the time the client allowed. This is due to the initial setup time of the secure random number generator

used by the SSL server socket. If you see this issue, simply rerun the example again or increase the deployment wait time in the `build-security.xml` Ant script.

9. Configuring JBoss for use Behind a Firewall

JBoss comes with many socket based services that open listening ports. In this section we list the services that open ports that might need to be configured to work when accessing JBoss behind a firewall. The following table shows the ports, socket type, associated service for the services in the default configuration file set. *Table 8.2, "Additional ports in the all configuration"* shows the same information for the additional ports that exist in the all configuration file set.

Port	Type	Service
1098	TCP	`org.jboss.naming.NamingService`
1099	TCP	`org.jboss.naming.NamingService`
4444	TCP	`org.jboss.invocation.jrmp.server.JRMPInvoker`
4445	TCP	`org.jboss.invocation.pooled.server.PooledInvoker`
8009	TCP	`org.jboss.web.tomcat.tc4.EmbeddedTomcatService`
8080	TCP	`org.jboss.web.tomcat.tc4.EmbeddedTomcatService`
8083	TCP	`org.jboss.web.WebService`
8093	TCP	`org.jboss.mq.il.uil2.UILServerILService`

Table 8.1. The ports found in the default configuration

Port	Type	Service
1100	TCP	`org.jboss.ha.jndi.HANamingService`
1101	TCP	`org.jboss.ha.jndi.HANamingService`
1102	UDP	`org.jboss.ha.jndi.HANamingService`
1161	UDP	`org.jboss.jmx.adaptor.snmp.agent.SnmpAgentService`
1162	UDP	`org.jboss.jmx.adaptor.snmp.trapd.TrapdService`
3528	TCP	`org.jboss.invocation.iiop.IIOPInvoker`
4447	TCP	`org.jboss.invocation.jrmp.server.JRMPInvokerHA`
45566[a]	UDP	`org.jboss.ha.framework.server.ClusterPartition`

[a] Plus two additional anonymous UDP ports, one can be set using the `rcv_port`, and the other cannot be set.

Table 8.2. Additional ports in the all configuration

10. How to Secure the JBoss Server

JBoss comes with several admin access points that need to be secured or removed to prevent

unauthorized access to administrative functions in a deployment. This section describes the various admin services and how to secure them.

10.1. The JMX Console

The `jmx-console.war` found in the deploy directory provides an html view into the JMX microkernel. As such, it provides access to arbitrary admin type access like shutting down the server, stopping services, deploying new services, etc. It should either be secured like any other web application, or removed.

10.2. The Web Console

The `web-console.war` found in the `deploy/management` directory is another web application view into the JMX microkernel. This uses a combination of an applet and a HTML view and provides the same level of access to admin functionality as the `jmx-console.war`. As such, it should either be secured or removed. The `web-console.war` contains commented out templates for basic security in its `WEB-INF/web.xml` as well as commented out setup for a security domain in `WEB-INF/jboss-web.xml`.

10.3. The HTTP Invokers

The `http-invoker.sar` found in the deploy directory is a service that provides RMI/HTTP access for EJBs and the JNDI `Naming` service. This includes a servlet that processes posts of marshalled `org.jboss.invocation.Invocation` objects that represent invocations that should be dispatched onto the `MBeanServer`. Effectively this allows access to MBeans that support the detached invoker operation via HTTP since one could figure out how to format an appropriate HTTP post. To security this access point you would need to secure the `JMXInvokerServlet` servlet found in the `http-invoker.sar/invoker.war/WEB-INF/web.xml` descriptor. There is a secure mapping defined for the `/restricted/JMXInvokerServlet` path by default, one would simply have to remove the other paths and configure the `http-invoker` security domain setup in the `http-invoker.sar/invoker.war/WEB-INF/jboss-web.xml` descriptor.

10.4. The JMX Invoker

The `jmx-invoker-adaptor-server.sar` is a service that exposes the JMX MBeanServer interface via an RMI compatible interface using the RMI/JRMP detached invoker service. The only way for this service to be secured currently would be to switch the protocol to RMI/HTTP and secure the `http-invoker.sar` as described in the previous section. In the future this service will be deployed as an XMBean with a security interceptor that supports role based access checks.

Web Services

The biggest new feature of J2EE 1.4 is the ability of J2EE components to act both as web service providers and consumers. J2EE applications can expose a web service from the EJB tier using a stateless session bean or from the web tier using a plain Java object. Additionally,J2EE components have a standard way of declaring references to external web services.

1. JAX-RPC Service Endpoints

JAX-RPC service endpoints (JSEs) provide web services from the web tier. They take the form of a simple Java objects that masquerade as servlets. To show how simple they are, we'll jump right in with a trivial hello web service implementation class.

```
package org.jboss.ws.hello;

public class HelloPojo
{
    public String hello(String name)
    {
        return "Hello " + name + "!";
    }
}
```

There is nothing remarkable about `HelloPojo`. It doesn't implement any special interfaces nor does it need any methods besides the business methods it decides to provide. The `hello` method is the operation that we will expose as a web service, and it does nothing but respond with a friendly greeting to the person passed in.

That is our web service implementation. In addition to this, we need a service endpoint interface (SEI) that defines the interface of the web service. That is shown here as the `Hello` interface.

```
package org.jboss.ws.hello;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello
    extends Remote
{
    public String hello(String name)
        throws RemoteException;
}
```

The service endpoint interface is declared `Remote` and the methods must throw `RemoteException`. Beyond this, it is a simple expression of the interface to our web service. This is all the code we need to write to expose a J2EE web service. Deploying it, however, does require a few additional deployment descriptors.

Although a JSE doesn't bears any direct resemblance to a servlet, it is nonetheless deployed as a servlet in the `web.xml` file. We'll need to declare the web service implementation class as a servlet and provide a servlet mapping that will respond to the web service invocations. Here is the definition required to deploy the hello web service.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
xmlns="http://java.sun.com/xml/ns/"Whats_new_in_JBoss_4-J2EE_Certification_and_Standards_Com
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4">
    <servlet>
        <servlet-name>HelloWorldWS</servlet-name>
        <servlet-class>org.jboss.ws.hello.HelloPojo</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloWorldWS</servlet-name>
        <url-pattern>/Hello</url-pattern>
    </servlet-mapping>
</web-app>
```

The URL pattern in the servlet mapping is the only externally visible configuration element. It controls what URL the web service lives at. This will be primarily noticed as the location of the WSDL file for this service.

The `web.xml` file doesn't contain any web service related configuration. A new deployment descriptor, `webservices.xml`, is needed to instruct JBoss to treat this servlet as a web service and not as a normal servlet. We'll need two additional configuration files, a WSDL file and a JAX-RPC mapping file. All of these files can be generated using the `wstool` generator that ships with JBoss.

`wstool` can be run from from the command line or as an Ant task. The JBossWS guide explains in more detail how to run the tool. In both cases, wstool needs to be pointed to the code and to a configuration file which describes the files to generate and the endpoint to generate them for. Here is the configuration file for the hello web service.

```
<configuration xmlns="http://www.jboss.org/jbossws-tools">
    <java-wsdl>
        <service name="HelloService"
                 style="rpc"
                 endpoint="org.jboss.ws.hello.Hello"/>
        <namespaces target-namespace="http://hello.ws.jboss.org/"
                    type-namespace="http://hello.ws.jboss.org/types"/>
        <mapping file="jaxrpc-mapping.xml"/>
        <webservices servlet-link="HelloWorldWS"/>
    </java-wsdl>
</configuration>
```

For a complete description of this file, see the JBossWS documentation.

The WSDL file that wscompile generated for our `config.xml` file is shown below. Note that the SOAP address isn't provided in the WSDL file. JBoss will insert the correct URL for the WSDL when it deploys the web service.

```
<definitions name="HelloService"
             targetNamespace="http://hello.ws.jboss.org/"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:tns="http://hello.ws.jboss.org/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <types/>
    <message name="Hello_hello">
        <part name="String_1" type="xsd:string"/>
    </message>
    <message name="Hello_helloResponse">
        <part name="result" type="xsd:string"/>
    </message>
    <portType name="Hello">
        <operation name="hello" parameterOrder="String_1">
            <input message="tns:Hello_hello"/>
            <output message="tns:Hello_helloResponse"/>
        </operation>
    </portType>
    <binding name="HelloBinding" type="tns:Hello">
        <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="hello">
            <soap:operation soapAction=""/>
            <input>
                <soap:body namespace="http://hello.ws.jboss.org/"
use="literal"/>
            </input>
            <output>
                <soap:body namespace="http://hello.ws.jboss.org/"
use="literal"/>
            </output>
        </operation>
    </binding>
    <service name="HelloService">
        <port binding="tns:HelloBinding" name="HelloPort">
            <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
        </port>
    </service>
</definitions>
```

We also asked wscompile to generate a JAX-RPC mapping file. This is shown below.

```
<?xml version='1.0' encoding='UTF-8'?>
<java-wsdl-mapping version="1.1"
xmlns="http://java.sun.com/xml/ns/"Whats_new_in_JBoss_4-J2EE_Certification_and_Standards_Com
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
```

```
http://www.ibm.com/webservices/xsd/j2ee_jaxrpc_mapping_1_1.xsd">
 <package-mapping>
  <package-type>org.jboss.ws.hello</package-type>
  <namespaceURI>http://hello.ws.jboss.org/types</namespaceURI>
 </package-mapping>
 <service-interface-mapping>
  <service-interface>org.jboss.ws.hello.HelloService</service-interface>
  <wsdl-service-name xmlns:serviceNS="http://hello.ws.jboss.org/">
    serviceNS:HelloService
  </wsdl-service-name>
  <port-mapping>
   <port-name>HelloPort</port-name>
   <java-port-name>HelloPort</java-port-name>
  </port-mapping>
 </service-interface-mapping>
 <service-endpoint-interface-mapping>
<service-endpoint-interface>org.jboss.ws.hello.Hello</service-endpoint-interface>
  <wsdl-port-type xmlns:portTypeNS="http://hello.ws.jboss.org/">
    portTypeNS:Hello
  </wsdl-port-type>
  <wsdl-binding xmlns:bindingNS="http://hello.ws.jboss.org/">
    bindingNS:HelloBinding
  </wsdl-binding>
  <service-endpoint-method-mapping>
   <java-method-name>hello</java-method-name>
   <wsdl-operation>hello</wsdl-operation>
   <method-param-parts-mapping>
    <param-position>0</param-position>
    <param-type>java.lang.String</param-type>
    <wsdl-message-mapping>
     <wsdl-message xmlns:wsdlMsgNS="http://hello.ws.jboss.org/">
       wsdlMsgNS:Hello_hello
     </wsdl-message>
     <wsdl-message-part-name>String_1</wsdl-message-part-name>
     <parameter-mode>IN</parameter-mode>
    </wsdl-message-mapping>
   </method-param-parts-mapping>
   <wsdl-return-value-mapping>
    <method-return-value>java.lang.String</method-return-value>
    <wsdl-message xmlns:wsdlMsgNS="http://hello.ws.jboss.org/">
      wsdlMsgNS:Hello_helloResponse
    </wsdl-message>
    <wsdl-message-part-name>result</wsdl-message-part-name>
   </wsdl-return-value-mapping>
  </service-endpoint-method-mapping>
 </service-endpoint-interface-mapping>
</java-wsdl-mapping>
```

Finally, we generates a `webservices.xml` file. This file links to our WSDL file with the `wsdl-file` element and to the mapping file using the `jaxrpc-mapping-file` element.

In addition to this, a `port-component` element is needed that maps a port in the WSDL file to a particular service implementation. For our JSE, this is done with a `servlet-link` inside the `service-impl-bean` element. The servlet link must be the same as the name of the

pseudo-servlet we declared in the `web.xml` file.

```
<webservices version="1.1"
xmlns="http://java.sun.com/xml/ns/"Whats_new_in_JBoss_4-J2EE_Certification_and_Standards_Com
xmlns:impl="http://hello.ws.jboss.org/"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://www.ibm.com/webservices/xsd/j2ee_web_services_1_1.xsd">
    <webservice-description>
<webservice-description-name>HelloService</webservice-description-name>
        <wsdl-file>WEB-INF/wsdl/HelloService.wsdl</wsdl-file>
<jaxrpc-mapping-file>WEB-INF/jaxrpc-mapping.xml</jaxrpc-mapping-file>
        <port-component>
            <port-component-name>HelloPort</port-component-name>
            <wsdl-port>impl:HelloPort</wsdl-port>
<service-endpoint-interface>org.jboss.ws.hello.Hello</service-endpoint-interface>
            <service-impl-bean>
                <servlet-link>HelloWorldWS</servlet-link>
            </service-impl-bean>
        </port-component>
    </webservice-description>
</webservices>
```

With these completed we can deploy the WAR file containing our web service. All the deployment descriptors go in the `WEB-INF` directory, as shown in *Figure 9.1, "The structure of hello-servlet.war"*. It's important to note that the WSDL file is required to be in the wsdl subdirectory.

Figure 9.1. The structure of hello-servlet.war

To deploy and test the hello web service, run the following from the examples directory:

```
[examples]$ ant -Dchap=ws -Dex=1 run-example
...
```

```
run-example1:
     [echo] Waiting for 5 seconds for deploy...
     [java] Contacting webservice at
http://localhost:8080/hello-servlet/Hello?wsdl
     [java] hello.hello(JBoss user)
     [java] output:Hello JBoss user!
```

The server log will contain information about the deployment including the temporary location of the generated WSDL and wsdd files. It also shows the full URL of the web service.

Note the URL the JBoss publishes the WSDL file at. Our web application name is `hello-servlet` and we mapped the servlet to `/Hello` in the `web.xml` file so the web service is mappend to `/hello-servlet/Hello`. The `?wsdl` query returns the WSDL file.

If you aren't sure what the URL of the WSDL file will be, JBoss provides a way to list the web services available on the system at `/jbossws/services`. *Figure 9.2, "The web services list"* shows a view of the services list.

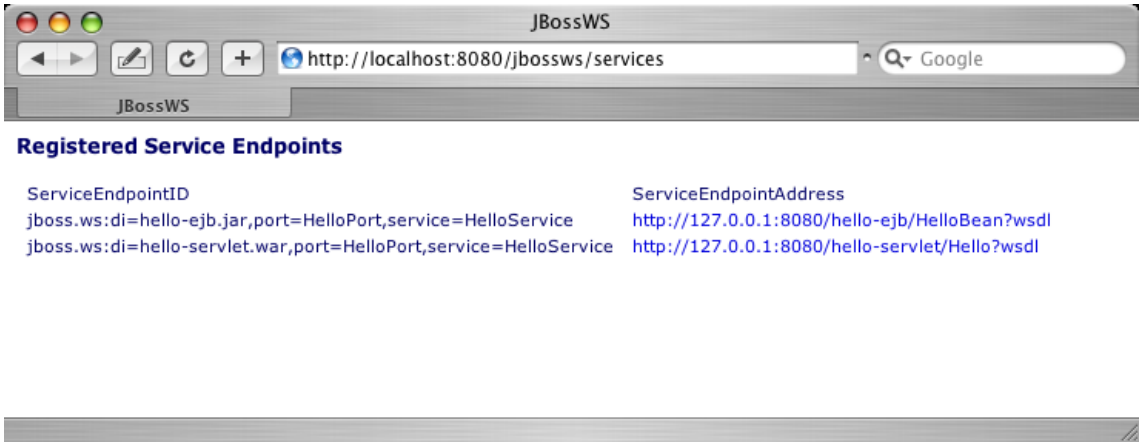


Figure 9.2. The web services list

The services list shows all of the deployed web services along with the name of the deployment unit and a link to the WSDL file for that service.

2. EJB Endpoints

Web services can also be provided from the EJB tier. Any stateless session bean can serve as the endpoint for a web service in almost the same way as the JAX-RPC endpoints. To see how this works, we will adapt the `HelloServlet` example into a session bean. Here is the code:

```
package org.jboss.ws.hello;

import javax.ejb.EJBException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
```

```
public class HelloBean
    implements SessionBean
{
    public String hello(String name)
    {
        return "Hello " + name + "!";
    }

    public void ejbCreate() {};
    public void ejbRemove() {};

    public void ejbActivate() {}
    public void ejbPassivate() {}

    public void setSessionContext(SessionContext ctx) {}
}
```

This is a very trivial session bean. Session beans normally require a home interface and either a local or remote interface. However, it is possible to omit them if the session bean is only serving as a web services endpoint. However, we do still need the `Hello` service endpoint interface that we used in the JSE example.

The `ejb-jar.xml` file is very standard for a session bean. The normal session bean parameters are explained in *Chapter 14, EJBs on JBoss*. The only new element is the `service-endpoint` element, which declares the service endpoint interface for the web service.

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar
xmlns="http://java.sun.com/xml/ns/"Whats_new_in_JBoss_4-J2EE_Certification_and_Standards_Com
version="2.1"
    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/ejb-jar_2_1.xsd">
    <display-name>chapter 12 EJB JAR</display-name>
    <enterprise-beans>
        <session>
            <ejb-name>HelloBean</ejb-name>
            <service-endpoint>org.jboss.ws.hello.Hello</service-endpoint>
            <ejb-class>org.jboss.ws.hello.HelloBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
    <assembly-descriptor>
        <method-permission>
            <unchecked/>
            <method>
                <ejb-name>HelloBean</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <container-transaction>
            <method>
```

```
                <ejb-name>HelloBean</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

The accompanying deployment descriptor files can again be generated by the `wstool` program. The configuration file is nearly identical, except that instaead of linking to the pseudo-servlet, we link to the `HelloBean` EJB.

```
<configuration xmlns="http://www.jboss.org/jbossws-tools">
    <java-wsdl>
        <service name="HelloService"
                 style="rpc"
                 endpoint="org.jboss.ws.hello.Hello"/>
        <namespaces target-namespace="http://hello.ws.jboss.org/"
                    type-namespace="http://hello.ws.jboss.org/types"/>
        <mapping file="jaxrpc-mapping.xml"/>
        <webservices ejb-link="HelloBean"/>
    </java-wsdl>
</configuration>
```

The generated files are nearly identical to the ones for the previous example, except for the `webservice.xml` file `webservices.xml`. The file, shown below, contains configuration options appropriate to the EJB endpoint.

```
<webservices version="1.1"
xmlns="http://java.sun.com/xml/ns/"Whats_new_in_JBoss_4-J2EE_Certification_and_Standards_Comp
             xmlns:impl="http://hello.ws.jboss.org/"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://www.ibm.com/webservices/xsd/j2ee_web_services_1_1.xsd">
    <webservice-description>
<webservice-description-name>HelloService</webservice-description-name>
        <wsdl-file>META-INF/wsdl/HelloService.wsdl</wsdl-file>
<jaxrpc-mapping-file>META-INF/jaxrpc-mapping.xml</jaxrpc-mapping-file>
        <port-component>
            <port-component-name>HelloPort</port-component-name>
            <wsdl-port>impl:HelloPort</wsdl-port>
<service-endpoint-interface>org.jboss.ws.hello.Hello</service-endpoint-interface>
            <service-impl-bean>
                <ejb-link>HelloBean</ejb-link>
            </service-impl-bean>
        </port-component>
    </webservice-description>
</webservices>
```

The first difference is that the WSDL file should be in the `META-INF/wsdl` directory instead of the `WEB-INF/wsdl` directory. The second difference is that the `service-impl-bean` element

contains an `ejb-link` that refers to the `ejb-name` of the session bean.

To package and deploy the application, run the following command in the examples directory:

```
[examples]$ ant -Dchap=ws -Dex=2 run-example
...
run-example2:
     [echo] Waiting for 5 seconds for deploy...
     [java] Contacting webservice at
http://localhost:8080/hello-ejb/HelloBean?wsdl
     [java] hello.hello(JBoss user)
     [java] output:Hello JBoss user!
```

The test program run here is the same as with the servlet example, except that we use a different URL for the WSDL. JBoss composes the WSDL using the base name of the EJB JAR file and the name of the service interface. However, as with all web services in JBoss, you can use the `http://localhost:8080/jbossws/services` service view shown in *Figure 9.2, "The web services list"* to verify the deployed URL of the WSDL.

3. Web Services Clients

We will now turn our attention from providing web services to consuming them.

3.1. A JAX-RPC client

The full JAX-RPC programming model is available to J2EE applications and clients. We won't cover the full range of client programming techniques, but we swill look briefly at the client we've used so far to test the web services we've deployed. The client, shown in the following listing, illustrates the dynamic proxy invocation mechanism.

```
package org.jboss.ws.client;

import org.jboss.ws.hello.Hello;

import javax.xml.rpc.Service;
import javax.xml.rpc.ServiceFactory;

import javax.xml.namespace.QName;

import java.net.URL;

public class HelloClient
{
    public static void main(String[] args)
        throws Exception
    {
        String urlstr   = args[0];
        String argument = args[1];

        System.out.println("Contacting webservice at " + urlstr);

        URL url =  new URL(urlstr);
```

```
        QName qname = new QName("http://hello.ws.jboss.org/",
                                "HelloService");

        ServiceFactory factory = ServiceFactory.newInstance();
        Service        service = factory.createService(url, qname);

        Hello          hello   = (Hello) service.getPort(Hello.class);

        System.out.println("hello.hello(" + argument + ")");
        System.out.println("output:" + hello.hello(argument));
    }
}
```

This JAX-RPC client uses the `Hello` service endpoint interface and creates a dynamic proxy to speak to the service advertised by the WSDL at the URL that is passed in as a command line argument. For illustrative purposes, we'll show another variation of web services invocation that doesn't use the service endpoint interface. This is known as the Dynamic Invocation Interface (DII). Using DII, it is possible to refer to a specific port and operation by name. Think of it as reflection for web services. The client code is shown in the following listing.

```
package org.jboss.ws.client;

import org.jboss.ws.hello.Hello;

import javax.xml.rpc.Service;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Call;

import javax.xml.namespace.QName;

import java.net.URL;

public class HelloClientDII
{
    public static void main(String[] args)
        throws Exception
    {
        String urlstr   = args[0];
        String argument = args[1];

        System.out.println("Contacting webservice at " + urlstr);

        URL url =  new URL(urlstr);

        String ns        = "http://hello.ws.jboss.org/";
        QName  qname     = new QName(ns, "HelloService");
        QName  port      = new QName(ns, "HelloPort");
        QName  operation = new QName(ns, "hello");

        ServiceFactory factory = ServiceFactory.newInstance();
        Service        service = factory.createService(url, qname);
        Call           call    = service.createCall(port, operation);
```

```
        System.out.println("hello.hello(" + argument + ")");
        System.out.println("output:" + call.invoke(new Object[]
{argument}));
    }
}
```

The following two commands can be used to run the DII client against both the JSE and EJB web services we have created.

```
[examples]$ ant -Dchap=ws -Dex=1b run-example
```

```
[examples]$ ant -Dchap=ws -Dex=2b run-example
```

Note: When accessing a remote client using the JBossWS client code, is is necessary to set the `java.endorsed.dirs` system property to the lib/endorsed directory under your JBoss installation. For example: `-Djava.endorsed.dirs=/path/to/jboss-4.0.4/lib/endorsed`.

3.2. Service references

The JAX-RPC examples in *Section 3.1, "A JAX-RPC client"* all required manual configuration of the WSDL URL and knowledge of the XML nature of the web services in question. This can be a configuration nightmare, but if your code is a J2EE component there is another option. J2EE components can declare service references and look up preconfigured `Service` objects in JNDI without needing to hardcode any web service references in the code.

To show how this works, let's first look at a session bean that needs to make a call to the hello web service:

```
package org.jboss.ws.example;

import javax.ejb.*;
import javax.naming.*;
import java.rmi.RemoteException;

import javax.xml.rpc.Service;
import javax.xml.rpc.ServiceException;

import org.jboss.ws.hello.Hello;

public class ExampleBean
    implements SessionBean
{
    public String doWork()
    {
        try {
            Context ctx     = new InitialContext();

            Service service = (Service)
ctx.lookup("java:comp/env/services/hello");
```

```
            Hello   hello   = (Hello)   service.getPort(Hello.class);

            return hello.hello("example bean");
        } catch (NamingException e) {
            throw new EJBException(e);

        } catch (ServiceException e) {
            throw new EJBException(e);

        } catch (RemoteException e) {
            throw new EJBException(e);
        }
    }

    public void ejbCreate() {};
    public void ejbRemove() {};

    public void ejbActivate() {}
    public void ejbPassivate() {}

    public void setSessionContext(SessionContext ctx) {}
}
```

`ExampleBean` invokes the hello web service in its `doWork` method. We've used the dynamic proxy invocation method here, but any of the JAX-RPC supported invocation methods would be fine. The interesting point here is that the bean has obtained the `Service` reference from a JNDI lookup in its ENC.

Web service references are declared using a `service-ref` element in inside an `ejb-jar.xml` file.

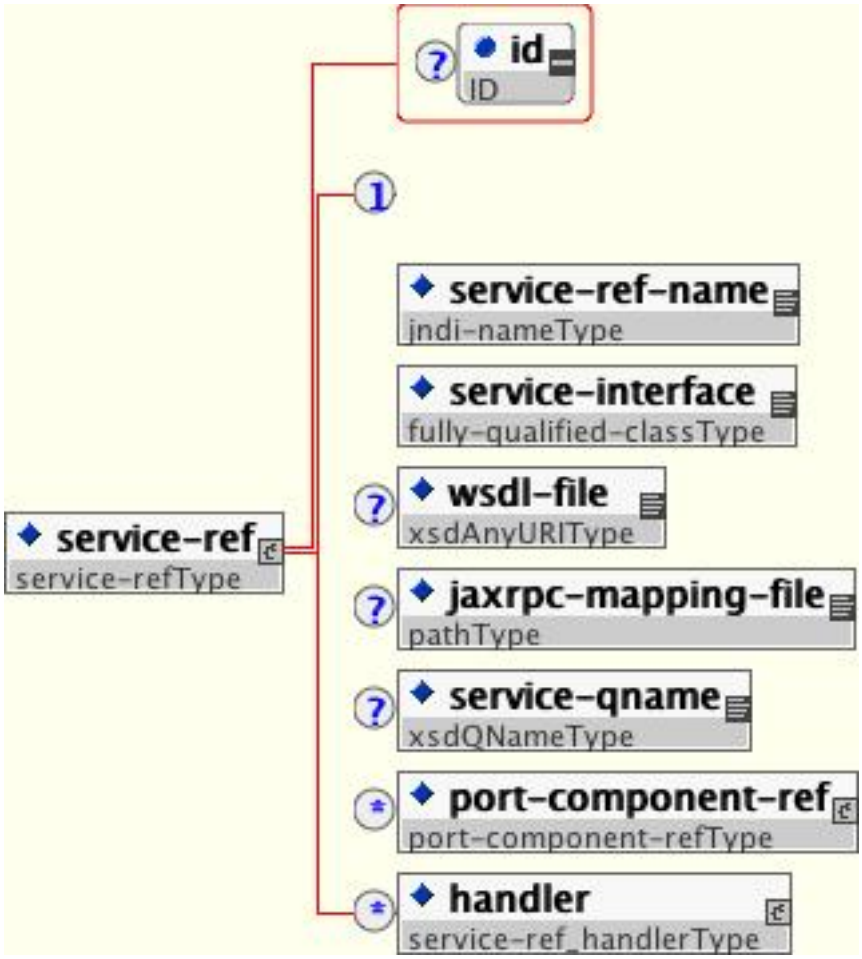


Figure 9.3. The service-ref content model

The following elements are supported by the `service-ref`:

- **service-ref-name**: This is the JNDI name that the service object will be bound under in the bean's ENC. It is relative to `java:comp/env/`.
- **service-interface**: This is the name of JAX-RPC service interface the client will use. Normally this is `javax.xml.rpc.Service`, but it's possible to provide your own service class.
- **wsdl-file**: This is the location of the WSDL file. The WSDL file should be under `META-INF/wsdl`.
- **jaxrpc-mapping-file**: This is the location of the JAX-RPC mapping file. It must be under the `META-INF` directory.
- **service-qname**: This element specifies the name of the service in the web services file. It is only mandatory if the WSDL file defines multiple services. The value must by a QName, which means it needs to be a namespace qualified value such as `ns:ServiceName` where `ns`

is an XML namespace valid at the scope of the `service-qname` element.

- **port-component-ref**: This element provides the mapping between a service endpoint interface and a port in a web service.
- **handler**: This allows the specification of handlers, which act like filters or interceptors on the current request or response.

The following `service-ref` declares a reference to the hello web service for the `Example` session bean.

```
<session>
    <ejb-name>Example</ejb-name>
    <home>org.jboss.ws.example.ExampleHome</home>
    <remote>org.jboss.ws.example.Example</remote>
    <ejb-class>org.jboss.ws.example.ExampleBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
    <service-ref>
        <service-ref-name>services/hello</service-ref-name>
        <service-interface>javax.xml.rpc.Service</service-interface>
        <wsdl-file>META-INF/wsdl/hello.wsdl</wsdl-file>
        <jaxrpc-mapping-file>META-INF/mapping.xml</jaxrpc-mapping-file>
        <service-qname xmlns:hello="http://hello.ws.jboss.org">
            hello:HelloService
        </service-qname>
    </service-ref>
</session>
```

This instructs the EJB deployer to make a `Service` object available for the bean in JNDI under the name `java:comp/env/services/hello` that talks to our hello web service. The session bean can then invoke normal web services operations on the service.

Since most of the web services configuration options are completely standard, there's little need to go into great depths here. However, JBoss does provide several additional web services configuration options through the `service-ref` element in the `jboss.xml` deployment descriptor. The content model for the service-ref element is shown in *Figure 9.4, “The jboss.xml service-ref content model”*.

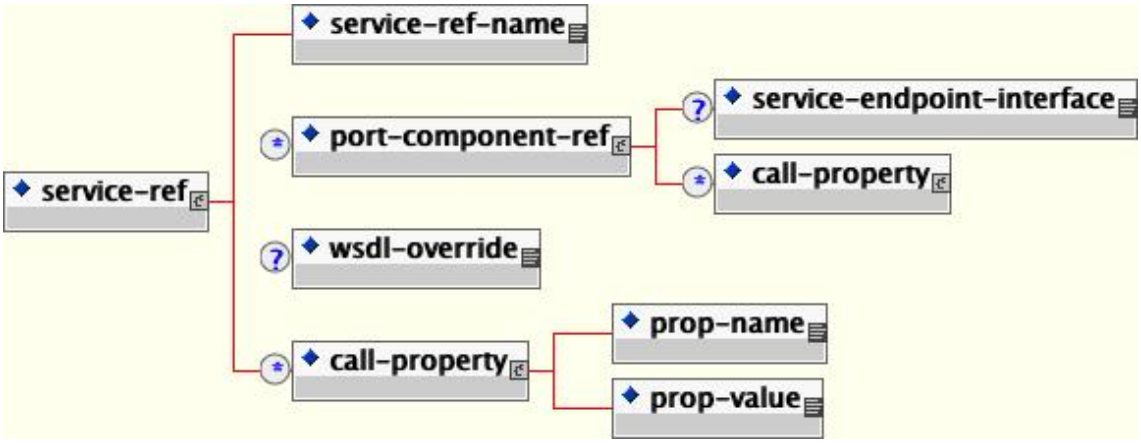


Figure 9.4. The jboss.xml service-ref content model

The configurable elements are:

- **service-ref-name**: This element should match the `service-ref-name` in the `ejb-jar.xml` file that is being configured.
- **port-component-ref**: The `port-component-ref` element provides additional information for a specific port. This includes properties that should be associated with the JAX-RPC stub for the port.
- **wsdl-override**: This provides an alternate location for the WSDL file. The value can be any valid URL. This can be used in co-ordination with the `wsdl-publish-location` to get the final WSDL file for a locally published web service. It could also be the URL of a remotely published WSDL that you don't want duplicated in the deployment file.
- **call-property**: This sets properties on the JAX-RPC stub.

Since the WSDL file generated by wscompile doesn't contain the SOAP address of our web service, we'll use the WSDL override feature to dynamically download the correct WSDL file from the server. While this might not be the best technique to use in a production application, it does illustrate the WSDL override functionality very well. The following `jboss.xml` file links the published URL for the `hello-servlet` version of the hello web service..

```
<!DOCTYPE jboss PUBLIC
          "-//JBoss//DTD JBOSS 4.0//EN"
          "http://www.jboss.org/j2ee/dtd/jboss_4_0.dtd">
<jboss>
    <enterprise-beans>
        <session>
            <ejb-name>Example</ejb-name>
            <service-ref>
                <service-ref-name>services/hello</service-ref-name>
                <wsdl-override>http://localhost:8080/hello-servlet/Hello?wsdl</wsdl-override
            </service-ref>
```

```
        </session>
    </enterprise-beans>
</jboss>
```

This example can be run as shown below:

```
[examples]$ ant -Dchap=ws -Dex=3 run-example
...
run-example3:
     [echo] Waiting for 5 seconds for deploy...
     [copy] Copying 1 file to /tmp/jboss-4.0.4/server/production/deploy
     [echo] Waiting for 5 seconds for deploy...
     [java] output:Hello example bean!
```

The `service-ref` element is not limited to the `ejb-jar.xml` file. It's available to any J2EE component. A service reference can be placed in the `web.xml` file for use by web tier components or in the `application-client.xml` file for use by J2EE client applications.

Additional Services

This chapter discusses useful MBean services that are not discussed elsewhere either because they are utility services not necessary for running JBoss, or they don't fit into a current section of the book.

1. Memory and Thread Monitoring

The `jboss.system:type=ServerInfo` MBean provides several attributes that can be used to monitor the thread and memory usage in a JBoss instance. These attributes can be monitored in many ways: through the JMX Console, from a third-party JMX management tool, from shell scripts using the twiddle command, etc... The most interesting attributes are shown below.

FreeMemory
: This is the current free memory available in the JVM.

ActiveThreadCount
: This is the number of active threads in the JVM.

ActiveThreadGroupCount
: This is the number of active thread groups in the JVM.

These are useful metrics for monitoring and alerting, but developers and administrators need a little more insite than this. The Java 5 JVMs from Sun provide more detailed information about the current state of the JVM. Some of these details are exposed by JBoss through operations on the SystemInfo MBean.

listMemoryPools
: This operations shows the size and current usage of all JVM memory pools. This operation is only available when using Java 5.

listThreadDump
: This operations shows all threads currently running in the JVM. When using Java 5, JBoss will display a complete stack trace for each thread, showing you exactly what code each thread is executing.

listThreadCpuUtilization
: This operations shows all threads currently running in the JVM along with the total CPU time each thread has used. The operation is only available in Java 5.

2. The Log4j Service

The `Log4jService` MBean configures the Apache log4j system. JBoss uses the log4j framework as its internal logging API.

- **ConfigurationURL**: The URL for the log4j configuration file. This can refer to either a XML document parsed by the `org.apache.log4j.xml.DOMConfigurator` or a Java properties file parsed by the `org.apache.log4j.PropertyConfigurator`. The type of the file is determined by the URL content type, or if this is null, the file extension. The default setting of `resource:log4j.xml` refers to the `conf/log4j.xml` file of the active server configuration file set.
- **RefreshPeriod**: The time in seconds between checks for changes in the log4 configuration specified by the `ConfigurationURL` attribute. The default value is 60 seconds.
- **CatchSystemErr**: This boolean flag if true, indicates if the `System.err` stream should be redirected onto a log4j category called `STDERR`. The default is true.
- **CatchSystemOut**: This boolean flag if true, indicates if the `System.out` stream should be redirected onto a log4j category called `STDOUT`. The default is true.
- **Log4jQuietMode**: This boolean flag if true, sets the `org.apache.log4j.helpers.LogLog.setQuiteMode`. As of log4j1.2.8 this needs to be set to avoid a possible deadlock on exception at the appender level. See bug#696819.

3. System Properties Management

The management of system properties can be done using the system properties service. It supports setting of the VM global property values just as `java.lang.System.setProperty` method and the VM command line arguments do.

Its configurable attributes include:

- **Properties**: a specification of multiple property `name=value` pairs using the `java.util.Properites.load(java.io.InputStream)` method format. Each `property=value` statement is given on a separate line within the body of the `Properties` attribute element.
- **URLList**: a comma separated list of URL strings from which to load properties file formatted content. If a component in the list is a relative path rather than a URL it will be treated as a file path relative to the `<jboss-dist>/server/<config>` directory. For example, a component of `conf/local.properties` would be treated as a file URL that points to the `<jboss-dist>/server/production/conf/local.properties` file when running with the `default` configuration file set.

The following illustrates the usage of the system properties service with an external properties file.

```
<mbean code="org.jboss.varia.property.SystemPropertiesService"
        name="jboss.util:type=Service,name=SystemProperties">

    <!-- Load properties from each of the given comma separated URLs -->
    <attribute name="URLList">
```

```
        http://somehost/some-location.properties,
        ./conf/somelocal.properties
    </attribute>
</mbean>
```

The following illustrates the usage of the system properties service with an embedded properties list.

```
<mbean code="org.jboss.varia.property.SystemPropertiesService"
        name="jboss.util:type=Service,name=SystemProperties">
    <!-- Set properties using the properties file style. -->
    <attribute name="Properties">
       property1=This is the value of my property
       property2=This is the value of my other property
    </attribute>

</mbean>
```

4. Property Editor Management

In JBoss, JavaBean property editors are used for reading data types from service files and for editing values in the JMX console. The `java.bean.PropertyEditorManager` class controls the `java.bean.PropertyEditor` instances in the system. The property editor manager can be managed in JBoss using the `org.jboss.varia.property.PropertyEditorManagerService` MBean. The property editor manager service is configured in `deploy/properties-service.xml` and supports the following attributes:

- **BootstrapEditors:** This is a listing of `property_editor_class=editor_value_type_class` pairs defining the property editor to type mappings that should be preloaded into the property editor manager. The value type of this attribute is a string so that it may be set from a string without requiring a custom property editor.
- **Editors**: This serves the same function as the `BootstrapEditors` attribute, but its type is `java.util.Properties`. Setting it from a string value in a service file requires a custom property editor for properties objects already be loaded. JBoss provides a suitable property editor.
- **EditorSearchPath**: This attribute allows one to set the editor packages search path on the `PropertyEditorManager` editor packages search path. Since there can be only one search path, setting this value overrides the default search path established by JBoss. If you set this, make sure to add the JBoss search path, `org.jboss.util.propertyeditor` and `org.jboss.mx.util.propertyeditor`, to the front of the new search path.

5. Services Binding Management

With all of the independently deployed services available in JBoss, running multiple instances

on a given machine can be a tedious exercise in configuration file editing to resolve port conflicts. The binding service allows you centrally configure the ports for multiple JBoss instances. After the service is normally loaded by JBoss, the `ServiceConfigurator` queries the service binding manager to apply any overrides that may exist for the service. The service binding manager is configured in `conf/jboss-service.xml`. The set of configurable attributes it supports include:

- **ServerName**: This is the name of the server configuration this JBoss instance is associated with. The binding manager will apply the overrides defined for the named configuration.
- **StoreFactoryClassName**: This is the name of the class that implements the `ServicesStoreFactory` interface. You may provide your own implementation, or use the default XML based store `org.jboss.services.binding.XMLServicesStoreFactory`. The factory provides a `ServicesStore` instance responsible for providing the names configuration sets.
- **StoreURL**: This is the URL of the configuration store contents, which is passed to the `ServicesStore` instance to load the server configuration sets from. For the XML store, this is a simple service binding file.

The following is a sample service binding manager configuration that uses the `ports-01` configuration from the `sample-bindings.xml` file provided in the JBoss examples directory.

```
<mbean code="org.jboss.services.binding.ServiceBindingManager"
      name="jboss.system:service=ServiceBindingManager">
    <attribute name="ServerName">ports-01</attribute>
    <attribute name="StoreURL">
        ../docs/examples/binding-manager/sample-bindings.xml
    </attribute>
    <attribute name="StoreFactoryClassName">
        org.jboss.services.binding.XMLServicesStoreFactory
    </attribute>
</mbean>
```

The structure of the binding file is shown in *Figure 10.1, "The binding service file structure"*.

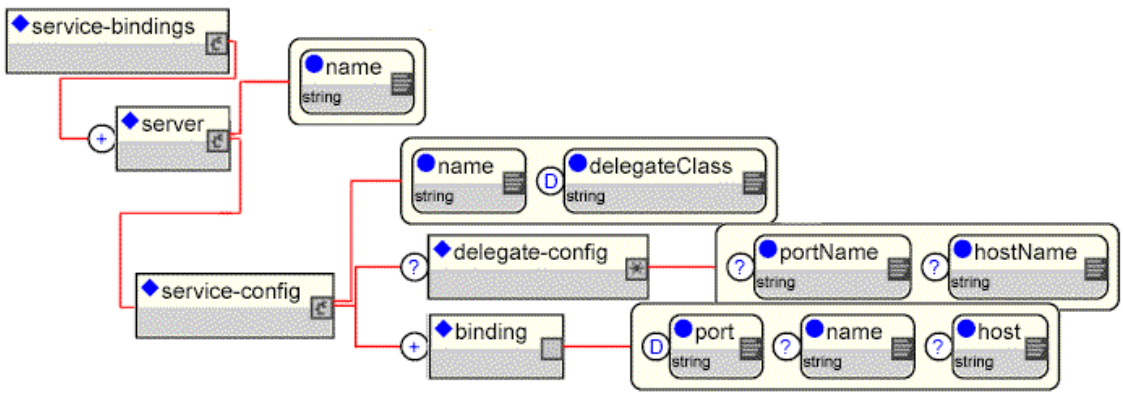


Figure 10.1. The binding service file structure

The elements are:

- **service-bindings**: The root element of the configuration file. It contains one or more server elements.
- **server**: This is the base of a JBoss server instance configuration. It has a required `name` attribute that defines the JBoss instance name to which it applies. This is the name that correlates with the `ServiceBindingManagerServerName` attribute value. The server element content consists of one or more `service-config` elements.
- **service-config**: This element represents a configuration override for an MBean service. It has a required name attribute that is the JMX `ObjectName` string of the MBean service the configuration applies to. It also has a required `delegateClass` name attribute that specifies the class name of the `ServicesConfigDelegate` implementation that knows how to handle bindings for the target service. Its contents consists of an optional `delegate-config` element and one or more binding elements.
- **binding**: A `binding` element specifies a named port and address pair. It has an optional `name` that can be used to provide multiple binding for a service. An example would be multiple virtual hosts for a web container. The port and address are specified via the optional `port` and `host` attributes respectively. If the port is not specified it defaults to 0 meaning choose an anonymous port. If the host is not specified it defaults to null meaning any address.
- **delegate-config**: The `delegate-config` element is an arbitrary XML fragment for use by the `ServicesConfigDelegate` implementation. The `hostName` and `portName` attributes only apply to the `AttributeMappingDelegate` of the example and are there to prevent DTD aware editors from complaining about their existence in the `AttributeMappingDelegate` configurations. Generally both the attributes and content of the `delegate-config` are arbitrary, but there is no way to specify and a element can have any number of attributes with a DTD.

The three `ServicesConfigDelegate` implementations are `AttributeMappingDelegate`, `XSLTConfigDelegate`, and `XSLTFileDelegate`.

5.1. AttributeMappingDelegate

The `AttributeMappingDelegate` class is an implementation of the `ServicesConfigDelegate` that expects a `delegate-config` element of the form:

```
<delegate-config portName="portAttrName" hostName="hostAttrName">
    <attribute name="someAttrName">someHostPortExpr</attribute>
    <!-- ... -->
</delegate-config>
```

The `portAttrName` is the attribute name of the MBean service to which the binding port value should be applied, and the `hostAttrName` is the attribute name of the MBean service to which the binding host value should be applied. If the `portName` attribute is not specified then the binding port is not applied. Likewise, if the `hostName` attribute is not specified then the binding host is not applied. The optional attribute element(s) specify arbitrary MBean attribute names whose values are a function of the host and/or port settings. Any reference to `${host}` in the attribute content is replaced with the host binding and any `${port}` reference is replaced with the port binding. The `portName`, `hostName` attribute values and attribute element content may reference system properties using the `${x}` syntax that is supported by the JBoss services descriptor.

The sample listing illustrates the usage of `AttributeMappingDelegate`.

```
<service-config name="jboss:service=Naming"
delegateClass="org.jboss.services.binding.AttributeMappingDelegate">
     <delegate-config portName="Port"/>
     <binding port="1099" />
</service-config>
```

Here the `jboss:service=Naming` MBean service has its `Port` attribute value overridden to 1099. The corresponding setting from the jboss1 server configuration overrides the port to 1199.

5.2. XSLTConfigDelegate

The `XSLTConfigDelegate` class is an implementation of the `ServicesConfigDelegate` that expects a `delegate-config` element of the form:

```
<delegate-config>
    <xslt-config configName="ConfigurationElement"><![CDATA[
        Any XSL document contents...
        ]]>
     </xslt-config>
     <xslt-param name="param-name">param-value</xslt-param>
     <!-- ... -->
</delegate-config>
```

The `xslt-config` child element content specifies an arbitrary XSL script fragment that is to be applied to the MBean service attribute named by the `configName` attribute. The named attribute must be of type `org.w3c.dom.Element`. The optional `xslt-param` elements specify XSL script parameter values for parameters used in the script. There are two XSL parameters defined by default called `host` and `port`, and their values are set to the configuration host and port bindings.

The `XSLTConfigDelegate` is used to transform services whose `port/interface` configuration is specified using a nested XML fragment. The following example maps the port number on hypersonic datasource:

```

```

```
<service-config
name="jboss.jca:service=ManagedConnectionFactory,name=DefaultDS"
delegateClass="org.jboss.services.binding.XSLTConfigDelegate">
    <delegate-config>
        <xslt-config
configName="ManagedConnectionFactoryProperties"><![CDATA[
<xsl:stylesheet
      xmlns:xsl='http://www.w3.org/1999/XSL/Transform' version='1.0'>

  <xsl:output method="xml" />
  <xsl:param name="host"/>
  <xsl:param name="port"/>

  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="config-property[@name='ConnectionURL']">
    <config-property type="java.lang.String" name="ConnectionURL">
       jdbc:hsqldb:hsql://<xsl:value-of select='$host'/>:<xsl:value-of
select='$port'/>
    </config-property>
  </xsl:template>

  <xsl:template match="*|@*">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
]]>
        </xslt-config>
     </delegate-config>
     <binding host="localhost" port="1901"/>
</service-config>
```

5.3. XSLTFileDelegate

The `XSLTFileDelegate` class works similarly to the `XSLTConfigDelegate` except that instead of transforming an embedded XML fragment, the XSLT script transforms a file read in from the file system. The `delegate-config` takes exactly the same form:

```
<delegate-config>
    <xslt-config configName="ConfigurationElement"><![CDATA[
        Any XSL document contents...
        ]]>
     </xslt-config>
     <xslt-param name="param-name">param-value</xslt-param>
     <!-- ... -->
</delegate-config>
```

The `xslt-config` child element content specifies an arbitrary XSL script fragment that is to be

applied to the MBean service attribute named by the `configName` attribute. The named attribute must be a String value corresponding to an XML file that will be transformed. The optional `xslt-param` elements specify XSL script parameter values for parameters used in the script. There are two XSL parameters defined by default called `host` and `port`, and their values are set to the configuration `host` and `port` bindings.

The following example maps the host and port values for the Tomcat connectors:

```
<service-config name="jboss.web:service=WebServer"
                delegateClass="org.jboss.services.binding.XSLTFileDelegate">
    <delegate-config>
        <xslt-config configName="ConfigFile"><![CDATA[
   <xsl:stylesheet
         xmlns:xsl='http://www.w3.org/1999/XSL/Transform' version='1.0'>

     <xsl:output method="xml" />
     <xsl:param name="port"/>

     <xsl:variable name="portAJP" select="$port - 71"/>
     <xsl:variable name="portHttps" select="$port + 363"/>

     <xsl:template match="/">
       <xsl:apply-templates/>
     </xsl:template>

      <xsl:template match = "Connector">
         <Connector>
            <xsl:for-each select="@*">
            <xsl:choose>
               <xsl:when test="(name() = 'port' and . = '8080')">
                  <xsl:attribute name="port">
                      <xsl:value-of select="$port" />
                  </xsl:attribute>
               </xsl:when>
               <xsl:when test="(name() = 'port' and . = '8009')">
                  <xsl:attribute name="port">
                      <xsl:value-of select="$portAJP" />
                  </xsl:attribute>
               </xsl:when>
               <xsl:when test="(name() = 'redirectPort')">
                  <xsl:attribute name="redirectPort">
                      <xsl:value-of select="$portHttps" />
                  </xsl:attribute>
               </xsl:when>
               <xsl:when test="(name() = 'port' and . = '8443')">
                  <xsl:attribute name="port">
                      <xsl:value-of select="$portHttps" />
                  </xsl:attribute>
               </xsl:when>
               <xsl:otherwise>
                  <xsl:attribute name="{name()}"><xsl:value-of select="."
/></xsl:attribute>
               </xsl:otherwise>
            </xsl:choose>
            </xsl:for-each>
```

```
            <xsl:apply-templates/>
          </Connector>
       </xsl:template>

      <xsl:template match="*|@*">
        <xsl:copy>
          <xsl:apply-templates select="@*|node()"/>
        </xsl:copy>
      </xsl:template>
    </xsl:stylesheet>
    ]]>
        </xslt-config>
     </delegate-config>
     <binding port="8280"/>
</service-config>
```

5.4. The Sample Bindings File

JBoss ships with service binding configuration file for starting up to three separate JBoss instances on one host. Here we will walk through the steps to bring up the two instances and look at the sample configuration. Start by making two server configuration file sets called `jboss0` and `jboss1` by running the following command from the book examples directory:

```
[examples]$ ant -Dchap=misc -Dex=1 run-example
```

This creates duplicates of the `server/production` configuration file sets as `server/jboss0` and `server/jboss1`, and then replaces the `conf/jboss-service.xml` descriptor with one that has the `ServiceBindingManager` configuration enabled as follows:

```
<mbean code="org.jboss.services.binding.ServiceBindingManager"
       name="jboss.system:service=ServiceBindingManager">
    <attribute name="ServerName">${jboss.server.name}</attribute>
    <attribute
name="StoreURL">${jboss.server.base.dir}/misc-ex1-bindings.xml</attribute>
    <attribute name="StoreFactoryClassName">
        org.jboss.services.binding.XMLServicesStoreFactory
    </attribute>
</mbean>
```

Here the configuration name is `${jboss.server.name}`. JBoss will replace that with name of the actual JBoss server configuration that we pass to the run script with the `-c` option. That will be either `jboss0` or `jboss1`, depending on which configuration is being run. The binding manager will find the corresponding server configuration section from the `misc-ex1-bindings.xml` and apply the configured overrides. The `jboss0` configuration uses the default settings for the ports, while the `jboss1` configuration adds 100 to each port number.

To test the sample configuration, start two JBoss instances using the `jboss0` and `jboss1` configuration file sets created previously. You can observe that the port numbers in the console

log are different for the `jboss1` server. To test out that both instances work correctly, try accessing the web server of the first JBoss on port 8080 and then try the second JBoss instance on port 8180.

6. RMI Dynamic Class Loading

The `WebService` MBean provides dynamic class loading for RMI access to the server EJBs. The configurable attributes for the service are as follows:

- **Port**: the `WebService` listening port number. A port of 0 will use any available port.
- **Host**: Set the name of the public interface to use for the host portion of the RMI codebase URL.
- **BindAddress**: the specific address the `WebService` listens on. This can be used on a multi-homed host for a `java.net.ServerSocket` that will only accept connect requests to one of its addresses.
- **Backlog**: The maximum queue length for incoming connection indications (a request to connect) is set to the `backlog` parameter. If a connection indication arrives when the queue is full, the connection is refused.
- **DownloadServerClasses**: A flag indicating if the server should attempt to download classes from thread context class loader when a request arrives that does not have a class loader key prefix.
- **DownloadResources**: A flag indicating whether the server should attempt to download non-class file resources using the thread context class loader. Note that allowing this is generally a security risk as it allows access to server configuration files which may contain security settings.
- **ThreadPool**: The `org.jboss.util.threadpool.BasicThreadPoolMBean` instance thread pool used for the class loading.

7. Scheduling Tasks

Java includes a simple timer based capability through the `java.util.Timer` and `java.util.TimerTask` utility classes. JMX also includes a mechanism for scheduling JMX notifications at a given time with an optional repeat interval as the `javax.management.timer.TimerMBean` agent service.

JBoss includes two variations of the JMX timer service in the `org.jboss.varia.scheduler.Scheduler` and `org.jboss.varia.scheduler.ScheduleManager` MBeans. Both MBeans rely on the JMX timer service for the basic scheduling. They extend the behavior of the timer service as described in the following sections.

7.1. org.jboss.varia.scheduler.Scheduler

The Scheduler differs from the `TimerMBean` in that the `Scheduler` directly invokes a callback on an instance of a user defined class, or an operation of a user specified MBean.

- **InitialStartDate**: Date when the initial call is scheduled. It can be either:
  - `NOW`: date will be the current time plus 1 seconds
  - A number representing the milliseconds since 1/1/1970
  - Date as String able to be parsed by `SimpleDateFormat` with default format pattern "`M/d/yy h:mm a`". If the date is in the past the `Scheduler` will search a start date in the future with respect to the initial repetitions and the period between calls. This means that when you restart the MBean (restarting JBoss etc.) it will start at the next scheduled time. When no start date is available in the future the `Scheduler` will not start.

  For example, if you start your `Schedulable` everyday at Noon and you restart your JBoss server then it will start at the next Noon (the same if started before Noon or the next day if start after Noon).
- **InitialRepetitions**: The number of times the scheduler will invoke the target's callback. If -1 then the callback will be repeated until the server is stopped.
- **StartAtStartup**: A flag that determines if the `Scheduler` will start when it receives its startService life cycle notification. If true the `Scheduler` starts on its startup. If false, an explicit `startSchedule` operation must be invoked on the `Scheduler` to begin.
- **SchedulePeriod**: The interval between scheduled calls in milliseconds. This value must be bigger than 0.
- **SchedulableClass**: The fully qualified class name of the `org.jboss.varia.scheduler.Schedulable` interface implementation that is to be used by the `Scheduler` . The `SchedulableArguments` and `SchedulableArgumentTypes` must be populated to correspond to the constructor of the `Schedulable` implementation.
- **SchedulableArguments**: A comma separated list of arguments for the `Schedulable` implementation class constructor. Only primitive data types, `String` and classes with a constructor that accepts a `String` as its sole argument are supported.
- **SchedulableArgumentTypes**: A comma separated list of argument types for the `Schedulable` implementation class constructor. This will be used to find the correct constructor via reflection. Only primitive data types, `String` and classes with a constructor that accepts a `String` as its sole argument are supported.
- **SchedulableMBean**: Specifies the fully qualified JMX `ObjectName` name of the schedulable MBean to be called. If the MBean is not available it will not be called but the remaining repetitions will be decremented. When using `SchedulableMBean` the `SchedulableMBeanMethod` must also be specified.

- **SchedulableMBeanMethod**: Specifies the operation name to be called on the schedulable MBean. It can optionally be followed by an opening bracket, a comma separated list of parameter keywords, and a closing bracket. The supported parameter keywords include:
  - `NOTIFICATION` which will be replaced by the timers notification instance (javax.management.Notification)
  - `DATE` which will be replaced by the date of the notification call (java.util.Date)
  - `REPETITIONS` which will be replaced by the number of remaining repetitions (long)
  - `SCHEDULER_NAME` which will be replaced by the `ObjectName` of the `Scheduler`
  - Any fully qualified class name which the `Scheduler` will set to null.

A given Scheduler instance only support a single schedulable instance. If you need to configure multiple scheduled events you would use multiple `Scheduler` instances, each with a unique `ObjectName`. The following is an example of configuring a `Scheduler` to call a `Schedulable` implementation as well as a configuration for calling a MBean.

```
<server>

    <mbean code="org.jboss.varia.scheduler.Scheduler"
           name="jboss.docs:service=Scheduler">
        <attribute name="StartAtStartup">true</attribute>
        <attribute
name="SchedulableClass">org.jboss.book.misc.ex2.ExSchedulable</attribute>
        <attribute name="SchedulableArguments">TheName,123456789</attribute>
        <attribute
name="SchedulableArgumentTypes">java.lang.String,long</attribute>

        <attribute name="InitialStartDate">NOW</attribute>
        <attribute name="SchedulePeriod">60000</attribute>
        <attribute name="InitialRepetitions">-1</attribute>
    </mbean>

</server>
```

The `SchedulableClassorg.jboss.book.misc.ex2.ExSchedulable` example class is given below.

```
package org.jboss.book.misc.ex2;

import java.util.Date;
import org.jboss.varia.scheduler.Schedulable;

import org.apache.log4j.Logger;

/**
 * A simple Schedulable example.
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.4 $
 */
```

```
public class ExSchedulable implements Schedulable
{
    private static final Logger log = Logger.getLogger(ExSchedulable.class);

    private String name;
    private long value;

    public ExSchedulable(String name, long value)
    {
        this.name = name;
        this.value = value;
        log.info("ctor, name: " + name + ", value: " + value);
    }

    public void perform(Date now, long remainingRepetitions)
    {
        log.info("perform, now: " + now +
                 ", remainingRepetitions: " + remainingRepetitions +
                 ", name: " + name + ", value: " + value);
    }
}
```

Deploy the timer SAR by running:

```
[examples]$ ant -Dchap=misc -Dex=2 run-example
```

The server console shows the following which includes the first two timer invocations, separated by 60 seconds:

```
21:09:27,716 INFO  [ExSchedulable] ctor, name: TheName, value: 123456789
21:09:28,925 INFO  [ExSchedulable] perform, now: Mon Dec 20 21:09:28 CST
2004,
  remainingRepetitions: -1, name: TheName, value: 123456789
21:10:28,899 INFO  [ExSchedulable] perform, now: Mon Dec 20 21:10:28 CST
2004,
  remainingRepetitions: -1, name: TheName, value: 123456789
21:11:28,897 INFO  [ExSchedulable] perform, now: Mon Dec 20 21:11:28 CST
2004,
  remainingRepetitions: -1, name: TheName, value: 123456789
```

8. The Timer Service

The JMX standard defines a timer MBean (`javax.management.timer.Timer`) which can send notifications at predetermined times. The a timer MBean can be instantiated within JBoss as any other MBean.

```
<mbean code="javax.management.timer.Timer"
name="jboss.monitor:name=Heartbeat,type=Timer"/>
```

A standard JMX timer doesn't produce any timer events unless it is asked to. To aid in the configuration of the timer MBean, JBoss provides a complementary `TimerService` MBean. It interacts with the timer MBean to configure timer events at regular intervals and to transform them into JMX notifications more suitable for other services. The `TimerService` MBean takes the following attributes:

- **NotificationType**: This is the type of the notification to be generated.
- **NotificationMessage**: This is the message that should be associated with the generated notification.
- **TimerPeriod**: This is the time period between notification. The time period is in milliseconds, unless otherwise specified with a unit like "30min" or "4h". Valid time suffixes are `msec`, `sec`, `min` and `h`.
- **Repeatitions**: This is the number of times the alert should be generated. A value of 0 indicates the alert should repeat indefinitely.
- **TimerMbean**: This is the `ObjectName` of the time MBean that this `TimerService` instance should configure notifications for.

The following sample illustrates the the use of the `TimerService` MBean.

```
<mbean code="org.jboss.monitor.services.TimerService"
       name="jboss.monitor:name=Heartbeat,type=TimerService">
    <attribute name="NotificationType">jboss.monitor.heartbeat</attribute>
    <attribute name="NotificationMessage">JBoss is alive!</attribute>
    <attribute name="TimerPeriod">60sec</attribute>
    <depends optional-attribute-name="TimerMBean">
        jboss.monitor:name=Heartbeat,type=Timer
    </depends>
</mbean>
```

This MBean configuration configures the `jboss.monitor:name=Heartbeat,type=Timer` timer to generate a `jboss.monitor.heartbeat` notification every 60 seconds. Any service that that wants to receive this periodic notifications can subscribe to the notification.

As an example, JBoss provides a simple `NotificationListener` MBean that can listen for a particular notification and log a log message when an event is generated. This MBean is very useful for debugging or manually observing notifications. The following MBean definition listens for any events generated by the heartbeat timer used in the previous examples.

```
<mbean code="org.jboss.monitor.services.NotificationListener"
       name="jboss.monitor:service=NotificationListener">
    <attribute name="SubscriptionList">
        <subscription-list>
            <mbean name="jboss.monitor:name=Heartbeat,type=Timer" />
        </subscription-list>
    </attribute>
```

```
</mbean>
```

The `subscription-list` element lists which MBeans the listener should listen to. Notice that the MBean we are listening to is the name of the actual timer MBean and not the `TimerService` MBean. Because the timer might generate multiple events, configured by multiple `TimerService` instances, you may need to filter by notification type. The `filter` element can be used to create notification filters that select only the notification types desired. The following listing shows how we can limit notifications to only the `jboss.monitor.heartbeat` type the timer service configured.

```
<mbean code="org.jboss.monitor.services.NotificationListener"
      name="jboss.monitor:service=NotificationListener">
    <attribute name="SubscriptionList">
        <subscription-list>
            <mbean name="jboss.monitor:name=Heartbeat,type=Timer">
                <filter factory="NotificationFilterSupportFactory">
                    <enable type="jboss.monitor.heartbeat"/>
                </filter>
            </mbean>
        </subscription-list>
    </attribute>
</mbean>
```

As an example of a slightly more interesting listener, we'll look at the ScriptingListener. This listener listens for particular events and then executes a specified script when events are received. The script can be written in any bean shell scripting language. The ScriptingListener accepts has the following parameters.

- **ScriptLanguage**: This is the language the script is written in. This should be `beanshell`, unless you have loaded libraries for another beanshell compatible language.
- **Script**: This is the text of the script to evaluate. It is good practice to enclose the script in a CDATA section to minimize conflicts between scripting language syntax and XML syntax.
- **SubscriptionList**: This is the list of MBeans that this MBean will listen to for events that will trigger the script.

The following example illustrates the use of the `ScriptingListener`. When the previously configured timer generates a heartbeat notification, the beanshell script will execute, printing the current memory values to STDOUT. (This output will be redirected to the log files) Notice that the beanshell script has a reference to the MBean server and can execute operations against other MBeans.

```
<mbean code="org.jboss.monitor.services.ScriptingListener"
       name="jboss.monitor:service=ScriptingListener">
    <attribute name="SubscriptionList">
        <subscription-list>
```

```
            <mbean name="jboss.monitor:name=Heartbeat,type=Timer"/>
        </subscription-list>
    </attribute>
    <attribute name="ScriptLanguage">beanshell</attribute>
    <attribute name="Script">
                <![CDATA[
   import javax.management.ObjectName;

   /* poll free memory and thread count */
   ObjectName target = new ObjectName("jboss.system:type=ServerInfo");

   long freeMemory = server.getAttribute(target, "FreeMemory");
   long threadCount = server.getAttribute(target, "ActiveThreadCount");

   log.info("freeMemory" + freeMemory + ", threadCount" + threadCount);
]]>
    </attribute>
</mbean>
```

Of course, you are not limited to these JBoss-provided notification listeners. Other services such as the barrier service (see *Section 9, "The BarrierController Service"*) receive and act on notifications that could be generated from a timer. Additionally, any MBean can be coded to listen for timer-generated notifications.

9. The BarrierController Service

Expressing dependencies between services using the <depends> tag is a convenient way to make the lifecycle of one service depend on the lifecycle of another. For example, when `serviceA` depends on `serviceB` JBoss will ensure the `serviceB.create()` is called before `serviceA.create()` and `serviceB.start()` is called before `serviceA.start()`.

However, there are cases where services do not conform to the JBoss lifecycle model, i.e. they don't expose create/start/stop/destroy lifecycle methods). This is the case for `jboss.system:type=Server MBean`, which represents the JBoss server itself. No lifecycle operations are exposed so you cannot simply express a dependency like: if JBoss is fully started then start my own service.

Or, even if they do conform to the JBoss lifecycle model, the completion of a lifecycle method (e.g. the `start` method) may not be sufficient to describe a dependency. For example the `jboss.web:service=WebServer` MBean that wraps the embedded Tomcat server in JBoss does not start the Tomcat connectors until after the server is fully started. So putting a dependency on this MBean, if we want to hit a webpage through Tomcat, will do no good.

Resolving such non-trivial dependencies is currently performed using JMX notifications. For example the `jboss.system:type=Server` MBean emits a notification of type `org.jboss.system.server.started` when it has completed startup, and a notification of type `org.jboss.system.server.stopped` when it shuts down. Similarly, `jboss.web:service=WebServer` emits a notification of type `jboss.tomcat.connectors.started` when it starts up. Services can subscribe to those

notifications in order to implement more complex dependencies. This technique has been generalized with the barrier controller service.

The barrier controller is a relatively simple MBean service that extends `ListenerServiceMBeanSupport` and thus can subscribe to any notification in the system. It uses the received notifications to control the lifecycle of a dynamically created MBean called the barrier.

The barrier is instantiated, registered and brought to the create state when the barrier controller is deployed. After that, the barrier is started and stopped when matching notifications are received. Thus, other services need only depend on the barrier MBean using the usual `<depends>` tag, without having to worry about complex lifecycle issues. They will be started and stopped in tandem with the Barrier. When the barrier controller is undeployed the barrier is destroyed.

The notifications of interest are configured in the barrier controller using the `SubscriptionList` attribute. In order to identify the starting and stopping notifications we associate with each subscription a handback string object. Handback objects, if specified, are passed back along with the delivered notifications at reception time (i.e. when `handleNotification()` is called) to qualify the received notifications, so that you can identify quickly from which subscription a notification is originating (because your listener can have many active subscriptions).

So we tag the subscriptions that produce the starting/stopping notifications of interest using any handback strings, and we configure this same string to the `StartBarrierHandback` (and `StopBarrierHandback` correspondingly) attribute of the barrier controller. Thus we can have more than one notifications triggering the starting or stopping of the barrier.

The following example shows a service that depends on the Tomcat connectors. In fact, this is a very common pattern for services that want to hit a servlet inside tomcat. The service that depends on the Barrier in the example, is a simple memory monitor that creates a background thread and monitors the memory usage, emitting notifications when thresholds get crossed, but it could be anything. We've used this because it prints out to the console starting and stopping messages, so we know when the service gets activated/deactivated.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- $Id: J2EE_Additional_Services.xml,v 1.4 2007/06/22 08:38:25 skittoli
Exp $ -->

<server>
  <!--
    In this example we have the BarrierController controlling a Barrier
    that is started when we receive the "jboss.tomcat.connectors.started"
    notification from the Tomcat mbean, and stopped when we receive the
    "org.jboss.system.server.stopped" notification from the server mbean.

    The dependent services need only define a dependency on the Barrier
mbean!
  -->
  <mbean code="org.jboss.system.BarrierController"
         name="jboss:service=BarrierController">

    <!-- Whether to have the Barrier initially started or not -->
```

```
    <attribute name="BarrierEnabledOnStartup">false</attribute>

    <!-- Whether to subscribe for notifications after startup -->
    <attribute name="DynamicSubscriptions">true</attribute>

    <!-- Dependent services will depend on this mbean -->
    <attribute
name="BarrierObjectName">jboss:name=TomcatConnector,type=Barrier</attribute>

    <!-- The notification subscription handback that starts the barrier -->
    <attribute name="StartBarrierHandback">start</attribute>

    <!-- The notification subscription handback that stops the barrier -->
    <attribute name="StopBarrierHandback">stop</attribute>

    <!-- The notifications to subscribe for, along with their handbacks -->
    <attribute name="SubscriptionList">
      <subscription-list>
        <mbean name="jboss.web:service=WebServer" handback="start">
          <filter factory="NotificationFilterSupportFactory">
            <enable type="jboss.tomcat.connectors.started"/>
          </filter>
        </mbean>
        <mbean name="jboss.system:type=Server" handback="stop">
          <filter factory="NotificationFilterSupportFactory">
            <enable type="org.jboss.system.server.stopped"/>
          </filter>
        </mbean>
      </subscription-list>
    </attribute>
  </mbean>

  <!--
    An example service that depends on the Barrier we declared above.
    This services creates a background thread and monitors the memory
    usage. When it exceeds the defined thresholds it emits notifications
  -->
  <mbean code="org.jboss.monitor.services.MemoryMonitor"
         name="jboss.monitor:service=MemoryMonitor">

    <attribute name="FreeMemoryWarningThreshold">20m</attribute>
    <attribute name="FreeMemoryCriticalThreshold">15m</attribute>

    <!-- The BarrierObjectName configured in the BarrierController -->
    <depends>jboss:name=TomcatConnector,type=Barrier</depends>
  </mbean>

</server>
```

If you hot-deploy this on a running server the Barrier will be stopped because by the time the barrier controller is deployed the starting notification is already seen. (There are ways to overcome this.) However, if you re-start the server, the barrier will be started just after the Tomcat connectors get activated. You can also manually start or stop the barrier by using the `startBarrier()` and `stopBarrier()` operations on the barrier controller. The attribute

`BarrierStateString` indicates the status of the barrier.

10. Exposing MBean Events via SNMP

JBoss has an SNMP adaptor service that can be used to intercept JMX notifications emitted by MBeans, convert them to traps and send them to SNMP managers. In this respect the snmp-adaptor acts as a SNMP agent. Future versions may offer support for full agent get/set functionality that maps onto MBean attributes or operations.

This service can be used to integrate JBoss with higher order system/network management platforms (HP OpenView, for example), making the MBeans visible to those systems. The MBean developer can instrument the MBeans by producing notifications for any significant event (e.g. server coldstart), and adaptor can then be configured to intercept the notification and map it onto an SNMP traps. The adaptor uses the JoeSNMP package from OpenNMS as the SNMP engine.

The SNMP service is configured in `snmp-adaptor.sar`. This service is only available in the `all` configuration, so you'll need to copy it to your configuration if you want to use it. Inside the snmp-adaptor.sar directory, there are two configuration files that control the SNMP service.

- **managers.xml**: configures where to send traps. The content model for this file is shown in *Figure 10.2, "The schema for the SNMP managers file"*.
- **notifications.xml**: specifies the exact mapping of each notification type to a corresponding SNMP trap. The content model for this file is shown in *Figure 10.3, "The schema for the notification to trap mapping file"*.

The `SNMPAgentService` MBean is configured in `snmp-adaptor.sar/META-INF/jboss-service.xml`. The configurable parameters are:

- **HeartBeatPeriod**: The period in seconds at which heartbeat notifications are generated.
- **ManagersResName**: Specifies the resource name of the `managers.xml` file.
- **NotificationMapResName**: Specifies the resource name of the `notications.xml` file.
- **TrapFactoryClassName**: The `org.jboss.jmx.adaptor.snmp.agent.TrapFactory` implementation class that takes care of translation of JMX Notifications into SNMP V1 and V2 traps.
- **TimerName**: Specifies the JMX ObjectName of the JMX timer service to use for heartbeat notifications.
- **SubscriptionList**: Specifies which MBeans and notifications to listen for.

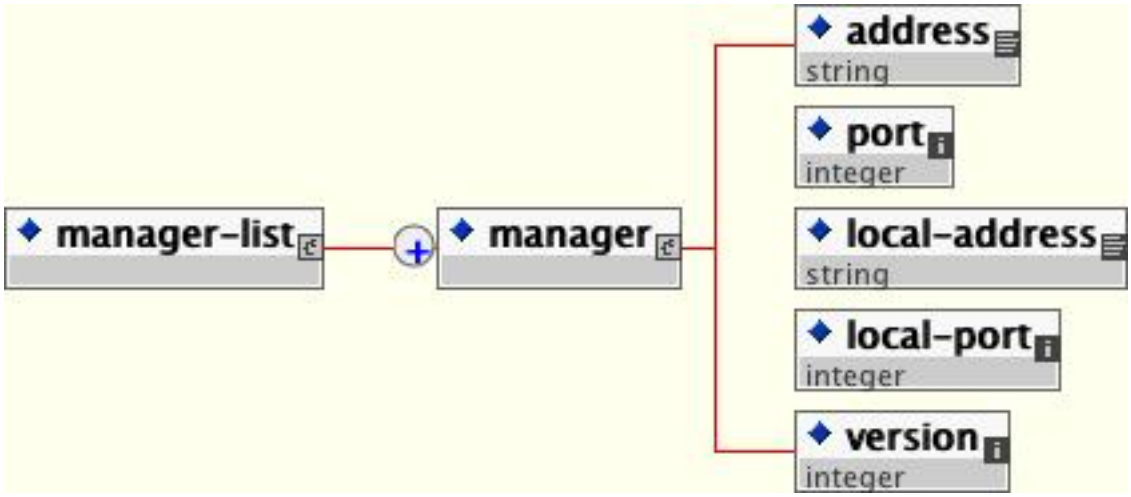


Figure 10.2. The schema for the SNMP managers file

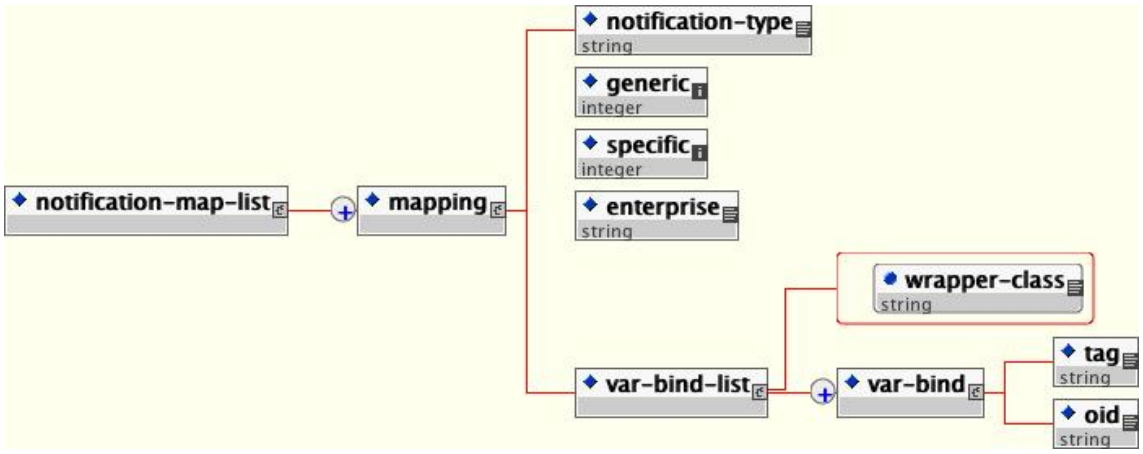


Figure 10.3. The schema for the notification to trap mapping file

`TrapdService` is a simple MBean that acts as an SNMP Manager. It listens to a configurable port for incoming traps and logs them as DEBUG messages using the system logger. You can modify the log4j configuration to redirect the log output to a file. `SnmpAgentService` and `TrapdService` are not dependent on each other.

Part III. Clustering Configuration

Quick Tutorial to Setup a Clustered Web Application

Web server clustering is to use multiple JBoss AS server instances to serve the same web address (e.g., http://www.jboss.com/). It is one of the most common clustering architectures JBoss AS users deploy. In this tutorial, we discuss the basic architecture of a web cluster and give step-by-step instructions on how to setup the cluster. A very simple web cluster looks like *Figure 11.1, "A very simple web cluster"*.

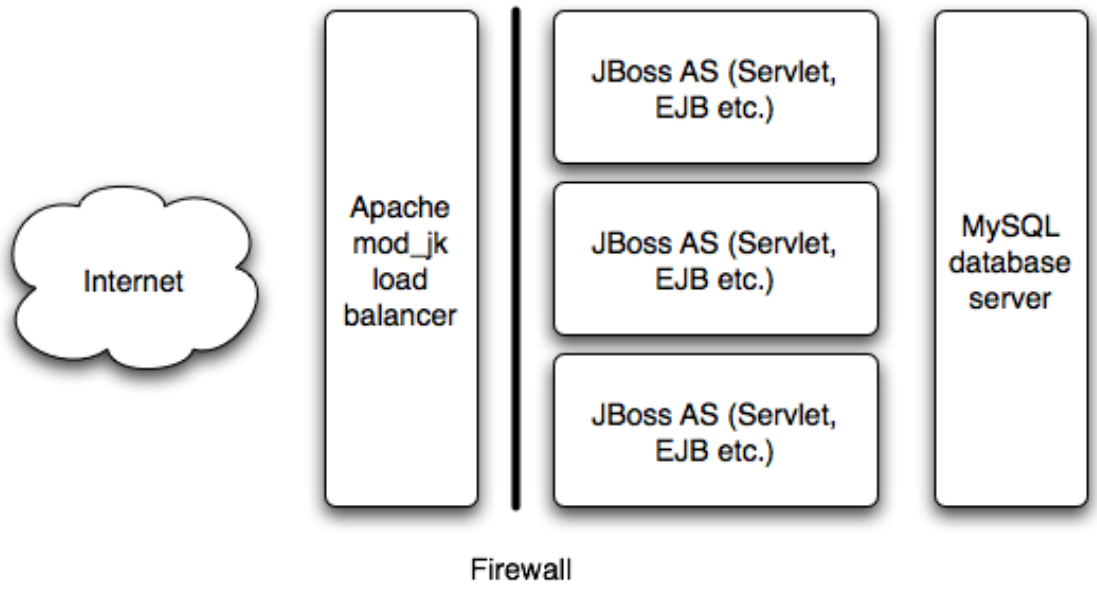


Figure 11.1. A very simple web cluster

A lightweight web server, known as the load balancer, receives all HTTP requests from that web address and dispatches them to JBoss AS nodes in the cluster. The JBoss AS node does the heavy lifting of processing the request and generating the response. All the JBoss AS nodes share the same database server for persistent data in the application. Notice that each node contains the entire JBoss AS stack, including the web server, servlet container, EJB container, persistence manager etc. Breaking the JBoss AS stack to different physical servers would substantially complicate the architecture but without much scalability benefit.

1. Setup the simple web cluster

In our simple cluster depicted in *Figure 11.1, "A very simple web cluster"*, every JBoss AS node is independent of each other. The load balancer "remembers" the user sessions and always forwards requests from the same session to the same JBoss AS node (a.k.a sticky session). So, each JBoss AS node stores its own set of HTTP session data. There is no information sharing between JBoss AS nodes, besides the persistent data in the shared database.

Assuming that the load balancer and the shared database are not performance bottlenecks, this cluster can scale linearly with the number of JBoss AS nodes. Of course, the simplicity and scalability has their price. The most obvious shortcomings of the this simple architecture are the lack of support for failover and database cache. We will look at solutions for those problems in *Section 2.2, "Database cache"*.

1.1. Setup the load balancer

The easiest way to setup a load balancer is to use the Apache web server with the mod_jk module to interact with JBoss AS. After installing the Apache web server, you should download the mod_jk 1.2.x binary from *http://www.apache.org/dist/jakarta/tomcat-connectors/jk/binaries/* [http://www.apache.org/dist/jakarta/tomcat-connectors/jk/binaries/]. Rename the downloaded file to mod_jk.so and copy it under APACHE_HOME/modules/ directory.

Use mod_jk 1.2.x

We strongly advise you to use mod_jk 1.2.x, as both mod_jk and mod_jk2 are deprecated, unsupported and no further developments are going on in the community.

Note

Aside from loading balancing, using Apache mod_jk to front JBoss AS servers have other benefits. The most important one is that Apache serves static content (e.g., images) much faster than Tomcat (i.e., the embedded servlet server in JBoss AS). Apache also handles SSL and keep-alive connections much more efficiently than Tomcat. In addition, Apache provides modules for access control, URL rewriting, CGI/PHP execution. So, we recommend fronting JBoss AS with Apache mod_jk even if you have only one JBoss AS node.

To tell the Apache server to use mod_jk, you need to add the following line to the APACHE_HOME/conf/httpd.conf file.

```
# Include mod_jk's specific configuration file
Include conf/mod_jk.conf
```

Example 11.1. Add this to the httpd.conf file

Next, you should configure the interaction between Apache server and mod_jk by creating the APACHE_HOME/conf/mod_jk.conf file as follows. The LoadModule directive must reference the mod_jk library you have downloaded in the previous section. You must indicate the exact same name with the "modules" file path prefix. The JkMount directive tells Apache which URLs it should forward to the mod_jk module (and, in turn, to the JBoss AS instances). In the above file, all requests with URL path /application/* are sent to the mod_jk load-balancer. This way, you can configure Apache to server static contents (or PHP contents) directly and only use the loadbalancer for Java applications. If you only use mod_jk as a loadbalancer, you can also forward all URLs (i.e., /*) to mod_jk.

```
# Load mod_jk module
# Specify the filename of the mod_jk lib
LoadModule jk_module modules/mod_jk.so

# Where to find workers.properties
JkWorkersFile conf/workers.properties

# Where to put jk logs
JkLogFile logs/mod_jk.log

# Set the jk log level [debug/error/info]
JkLogLevel info

# Select the log format
JkLogStampFormat  "[%a %b %d %H:%M:%S %Y]"

# JkRequestLogFormat
JkRequestLogFormat "%w %V %T"

# Mount your applications
JkMount /application/* loadbalancer

# Add shared memory.
# This directive is present with 1.2.10 and
# later versions of mod_jk, and is needed for
# for load balancing to work properly
JkShmFile logs/jk.shm

# Add jkstatus for managing runtime data
<Location /jkstatus/>
    JkMount status
    Order deny,allow
    Deny from all
    Allow from 127.0.0.1
</Location>
```

Example 11.2. The mod_jk.conf file

Then, you need to configure mod_jk itself to load balance the JBoss AS cluster. The workers file conf/workers.properties specifies the locations of JBoss AS server nodes and how calls should be load-balanced across them. For a two nodes setup, the file could look like the following.

```
# Define list of workers that will be used
# for mapping requests
worker.list=loadbalancer,status

# Define Node1
# modify the host as your host IP or DNS name.
worker.node1.port=8009
worker.node1.host=node1.mydomain.com
worker.node1.type=ajp13
worker.node1.lbfactor=1
worker.node1.cachesize=10

# Define Node2
# modify the host as your host IP or DNS name.
worker.node2.port=8009
worker.node2.host= node2.mydomain.com
worker.node2.type=ajp13
worker.node2.lbfactor=1
worker.node2.cachesize=10

# Load-balancing behaviour
worker.loadbalancer.type=lb
worker.loadbalancer.balance_workers=node1,node2
worker.loadbalancer.sticky_session=1
#worker.list=loadbalancer

# Status worker for managing load balancer
worker.status.type=status
```

Example 11.3. The workers.properties file

The above file configures mod_jk to perform weighted round-robin load balancing between two JBoss AS instances, node1 and node2, listening on port 8009. In the works.properties file, each node is defined using the worker.XXX naming convention where XXX represents an arbitrary name you choose for one of the target JBoss AS node. For each worker, you must give the host name (or IP address) and port number of the AJP13 connector (the port 8009 is default). The lbfactor attribute is the load-balancing factor for this specific worker. It is used to define the priority (or weight) a node should have over other nodes. The higher this number is, the more HTTP requests it will receive. This setting can be used to differentiate servers with different processing power. The cachesize attribute defines the size of the thread pools associated to the JBoss AS node (i.e. the number of concurrent requests it will forward to the node). Make sure this number does not outnumber the number of threads configured on the AJP13 connector of the node. The last part of the conf/workers.properties file defines the loadbalancer worker. The

worker.loadbalancer.balanced_workers line must list all workers previously defined in the same file, and then load-balancing will happen over these workers.

Sticky sessions

In mod_jk, you almost always configure the load balancer to use "sticky sessions". With sticky sessions, mod_jk forwards requests in the same web session to the same JBoss AS node. The sticky session ensures that the user is always served by the JBoss AS instance that has the correct session state. Sticky session is required for our simple cluster setup.

1.2. Configure JBoss AS nodes

The JBoss AS nodes in the cluster are almost identical to each other. They must run the same version of JBoss AS, the same applications, and have the same configuration except for few clustering related configuration options. In this section, we cover two JBoss AS configuration options required for the web cluster. One of them is universal for all nodes and the other is unique for each node.

The clustering profile for JBoss AS

For this simple clustering architecture, you do not even need the "clustering" profile of JBoss AS -- you can just use the default JBoss AS profile since there is no communication (i.e., clustering) between JBoss AS nodes.

First, we must configure all JBoss AS nodes to add an "identifier" of itself to all HTTP session IDs it serves. This way, the load balancer would know which JBoss AS node to forward each request just by checking at the session ID of the request. To do that, edit the JBOSS_HOME/server/all/deploy/jboss-web.deploy/META-INF/jboss-service.xml file (replace all with your own server configuration name). Locate the <attribute> element with a name of UseJK, and set its value to true as follows.

```
<attribute name="UseJK">true</attribute>
```

Example 11.4. Configure jboss-service.xml to support JBoss AS node identity in HTTP session IDs

Second, you need to actually give an identity to each JBoss AS node. On each clustered JBoss

node, you need to name the node according to the name specified in workers.properties. For instance, on JBoss instance node1, edit the JBOSS_HOME/server/all/deploy/jboss-web.deployer/server.xml file (replace all with your own server configuration name if necessary). Locate the <Engine> element and add an attribute jvmRoute as follows.

```
<Engine name="jboss.web"
        defaultHost="localhost" jvmRoute="node1">
... ...
</Engine>
```

Example 11.5. Add node information to the server.xml file

Each JBoss AS instance appends its own jvmRoute value to its HTTP session IDs so that mod_jk can route incoming requests. That's all you need on the JBoss AS node.

1.3. Shared Database

By default, JBoss AS uses an embedded HSQL database for persistence data storage. For a web cluster to work properly, you must use a shared database for all server nodes. We recommend you setup a separate MySQL database server and then configure it as the DataSource for applications deployed on all nodes. Please see *Appendix B, Use Alternative Databases with JBoss AS* for more information on how to setup external databases as DataSources.

2. Optional improvements to the simple cluster

The simple web cluster we discussed above is simply a load balanced architecture for servers that only share information at the database level. This architecture is highly scalable but it does not provide crucial clustering features such as failover and distributed caching. With some simple optional steps, you can easily add those features to your cluster.

Please note that you need to install a clustering enabled JBoss AS profile (i.e., the all or ejb3-cluster profiles) on all your server nodes for the optional setups.

2.1. Failover support

The simple web cluster does not support failover if one of the server nodes crashes. Since each JBoss AS node stores its own HTTP session state data -- the users on the crashed server would lose their sessions when the load balancer forwards their requests to another node. The solution is for each node in the cluster to replicate its HTTP session data to other nodes in the cluster, so that when the node crashes, there is another node node to pick up all users it left off without any session loss.

When you start multiple clustering enabled JBoss AS instances on the same network, they automatically form a cluster. Therefore, all you need to do is to enable HTTP session replication in your web application. That is by adding a <distributable/> tag in your web.xml file.

```
<web-app  ...>
    <distributable/>
    <!-- ... -->
</web-app>
```

However, failover does not come free. Since extra object serialization (CPU intensive) and communication between server nodes (network intensive) are required, the cluster can no longer scale linearly with the number of nodes. With the default setup, the HTTP sessions on each node are replicated to all other nodes in the cluster. So the clustering workload increases geometrically with the number of server nodes. The cluster would not scale beyond 8 to 10 servers with the default setup. For a truly scalable failover solution, we need to setup buddy replication in the cluster so that each node only replicates its HTTP session data to another "buddy" failover node.

2.2. Database cache

Database cache is very useful for improving performance of ORM solutions like Hibernate and EJB 3.0. It stores frequently accessed data objects in the application server's memory and hence reduces the round trips to the database server. However, the challenge is to keep the cache on all server nodes in sync. For instance, data object O might be cached on both server nodes A and B. Now, node A updates object O and flushes the change back to the database. If A and B are not correlated, there is no way for B to know that object O already has an updated value in the database -- so users on server B would continue to access the expired value of O and get errors.

The solution here is to use a distributed database cache. The cache is "shared" by all server nodes. So, if one node adds / updates / removes, any object into / from the cache, all other nodes get the updated cache instantly as well. Here we give an example configuration for EJB 3.0 entity bean, which is supported by Hibernate 3.2+.

The distributed database cache is supported out of the box in the ejb3-cluster profile of JBoss AS. As the case with HTTP session replication, all you need is to enable this feature in your application. First, you need to specify JBoss TreeCache as your database cache provider in your persistence.xml file.

```
<!-- Clustered cache with TreeCache -->
<property name="cache.provider_class">
    org.jboss.ejb3.entity.TreeCacheProviderHook
</property>

<property name="treecache.mbean.object_name">
    jboss.cache:service=EJB3EntityTreeCache
```

```
</property>
```

The on each entity bean class you want to cache, add the @Cache annotation.

```
@Entity
@Cache(usage=CacheConcurrencyStrategy.TRANSACTIONAL)
public class Product implements Serializable {
  // ... ...
}
```

That's it. Now the Product data objects are cached in the shared cache across all JBoss AS server nodes in the cluster.

3. Basic optimization

In this section, we will give a few optimization tips to improve the performance of the cluster. Details of those optimization techniques will be covered later in this book.

The first tip is to balance the connection pools in JBoss AS server nodes and the load balancer. The goal is to make the Apache connection poll 80% the size of the combined connection pools of the JBoss AS nodes. In addition, to optimize thread performance in the load balancer, we highly recommend you install the "worker" MPM for Apache on Unix / Linux servers and the "winnt" MPM for Windows servers.

Second, we need to understand that HTTP session replication is very expensive and should be minimized when possible. Choosing the right replication trigger and granularity is an important step. For instance, setting the replication trigger to SET might require you to write more code to push changed data back into the session. But it can drastically reduce the replication work load for a mostly read-only application. Setting the granularity level to ATTRIBUTE would be much more efficient than SESSION if the session size is large. However, setting the granularity level to FIELD may not have a great benefit due to the AOP overhead associated with dirty checking on fields.

In addition, as we discussed before, the default "replicate-to-all-nodes" approach for HTTP session replication does scale for large clusters.

At the network level, it is important to make sure that you use an asynchronous communication mechanism to replicate HTTP sessions. Since the replicated HTTP sessions are only used when a rare failover happens, it is not essential for the failover node to always have the exact same state as the primary node at real time.

It also worth noting that the choice of underlying network protocols for the inter-node communication also affects performance. In general, on a small network (i.e., 4 or less nodes)

the node-to-node TCP/IP protocol would out perform the broadcast-based UDP protocol. It is the opposite when the cluster size grows larger.

JBossCache and JGroups Services

JGroups and JBossCache provide the underlying communication, node replication and caching services, for JBoss AS clusters. Those services are configured as MBeans. There is a set of JBossCache and JGroups MBeans for each type of clustering applications (e.g., the Stateful Session EJBs, the distributed entity EJBs etc.).

The JBoss AS ships with a reasonable set of default JGroups and JBossCache MBean configurations. Most applications just work out of the box with the default MBean configurations. You only need to tweak them when you are deploying an application that has special network or performance requirements.

1. JGroups Configuration

The JGroups framework provides services to enable peer-to-peer communications between nodes in a cluster. It is built on top a stack of network communication protocols that provide transport, discovery, reliability and failure detection, and cluster membership management services. *Figure 12.1, "Protocol stack in JGroups"* shows the protocol stack in JGroups.

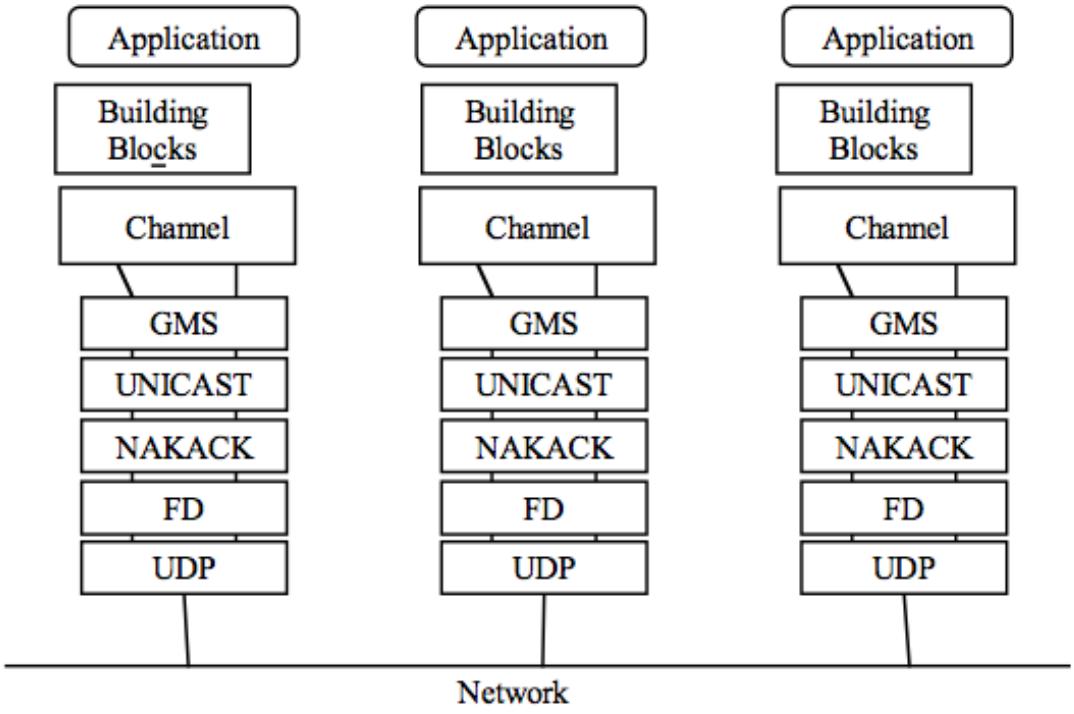


Figure 12.1. Protocol stack in JGroups

JGroups configurations often appear as a nested attribute in cluster related MBean services, such as the `PartitionConfig` attribute in the `ClusterPartition` MBean or the `ClusterConfig` attribute in the `TreeCache` MBean. You can configure the behavior and

properties of each protocol in JGroups via those MBean attributes. Below is an example JGroups configuration in the `ClusterPartition` MBean.

```
<mbean code="org.jboss.ha.framework.server.ClusterPartition"
    name="jboss:service=DefaultPartition">

    ... ...

    <attribute name="PartitionConfig">
        <Config>
            <UDP mcast_addr="228.1.2.3" mcast_port="45566"
               ip_ttl="8" ip_mcast="true"
               mcast_send_buf_size="800000" mcast_recv_buf_size="150000"
               ucast_send_buf_size="800000" ucast_recv_buf_size="150000"
               loopback="false"/>
            <PING timeout="2000" num_initial_members="3"
               up_thread="true" down_thread="true"/>
            <MERGE2 min_interval="10000" max_interval="20000"/>
            <FD shun="true" up_thread="true" down_thread="true"
               timeout="2500" max_tries="5"/>
            <VERIFY_SUSPECT timeout="3000" num_msgs="3"
               up_thread="true" down_thread="true"/>
            <pbcast.NAKACK gc_lag="50"
               retransmit_timeout="300,600,1200,2400,4800"
               max_xmit_size="8192"
               up_thread="true" down_thread="true"/>
            <UNICAST timeout="300,600,1200,2400,4800"
               window_size="100" min_threshold="10"
               down_thread="true"/>
            <pbcast.STABLE desired_avg_gossip="20000"
               up_thread="true" down_thread="true"/>
            <FRAG frag_size="8192"
               down_thread="true" up_thread="true"/>
            <pbcast.GMS join_timeout="5000" join_retry_timeout="2000"
               shun="true" print_local_addr="true"/>
            <pbcast.STATE_TRANSFER up_thread="true" down_thread="true"/>
        </Config>
    </attribute>
</mbean>
```

All the JGroups configuration data is contained in the `<Config>` element under the JGroups config MBean attribute. In the next several sections, we will dig into the options in the `<Config>` element and explain exactly what they mean.

1.1. Transport Protocols

The transport protocols send messages from one cluster node to another (unicast) or from cluster node to all other nodes in the cluster (mcast). JGroups supports UDP, TCP, and TUNNEL as transport protocols.

Note

The `UDP`, `TCP`, and `TUNNEL` elements are mutually exclusive. You can only have one transport protocol in each JGroups `Config` element

1.1.1. UDP configuration

UDP is the preferred protocol for JGroups. UDP uses multicast or multiple unicasts to send and receive messages. If you choose UDP as the transport protocol for your cluster service, you need to configure it in the `UDP` sub-element in the JGroups `Config` element. Here is an example.

```
<UDP mcast_send_buf_size="32000"
    mcast_port="45566"
    ucast_recv_buf_size="64000"
    mcast_addr="228.8.8.8"
    bind_to_all_interfaces="true"
    loopback="true"
    mcast_recv_buf_size="64000"
    max_bundle_size="30000"
    max_bundle_timeout="30"
    use_incoming_packet_handler="false"
    use_outgoing_packet_handler="false"
    ucast_send_buf_size="32000"
    ip_ttl="32"
    enable_bundling="false"/>
```

The available attributes in the above JGroups configuration are listed below.

- **ip_mcast** specifies whether or not to use IP multicasting. The default is `true`.
- **mcast_addr** specifies the multicast address (class D) for joining a group (i.e., the cluster). The default is `228.8.8.8`.
- **mcast_port** specifies the multicast port number. The default is `45566`.
- **bind_addr** specifies the interface on which to receive and send multicasts (uses the `bind.address` system property, if present). If you have a multihomed machine, set the `bind_addr` attribute to the appropriate NIC IP address. Ignored if the `ignore.bind.address` property is true.
- **bind_to_all_interfaces** specifies whether this node should listen on all interfaces for multicasts. The default is `false`. It overrides the `bind_addr` property for receiving multicasts. However, `bind_addr` (if set) is still used to send multicasts.

- **ip_ttl** specifies the TTL for multicast packets.
- **use_incoming_packet_handler** specifies whether to use a separate thread to process incoming messages.
- **use_outgoing_packet_handler** specifies whether to use a separate thread to process outgoing messages.
- **enable_bundling** specifies whether to enable bundling. If it is `true`, the node would queue outgoing messages until `max_bundle_size` bytes have accumulated, or `max_bundle_time` milliseconds have elapsed, whichever occurs first. Then bundle queued messages into a large message and send it. The messages are unbundled at the receiver. The default is `false`.
- **loopback** specifies whether to loop outgoing message back up the stack. In `unicast` mode, the messages are sent to self. In `mcast` mode, a copy of the mcast message is sent.
- **discard_incompatibe_packets** specifies whether to discard packets from different JGroups versions. Each message in the cluster is tagged with a JGroups version. When a message from a different version of JGroups is received, it will be discarded if set to true, otherwise a warning will be logged.
- **mcast_send_buf_size, mcast_recv_buf_size, ucast_send_buf_size, ucast_recv_buf_size** define receive and send buffer sizes. It is good to have a large receiver buffer size, so packets are less likely to get dropped due to buffer overflow.

Note

On Windows 2000 machines, because of the media sense feature being broken with multicast (even after disabling media sense), you need to set the UDP protocol's `loopback` attribute to `true`.

1.1.2. TCP configuration

Alternatively, a JGroups-based cluster can also work over TCP connections. Compared with UDP, TCP generates more network traffic when the cluster size increases but TCP is more reliable. TCP is fundamentally a unicast protocol. To send multicast messages, JGroups uses multiple TCP unicasts. To use TCP as a transport protocol, you should define a `TCP` element in the JGroups `Config` element. Here is an example of the `TCP` element.

```
<TCP start_port="7800"
    bind_addr="192.168.5.1"
    loopback="true"/>
```

Below are the attributes available in the `TCP` element.

- **bind_addr** specifies the binding address. It can also be set with the `-Dbind.address` command line option at server startup.
- **start_port, end_port** define the range of TCP ports the server should bind to. The server socket is bound to the first available port from `start_port`. If no available port is found (e.g., because of a firewall) before the `end_port`, the server throws an exception.
- **loopback** specifies whether to loop outgoing message back up the stack. In `unicast` mode, the messages are sent to self. In `mcast` mode, a copy of the mcast message is sent.
- **mcast_send_buf_size, mcast_recv_buf_size, ucast_send_buf_size, ucast_recv_buf_size** define receive and send buffer sizes. It is good to have a large receiver buffer size, so packets are less likely to get dropped due to buffer overflow.
- **conn_expire_time** specifies the time (in milliseconds) after which a connection can be closed by the reaper if no traffic has been received.
- **reaper_interval** specifies interval (in milliseconds) to run the reaper. If both values are 0, no reaping will be done. If either value is > 0, reaping will be enabled.

1.1.3. TUNNEL configuration

The TUNNEL protocol uses an external router to send messages. The external router is known as a `GossipRouter`. Each node has to register with the router. All messages are sent to the router and forwarded on to their destinations. The TUNNEL approach can be used to setup communication with nodes behind firewalls. A node can establish a TCP connection to the GossipRouter through the firewall (you can use port 80). The same connection is used by the router to send messages to nodes behind the firewall. The TUNNEL configuration is defined in the `TUNNEL` element in the JGroups `Config` element. Here is an example.

```
<TUNNEL router_port="12001"
    router_host="192.168.5.1"/>
```

The available attributes in the `TUNNEL` element are listed below.

- **router_host** specifies the host on which the GossipRouter is running.
- **router_port** specifies the port on which the GossipRouter is listening.
- **loopback** specifies whether to loop messages back up the stack. The default is `true`.

1.2. Discovery Protocols

The cluster need to maintain a list of current member nodes at all times so that the load balancer and client interceptor know how to route their requests. The discovery protocols are used to discover active nodes in the cluster. All initial nodes are discovered when the cluster starts up. When a new node joins the cluster later, it is only discovered after the group membership protocol (GMS, see *Section 1.5.1, "Group Membership"*) admits it into the group.

Since the discovery protocols sit on top of the transport protocol. You can choose to use different discovery protocols based on your transport protocol. The discovery protocols are also configured as sub-elements in the JGroups MBean `Config` element.

1.2.1. PING

The PING discovery protocol normally sits on top of the UDP transport protocol. Each node responds with a unicast UDP datagram back to the sender. Here is an example PING configuration under the JGroups `Config` element.

```
<PING timeout="2000"
    num_initial_members="2"/>
```

The available attributes in the `PING` element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for any responses.
- **num_initial_members** specifies the maximum number of responses to wait for.
- **gossip_host** specifies the host on which the GossipRouter is running.
- **gossip_port** specifies the port on which the GossipRouter is listening on.
- **gossip_refresh** specifies the interval (in milliseconds) for the lease from the GossipRouter.
- **initial_hosts** is a comma-separated list of addresses (e.g., `host1[12345],host2[23456]`), which are pinged for discovery.

If both `gossip_host` and `gossip_port` are defined, the cluster uses the GossipRouter for the initial discovery. If the `initial_hosts` is specified, the cluster pings that static list of addresses for discovery. Otherwise, the cluster uses IP multicasting for discovery.

Note

The discovery phase returns when the `timeout` ms have elapsed or the `num_initial_members` responses have been received.

1.2.2. TCPGOSSIP

The TCPGOSSIP protocol only works with a GossipRouter. It works essentially the same way as the PING protocol configuration with valid `gossip_host` and `gossip_port` attributes. It works on top of both UDP and TCP transport protocols. Here is an example.

```
<PING timeout="2000"
    initial_hosts="192.168.5.1[12000],192.168.0.2[12000]"
    num_initial_members="3"/>
```

The available attributes in the `TCPGOSSIP` element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for any responses.
- **num_initial_members** specifies the maximum number of responses to wait for.
- **initial_hosts** is a comma-separated list of addresses (e.g., `host1[12345],host2[23456]`) for GossipRouters to register with.

1.2.3. TCPPING

The TCPPING protocol takes a set of known members and ping them for discovery. This is essentially a static configuration. It works on top of TCP. Here is an example of the `TCPPING` configuration element in the JGroups `Config` element.

```
<TCPPING timeout="2000"
    initial_hosts="192.168.5.1[7800],192.168.0.2[7800]"
    port_range="2"
    num_initial_members="3"/>
```

The available attributes in the `TCPPING` element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for any responses.
- **num_initial_members** specifies the maximum number of responses to wait for.
- **initial_hosts** is a comma-separated list of addresses (e.g., `host1[12345],host2[23456]`) for pinging.
- **port_range** specifies the range of ports to ping on each host in the `initial_hosts` list. That is because multiple nodes can run on the same host. In the above example, the cluster would ping ports 7800, 7801, and 7802 on both hosts.

1.2.4. MPING

The MPING protocol is a multicast ping over TCP. It works almost the same way as PING works on UDP. It does not require external processes (GossipRouter) or static configuration (initial host list). Here is an example of the `MPING` configuration element in the JGroups `Config` element.

```
<MPING timeout="2000"
    bind_to_all_interfaces="true"
    mcast_addr="228.8.8.8"
    mcast_port="7500"
    ip_ttl="8"
    num_initial_members="3"/>
```

The available attributes in the `MPING` element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for any responses.
- **num_initial_members** specifies the maximum number of responses to wait for.
- **bind_addr** specifies the interface on which to send and receive multicast packets.
- **bind_to_all_interfaces** overrides the `bind_addr` and uses all interfaces in multihome nodes.
- **mcast_addr, mcast_port, ip_ttl** attributes are the same as related attributes in the UDP protocol configuration.

1.3. Failure Detection Protocols

The failure detection protocols are used to detect failed nodes. Once a failed node is detected, the cluster updates its view so that the load balancer and client interceptors know to avoid the dead node. The failure detection protocols are configured as sub-elements in the JGroups MBean `Config` element.

1.3.1. FD

The FD discovery protocol requires each node periodically sends are-you-alive messages to its neighbor. If the neighbor fails to respond, the calling node sends a SUSPECT message to the cluster. The current group coordinator double checks that the suspect node is indeed dead and updates the cluster's view. Here is an example FD configuration.

```
<FD timeout="2000"
    max_tries="3"
    shun="true"/>
```

The available attributes in the `FD` element are listed below.

- **timeout** specifies the maximum number of milliseconds to wait for the responses to the are-you-alive messages.
- **max_tries** specifies the number of missed are-you-alive messages from a node before the node is suspected.
- **shun** specifies whether a failed node will be shunned. Once shunned, the node will be expelled from the cluster even if it comes back later. The shunned node would have to re-join the cluster through the discovery process.

Note

Regular traffic from a node counts as if it is a live. So, the are-you-alive messages are only sent when there is no regular traffic to the node for sometime.

1.3.2. FD_SOCK

The are-you-alive messages in the FD protocol could increase the network load when there are many nodes. It could also produce false suspicions. For instance, if the network is too busy and the timeout is too short, nodes could be falsely suspected. Also, if one node is suspended in a debugger or profiler, it could also be suspected and shunned. The FD_SOCK protocol addresses the above issues by suspecting node failures only when a regular TCP connection to the node fails. However, the problem with such passive detection is that hung nodes will not be detected until it is accessed and the TCP timeouts after several minutes. FD_SOCK works best in high load networks where all nodes are frequently accessed. The simplest FD_SOCK configuration does not take any attribute. You can just declare an empty `FD_SOCK` element in JGroups's `Config` element.

```
<FD_SOCK/>
```

There is only one optional attribute in the `FD_SOCK` element.

- **srv_sock_bind_addr** specifies the interface to which the server socket should bind to. If it is omitted, the `-D bind.address` property from the server startup command line is used.

1.3.3. FD_SIMPLE

The FD_SIMPLE protocol is a more tolerant (less false suspicions) protocol based on

are-you-alive messages. Each node periodically sends are-you-alive messages to a randomly chosen node and wait for a response. If a response has not been received within a certain timeout time, a counter associated with that node will be incremented. If the counter exceeds a certain value, that node will be suspected. When a response to an are-you-alive message is received, the counter resets to zero. Here is an example configuration for the `FD_SIMPLE` protocol.

```
<FD_SIMPLE timeout="2000"
    max_missed_hbs="10"/>
```

The available attributes in the `FD_SIMPLE` element are listed below.

- **timeout** specifies the timeout (in milliseconds) for the are-you-alive message. If a response is not received within timeout, the counter for the target node is increased.
- **max_missed_hbs** specifies maximum number of are-you-alive messages (i.e., the counter value) a node can miss before it is suspected failure.

1.4. Reliable Delivery Protocols

The reliable delivery protocols in the JGroups stack ensure that data pockets are actually delivered in the right order (FIFO) to the destination node. The basis for reliable message delivery is positive and negative delivery acknowledgments (ACK and NAK). In the ACK mode, the sender resends the message until the acknowledgment is received from the receiver. In the NAK mode, the receiver requests retransmission when it discovers a gap.

1.4.1. UNICAST

The UNICAST protocol is used for unicast messages. It uses ACK. It is configured as a sub-element under the JGroups `Config` element. Here is an example configuration for the `UNICAST` protocol.

```
<UNICAST timeout="100,200,400,800"/>
```

There is only one configurable attribute in the `UNICAST` element.

- **timeout** specifies the retransmission timeout (in milliseconds). For instance, if the timeout is "100,200,400,800", the sender resends the message if it hasn't received an ACK after 100 ms the first time, and the second time it waits for 200 ms before resending, and so on.

1.4.2. NAKACK

The NAKACK protocol is used for multicast messages. It uses NAK. Under this protocol, each message is tagged with a sequence number. The receiver keeps track of the sequence numbers and deliver the messages in order. When a gap in the sequence number is detected, the receiver asks the sender to retransmit the missing message. The NAKACK protocol is configured as the `pbcast.NAKACK` sub-element under the JGroups `Config` element. Here is an example configuration.

```
<pbcast.NAKACK
    max_xmit_size="8192"
    use_mcast_xmit="true"
    retransmit_timeout="600,1200,2400,4800"/>
```

The configurable attributes in the `pbcast.NAKACK` element are as follows.

- **retransmit_timeout** specifies the retransmission timeout (in milliseconds). It is the same as the `timeout` attribute in the UNICAST protocol.
- **use_mcast_xmit** determines whether the sender should send the retransmission to the entire cluster rather than just the node requesting it. This is useful when the sender drops the pocket -- so we do not need to retransmit for each node.
- **max_xmit_size** specifies maximum size for a bundled retransmission, if multiple packets are reported missing.
- **discard_delivered_msgs** specifies whether to discard delivery messages on the receiver nodes. By default, we save all delivered messages. However, if we only ask the sender to resend their messages, we can enable this option and discard delivered messages.

1.5. Other Configuration Options

In addition to the protocol stacks, you can also configure JGroups network services in the `Config` element.

1.5.1. Group Membership

The group membership service in the JGroups stack maintains a list of active nodes. It handles the requests to join and leave the cluster. It also handles the SUSPECT messages sent by failure detection protocols. All nodes in the cluster, as well as the load balancer and client side interceptors, are notified if the group membership changes. The group membership service is configured in the `pbcast.GMS` sub-element under the JGroups `Config` element. Here is an example configuration.

```
```

```
<pbcast.GMS print_local_addr="true"
    join_timeout="3000"
    down_thread="false"
    join_retry_timeout="2000"
    shun="true"/>
```

The configurable attributes in the `pbcast.GMS` element are as follows.

- **join_timeout** specifies the maximum number of milliseconds to wait for a new node JOIN request to succeed. Retry afterwards.
- **join_retry_timeout** specifies the maximum number of milliseconds to wait after a failed JOIN to re-submit it.
- **print_local_addr** specifies whether to dump the node's own address to the output when started.
- **shun** specifies whether a node should shun itself if it receives a cluster view that it is not a member node.
- **disable_initial_coord** specifies whether to prevent this node as the cluster coordinator.

1.5.2. Flow Control

The flow control service tries to adapt the sending data rate and the receiving data among nodes. If a sender node is too fast, it might overwhelm the receiver node and result in dropped packets that have to be retransmitted. In JGroups, the flow control is implemented via a credit-based system. The sender and receiver nodes have the same number of credits (bytes) to start with. The sender subtracts credits by the number of bytes in messages it sends. The receiver accumulates credits for the bytes in the messages it receives. When the sender's credit drops to a threshold, the receivers sends some credit to the sender. If the sender's credit is used up, the sender blocks until it receives credits from the receiver. The flow control service is configured in the `FC` sub-element under the JGroups `Config` element. Here is an example configuration.

```
<FC max_credits="1000000"
    down_thread="false"
    min_threshold="0.10"/>
```

The configurable attributes in the `FC` element are as follows.

- **max_credits** specifies the maximum number of credits (in bytes). This value should be

smaller than the JVM heap size.

- **min_credits** specifies the threshold credit on the sender, below which the receiver should send in more credits.
- **min_threshold** specifies percentage value of the threshold. It overrides the `min_credits` attribute.

1.5.3. State Transfer

The state transfer service transfers the state from an existing node (i.e., the cluster coordinator) to a newly joining node. It is configured in the `pbcast.STATE_TRANSFER` sub-element under the JGroups `Config` element. It does not have any configurable attribute. Here is an example configuration.

```
<pbcast.STATE_TRANSFER
    down_thread="false"
    up_thread="false"/>
```

1.5.4. Distributed Garbage Collection

In a JGroups cluster, all nodes have to store all messages received for potential retransmission in case of a failure. However, if we store all messages forever, we will run out of memory. So, the distributed garbage collection service in JGroups periodically purges messages that have seen by all nodes from the memory in each node. The distributed garbage collection service is configured in the `pbcast.STABLE` sub-element under the JGroups `Config` element. Here is an example configuration.

```
<pbcast.STABLE stability_delay="1000"
    desired_avg_gossip="5000"
    down_thread="false"
    max_bytes="250000"/>
```

The configurable attributes in the `pbcast.STABLE` element are as follows.

- **desired_avg_gossip** specifies intervals (in milliseconds) of garbage collection runs. Value `0` disables this service.
- **max_bytes** specifies the maximum number of bytes received before the cluster triggers a garbage collection run. Value `0` disables this service.
- **max_gossip_runs** specifies the maximum garbage collections runs before any changes. After this number is reached, there is no garbage collection until the message is received.

Note

Set the `max_bytes` attribute when you have a high traffic cluster.

1.5.5. Merging

When a network error occurs, the cluster might be partitioned into several different partitions. JGroups has a MERGE service that allows the coordinators in partitions to communicate with each other and form a single cluster back again. The flow control service is configured in the `MERGE2` sub-element under the JGroups `Config` element. Here is an example configuration.

```
<MERGE2 max_interval="10000"
    min_interval="2000"/>
```

The configurable attributes in the `FC` element are as follows.

- **max_interval** specifies the maximum number of milliseconds to send out a MERGE message.
- **min_interval** specifies the minimum number of milliseconds to send out a MERGE message.

JGroups chooses a random value between `min_interval` and `max_interval` to send out the MERGE message.

Note

The cluster states are not merged in a merger. This has to be done by the application.

2. JBossCache Configuration

JBoss Cache provides distributed cache and state replication services for the JBoss cluster. A JBoss cluster can have multiple JBoss Cache MBeans (known as the `TreeCache` MBean), one for HTTP session replication, one for stateful session beans, one for cached entity beans, etc. A generic `TreeCache` MBean configuration is listed below. Application specific `TreeCache` MBean configurations are covered in later chapters when those applications are discussed.

```
<mbean code="org.jboss.cache.TreeCache"
       name="jboss.cache:service=TreeCache">
```

```
<depends>jboss:service=Naming</depends>
<depends>jboss:service=TransactionManager</depends>

<! -- Configure the TransactionManager -->
<attribute name="TransactionManagerLookupClass">
    org.jboss.cache.DummyTransactionManagerLookup
</attribute>

<! --
        Node locking level : SERIALIZABLE
                             REPEATABLE_READ (default)
                             READ_COMMITTED
                             READ_UNCOMMITTED
                             NONE
-->
<attribute name="IsolationLevel">REPEATABLE_READ</attribute>

<! --     Valid modes are LOCAL
                         REPL_ASYNC
                         REPL_SYNC
-->
<attribute name="CacheMode">LOCAL</attribute>

<! -- Name of cluster. Needs to be the same for all clusters, in order
         to find each other -->
<attribute name="ClusterName">TreeCache-Cluster</attribute>

<! --    The max amount of time (in milliseconds) we wait until the
        initial state (ie. the contents of the cache) are
        retrieved from existing members in a clustered environment
-->
<attribute name="InitialStateRetrievalTimeout">5000</attribute>

<! --    Number of milliseconds to wait until all responses for a
        synchronous call have been received.
-->
<attribute name="SyncReplTimeout">10000</attribute>

<! --  Max number of milliseconds to wait for a lock acquisition -->
<attribute name="LockAcquisitionTimeout">15000</attribute>

<! --  Name of the eviction policy class. -->
<attribute name="EvictionPolicyClass">
    org.jboss.cache.eviction.LRUPolicy
</attribute>

<! --  Specific eviction policy configurations. This is LRU -->
<attribute name="EvictionPolicyConfig">
    <config>
        <attribute name="wakeUpIntervalSeconds">5</attribute>
        <!--  Cache wide default -->
        <region name="/_default_">
            <attribute name="maxNodes">5000</attribute>
            <attribute name="timeToLiveSeconds">1000</attribute>
        </region>
```

```
            <region name="/org/jboss/data">
                <attribute name="maxNodes">5000</attribute>
                <attribute name="timeToLiveSeconds">1000</attribute>
            </region>

            <region name="/org/jboss/test/data">
                <attribute name="maxNodes">5</attribute>
                <attribute name="timeToLiveSeconds">4</attribute>
            </region>
        </config>
    </attribute>

    <attribute name="CacheLoaderClass">
        org.jboss.cache.loader.bdbje.BdbjeCacheLoader
    </attribute>

    <attribute name="CacheLoaderConfig">
       location=c:\\tmp
    </attribute>
    <attribute name="CacheLoaderShared">true</attribute>
    <attribute name="CacheLoaderPreload">
        /a/b/c,/all/my/objects
    </attribute>
    <attribute name="CacheLoaderFetchTransientState">false</attribute>
    <attribute name="CacheLoaderFetchPersistentState">true</attribute>

    <attribute name="ClusterConfig">
        ... JGroups config for the cluster ...
    </attribute>
</mbean>
```

The JGroups configuration element (i.e., the `ClusterConfig` attribute) is omitted from the above listing. You have learned how to configure JGroups earlier in this chapter (*Section 1, "JGroups Configuration"*). The `TreeCache` MBean takes the following attributes.

- **CacheLoaderClass** specifies the fully qualified class name of the `CacheLoader` implementation.
- **CacheLoaderConfig** contains a set of properties from which the specific CacheLoader implementation can configure itself.
- **CacheLoaderFetchPersistentState** specifies whether to fetch the persistent state from another node. The persistence is fetched only if `CacheLoaderShared` is `false`. This attribute is only used if `FetchStateOnStartup` is `true`.
- **CacheLoaderFetchTransientState** specifies whether to fetch the in-memory state from another node. This attribute is only used if `FetchStateOnStartup` is `true`.
- **CacheLoaderPreload** contains a list of comma-separate nodes that need to be preloaded (e.g., `/aop`, `/productcatalogue`).

- **CacheLoaderShared** specifies whether we want to shared a datastore, or whether each node wants to have its own local datastore.
- **CacheMode** specifies how to synchronize cache between nodes. The possible values are `LOCAL`, `REPL_SYNC`, or `REPL_ASYNC`.
- **ClusterName** specifies the name of the cluster. This value needs to be the same for all nodes in a cluster in order for them to find each other.
- **ClusterConfig** contains the configuration of the underlying JGroups stack (see *Section 1, "JGroups Configuration"*.
- **EvictionPolicyClass** specifies the name of a class implementing `EvictionPolicy`. You can use a JBoss Cache provided `EvictionPolicy` class or provide your own policy implementation. If this attribute is empty, no eviction policy is enabled.
- **EvictionPolicyConfig** contains the configuration parameter for the specified eviction policy. Note that the content is provider specific.
- **FetchStateOnStartup** specifies whether or not to acquire the initial state from existing members. It allows for warm/hot caches (`true/false`). This can be further defined by `CacheLoaderFetchTransientState` and `CacheLoaderFetchPersistentState`.
- **InitialStateRetrievalTimeout** specifies the time in milliseconds to wait for initial state retrieval.
- **IsolationLevel** specifies the node locking level. Possible values are `SERIALIZABLE`, `REPEATABLE_READ` (default), `READ_COMMITTED`, `READ_UNCOMMITTED`, and `NONE`.
- **LockAcquisitionTimeout** specifies the time in milliseconds to wait for a lock to be acquired. If a lock cannot be acquired an exception will be thrown.
- **ReplQueueInterval** specifies the time in milliseconds for elements from the replication queue to be replicated.
- **SyncReplTimeout** specifies the time in milliseconds to wait until replication ACKs have been received from all nodes in the cluster. This attribute applies to synchronous replication mode only (i.e., `CacheMode` attribute is `REPL_SYNC`).
- **UseReplQueue** specifies whether or not to use a replication queue (`true/false`). This attribute applies to synchronous replication mode only (i.e., `CacheMode` attribute is `REPL_ASYNC`).
- **ReplQueueMaxElements** specifies the maximum number of elements in the replication queue until replication kicks in.
- **TransactionManagerLookupClass** specifies the fully qualified name of a class implementing `TransactionManagerLookup`. The default is `JBossTransactionManagerLookup` for the transaction manager inside the JBoss AS. There is also an option of `DummyTransactionManagerLookup` for simple standalone examples.

Clustering

High Availability Enterprise Services via JBoss Clusters

1. Introduction

Clustering allows us to run an applications on several parallel servers (a.k.a cluster nodes). The load is distributed across different servers, and even if any of the servers fails, the application is still accessible via other cluster nodes. Clustering is crucial for scalable enterprise applications, as you can improve performance by simply adding more nodes to the cluster.

The JBoss Application Server (AS) comes with clustering support out of the box. The simplest way to start a JBoss server cluster is to start several JBoss instances on the same local network, using the `run -c all` command for each instance. Those server instances, all started in the `all` configuration, detect each other and automatically form a cluster.

In the first section of this chapter, I discuss basic concepts behind JBoss's clustering services. It is important that you understand those concepts before reading the rest of the chapter. Clustering configurations for specific types of applications are covered after this section.

1.1. Cluster Definition

A cluster is a set of nodes. In a JBoss cluster, a node is a JBoss server instance. Thus, to build a cluster, several JBoss instances have to be grouped together (known as a "partition"). On a same network, we may have different clusters. In order to differentiate them, each cluster must have an individual name.

Figure 13.1, “Clusters and server nodes” shows an example network of JBoss server instances divided into three clusters, with each cluster only having one node. Nodes can be added to or removed from clusters at any time.

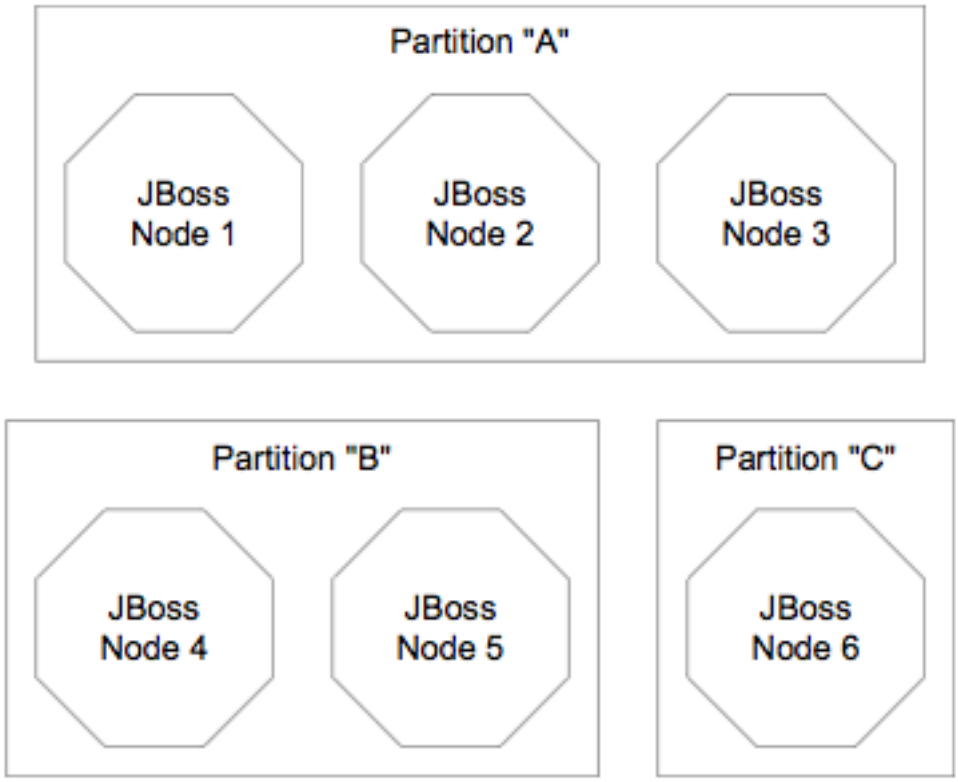


Figure 13.1. Clusters and server nodes

Note

While it is technically possible to put a JBoss server instance into multiple clusters at the same time, this practice is generally not recommended, as it increases the management complexity.

Each JBoss server instance (node) specifies which cluster (i.e., partition) it joins in the `ClusterPartition` MBean in the `deploy/cluster-service.xml` file. All nodes that have the same `ClusterPartition` MBean configuration join the same cluster. Hence, if you want to divide JBoss nodes in a network into two clusters, you can just come up with two different `ClusterPartition` MBean configurations, and each node would have one of the two configurations depending on which cluster it needs to join. If the designated cluster does not exist when the node is started, the cluster would be created. Likewise, a cluster is removed when all its nodes are removed.

The following example shows the MBean definition packaged with the standard JBoss AS distribution. So, if you simply start JBoss servers with their default clustering settings on a local network, you would get a default cluster named `DefaultPartition` that includes all server instances as its nodes.

```
<mbean code="org.jboss.ha.framework.server.ClusterPartition"
    name="jboss:service=DefaultPartition">

    <! -- Name of the partition being built -->
    <attribute name="PartitionName">
        ${jboss.partition.name:DefaultPartition}
    </attribute>

    <! -- The address used to determine the node name -->
    <attribute name="NodeAddress">${jboss.bind.address}</attribute>

    <! -- Determine if deadlock detection is enabled -->
    <attribute name="DeadlockDetection">False</attribute>

    <! -- Max time (in ms) to wait for state transfer to complete.
        Increase for large states -->
    <attribute name="StateTransferTimeout">30000</attribute>

    <! -- The JGroups protocol configuration -->
    <attribute name="PartitionConfig">
        ... ...
    </attribute>
</mbean>
```

Here, we omitted the detailed JGroups protocol configuration for this cluster. JGroups handles the underlying peer-to-peer communication between nodes, and its configuration is discussed in *Section 1, "JGroups Configuration"*. The following list shows the available configuration attributes in the `ClusterPartition` MBean.

- **PartitionName** is an optional attribute to specify the name of the cluster. Its default value is `DefaultPartition`.
- **NodeAddress** is an optional attribute to specify the binding IP address of this node.
- **DeadlockDetection** is an optional boolean attribute that tells JGroups to run message deadlock detection algorithms with every request. Its default value is `false`.
- **StateTransferTimeout** is an optional attribute to specify the timeout for state replication across the cluster (in milliseconds). Its default value is `30000`.
- **PartitionConfig** is an element to specify JGroup configuration options for this cluster (see *Section 1, "JGroups Configuration"*).

In order for nodes to form a cluster, they must have the exact same `PartitionName` and the `ParitionConfig` elements. Changes in either element on some but not all nodes would cause the cluster to split. It is generally easier to change the `ParitionConfig` (i.e., the address/port) to run multiple cluster rather than changing the `PartitionName` due to the mulititude of places the former needs to be changed in other configuration files. However, changing the

`PartitionName` is made easier in 4.2 due to the use of the `${jboss.partition.name}` property which allows the name to be change via a single `jboss.partition.name` system property

You can view the current cluster information by pointing your browser to the JMX console of any JBoss instance in the cluster (i.e., *http://hostname:8080/jmx-console/*) and then clicking on the `jboss:service=DefaultPartition` MBean (change the MBean name to reflect your cluster name if this node does not join `DefaultPartition`). A list of IP addresses for the current cluster members is shown in the `CurrentView` field.

Note

A cluster (partition) contains a set of nodes that work toward a same goal. Some clustering features require to sub-partition the cluster to achieve a better scalability. For example, let's imagine that we have a 10-node cluster and we want to replicate in memory the state of stateful session beans on all 10 different nodes to provide for fault-tolerant behaviour. It would mean that each node has to store a backup of the 9 other nodes. This would not scale at all (each node would need to carry the whole state cluster load). It is probably much better to have some kind of sub-partitions inside a cluster and have beans state exchanged only between nodes that are part of the same sub-partition. The future JBoss clustering implementation will support sub-partitions and it will allow the cluster administrator to determine the optimal size of a sub-partition. The sub-partition topology computation will be done dynamically by the cluster.

1.2. Service Architectures

The clustering topography defined by the `ClusterPartition` MBean on each node is of great importance to system administrators. But for most application developers, you are probably more concerned about the cluster architecture from a client application's point of view. JBoss AS supports two types of clustering architectures: client-side interceptors (a.k.a proxies or stubs) and load balancers.

1.2.1. Client-side interceptor

Most remote services provided by the JBoss application server, including JNDI, EJB, RMI and JBoss Remoting, require the client to obtain (e.g., to look up and download) a stub (or proxy) object. The stub object is generated by the server and it implements the business interface of the service. The client then makes local method calls against the stub object. The call is automatically routed across the network and invoked against service objects managed in the server. In a clustering environment, the server-generated stub object is also an interceptor that understand how to route calls to nodes in the cluster. The stub object figures out how to find the appropriate server node, marshal call parameters, un-marshall call results, return the results to the caller client.

The stub interceptors have updated knowledge about the cluster. For instance, they know the IP

addresses of all available server nodes, the algorithm to distribute load across nodes (see next section), and how to failover the request if the target node not available. With every service request, the server node updates the stub interceptor with the latest changes in the cluster. For instance, if a node drops out of the cluster, each of the client stub interceptor is updated with the new configuration the next time it connects to any active node in the cluster. All the manipulations on the service stub are transparent to the client application. The client-side interceptor clustering architecture is illustrated in *Figure 13.2, "The client-side interceptor (proxy) architecture for clustering"*.

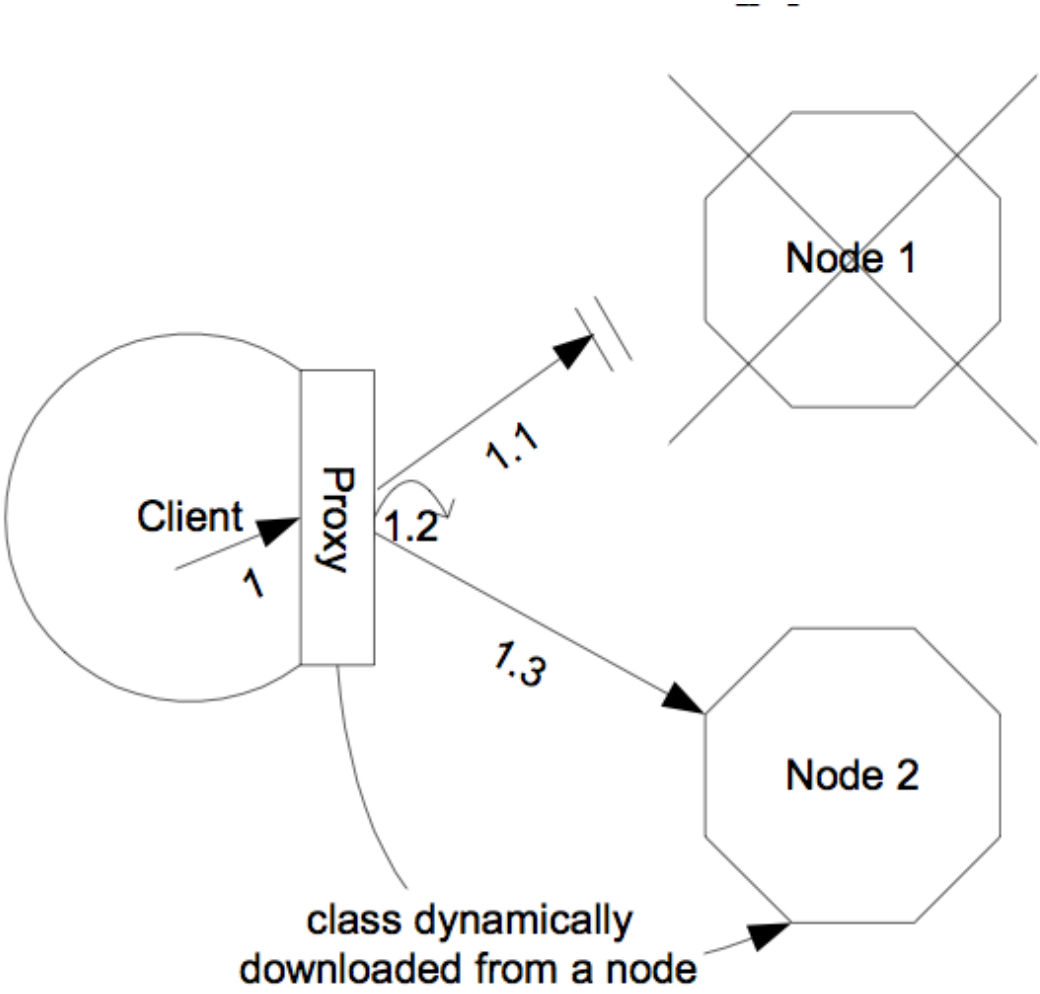


Figure 13.2. The client-side interceptor (proxy) architecture for clustering

Note

Section 3.1.1, "Handle Cluster Restart" describes how to enable the client proxy to handle the entire cluster restart.

1.2.2. Load balancer

Other JBoss services, in particular the HTTP web services, do not require the client to download anything. The client (e.g., a web browser) sends in requests and receives responses directly over the wire according to certain communication protocols (e.g., the HTTP protocol). In this case, a load balancer is required to process all requests and dispatch them to server nodes in the cluster. The load balancer is typically part of the cluster. It understands the cluster configuration as well as failover policies. The client only needs to know about the load balancer. The load balancer clustering architecture is illustrated in *Figure 13.3, "The load balancer architecture for clustering"*.

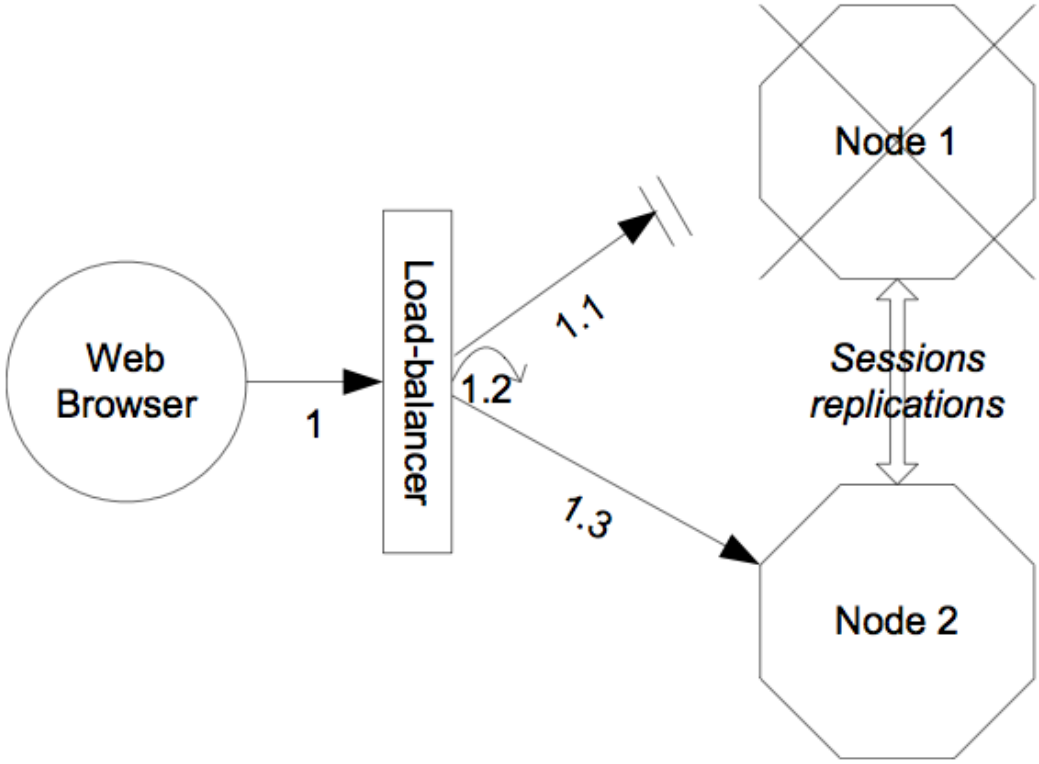


Figure 13.3. The load balancer architecture for clustering

A potential problem with the load balancer solution is that the load balancer itself is a single point of failure. It needs to be monitored closely to ensure high availability of the entire cluster services.

1.3. Load-Balancing Policies

Both the JBoss client-side interceptor (stub) and load balancer use load balancing policies to determine which server node to send a new request to. In this section, let's go over the load balancing policies available in JBoss AS.

1.3.1. JBoss AS 3.0.x

In JBoss 3.0.x, the following two load balancing options are available.

- Round-Robin (`org.jboss.ha.framework.interfaces.RoundRobin`): each call is dispatched to a new node. The first target node is randomly selected from the list.
- First Available (`org.jboss.ha.framework.interfaces.FirstAvailable`): one of the available target nodes is elected as the main target and is used for every call: this elected member is randomly chosen from the list of members in the cluster. When the list of target nodes changes (because a node starts or dies), the policy will re-elect a target node unless the currently elected node is still available. Each client-side interceptor or load balancer elects its own target node independently of the other proxies.

1.3.2. JBoss AS 3.2+

In JBoss 3.2+, three load balancing options are available. The Round-Robin and First Available options have the same meaning as the ones in JBoss AS 3.0.x.

The new load balancing option in JBoss 3.2 is "First AvailableIdenticalAllProxies" (`org.jboss.ha.framework.interfaces.FirstAvailableIdenticalAllProxies`). It has the same behaviour as the "First Available" policy but the elected target node is shared by all client-side interceptors of the same "family".

In JBoss 3.2 (and later), the notion of "Proxy Family" is defined. A Proxy Family is a set of stub interceptors that all make invocations against the same replicated target. For EJBs for example, all stubs targeting the same EJB in a given cluster belong to the same proxy family. All interceptors of a given family share the same list of target nodes. Each interceptor also has the ability to share arbitrary information with other interceptors of the same family. A use case for the proxy family is give in *Section 3.1, “Stateless Session Bean in EJB 2.x”*.

1.4. Farming Deployment

The easiest way to deploy an application into the cluster is to use the farming service. That is to hot-deploy the application archive file (e.g., the EAR, WAR or SAR file) in the`all/farm/`directory of any of the cluster member and the application is automatically duplicated across all nodes in the same cluster. If node joins the cluster later, it will pull in all farm deployed applications in the cluster and deploy them locally at start-up time. If you delete the application from one of the running cluster server node's `farm/` folder, the application will be undeployed locally and then removed from all other cluster server nodes farm folder (triggers undeployment.) You should manually delete the application from the farm folder of any server node not currently connected to the cluster.

Note

You can only put archive files, not exploded directories, in the `farm` directory. This way, the application on a remote node is only deployed when the entire archive file is copied over. Otherwise, the application might be deployed (and failed) when the directory is only partially copied.

Farming is enabled by default in the `all` configuration in JBoss AS distributions, so you will not have to set it up yourself. The configuration file is located in the `deploy/deploy.last` directory. If you want to enable farming in your custom configuration, simply create the XML file shown below (named it `farm-service.xml`) and copy it to the JBoss deploy directory `$JBOSS_HOME/server/your_own_config/deploy`. Make sure that you custom configuration has clustering enabled.

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

    <mbean code="org.jboss.ha.framework.server.FarmMemberService"
           name="jboss:service=FarmMember,partition=DefaultPartition">
        ...
        <attribute name="PartitionName">DefaultPartition</attribute>
        <attribute name="ScanPeriod">5000</attribute>
        <attribute name="URLs">farm/</attribute>
    </mbean>
</server>
```

After deploying `farm-service.xml` you are ready to rumble. The required `FarmMemberService` MBean attributes for configuring a farm are listed below.

- **PartitionName** specifies the name of the cluster for this deployed farm. Its default value is `DefaultPartition`.
- **URLs** points to the directory where deployer watches for files to be deployed. This MBean will create this directory is if does not already exist. Also, "." pertains to the configuration directory (i.e., `$JBOSS_HOME/server/all/`).
- **ScanPeriod** specifies the interval at which the folder must be scanned for changes.. Its default value is `5000`.

The Farming service is an extension of the `URLDeploymentScanner`, which scans for hot deployments in `deploy/` directory. So, you can use all the attributes defined in the `URLDeploymentScanner` MBean in the `FarmMemberService` MBean. In fact, the `URLs` and `ScanPeriod` attributes listed above are inherited from the `URLDeploymentScanner` MBean.

Frequently Asked Questions on JBossFarm Deployment

1.4.1. I have multiple nodes running in a cluster. On one of the nodes that were down, I copied a new application archive to the farm deployment directory and then started the server. Why didn't the new application deploy to other cluster server nodes?

The new application is not supposed to deploy while the server is down. You have to start the server before updating the farm deployment directory. After starting the server, copy the application archive again to the farm deployment directory (or touch it). The application will then be deployed on the cluster as mentioned above.

1.4.2. I deleted an application from the farm deployment directory and then started the server. Why didn't the application get deleted from other cluster server nodes?

Applications will not be undeployed when a server is down. You have to start the server before updating the farm deployment directory. After starting the server, delete the application archive from the farm deployment directory. The application will then be removed from other cluster server nodes.

1.4.3. I deleted an application from the cluster but one of the server nodes was down at the time, when I started that node, the application was still on it. How do I delete the application from that node?

You have to manually delete the application from the node(s) that were down when the application was removed from the cluster. Delete the application archive from the farm deployment directory on the node(s) that were down.

1.5. Distributed state replication services

In a clustered server environment, distributed state management is a key service the cluster must provide. For instance, in a stateful session bean application, the session state must be synchronized among all bean instances across all nodes, so that the client application reaches the same session state no matter which node serves the request. In an entity bean application, the bean object sometimes needs to be cached across the cluster to reduce the database load. Currently, the state replication and distributed cache services in JBoss AS are provided via two ways: the `HASessionState` MBean and the JBoss Cache framework.

- The `HASessionState` MBean provides session replication and distributed cache services for EJB 2.x stateful session beans and HTTP load balancers in JBoss 3.x and 4.x. The MBean is defined in the `all/deploy/cluster-service.xml` file. We will show its configuration options in the EJB 2.x stateful session bean section later.

- JBoss Cache is a fully featured distributed cache framework that can be used in any application server environment and standalone. It gradually replaces the `HASessionState` service. JBoss AS integrates JBoss Cache to provide cache services for HTTP sessions, EJB 3.0 session and entity beans, as well as Hibernate persistence objects. Each of these cache services is defined in a separate MBean. We will cover those MBeans when we discuss specific services in the next several sections.

2. Clustered JNDI Services

JNDI is one of the most important services provided by the application server. The JBoss clustered JNDI service is based on the client-side interceptor architecture. The client must obtain a JNDI stub object (via the `InitialContext` object) and invoke JNDI lookup services on the remote server through the stub. Furthermore, JNDI is the basis for many other interceptor-based clustering services: those services register themselves with the JNDI so that the client can lookup their stubs and make use of their services.

2.1. How it works

The JBoss HA-JNDI (High Availability JNDI) service maintains a cluster-wide context tree. The cluster wide tree is always available as long as there is one node left in the cluster. Each JNDI node in the cluster also maintains its own local JNDI context. The server side application can bind its objects to either trees. In this section, you will learn the distinctions of the two trees and the best practices in application development. The design rational of this architecture is as follows.

- We didn't want any migration issues with applications already assuming that their JNDI implementation was local. We wanted clustering to work out-of-the-box with just a few tweaks of configuration files.
- We needed a clean distinction between locally bound objects and cluster-wide objects.
- In a homogeneous cluster, this configuration actually cuts down on the amount of network traffic.
- Designing it in this way makes the HA-JNDI service an optional service since all underlying cluster code uses a straight new `InitialContext()` to lookup or create bindings.

On the server side, `new InitialContext()`, will be bound to a local-only, non-cluster-wide JNDI Context (this is actually basic JNDI). So, all EJB homes and such will not be bound to the cluster-wide JNDI Context, but rather, each home will be bound into the local JNDI. When a remote client does a lookup through HA-JNDI, HA-JNDI will delegate to the local JNDI Context when it cannot find the object within the global cluster-wide Context. The detailed lookup rule is as follows.

- If the binding is available in the cluster-wide JNDI tree and it returns it.

- If the binding is not in the cluster-wide tree, it delegates the lookup query to the local JNDI service and returns the received answer if available.
- If not available, the HA-JNDI services asks all other nodes in the cluster if their local JNDI service owns such a binding and returns the an answer from the set it receives.
- If no local JNDI service owns such a binding, a `NameNotFoundException` is finally raised.

So, an EJB home lookup through HA-JNDI, will always be delegated to the local JNDI instance. If different beans (even of the same type, but participating in different clusters) use the same JNDI name, it means that each JNDI server will have a different "target" bound (JNDI on node 1 will have a binding for bean A and JNDI on node 2 will have a binding, under the same name, for bean B). Consequently, if a client performs a HA-JNDI query for this name, the query will be invoked on any JNDI server of the cluster and will return the locally bound stub. Nevertheless, it may not be the correct stub that the client is expecting to receive!

Note

You cannot currently use a non-JNP JNDI implementation (i.e. LDAP) for your local JNDI implementation if you want to use HA-JNDI. However, you can use JNDI federation using the `ExternalContext` MBean to bind non-JBoss JNDI trees into the JBoss JNDI namespace. Furthermore, nothing prevents you though of using one centralized JNDI server for your whole cluster and scrapping HA-JNDI and JNP.

Note

If a binding is only made available on a few nodes in the cluster (for example because a bean is only deployed on a small subset of nodes in the cluster), the probability to lookup a HA-JNDI server that does not own this binding is higher and the lookup will need to be forwarded to all nodes in the cluster. Consequently, the query time will be longer than if the binding would have been available locally. Moral of the story: as much as possible, cache the result of your JNDI queries in your client.

If you want to access HA-JNDI from the server side, you must explicitly get an `InitialContext` by passing in JNDI properties. The following code shows how to access the HA-JNDI.

```
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
      "org.jnp.interfaces.NamingContextFactory");
p.put(Context.URL_PKG_PREFIXES, "jboss.naming:org.jnp.interfaces");
p.put(Context.PROVIDER_URL, "localhost:1100"); // HA-JNDI port.
```

```
return new InitialContext(p);
```

The `Context.PROVIDER_URL` property points to the HA-JNDI service configured in the `HANamingService` MBean (see *Section 2.3, "JBoss configuration"*).

2.2. Client configuration

The JNDI client needs to be aware of the HA-JNDI cluster. You can pass a list of JNDI servers (i.e., the nodes in the HA-JNDI cluster) to the `java.naming.provider.url` JNDI setting in the `jndi.properties` file. Each server node is identified by its IP address and the JNDI port number. The server nodes are separated by commas (see *Section 2.3, "JBoss configuration"* on how to configure the servers and ports).

```
java.naming.provier.url=server1:1100,server2:1100,server3:1100,server4:1100
```

When initialising, the JNP client code will try to get in touch with each server node from the list, one after the other, stopping as soon as one server has been reached. It will then download the HA-JNDI stub from this node.

Note

There is no load balancing behavior in the JNP client lookup process. It just goes through the provider list and use the first available server. The HA-JNDI provider list only needs to contain a subset of HA-JNDI nodes in the cluster.

The downloaded smart stub contains the logic to fail-over to another node if necessary and the updated list of currently running nodes. Furthermore, each time a JNDI invocation is made to the server, the list of targets in the stub interceptor is updated (only if the list has changed since the last call).

If the property string `java.naming.provider.url` is empty or if all servers it mentions are not reachable, the JNP client will try to discover a bootstrap HA-JNDI server through a multicast call on the network (auto-discovery). See *Section 2.3, "JBoss configuration"* on how to configure auto-discovery on the JNDI server nodes. Through auto-discovery, the client might be able to get a valid HA-JNDI server node without any configuration. Of course, for the auto-discovery to work, the client must reside in the same LAN as the server cluster (e.g., the web servlets using the EJB servers). The LAN or WAN must also be configured to propagate such multicast datagrams.

Note

The auto-discovery feature uses multicast group address 230.0.0.4:1102.

In addition to the `java.naming.provier.url` property, you can specify a set of other properties. The following list shows all client side properties you can specify, when creating a new `InitialContext`.

- `java.naming.provier.url`: Provides a list of IP addresses and port numbers for HA-JNDI provider nodes in the cluster. The client tries those providers one by one and uses the first one that responds.
- `jnp.disableDiscovery`: When set to `true`, this property disables the automatic discovery feature. Default is `false`.
- `jnp.partitionName`: In an environment where multiple HA-JNDI services, which are bound to distinct clusters (i.e., partitions), are started, this property allows you to configure which cluster you broadcast to when the automatic discovery feature is used. If you do not use the automatic discovery feature (e.g., you could explicitly provide a list of valid JNDI nodes in `java.naming.provider.url`), this property is not used. By default, this property is not set and the automatic discovery select the first HA-JNDI server that responds, independently of the cluster partition name.
- `jnp.discoveryTimeout`: Determines how much time the context will wait for a response to its automatic discovery packet. Default is 5000 ms.
- `jnp.discoveryGroup`: Determines which multicast group address is used for the automatic discovery. Default is `230.0.0.4`.
- `jnp.discoveryPort`: Determines which multicast group port is used for the automatic discovery. Default is `1102`.

2.3. JBoss configuration

The `cluster-service.xml` file in the `all/deploy` directory includes the following MBean to enable HA-JNDI services.

```
<mbean code="org.jboss.ha.jndi.HANamingService"
       name="jboss:service=HAJNDI">
    <depends>jboss:service=DefaultPartition</depends>
</mbean>
```

You can see that this MBean depends on the `DefaultPartition` MBean defined above it

(discussed in an earlier section in this chapter). In other configurations, you can put that element in the `jboss-services.xml` file or any other JBoss configuration files in the `/deploy` directory to enable HA-JNDI services. The available attributes for this MBean are listed below.

- **PartitionName** is an optional attribute to specify the name of the cluster for the different nodes of the HA-JNDI service to communicate. The default value is `DefaultPartition`.
- **BindAddress** is an optional attribute to specify the address to which the HA-JNDI server will bind waiting for JNP clients. Only useful for multi-homed computers.
- **Port** is an optional attribute to specify the port to which the HA-JNDI server will bind waiting for JNP clients. The default value is `1100`.
- **Backlog** is an optional attribute to specify the backlog value used for the TCP server socket waiting for JNP clients. The default value is `50`.
- **RmiPort** determines which port the server should use to communicate with the downloaded stub. This attribute is optional. If it is missing, the server automatically assigns a RMI port.
- **AutoDiscoveryAddress** is an optional attribute to specify the multicast address to listen to for JNDI automatic discovery. The default value is `230.0.0.4`.
- **AutoDiscoveryGroup** is an optional attribute to specify the multicast group to listen to for JNDI automatic discovery.. The default value is `1102`.
- **LookupPool** specifies the thread pool service used to control the bootstrap and auto discovery lookups.
- **DiscoveryDisabled** is a boolean flag that disables configuration of the auto discovery multicast listener.
- **AutoDiscoveryBindAddress** sets the auto-discovery bootstrap multicast bind address. If this attribute is not specified and a `BindAddress` is specified, the `BindAddress` will be used..
- **AutoDiscoveryTTL** specifies the TTL (time-to-live) for autodiscovery IP multicast packets.

The full default configuration of the `HANamingService` MBean is as follows.

```
<mbean code="org.jboss.ha.jndi.HANamingService"
      name="jboss:service=HAJNDI">
    <depends>
        jboss:service=${jboss.partition.name:DefaultPartition}
    </depends>
    <! -- Name of the partition to which the service is linked -->
    <attribute name="PartitionName">
        ${jboss.partition.name:DefaultPartition}
    </attribute>
    <! -- Bind address of bootstrap and HA-JNDI RMI endpoints -->
    <attribute name="BindAddress">${jboss.bind.address}</attribute>
    <! -- Port on which the HA-JNDI stub is made available -->
    <attribute name="Port">1100</attribute>
```

```
    <! -- RmiPort to be used by the HA-JNDI service once bound.
        0 is for auto. -->
    <attribute name="RmiPort">1101</attribute>
    <! -- Accept backlog of the bootstrap socket -->
    <attribute name="Backlog">50</attribute>
    <! -- The thread pool service used to control the bootstrap and
      auto discovery lookups -->
    <depends optional-attribute-name="LookupPool"
        proxy-type="attribute">jboss.system:service=ThreadPool</depends>

    <! -- A flag to disable the auto discovery via multicast -->
    <attribute name="DiscoveryDisabled">false</attribute>
    <! -- Set the auto-discovery bootstrap multicast bind address. -->
    <attribute name="AutoDiscoveryBindAddress">
        ${jboss.bind.address}
    </attribute>

    <! -- Multicast Address and group port used for auto-discovery -->
    <attribute name="AutoDiscoveryAddress">
        ${jboss.partition.udpGroup:230.0.0.4}
    </attribute>
    <attribute name="AutoDiscoveryGroup">1102</attribute>
    <! -- The TTL (time-to-live) for autodiscovery IP multicast packets -->
    <attribute name="AutoDiscoveryTTL">16</attribute>

    <! -- Client socket factory to be used for client-server
           RMI invocations during JNDI queries
    <attribute name="ClientSocketFactory">custom</attribute>
    -->
    <! -- Server socket factory to be used for client-server
           RMI invocations during JNDI queries
    <attribute name="ServerSocketFactory">custom</attribute>
    -->
</mbean>
```

It is possible to start several HA-JNDI services that use different clusters. This can be used, for example, if a node is part of many clusters. In this case, make sure that you set a different port or IP address for both services. For instance, if you wanted to hook up HA-JNDI to the example cluster you set up and change the binding port, the Mbean descriptor would look as follows.

```
<mbean code="org.jboss.ha.jndi.HANamingService"
       name="jboss:service=HAJNDI">
    <depends>jboss:service=MySpecialPartition</depends>
    <attribute name="PartitionName">MySpecialPartition</attribute>
    <attribute name="Port">56789</attribute>
</mbean>
```

3. Clustered Session EJBs

Session EJBs provide remote invocation services. They are clustered based on the client-side interceptor architecture. The client application for a clustered session bean is exactly the same as the client for the non-clustered version of the session bean, except for a minor change to the `java.naming.provier.url` system property to enable HA-JNDI lookup (see previous section). No code change or re-compilation is needed on the client side. Now, let's check out how to configure clustered session beans in EJB 2.x and EJB 3.0 server applications respectively.

3.1. Stateless Session Bean in EJB 2.x

Clustering stateless session beans is most probably the easiest case: as no state is involved, calls can be load-balanced on any participating node (i.e. any node that has this specific bean deployed) of the cluster. To make a bean clustered, you need to modify its `jboss.xml` descriptor to contain a `<clustered>` tag.

```
<jboss>
    <enterprise-beans>
        <session>
            <ejb-name>nextgen.StatelessSession</ejb-name>
            <jndi-name>nextgen.StatelessSession</jndi-name>
            <clustered>True</clustered>
            <cluster-config>
                <partition-name>DefaultPartition</partition-name>
                <home-load-balance-policy>
                    org.jboss.ha.framework.interfaces.RoundRobin
                </home-load-balance-policy>
                <bean-load-balance-policy>
                    org.jboss.ha.framework.interfaces.RoundRobin
                </bean-load-balance-policy>
            </cluster-config>
        </session>
    </enterprise-beans>
</jboss>
```

Note

The `<clustered>True</clustered>` element is really just an alias for the `<configuration-name>Clustered Stateless SessionBean</configuration-name>` element.

In the bean configuration, only the `<clustered>` element is mandatory. It indicates that the bean works in a cluster. The `<cluster-config>` element is optional and the default values of its attributes are indicated in the sample configuration above. Below is a description of the attributes in the `<cluster-config>` element.

- **partition-name** specifies the name of the cluster the bean participates in. The default value is `DefaultPartition`. The default partition name can also be set system-wide using the `jboss.partition.name` system property.
- **home-load-balance-policy** indicates the class to be used by the home stub to balance calls made on the nodes of the cluster. By default, the proxy will load-balance calls in a `RoundRobin` fashion. You can also implement your own load-balance policy class or use the class `FirstAvailable` that persists to use the first node available that it meets until it fails.
- **bean-load-balance-policy** Indicates the class to be used by the bean stub to balance calls made on the nodes of the cluster. Comments made for the `home-load-balance-policy` attribute also apply.

In JBoss 3.0.x, each client-side stub has its own list of available target nodes. Consequently, some side-effects can occur. For example, if you cache your home stub and re-create a remote stub for a stateless session bean (with the Round-Robin policy) each time you need to make an invocation, a new remote stub, containing the list of available targets, will be downloaded for each invocation. Consequently, as the first target node is always the first in the list, calls will not seemed to be load-balanced because there is no usage-history between different stubs. In JBoss 3.2+, the proxy families (i.e., the "First AvailableIdenticalAllProxies" load balancing policy, see *Section 1.3.2, "JBoss AS 3.2+"*) remove this side effect as the home and remote stubs of a given EJB are in two different families.

3.1.1. Handle Cluster Restart

We have covered the HA smart client architecture in *Section 1.2.1, "Client-side interceptor"*. The default HA smart proxy client can only failover as long as one node in the cluster exists. If there is a complete cluster shutdown, the proxy becomes orphanned and looses knowledge of the available nodes in the cluster. There is no way for the proxy to recover from this. The proxy needs to be looked up out of JNDI/HAJNDI when the nodes are restarted.

The 3.2.7+/4.2+ releases contain a `RetryInterceptor` that can be added to the proxy client side interceptor stack to allow for a transparent recovery from such a restart failure. To enable it for an EJB, setup an `invoker-proxy-binding` that includes the `RetryInterceptor`. Below is an example `jboss.xml` configuration.

```
<jboss>
    <session>
        <ejb-name>nextgen_RetryInterceptorStatelessSession</ejb-name>
        <invoker-bindings>
            <invoker>
                <invoker-proxy-binding-name>
                    clustered-retry-stateless-rmi-invoker
                </invoker-proxy-binding-name>
                <jndi-name>
                    nextgen_RetryInterceptorStatelessSession
                </jndi-name>
            </invoker>
        </invoker-bindings>
```

```
        <clustered>true</clustered>
    </session>

    <invoker-proxy-binding>
        <name>clustered-retry-stateless-rmi-invoker</name>
        <invoker-mbean>jboss:service=invoker,type=jrmpha</invoker-mbean>
        <proxy-factory>org.jboss.proxy.ejb.ProxyFactoryHA</proxy-factory>
        <proxy-factory-config>
            <client-interceptors>
                <home>
                    <interceptor>
                        org.jboss.proxy.ejb.HomeInterceptor
                    </interceptor>
                    <interceptor>
                        org.jboss.proxy.SecurityInterceptor
                    </interceptor>
                    <interceptor>
                        org.jboss.proxy.TransactionInterceptor
                    </interceptor>
                    <interceptor>
                        org.jboss.proxy.ejb.RetryInterceptor
                    </interceptor>
                    <interceptor>
                        org.jboss.invocation.InvokerInterceptor
                    </interceptor>
                </home>
                <bean>
                    <interceptor>
                        org.jboss.proxy.ejb.StatelessSessionInterceptor
                    </interceptor>
                    <interceptor>
                        org.jboss.proxy.SecurityInterceptor
                    </interceptor>
                    <interceptor>
                        org.jboss.proxy.TransactionInterceptor
                    </interceptor>
                    <interceptor>
                        org.jboss.proxy.ejb.RetryInterceptor
                    </interceptor>
                    <interceptor>
                        org.jboss.invocation.InvokerInterceptor
                    </interceptor>
                </bean>
            </client-interceptors>
        </proxy-factory-config>
    </invoker-proxy-binding>
```

3.2. Stateful Session Bean in EJB 2.x

Clustering stateful session beans is more complex than clustering their stateless counterparts since JBoss needs to manage the state information. The state of all stateful session beans are replicated and synchronized across the cluster each time the state of a bean changes. The JBoss AS uses the `HASessionState` MBean to manage distributed session states for clustered

EJB 2.x stateful session beans. In this section, we cover both the session bean configuration and the `HASessionState` MBean configuration.

3.2.1. The EJB application configuration

In the EJB application, you need to modify the `jboss.xml` descriptor file for each stateful session bean and add the `<clustered>` tag.

```
<jboss>
    <enterprise-beans>
        <session>
            <ejb-name>nextgen.StatefulSession</ejb-name>
            <jndi-name>nextgen.StatefulSession</jndi-name>
            <clustered>True</clustered>
            <cluster-config>
                <partition-name>DefaultPartition</partition-name>
                <home-load-balance-policy>
                    org.jboss.ha.framework.interfaces.RoundRobin
                </home-load-balance-policy>
                <bean-load-balance-policy>
                    org.jboss.ha.framework.interfaces.FirstAvailable
                </bean-load-balance-policy>
                <session-state-manager-jndi-name>
                    /HASessionState/Default
                </session-state-manager-jndi-name>
            </cluster-config>
        </session>
    </enterprise-beans>
</jboss>
```

In the bean configuration, only the `<clustered>` tag is mandatory to indicate that the bean works in a cluster. The `<cluster-config>` element is optional and its default attribute values are indicated in the sample configuration above.

The `<session-state-manager-jndi-name>` tag is used to give the JNDI name of the `HASessionState` service to be used by this bean.

The description of the remaining tags is identical to the one for stateless session bean. Actions on the clustered stateful session bean's home interface are by default load-balanced, round-robin. Once the bean's remote stub is available to the client, calls will not be load-balanced round-robin any more and will stay "sticky" to the first node in the list.

3.2.2. Optimize state replication

As the replication process is a costly operation, you can optimise this behaviour by optionally implementing in your bean class a method with the following signature:

```
public boolean isModified ();
```

Before replicating your bean, the container will detect if your bean implements this method. If your bean does, the container calls the `isModified()` method and it only replicates the bean when the method returns `true`. If the bean has not been modified (or not enough to require replication, depending on your own preferences), you can return `false` and the replication would not occur. This feature is available on JBoss AS 3.0.1+ only.

3.2.3. The HASessionState service configuration

The `HASessionState` service MBean is defined in the`all/deploy/cluster-service.xml`file.

```
<mbean code="org.jboss.ha.hasessionstate.server.HASessionStateService"
      name="jboss:service=HASessionState">
    <depends>
        jboss:service=${jboss.partition.name:DefaultPartition}
    </depends>
    <!-- Name of the partition to which the service is linked -->
    <attribute name="PartitionName">
        ${jboss.partition.name:DefaultPartition}
    </attribute>
    <!-- JNDI name under which the service is bound -->
    <attribute name="JndiName">/HASessionState/Default</attribute>
    <!-- Max delay before cleaning unreclaimed state.
           Defaults to 30*60*1000 => 30 minutes -->
    <attribute name="BeanCleaningDelay">0</attribute>
</mbean>
```

The configuration attributes in the `HASessionState` MBean are listed below.

- **JndiName** is an optional attribute to specify the JNDI name under which this `HASessionState` service is bound. The default value is `/HAPartition/Default`.
- **PartitionName** is an optional attribute to specify the name of the cluster in which the current `HASessionState` protocol will work. The default value is `DefaultPartition`.
- **BeanCleaningDelay** is an optional attribute to specify the number of miliseconds after which the `HASessionState` service can clean a state that has not been modified. If a node, owning a bean, crashes, its brother node will take ownership of this bean. Nevertheless, the container cache of the brother node will not know about it (because it has never seen it before) and will never delete according to the cleaning settings of the bean. That is why the `HASessionState` service needs to do this cleanup sometimes. The default value is `30*60*1000` milliseconds (i.e., 30 minutes).

3.3. Stateless Session Bean in EJB 3.0

To cluster a stateless session bean in EJB 3.0, all you need to do is to annotate the bean class withe the `@Cluster` annotation. You can pass in the load balance policy and cluster partition as parameters to the annotation. The default load balance policy is `org.jboss.ha.framework.interfaces.RandomRobin` and the default cluster is `DefaultPartition`. Below is the definition of the `@Cluster` annotation.

```
public @interface Clustered {
   Class loadBalancePolicy() default LoadBalancePolicy.class;
   String partition() default "DefaultPartition";
}
```

Here is an example of a clustered EJB 3.0 stateless session bean implementation.

```
@Stateless
@Clustered
public class MyBean implements MySessionInt {

   public void test() {
      // Do something cool
   }
}
```

3.4. Stateful Session Bean in EJB 3.0

To cluster stateful session beans in EJB 3.0, you need to tag the bean implementation class with the `@Cluster` annotation, just as we did with the EJB 3.0 stateless session bean earlier.

```
@Stateful
@Clustered
public class MyBean implements MySessionInt {

   private int state = 0;

   public void increment() {
      System.out.println("counter: " + (state++));
   }
}
```

JBoss Cache provides the session state replication service for EJB 3.0 stateful session beans. The related MBean service is defined in the `ejb3-clustered-sfsbcache-service.xml` file in the `deploy` directory. The contents of the file are as follows.

```
```

```
<server>
   <mbean code="org.jboss.ejb3.cache.tree.PassivationTreeCache"
       name="jboss.cache:service=EJB3SFSBClusteredCache">

        <attribute name="IsolationLevel">READ_UNCOMMITTED</attribute>
        <attribute name="CacheMode">REPL_SYNC</attribute>
        <attribute name="ClusterName">SFSB-Cache</attribute>
        <attribute name="ClusterConfig">
            ... ...
        </attribute>

        <!--  Number of milliseconds to wait until all responses for a
              synchronous call have been received.
        -->
        <attribute name="SyncReplTimeout">10000</attribute>

        <!--  Max number of milliseconds to wait for a lock acquisition -->
        <attribute name="LockAcquisitionTimeout">15000</attribute>

        <!--  Name of the eviction policy class. -->
        <attribute name="EvictionPolicyClass">
            org.jboss.ejb3.cache.tree.StatefulEvictionPolicy
        </attribute>

        <!--  Specific eviction policy configurations. This is LRU -->
        <attribute name="EvictionPolicyConfig">
            <config>
                <attribute name="wakeUpIntervalSeconds">1</attribute>
                <name>statefulClustered</name>
                <region name="/_default_">
                    <attribute name="maxNodes">1000000</attribute>
                    <attribute name="timeToIdleSeconds">300</attribute>
                </region>
            </config>
        </attribute>

        <attribute name="CacheLoaderFetchPersistentState">false</attribute>
        <attribute name="CacheLoaderFetchTransientState">true</attribute>
        <attribute name="FetchStateOnStartup">true</attribute>
        <attribute name="CacheLoaderClass">
            org.jboss.ejb3.cache.tree.StatefulCacheLoader
        </attribute>
        <attribute name="CacheLoaderConfig">
            location=statefulClustered
        </attribute>
   </mbean>
</server>
```

The configuration attributes in the `PassivationTreeCache` MBean are essentially the same as the attributes in the standard JBoss Cache `TreeCache` MBean discussed in *Chapter 12, JBossCache and JGroups Services*. Again, we omitted the JGroups configurations in the `ClusterConfig` attribute (see more in *Section 1, "JGroups Configuration"*).

4. Clustered Entity EJBs

In a JBoss AS cluster, the entity bean instances need to replicated across all nodes. If an entity bean provides remote services, the service methods need to be load balanced as well.

To use a clustered entity bean, the application does not need to do anything special, except for looking up bean references from the clustered HA-JNDI.

4.1. Entity Bean in EJB 2.x

First of all, it is worth to note that clustering 2.x entity beans is a bad thing to do. Its exposes elements that generally are too fine grained for use as remote objects to clustered remote objects and introduces data synchronization problems that are non-trivial. Do NOT use EJB 2.x entity bean clustering unless you fit into the sepecial case situation of read-only, or one read-write node with read-only nodes synched with the cache invalidation services.

To cluster EJB 2.x entity beans, you need to add the `<clustered>` element to the application's `jboss.xml` descriptor file. Below is a typical `jboss.xml` file.

```
<jboss>
    <enterprise-beans>
        <entity>
            <ejb-name>nextgen.EnterpriseEntity</ejb-name>
            <jndi-name>nextgen.EnterpriseEntity</jndi-name>
            <clustered>True</clustered>
            <cluster-config>
                <partition-name>DefaultPartition</partition-name>
                <home-load-balance-policy>
                    org.jboss.ha.framework.interfaces.RoundRobin
                </home-load-balance-policy>
                <bean-load-balance-policy>
                    org.jboss.ha.framework.interfaces.FirstAvailable
                </bean-load-balance-policy>
            </cluster-config>
        </entity>
    </enterprise-beans>
</jboss>
```

The EJB 2.x entity beans are clustered for load balanced remote invocations. All the bean instances are synchronized to have the same contents on all nodes.

However, clustered EJB 2.x Entity Beans do not have a distributed locking mechanism or a distributed cache. They can only be synchronized by using row-level locking at the database level (see `<row-lock>` in the CMP specification) or by setting the Transaction Isolation Level of your JDBC driver to be `TRANSACTION_SERIALIZABLE`. Because there is no supported distributed locking mechanism or distributed cache Entity Beans use Commit Option "B" by default (See `standardjboss.xml` and the container configurations Clustered CMP 2.x EntityBean, Clustered CMP EntityBean, or Clustered BMP EntityBean). It is not recommended that you use Commit

Option "A" unless your Entity Bean is read-only. (There are some design patterns that allow you to use Commit Option "A" with read-mostly beans. You can also take a look at the Seppuku pattern *http://dima.dhs.org/misc/readOnlyUpdates.html*. JBoss may incorporate this pattern into later versions.)

Note

If you are using Bean Managed Persistence (BMP), you are going to have to implement synchronization on your own. The MVCSoft CMP 2.0 persistence engine (see *http://www.jboss.org/jbossgroup/partners.jsp*) provides different kinds of optimistic locking strategies that can work in a JBoss cluster.

4.2. Entity Bean in EJB 3.0

In EJB 3.0, the entity beans primarily serve as a persistence data model. They do not provide remote services. Hence, the entity bean clustering service in EJB 3.0 primarily deals with distributed caching and replication, instead of load balancing.

4.2.1. Configure the distributed cache

To avoid round trips to the database, you can use a cache for your entities. JBoss EJB 3.0 is implemented by Hibernate, which has support for a second-level cache. The Hibernate setup used for the JBoss EJB 3.0 implementation uses JBoss Cache as its underlying cache implementation. The cache provides the following functionalities.

- If you persist a cache enabled entity bean instance to the database via the entity manager the entity will inserted into the cache.
- If you update an entity bean instance and save the changes to the database via the entity manager the entity will updated in the cache.
- If you remove an entity bean instance from the database via the entity manager the entity will removed from the cache.
- If loading a cached entity from the database via the entity manager, and that entity does not exist in the database, it will be inserted into the cache.

JBoss Cache service for EJB 3.0 entity beans is configured in a `TreeCache` MBean (see *Section 2, "JBossCache Configuration"*) in the `deploy/ejb3-entity-cache-service.xml` file. The name of the cache MBean service is `jboss.cache:service=EJB3EntityTreeCache`. Below is the contents of the `ejb3-entity-cache-service.xml` file in the standard JBoss distribution. Again, we omitted the JGroups configuration element `ClusterConfig`.

```
<server>
    <mbean code="org.jboss.cache.TreeCache"
```

```
            name="jboss.cache:service=EJB3EntityTreeCache">

        <depends>jboss:service=Naming</depends>
        <depends>jboss:service=TransactionManager</depends>

        <!-- Configure the TransactionManager -->
        <attribute name="TransactionManagerLookupClass">
            org.jboss.cache.JBossTransactionManagerLookup
        </attribute>

        <attribute name="IsolationLevel">REPEATABLE_READ</attribute>
        <attribute name="CacheMode">REPL_SYNC</attribute>

        <!--Name of cluster. Needs to be the same for all clusters,
            in order to find each other -->
        <attribute name="ClusterName">EJB3-entity-cache</attribute>

        <attribute name="ClusterConfig">
            ... ...
        </attribute>

        <attribute name="InitialStateRetrievalTimeout">5000</attribute>
        <attribute name="SyncReplTimeout">10000</attribute>
        <attribute name="LockAcquisitionTimeout">15000</attribute>

        <attribute name="EvictionPolicyClass">
            org.jboss.cache.eviction.LRUPolicy
        </attribute>

        <!--  Specific eviction policy configurations. This is LRU -->
        <attribute name="EvictionPolicyConfig">
            <config>
                <attribute name="wakeUpIntervalSeconds">5</attribute>
                <!--  Cache wide default -->
                <region name="/_default_">
                    <attribute name="maxNodes">5000</attribute>
                    <attribute name="timeToLiveSeconds">1000</attribute>
                </region>
            </config>
        </attribute>
    </mbean>
</server>
```

As we discussed in *Section 2, "JBossCache Configuration"*, JBoss Cache allows you to specify timeouts to cached entities. Entities not accessed within a certain amount of time are dropped from the cache in order to save memory. If running within a cluster, and the cache is updated, changes to the entries in one node will be replicated to the corresponding entries in the other nodes in the cluster.

Now, we have JBoss Cache configured to support distributed caching of EJB 3.0 entity beans. We still have to configure individual entity beans to use the cache service.

4.2.2. Configure the entity beans for cache

You define your entity bean classes the normal way. Future versions of JBoss EJB 3.0 will support annotating entities and their relationship collections as cached, but for now you have to configure the underlying hibernate engine directly. Take a look at the `persistence.xml` file, which configures the caching options for hibernate via its optional `property` elements. The following element in `persistence.xml` defines that caching should be enabled:

```
<!-- Clustered cache with TreeCache -->
<property name="cache.provider_class">
    org.jboss.ejb3.entity.TreeCacheProviderHook
</property>
```

The following property element defines the object name of the cache to be used, and the MBean name.

```
<property name="treecache.mbean.object_name">
    jboss.cache:service=EJB3EntityTreeCache
</property>
```

Next we need to configure what entities be cached. The default is to not cache anything, even with the settings shown above. We use the `@Cache` annotation to tag entity beans that needs to be cached.

```
@Entity
@Cache(usage=CacheConcurrencyStrategy.TRANSACTIONAL)
public class Customer implements Serializable {
  // ... ...
}
```

A very simplified rule of thumb is that you will typically want to do caching for objects that rarely change, and which are frequently read. You can fine tune the cache for each entity bean in the `ejb3-entity-cache-service.xml` configuration file. For instance, you can specify the size of the cache. If there are too many objects in the cache, the cache could evict oldest objects (or least used objects, depending on configuration) to make room for new objects. The cache for the `mycompany.Customer` entity bean is `/mycompany/Customer` cache region.

```
<server>
  <mbean code="org.jboss.cache.TreeCache"
         name="jboss.cache:service=EJB3EntityTreeCache">
    <depends>jboss:service=Naming
```

```
    <depends>jboss:service=TransactionManager
    ... ...
    <attribute name="EvictionPolicyConfig">
      <config>
        <attribute name="wakeUpIntervalSeconds">5</attribute>
        <region name="/_default_">
          <attribute name="maxNodes">5000</attribute>
          <attribute name="timeToLiveSeconds">1000</attribute>
        </region>
        <region name="/mycompany/Customer">
          <attribute name="maxNodes">10</attribute>
          <attribute name="timeToLiveSeconds">5000</attribute>
        </region>
        ... ...
      </config>
    </attribute>
  </mbean>
</server>
```

If you do not specify a cache region for an entity bean class, all instances of this class will be cached in the `/_default` region as defined above. The EJB3 `Query` API provides means for you to save to load query results (i.e., collections of entity beans) from specified cache regions.

5. HTTP Services

HTTP session replication is used to replicate the state associated with your web clients on other nodes of a cluster. Thus, in the event one of your node crashes, another node in the cluster will be able to recover. Two distinct functions must be performed:

- Session state replication
- Load-balance of incoming invocations

State replication is directly handled by JBoss. When you run JBoss in the `all` configuration, session state replication is enabled by default. Just deploy your web application and its session state is already replicated across all JBoss instances in the cluster.

However, Load-balancing is a different story, it is not handled by JBoss itself and requires additional software. As a very common scenario, we will demonstrate how to setup Apache and mod_jk. This activity could be either performed by specialized hardware switches or routers (Cisco LoadDirector for example) or any other dedicated software though.

Note

A load-balancer tracks the HTTP requests and, depending on the session to which is linked the request, it dispatches the request to the appropriate node.

> This is called a load-balancer with sticky-sessions: once a session is created on a node, every future request will also be processed by the same node. Using a load-balancer that supports sticky-sessions without replicating the sessions allows you to scale very well without the cost of session state replication: each query will always be handled by the same node. But in the case a node dies, the state of all client sessions hosted by this node are lost (the shopping carts, for example) and the clients will most probably need to login on another node and restart with a new session. In many situations, it is acceptable not to replicate HTTP sessions because all critical state is stored in the database. In other situations, loosing a client session is not acceptable and, in this case, session state replication is the price one has to pay.

Apache is a well-known web server which can be extended by plugging modules. One of these modules, mod_jk (and the newest mod_jk2) has been specifically designed to allow forward requests from Apache to a Servlet container. Furthermore, it is also able to load-balance HTTP calls to a set of Servlet containers while maintaining sticky sessions, and this is what is actually interesting for us.

5.1. Download the software

First of all, make sure that you have Apache installed. You can download Apache directly from Apache web site at `http://httpd.apache.org/`. Its installation is pretty straightforward and requires no specific configuration. As several versions of Apache exist, we advise you to use version 2.0.x. We will consider, for the next sections, that you have installed Apache in the `APACHE_HOME` directory.

Next, download mod_jk binaries. Several versions of mod_jk exist as well. We strongly advise you to use mod_jk 1.2.x, as both mod_jk and mod_jk2 are deprecated, unsupported and no further developments are going on in the community. The mod_jk 1.2.x binary can be downloaded from `http://www.apache.org/dist/jakarta/tomcat-connectors/jk/binaries/`. Rename the downloaded file to `mod_jk.so` and copy it under `APACHE_HOME/modules/`.

5.2. Configure Apache to load mod_jk

Modify APACHE_HOME/conf/httpd.conf and add a single line at the end of the file:

```
# Include mod_jk's specific configuration file
Include conf/mod-jk.conf
```

Next, create a new file named `APACHE_HOME/conf/mod-jk.conf`:

```
# Load mod_jk module
# Specify the filename of the mod_jk lib
LoadModule jk_module modules/mod_jk.so

# Where to find workers.properties
JkWorkersFile conf/workers.properties

# Where to put jk logs
JkLogFile logs/mod_jk.log

# Set the jk log level [debug/error/info]
JkLogLevel info

# Select the log format
JkLogStampFormat  "[%a %b %d %H:%M:%S %Y]"

# JkOptions indicates to send SSK KEY SIZE
JkOptions +ForwardKeySize +ForwardURICompat -ForwardDirectories

# JkRequestLogFormat
JkRequestLogFormat "%w %V %T"

# Mount your applications
JkMount /application/* loadbalancer

# You can use external file for mount points.
# It will be checked for updates each 60 seconds.
# The format of the file is: /url=worker
# /examples/*=loadbalancer
JkMountFile conf/uriworkermap.properties

# Add shared memory.
# This directive is present with 1.2.10 and
# later versions of mod_jk, and is needed for
# for load balancing to work properly
JkShmFile logs/jk.shm

# Add jkstatus for managing runtime data
<Location /jkstatus/>
    JkMount status
    Order deny,allow
    Deny from all
    Allow from 127.0.0.1
</Location>
```

Please note that two settings are very important:

- The `LoadModule` directive must reference the mod_jk library you have downloaded in the previous section. You must indicate the exact same name with the "modules" file path prefix.
- The `JkMount` directive tells Apache which URLs it should forward to the mod_jk module (and,

in turn, to the Servlet containers). In the above file, all requests with URL path `/application/*` are sent to the mod_jk load-balancer. This way, you can configure Apache to server static contents (or PHP contents) directly and only use the loadbalancer for Java applications. If you only use mod_jk as a loadbalancer, you can also forward all URLs (i.e., `/*`) to mod_jk.

In addition to the `JkMount` directive, you can also use the `JkMountFile` directive to specify a mount points configuration file, which contains multiple Tomcat forwarding URL mappings. You just need to create a `uriworkermap.properties` file in the `APACHE_HOME/conf` directory. The format of the file is `/url=worker_name`. To get things started, paste the following example into the file you created:

```
# Simple worker configuration file

# Mount the Servlet context to the ajp13 worker
/jmx-console=loadbalancer
/jmx-console/*=loadbalancer
/web-console=loadbalancer
/web-console/*=loadbalancer
```

This will configure mod_jk to forward requests to `/jmx-console` and `/web-console` to Tomcat.

You will most probably not change the other settings in `mod_jk.conf`. They are used to tell mod_jk where to put its logging file, which logging level to use and so on.

5.3. Configure worker nodes in mod_jk

Next, you need to configure mod_jk workers file `conf/workers.properties`. This file specify where are located the different Servlet containers and how calls should be load-balanced across them. The configuration file contains one section for each target servlet container and one global section. For a two nodes setup, the file could look like this:

```
# Define list of workers that will be used
# for mapping requests
worker.list=loadbalancer,status

# Define Node1
# modify the host as your host IP or DNS name.
worker.node1.port=8009
worker.node1.host=node1.mydomain.com
worker.node1.type=ajp13
worker.node1.lbfactor=1
worker.node1.cachesize=10

# Define Node2
# modify the host as your host IP or DNS name.
worker.node2.port=8009
worker.node2.host= node2.mydomain.com
```

```
worker.node2.type=ajp13
worker.node2.lbfactor=1
worker.node2.cachesize=10

# Load-balancing behaviour
worker.loadbalancer.type=lb
worker.loadbalancer.balance_workers=node1,node2
worker.loadbalancer.sticky_session=1
#worker.list=loadbalancer

# Status worker for managing load balancer
worker.status.type=status
```

Basically, the above file configures mod_jk to perform weighted round-robin load balancing with sticky sessions between two servlet containers (JBoss Tomcat) node1 and node2 listening on port 8009.

In the `works.properties` file, each node is defined using the `worker.XXX` naming convention where `XXX` represents an arbitrary name you choose for one of the target Servlet container. For each worker, you must give the host name (or IP address) and port number of the AJP13 connector running in the Servlet container.

The `lbfactor` attribute is the load-balancing factor for this specific worker. It is used to define the priority (or weight) a node should have over other nodes. The higher this number is, the more HTTP requests it will receive. This setting can be used to differentiate servers with different processing power.

The `cachesize` attribute defines the size of the thread pools associated to the Servlet container (i.e. the number of concurrent requests it will forward to the Servlet container). Make sure this number does not outnumber the number of threads configured on the AJP13 connector of the Servlet container. Please review `http://jakarta.apache.org/tomcat/connectors-doc/config/workers.html` for comments on `cachesize` for Apache 1.3.x.

The last part of the `conf/workers.properties` file defines the loadbalancer worker. The only thing you must change is the `worker.loadbalancer.balanced_workers` line: it must list all workers previously defined in the same file: load-balancing will happen over these workers.

The `sticky_session` property specifies the cluster behavior for HTTP sessions. If you specify `worker.loadbalancer.sticky_session=0`, each request will be load balanced between node1 and node2. But when a user opens a session on one server, it is a good idea to always forward this user's requests to the same server. This is called a "sticky session", as the client is always using the same server he reached on his first request. Otherwise the user's session data would need to be synchronized between both servers (session replication, see *Section 5.5, "Configure HTTP session state replication"*). To enable session stickiness, you need to set `worker.loadbalancer.sticky_session` to 1.

Note

A non-loadbalanced setup with a single node required the `worker.list=node1` entry before mod_jk would function correctly.

5.4. Configure JBoss

Finally, we must configure the JBoss Tomcat instances on all clustered nodes so that they can expect requests forwarded from the mod_jk loadbalancer.

On each clustered JBoss node, we have to name the node according to the name specified in `workers.properties`. For instance, on JBoss instance node1, edit the `JBOSS_HOME/server/all/deploy/jboss-web.deployer/server.xml` file (replace `/all` with your own server name if necessary). Locate the `<Engine>` element and add an attribute `jvmRoute`:

```
<Engine name="jboss.web" defaultHost="localhost" jvmRoute="node1">
... ...
</Engine>
```

Then, for each JBoss Tomcat instance in the cluster, we need to tell it to add the `jvmRoute` value to its session cookies so that mod_jk can route incoming requests. Edit the `JBOSS_HOME/server/all/deploy/jboss-web.deployer/META-INF/jboss-service.xml` file (replace `/all` with your own server name). Locate the `<attribute>` element with a name of `UseJK`, and set its value to `true`:

```
<attribute name="UseJK">true</attribute>
```

At this point, you have a fully working Apache+mod_jk load-balancer setup that will balance call to the Servlet containers of your cluster while taking care of session stickiness (clients will always use the same Servlet container).

Note

For more updated information on using mod_jk 1.2 with JBoss Tomcat, please refer to the JBoss wiki page at `http://wiki.jboss.org/wiki/Wiki.jsp?page=UsingMod_jk1.2WithJBoss`.

5.5. Configure HTTP session state replication

In *Section 5.3, "Configure worker nodes in mod_jk"*, we covered how to use sticky sessions to make sure that a client in a session always hits the same server node in order to maintain the session state. However, that is not an ideal solution. The load might be unevenly distributed over the nodes over time and if a node goes down, all its session data is lost. A better and more reliable solution is to replicate session data across all nodes in the cluster. This way, the client can hit any server node and obtain the same session states.

The `jboss.cache:service=TomcatClusteringCache` MBean makes use of JBoss Cache to provide HTTP session replication service to the HTTP load balancer in a JBoss Tomcat cluster. This MBean is defined in the `deploy/jboss-web-cluster.sar/META-INF/jboss-service.xml` file.

Note

Before AS 4.0.4 CR2, the HTTP session cache configuration file is the `deploy/tc5-cluster-service.xml` file. Please see AS 4.2 documentation for more details.

Below is a typical `deploy/jboss-web-cluster.sar/META-INF/jboss-service.xml` file. The configuration attributes in the `TomcatClusteringCache` MBean is very similar to those in *Section 2, "JBossCache Configuration"*.

```
<mbean code="org.jboss.cache.aop.TreeCacheAop"
    name="jboss.cache:service=TomcatClusteringCache">

    <depends>jboss:service=Naming</depends>
    <depends>jboss:service=TransactionManager</depends>
    <depends>jboss.aop:service=AspectDeployer</depends>

    <attribute name="TransactionManagerLookupClass">
        org.jboss.cache.BatchModeTransactionManagerLookup
    </attribute>

    <attribute name="IsolationLevel">REPEATABLE_READ</attribute>

    <attribute name="CacheMode">REPL_ASYNC</attribute>

    <attribute name="ClusterName">
      Tomcat-${jboss.partition.name:Cluster}
    </attribute>

    <attribute name="UseMarshalling">false</attribute>

    <attribute name="InactiveOnStartup">false</attribute>

    <attribute name="ClusterConfig">
        ... ...
```

```
    </attribute>

    <attribute name="LockAcquisitionTimeout">15000</attribute>
</mbean>
```

The detailed configuration for the `TreeCache` MBean is covered in *Section 2, "JBossCache Configuration"*. Below, we will just discuss several attributes that are most relevant to the HTTP cluster session replication.

- **TransactionManagerLookupClass** sets the transaction manager factory. The default value is `org.jboss.cache.BatchModeTransactionManagerLookup`. It tells the cache NOT to participate in JTA-specific transactions. Instead, the cache manages its own transaction to support finely grained replications.
- **IsolationLevel** sets the isolation level for updates to the transactional distributed cache. The valid values are `SERIALIZABLE`, `REPEATABLE_READ`, `READ_COMMITTED`, `READ_UNCOMMITTED`, and `NONE`. These isolation levels mean the same thing as isolation levels on the database. The default isolation of `REPEATABLE_READ` makes sense for most web applications.
- **CacheMode** controls how the cache is replicated. The valid values are `REPL_SYNC` and `REPL_ASYNC`, which determine whether changes are made synchronously or asynchronously. Using synchronous replication makes sure changes propagated to the cluster before the web request completes. However, synchronous replication is much slower. For asyncrhonous access, you will want to enable and tune the replication queue.
- **ClusterName** specifies the name of the cluster that the cache works within. The default cluster name is the the word "Tomcat-" appended by the current JBoss partition name. All the nodes should use the same cluster name. Although session replication can share the same channel (multicast address and port) with other clustered services in JBoss, replication should have it's own cluster name.
- The **UseMarshalling** and **InactiveOnStartup** attributes must have the same value. They must be `true` if `FIELD` level session replication is needed (see later). Otherwise, they are default to `false`.
- **ClusterConfig** configures the underlying JGroups stack. The most import configuration elements are the muliticast adress and port, `mcast_addr` and `mcast_port` respectively, to use for clustered communication. These values should make sense for your network. Please refer to *Section 1, "JGroups Configuration"* for more information.
- **LockAcquisitionTimeout** sets the maximum number of milliseconds to wait for a lock acquisition. The default value is 15000.
- **UseReplQueue** determines whether to enable the replication queue when using asynchronous replication. This allows multiple cache updates to be bundled together to improve performance. The replication queue properties are controlled by the

`ReplQueueInterval` and `ReplQueueMaxElements` properties.

- **ReplQueueInterval** specifies the time in milliseconds JBoss Cache will wait before sending items in the replication queue.
- **ReplQueueMaxElements**: specifies the maximum number of elements allowed in the replication queue before JBoss Cache will send an update.

5.6. Enabling session replication in your application

To enable clustering of your web application you must it as distributable in the `web.xml` descriptor. Here's an example:

```
<?xml version="1.0"?>
<web-app
xmlns="http://java.sun.com/xml/ns/"Whats_new_in_JBoss_4-J2EE_Certification_and_Standards_Com
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
          version="2.4">
    <distributable/>
    <!-- ... -->
</web-app>
```

You can futher configure session replication using the `replication-config` element in the `jboss-web.xml` file. Here is an example:

```
<jboss-web>
    <replication-config>
        <replication-trigger>SET_AND_NON_PRIMITIVE_GET</replication-trigger>
        <replication-granularity>SESSION</replication-granularity>
        <replication-field-batch-mode>true</replication-field-batch-mode>
    </replication-config>
</jboss-web>
```

The `replication-trigger` element determines what triggers a session replication (or when is a session is considered dirty). It has 4 options:

- **SET**: With this policy, the session is considered dirty only when an attribute is set in the session. If your application always writes changed value back into the session, this option will be most optimized in term of performance. If an object is retrieved from the session and modified without being written back into the session, the change to that object will not be replicated.
- **SET_AND_GET**: With this policy, any attribute that is get or set will be marked as dirty. If an object is retrieved from the session and modified without being written back into the session, the change to that object will be replicated. This option can have significant performance

implications.

- **SET_AND_NON_PRIMITIVE_GET**: This policy is similar to the SET_AND_GET policy except that only non-primitive get operations are considered dirty. For example, the http session request may retrieve a non-primitive object instance from the attribute and then modify the instance. If we don't specify that non-primitive get is considered dirty, then the modification will not be replication properly. This is the default value.
- **ACCESS**: This option causes the session to be marked as dirty whenever it is accessed. Since a the session is accessed during each HTTP request, it will be replicated with each request. The access time stamp in the session instance will be updated as well. Since the time stamp may not be updated in other clustering nodes because of no replication, the session in other nodes may expire before the active node if the HTTP request does not retrieve or modify any session attributes. When this option is set, the session timestamps will be synchronized throughout the cluster nodes. Note that use of this option can have a significant performance impact, so use it with caution.

The `replication-granularity` element controls the size of the replication units. The supported values are:

- **SESSION**: Replication is per session instance. As long as it is considered modified when the snapshot manager is called, the whole session object will be serialized.
- **ATTRIBUTE**: Replication is only for the dirty attributes in the session plus some session data, like, lastAccessTime. For session that carries large amount of data, this option can increase replication performance.
- **FIELD**: Replication is only for data fields inside session attribute objects (see more later).

The `replication-field-batch-mode` element indicates whether you want to have batch update between each http request or not. Default is `true`.

If your sessions are generally small, SESSION is the better policy. If your session is larger and some parts are infrequently accessed, ATTRIBUTE replication will be more effective. If your application has very big data objects in session attributes and only fields in those objects are frequently modified, the FIELD policy would be the best. In the next section, let's discuss exactly how the FIELD level replication works.

5.7. Use FIELD level replication

FIELD-level replication only replicates modified data fields inside objects stored in the session. It could potentially drastically reduce the data traffic between clustered nodes, and hence improve the performance of the whole cluster. To use FIELD-level replication, you only need to annotated your POJOs the same way you do for EJB3 beans.

When you annotate your class with `@org.jboss.cache.aop.annotation.PojoCacheable`, you indicate that instances of this class will be used in FIELD-level replication. For exmaple,

```
@org.jboss.cache.aop.annotation.PojoCacheable
public class Address
{
...
}
```

If you annotate it with `@org.jboss.cache.aop.annotation.InstanceOfPojoCacheable` instead, then all of its sub-class will be automatically annotated as well. For example,

```
@org.jboss.cache.aop.annotation.InstanceOfPojoCacheable
public class Person
{
...
}
```

then when you have a sub-class like

```
public class Student extends Person
{
...
}
```

there will be no need to annotate `Student`. It will be annotated automatically because it is a sub-class of `Person`.

Use POJO Cache in JDK 1.4

The POJO cache annotations discussed in this chapter are JDK 5 annotations. JDK 5 is required for JBoss AS 4.2.0 and above. However, for earlier JBoss AS 4.0.x versions under JDK 1.4, you can still use POJO cache by embedding annotations in JavaDoc comments and then pre-process the class with annotation compiler. Please see the JBoss Cache documentation for more details.

You can see a complete example on how to build, deploy, and validate a FIELD-level replicated web application in JDK 1.4 from this page: `http://wiki.jboss.org/wiki/Wiki.jsp?page=Http_session_field_level_exa`

When you deploy the web application into JBoss AS, make sure that the following configurations are correct:

- In the server's `deploy/jboss-web-cluster.sar/META-INF/jboss-service.xml` file, the `inactiveOnStartup` and `useMarshalling` attributes must both be `true`.
- In the application's `jboss-web.xml` file, the `replication-granularity` attribute must be `FIELD`.

Finally, let's see an example on how to use FIELD-level replication on those data classes. Notice that there is no need to call `session.setAttribute()` after you make changes to the data object, and all changes to the fields are automatically replicated across the cluster.

```
// Do this only once. So this can be in init(), e.g.
if(firstTime)
{
  Person joe = new Person("Joe", 40);
  Person mary = new Person("Mary", 30);
  Address addr = new Address();
  addr.setZip(94086);

  joe.setAddress(addr);
  mary.setAddress(addr); // joe and mary share the same address!

  session.setAttribute("joe", joe); // that's it.
  session.setAttribute("mary", mary); // that's it.
}

Person mary = (Person)session.getAttribute("mary");
mary.getAddress().setZip(95123); // this will update and replicate the zip
code.
```

Besides plain objects, you can also use regular Java collections of those objects as session attributes. JBoss cache automatically figures out how to handle those collections and replicate field changes in their member objects.

5.8. Monitoring session replication

If you have deployed and accessed your application, go to the `jboss.cache:service=TomcatClusteringCache` MBean and invoke the `printDetails` operation. You should see output resembling the following.

```
/JSESSION

/quote

/FB04767C454BAB3B2E462A27CB571330
VERSION: 6
FB04767C454BAB3B2E462A27CB571330:
org.jboss.invocation.MarshalledValue@1f13a81c

/AxCI8Ovt5VQTfNyYy9Bomw**
VERSION: 4
```

```
AxCI8Ovt5VQTfNyYy9Bomw**: org.jboss.invocation.MarshalledValue@e076e4c8
```

This output shows two separate web sessions, in one application named *quote*, that are being shared via JBossCache. This example uses a `replication-granularity` of `session`. Had `attribute` level replication been used, there would be additional entries showing each replicated session attribute. In either case, the replicated values are stored in an opaque `MarshelledValue` container. There aren't currently any tools that allow you to inspect the contents of the replicated session values. If you don't see any output, either the application was not correctly marked as `distributable` or you haven't accessed a part of application that places values in the HTTP session. The `org.jboss.cache` and `org.jboss.web` logging categories provide additional insight into session replication useful for debugging purposes.

5.9. Using Single Sign On

JBoss supports clustered single sign-on, allowing a user to authenticate to one application on a JBoss server and to be recognized on all applications, on that same machine or on another node in the cluster, that are deployed on the same virtual host. Authentication replication is handled by the HTTP session replication service. Although session replication does not need to be explicitly enabled for the applications in question, the `jboss-web-cluster.sar` application does need to be deployed (e.g., in the `all` configuration).

To enable single sign-on, you must add the `ClusteredSingleSignOn` valve to the appropriate `Host` elements of the tomcat `server.xml` file. The valve configuration is shown here:

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

6. Clustered JMS Services

JBoss AS 3.2.4 and above support high availability JMS (HA-JMS) services in the `all` server configuration. In the current production release of JBoss AS, the HA-JMS service is implemented as a clustered singleton fail-over service.

Note

If you are willing to configure HA-JMS yourself, you can get it to work with earlier versions of JBoss AS. We have a customer who uses HA-JMS successfully in JBoss AS 3.0.7. Please contact JBoss support for more questions.

6.1. High Availability Singleton Fail-over

The JBoss HA-JMS service (i.e., message queues and topics) only runs on a single node (i.e., the master node) in the cluster at any given time. If that node fails, the cluster simply elects another node to run the JMS service (fail-over). This setup provides redundancy against server

failures but does not reduce the work load on the JMS server node.

Note

While you cannot load balance HA-JMS queues (there is only one master node that runs the queues), you can load balance the MDBs that process messages from those queues (see *Section 6.1.3, "Load Balanced HA-JMS MDBs"*).

6.1.1. Server Side Configuration

To use the singleton fail-over HA-JMS service, you must configure JMS services identically on all nodes in the cluster. That includes all JMS related service MBeans and all deployed JMS applications.

The JMS server is configured to persist its data in the `DefaultDS`. By default, that is the embedded HSQLDB. In most cluster environments, however, all nodes need to persist data against a shared database. So, the first thing to do before you start clustered JMS is to setup a shared database for JMS. You need to do the following:

- Configure `DefaultDS` to point to the database server of your choice. That is to replace the `deploy/hsqlsb-ds.xml` file with the `xxx-ds.xml` file in the `docs/examples/jca` directory, where `xxx` is the name of the target shared database (e.g., `mysql-ds.xml`, see *Section 3, "Configuring JDBC DataSources"*).
- Replace the `hsqldb-jdbc2-service.xml` file under the `server/all/deploy-hasingleton/jms` directory with one tuned to the specific database. For example if you use MySQL the file is `mysql-jdbc2-service.xml`. Configuration files for a number of RDBMS are bundled with the JBoss AS distribution. They can be found under `docs/examples/jms`.

Note

There is no need to replace the `hsqldb-jdbc-state-service.xml` file under the `server/all/deploy-hasingleton/jms` directory. Despite the `hsql` in its name, it works with all SQL92 compliant databases, including HSQL, MySQL, SQL Server, and more. It automatically uses the `DefaultDS` for storage, as we configured above.

6.1.2. HA-JMS Client

The HA-JMS client is different from regular JMS clients in two important aspects.

- The HA-JMS client must obtain JMS connection factories from the HA-JNDI (the default port is 1100).
- The client connection must listens for server exceptions. When the cluster fail-over to a different master node, all client operations on the current connection fails with exceptions. The client must know to re-connect.

Note

While the HA-JMS connection factory knows the current master node that runs JMS services, there is no smart client side interceptor. The client stub only knows the fixed master node and cannot adjust to server topography changes.

6.1.3. Load Balanced HA-JMS MDBs

While the HA-JMS queues and topics only run on a single node at a time, MDBs on multiple nodes can receive and process messages from the HA-JMS master node. The contested queues and topics result in load balancing behavior for MDBs. To enable loading balancing for MDBs, you can specify a receiver for the queue. The receiver records which node is waiting for a message and in which order the messages should be processed. JBoss provides three receiver implementations.

- The `org.jboss.mq.server.ReceiversImpl` is the default implementation using a `HashSet`.
- The `org.jboss.mq.server.ReceiversImplArrayList` is theimplementation using an `ArrayList`.
- The `org.jboss.mq.server.ReceiversImplLinkedList` is the implementation using a `LinkedList`.

You can specify the receiver implementation class name as an attribute in the MBean that defines the permanent JMS `Queue` or `DestinationManager` on each node (see *Chapter 7, Messaging on JBoss*). For best load balancing performance, we suggest you to use the `ReceiversImplArrayList` or `ReceiversImplArrayList` implementations due to an undesirable implementation detail of `HashSet` in the JVM.

Part IV. Legacy EJB Support

EJBs on JBoss

The EJB Container Configuration and Architecture

The JBoss EJB container architecture employs a modular plug-in approach. All key aspects of the EJB container may be replaced by custom versions of a plug-in and/or an interceptor by a developer. This approach allows for fine tuned customization of the EJB container behavior to optimally suite your needs. Most of the EJB container behavior is configurable through the EJB JAR `META-INF/jboss.xml` descriptor and the default server-wide equivalent `standardjboss.xml` descriptor. We will look at various configuration capabilities throughout this chapter as we explore the container architecture.

1. The EJB Client Side View

We will begin our tour of the EJB container by looking at the client view of an EJB through the home and remote proxies. It is the responsibility of the container provider to generate the `javax.ejb.EJBHome` and `javax.ejb.EJBObject` for an EJB implementation. A client never references an EJB bean instance directly, but rather references the `EJBHome` which implements the bean home interface, and the `EJBObject` which implements the bean remote interface. *Figure 14.1, "The composition of an EJBHome proxy in JBoss."* shows the composition of an EJB home proxy and its relation to the EJB deployment.

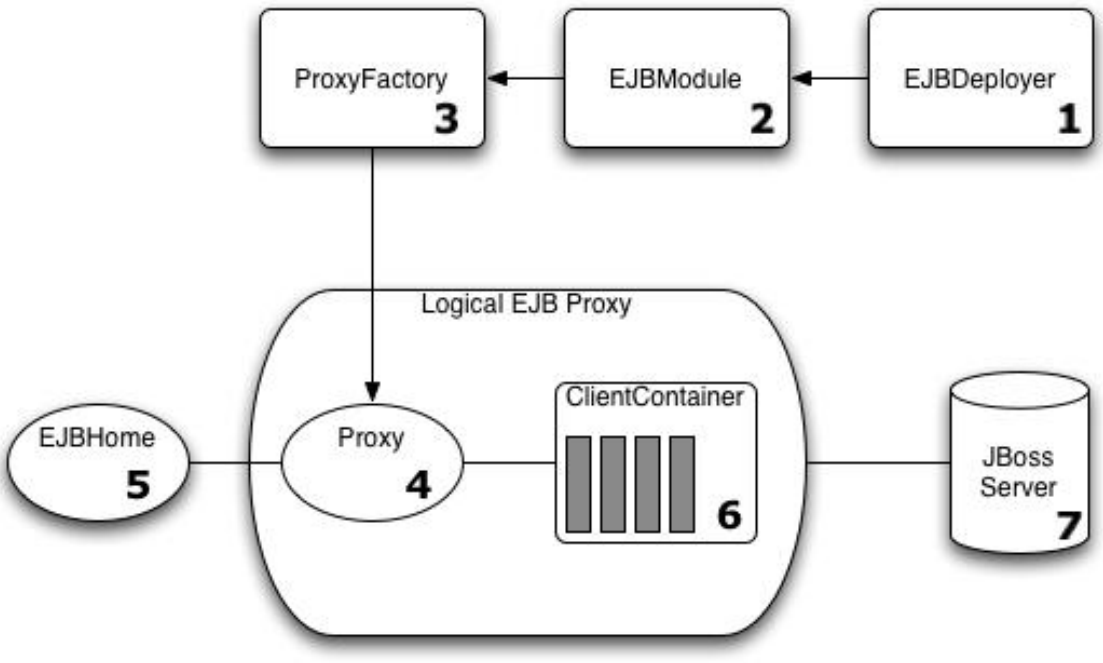


Figure 14.1. The composition of an EJBHome proxy in JBoss.

The numbered items in the figure are:

1. The EJBDeployer (`org.jboss.ejb.EJBDeployer`) is invoked to deploy an EJB JAR. An `EJBModule` (`org.jboss.ejb.EJBModule`) is created to encapsulate the deployment metadata.

2. The create phase of the `EJBModule` life cycle creates an `EJBProxyFactory` (`org.jboss.ejb.EJBProxyFactory`) that manages the creation of EJB home and remote interface proxies based on the `EJBModuleinvoker-proxy-bindings` metadata. There can be multiple proxy factories associated with an EJB and we will look at how this is defined shortly.

3. The `ProxyFactory` constructs the logical proxies and binds the homes into JNDI. A logical proxy is composed of a dynamic `Proxy` (`java.lang.reflect.Proxy`), the home interfaces of the EJB that the proxy exposes, the `ProxyHandler` (`java.lang.reflect.InvocationHandler`) implementation in the form of the `ClientContainer` (`org.jboss.proxy.ClientContainer`), and the client side interceptors.

4. The proxy created by the `EJBProxyFactory` is a standard dynamic proxy. It is a serializable object that proxies the EJB home and remote interfaces as defined in the `EJBModule` metadata. The proxy translates requests made through the strongly typed EJB interfaces into a detyped invocation using the `ClientContainer` handler associated with the proxy. It is the dynamic proxy instance that is bound into JNDI as the EJB home interface that clients lookup. When a client does a lookup of an EJB home, the home proxy is transported into the client VM along with the `ClientContainer` and its interceptors. The use of dynamic proxies avoids the EJB specific compilation step required by many other EJB containers.

5. The EJB home interface is declared in the ejb-jar.xml descriptor and available from the EJBModule metadata. A key property of dynamic proxies is that they are seen to implement the interfaces they expose. This is true in the sense of Java's strong type system. A proxy can be cast to any of the home interfaces and reflection on the proxy provides the full details of the interfaces it proxies.

6. The proxy delegates calls made through any of its interfaces to the `ClientContainer` handler. The single method required of the handler is: `public Object invoke(Object proxy, Method m, Object[] args) throws Throwable`. The `EJBProxyFactory` creates a `ClientContainer` and assigns this as the `ProxyHandler`. The `ClientContainer`'s state consists of an `InvocationContext` (`org.jboss.invocation.InvocationContext`) and a chain of interceptors (`org.jboss.proxy.Interceptor`). The `InvocationContext` contains:
   - the JMX `ObjectName` of the EJB container MBean the `Proxy` is associated with
   - the `javax.ejb.EJBMetaData` for the EJB
   - the JNDI name of the EJB home interface
   - the transport specific invoker (`org.jboss.invocation.Invoker`)

   The interceptor chain consists of the functional units that make up the EJB home or remote interface behavior. This is a configurable aspect of an EJB as we will see when we discuss the `jboss.xml` descriptor, and the interceptor makeup is contained in the `EJBModule` metadata. Interceptors (`org.jboss.proxy.Interceptor`) handle the different EJB types, security, transactions and transport. You can add your own interceptors as well.

7. The transport specific invoker associated with the proxy has an association to the server side

detached invoker that handles the transport details of the EJB method invocation. The detached invoker is a JBoss server side component.

The configuration of the client side interceptors is done using the `jboss.xmlclient-interceptors` element. When the `ClientContainer` invoke method is called it creates an untyped `Invocation` (`org.jboss.invocation.Invocation`) to encapsulate request. This is then passed through the interceptor chain. The last interceptor in the chain will be the transport handler that knows how to send the request to the server and obtain the reply, taking care of the transport specific details.

As an example of the client interceptor configuration usage, consider the default stateless session bean configuration found in the `server/production/standardjboss.xml` descriptor. *Example 14.1, "The client-interceptors from the Standard Stateless SessionBean configuration."* shows the `stateless-rmi-invoker` client interceptors configuration referenced by the Standard Stateless SessionBean.

```
<invoker-proxy-binding>
    <name>stateless-rmi-invoker</name>
    <invoker-mbean>jboss:service=invoker,type=jrmp</invoker-mbean>
    <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-factory>
        <proxy-factory-config>
        <client-interceptors>
            <home>
<interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
                <interceptor call-by-value="false">
                    org.jboss.invocation.InvokerInterceptor
                </interceptor>
                <interceptor call-by-value="true">
                    org.jboss.invocation.MarshallingInvokerInterceptor
                </interceptor>
            </home>
            <bean>
<interceptor>org.jboss.proxy.ejb.StatelessSessionInterceptor</interceptor>
<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
                <interceptor call-by-value="false">
                    org.jboss.invocation.InvokerInterceptor
                </interceptor>
                <interceptor call-by-value="true">
                    org.jboss.invocation.MarshallingInvokerInterceptor
                </interceptor>
            </bean>
        </client-interceptors>
    </proxy-factory-config>
</invoker-proxy-binding>
```

```
<container-configuration>
    <container-name>Standard Stateless SessionBean</container-name>
    <call-logging>false</call-logging>
```

```
<invoker-proxy-binding-name>stateless-rmi-invoker</invoker-proxy-binding-name>
    <!-- ... -->
</container-configuration>
```

Example 14.1. The client-interceptors from the Standard Stateless SessionBean configuration.

This is the client interceptor configuration for stateless session beans that is used in the absence of an EJB JAR `META-INF/jboss.xml` configuration that overrides these settings. The functionality provided by each client interceptor is:

- **org.jboss.proxy.ejb.HomeInterceptor**: handles the `getHomeHandle`, `getEJBMetaData`, and remove methods of the `EJBHome` interface locally in the client VM. Any other methods are propagated to the next interceptor.
- **org.jboss.proxy.ejb.StatelessSessionInterceptor**: handles the `toString`, `equals`, `hashCode`, `getHandle`, `getEJBHome` and `isIdentical` methods of the `EJBObject` interface locally in the client VM. Any other methods are propagated to the next interceptor.
- **org.jboss.proxy.SecurityInterceptor**: associates the current security context with the method invocation for use by other interceptors or the server.
- **org.jboss.proxy.TransactionInterceptor**: associates any active transaction with the invocation method invocation for use by other interceptors.
- **org.jboss.invocation.InvokerInterceptor**: encapsulates the dispatch of the method invocation to the transport specific invoker. It knows if the client is executing in the same VM as the server and will optimally route the invocation to a by reference invoker in this situation. When the client is external to the server VM, this interceptor delegates the invocation to the transport invoker associated with the invocation context. In the case of the *Example 14.1, "The client-interceptors from the Standard Stateless SessionBean configuration."* configuration, this would be the invoker stub associated with the `jboss:service=invoker,type=jrmp`, the `JRMPInvoker` service.

  org.jboss.invocation.MarshallingInvokerInterceptor: extends the `InvokerInterceptor` to not optimize in-VM invocations. This is used to force `call-by-value` semantics for method calls.

1.1. Specifying the EJB Proxy Configuration

To specify the EJB invocation transport and the client proxy interceptor stack, you need to define an `invoker-proxy-binding` in either the EJB JAR `META-INF/jboss.xml descriptor`, or the server `standardjboss.xml` descriptor. There are several default `invoker-proxy-bindings` defined in the `standardjboss.xml` descriptor for the various default

EJB container configurations and the standard RMI/JRMP and RMI/IIOP transport protocols. The current default proxy configurations are:

- **entity-rmi-invoker**: a RMI/JRMP configuration for entity beans
- **clustered-entity-rmi-invoker**: a RMI/JRMP configuration for clustered entity beans
- **stateless-rmi-invoker**: a RMI/JRMP configuration for stateless session beans
- **clustered-stateless-rmi-invoker**: a RMI/JRMP configuration for clustered stateless session beans
- **stateful-rmi-invoker**: a RMI/JRMP configuration for clustered stateful session beans
- **clustered-stateful-rmi-invoker**: a RMI/JRMP configuration for clustered stateful session beans
- **message-driven-bean**: a JMS invoker for message driven beans
- **singleton-message-driven-bean**: a JMS invoker for singleton message driven beans
- **message-inflow-driven-bean**: a JMS invoker for message inflow driven beans
- **jms-message-inflow-driven-bean**: a JMS inflow invoker for standard message driven beans
- **iiop**: a RMI/IIOP for use with session and entity beans.

To introduce a new protocol binding, or customize the proxy factory, or the client side interceptor stack, requires defining a new `invoker-proxy-binding`. The full `invoker-proxy-binding` DTD fragment for the specification of the proxy configuration is given in *Figure 14.2, "The invoker-proxy-binding schema"*.

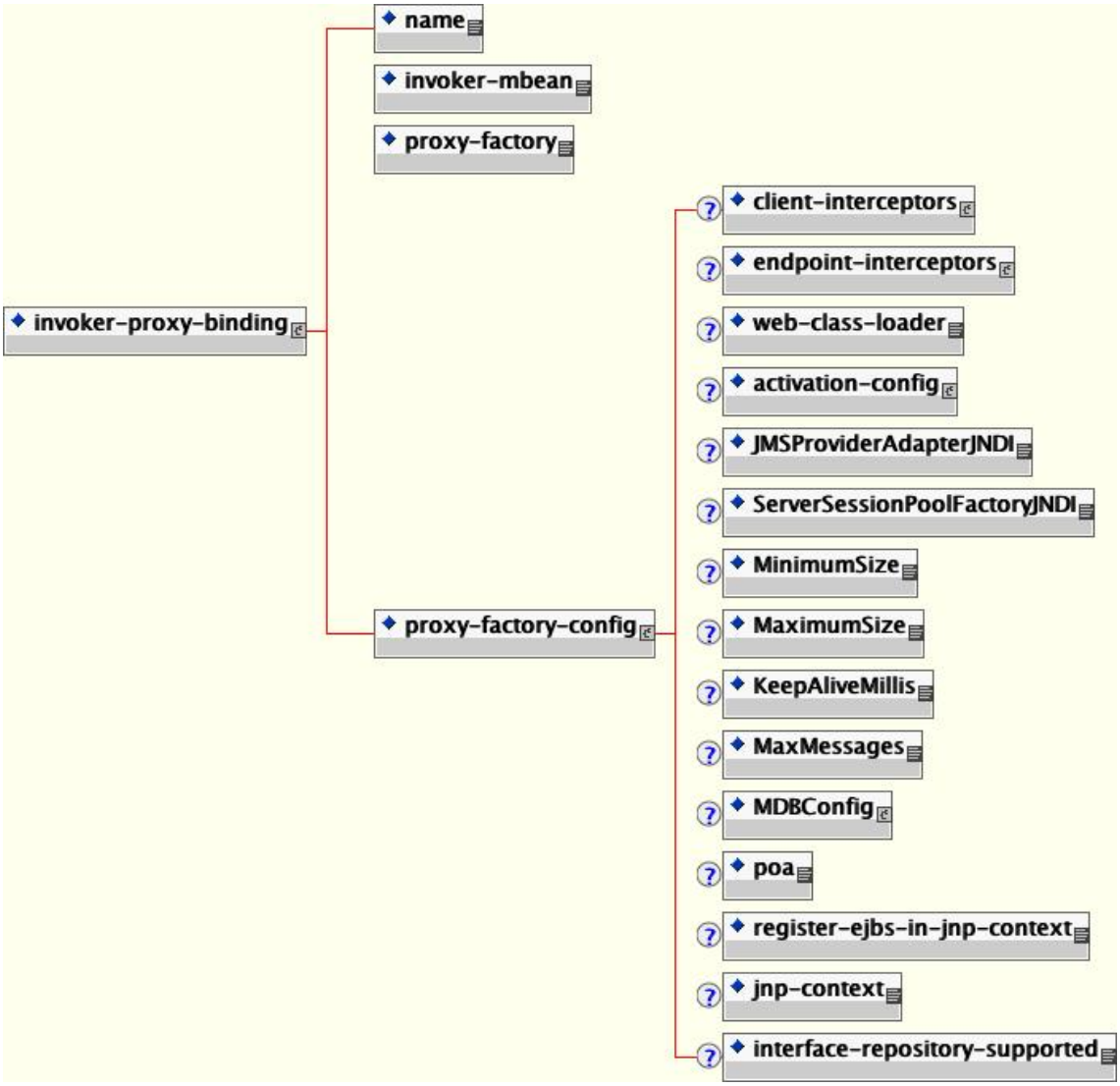


Figure 14.2. The invoker-proxy-binding schema

The `invoker-proxy-binding` child elements are:

- **name**: The `name` element gives a unique name for the `invoker-proxy-binding`. The name is used to reference the binding from the EJB container configuration when setting the default proxy binding as well as the EJB deployment level to specify addition proxy bindings. You will see how this is done when we look at the `jboss.xml` elements that control the server side EJB container configuration.
- **invoker-mbean**: The `invoker-mbean` element gives the JMX `ObjectName` string of the detached invoker MBean service the proxy invoker will be associated with.
- **proxy-factory**: The `proxy-factory` element specifies the fully qualified class name of the

proxy factory, which must implement the `org.jboss.ejb.EJBProxyFactory` interface. The `EJBProxyFactory` handles the configuration of the proxy and the association of the protocol specific invoker and context. The current JBoss implementations of the `EJBProxyFactory` interface include:

- **org.jboss.proxy.ejb.ProxyFactory**: The RMI/JRMP specific factory.
- **org.jboss.proxy.ejb.ProxyFactoryHA**: The cluster RMI/JRMP specific factory.
- **org.jboss.ejb.plugins.jms.JMSContainerInvoker**: The JMS specific factory.
- **org.jboss.proxy.ejb.IORFactory**: The RMI/IIOP specific factory.

- **proxy-factory-config**: The `proxy-factory-config` element specifies additional information for the `proxy-factory` implementation. Unfortunately, its currently an unstructured collection of elements. Only a few of the elements apply to each type of proxy factory. The child elements break down into the three invocation protocols: RMI/RJMP, RMI/IIOP and JMS.

For the RMI/JRMP specific proxy factories, `org.jboss.proxy.ejb.ProxyFactory` and `org.jboss.proxy.ejb.ProxyFactoryHA` the following elements apply:

- **client-interceptors**: The `client-interceptors` define the home, remote and optionally the multi-valued proxy interceptor stacks.
- **web-class-loader**: The web class loader defines the instance of the `org.jboss.web.WebClassLoader` that should be associated with the proxy for dynamic class loading.

The following `proxy-factory-config` is for an entity bean accessed over RMI.

```
<proxy-factory-config>
    <client-interceptors>
        <home>
            <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
            <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
            <interceptor call-by-value="false">
                org.jboss.invocation.InvokerInterceptor
            </interceptor>
            <interceptor call-by-value="true">
                org.jboss.invocation.MarshallingInvokerInterceptor
            </interceptor>
        </home>
        <bean>
            <interceptor>org.jboss.proxy.ejb.EntityInterceptor</interceptor>
            <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
            <interceptor call-by-value="false">
                org.jboss.invocation.InvokerInterceptor
            </interceptor>
            <interceptor call-by-value="true">
```

```
                org.jboss.invocation.MarshallingInvokerInterceptor
            </interceptor>
        </bean>
        <list-entity>
<interceptor>org.jboss.proxy.ejb.ListEntityInterceptor</interceptor>
            <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
            <interceptor call-by-value="false">
                org.jboss.invocation.InvokerInterceptor
            </interceptor>
            <interceptor call-by-value="true">
                org.jboss.invocation.MarshallingInvokerInterceptor
            </interceptor>
        </list-entity>
    </client-interceptors>
</proxy-factory-config>
```

For the RMI/IIOP specific proxy factory, `org.jboss.proxy.ejb.IORFactory`, the following elements apply:

- **web-class-loader**: The web class loader defines the instance of the `org.jboss.web.WebClassLoader` that should be associated with the proxy for dynamic class loading.
- **poa**: The portable object adapter usage. Valid values are `per-servant` and `shared`.
- **register-ejbs-in-jnp-context**: A flag indicating if the EJBs should be register in JNDI.
- **jnp-context**: The JNDI context in which to register EJBs.
- **interface-repository-supported**: This indicates whether or not a deployed EJB has its own CORBA interface repository.

The following shows a `proxy-factory-config` for EJBs accessed over IIOP.

```
<proxy-factory-config>
    <web-class-loader>org.jboss.iiop.WebCL</web-class-loader>
    <poa>per-servant</poa>
    <register-ejbs-in-jnp-context>true</register-ejbs-in-jnp-context>
    <jnp-context>iiop</jnp-context>
</proxy-factory-config>
```

For the JMS specific proxy factory, `org.jboss.ejb.plugins.jms.JMSContainerInvoker`, the following elements apply:

- **MinimumSize**: This specifies the minimum pool size for MDBs processing . This defaults to 1.
- **MaximumSize**: This specifies the upper limit to the number of concurrent MDBs that will be

allowed for the JMS destination. This defaults to 15.

- **MaxMessages**: This specifies the `maxMessages` parameter value for the `createConnectionConsumer` method of `javax.jms.QueueConnection` and `javax.jms.TopicConnection` interfaces, as well as the `maxMessages` parameter value for the `createDurableConnectionConsumer` method of `javax.jms.TopicConnection`. It is the maximum number of messages that can be assigned to a server session at one time. This defaults to 1. This value should not be modified from the default unless your JMS provider indicates this is supported.
- **KeepAliveMillis**: This specifies the keep alive time interval in milliseconds for sessions in the session pool. The default is 30000 (30 seconds).
- **MDBConfig**: Configuration for the MDB JMS connection behavior. Among the elements supported are:
  - **ReconnectIntervalSec**: The time to wait (in seconds) before trying to recover the connection to the JMS server.
  - **DeliveryActive**: Whether or not the MDB is active at startup. The default is true.
  - **DLQConfig**: Configuration for an MDB's dead letter queue, used when messages are redelivered too many times.
  - **JMSProviderAdapterJNDI**: The JNDI name of the JMS provider adapter in the `java:/` namespace. This is mandatory for an MDB and must implement `org.jboss.jms.jndi.JMSProviderAdapter`.
  - **ServerSessionPoolFactoryJNDI**: The JNDI name of the session pool in the `java:/` namespace of the JMS provider's session pool factory. This is mandatory for an MDB and must implement `org.jboss.jms.asf.ServerSessionPoolFactory`.

Example 14.2, "A sample JMSContainerInvoker proxy-factory-config" gives a sample `proxy-factory-config` fragment taken from the `standardjboss.xml` descriptor.

```
<proxy-factory-config>
    <JMSProviderAdapterJNDI>DefaultJMSProvider</JMSProviderAdapterJNDI>
    <ServerSessionPoolFactoryJNDI>StdJMSPool</ServerSessionPoolFactoryJNDI>
    <MinimumSize>1</MinimumSize>
    <MaximumSize>15</MaximumSize>
    <KeepAliveMillis>30000</KeepAliveMillis>
    <MaxMessages>1</MaxMessages>
    <MDBConfig>
        <ReconnectIntervalSec>10</ReconnectIntervalSec>
        <DLQConfig>
            <DestinationQueue>queue/DLQ</DestinationQueue>
            <MaxTimesRedelivered>10</MaxTimesRedelivered>
            <TimeToLive>0</TimeToLive>
        </DLQConfig>
    </MDBConfig>
</proxy-factory-config>
```

Example 14.2. A sample JMSContainerInvoker proxy-factory-config

2. The EJB Server Side View

Every EJB invocation must end up at a JBoss server hosted EJB container. In this section we will look at how invocations are transported to the JBoss server VM and find their way to the EJB container via the JMX bus.

2.1. Detached Invokers - The Transport Middlemen

We looked at the detached invoker architecture in the context of exposing RMI compatible interfaces of MBean services earlier. Here we will look at how detached invokers are used to expose the EJB container home and bean interfaces to clients. The generic view of the invoker architecture is presented in *Figure 14.3, "The transport invoker server side architecture"*.

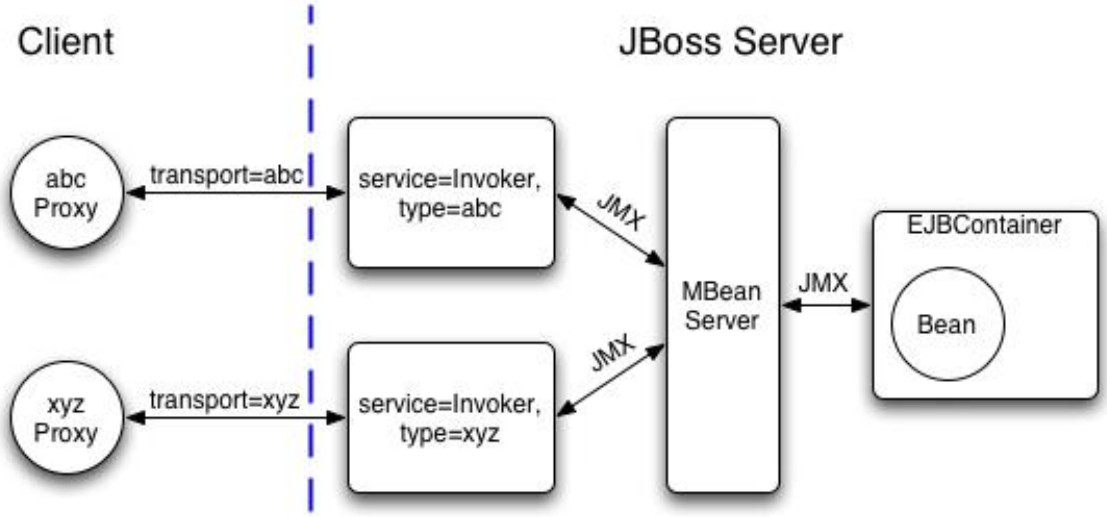


Figure 14.3. The transport invoker server side architecture

For each type of home proxy there is a binding to an invoker and its associated transport protocol. A container may have multiple invocation protocols active simultaneously. In the `jboss.xml` file, an `invoker-proxy-binding-name` maps to an `invoker-proxy-binding/name` element. At the `container-configuration` level this specifies the default invoker that will be used for EJBs deployed to the container. At the bean level, the `invoker-bindings` specify one or more invokers to use with the EJB container MBean.

When one specifies multiple invokers for a given EJB deployment, the home proxy must be given a unique JNDI binding location. This is specified by the `invoker/jndi-name` element

value. Another issue when multiple invokers exist for an EJB is how to handle remote homes or interfaces obtained when the EJB calls other beans. Any such interfaces need to use the same invoker used to call the outer EJB in order for the resulting remote homes and interfaces to be compatible with the proxy the client has initiated the call through. The `invoker/ejb-ref` elements allow one to map from a protocol independent ENC `ejb-ref` to the home proxy binding for `ejb-ref` target EJB home that matches the referencing invoker type.

An example of using a custom `JRMPInvoker` MBean that enables compressed sockets for session beans can be found in the `org.jboss.test.jrmp` package of the testsuite. The following example illustrates the custom `JRMPInvoker` configuration and its mapping to a stateless session bean.

```
<server>
    <mbean code="org.jboss.invocation.jrmp.server.JRMPInvoker"
name="jboss:service=invoker,type=jrmp,socketType=CompressionSocketFactory">
        <attribute name="RMIObjectPort">4445</attribute>
        <attribute name="RMIClientSocketFactory">
            org.jboss.test.jrmp.ejb.CompressionClientSocketFactory
        </attribute>
        <attribute name="RMIServerSocketFactory">
            org.jboss.test.jrmp.ejb.CompressionServerSocketFactory
        </attribute>
</mbean>
                </server>
```

Here the default `JRMPInvoker` has been customized to bind to port 4445 and to use custom socket factories that enable compression at the transport level.

```
<?xml version="1.0"?>
<!DOCTYPE jboss PUBLIC
          "-//JBoss//DTD JBOSS 3.2//EN"
          "http://www.jboss.org/j2ee/dtd/jboss_3_2.dtd">
<!-- The jboss.xml descriptor for the jrmp-comp.jar ejb unit -->
<jboss>
    <enterprise-beans>
        <session>
            <ejb-name>StatelessSession</ejb-name>
            <configuration-name>Standard Stateless
SessionBean</configuration-name>
            <invoker-bindings>
                <invoker>
                    <invoker-proxy-binding-name>
                        stateless-compression-invoker
                    </invoker-proxy-binding-name>
                    <jndi-name>jrmp-compressed/StatelessSession</jndi-name>
                </invoker>
            </invoker-bindings>
        </session>
    </enterprise-beans>

    <invoker-proxy-bindings>
        <invoker-proxy-binding>
            <name>stateless-compression-invoker</name>
            <invoker-mbean>
```

```
jboss:service=invoker,type=jrmp,socketType=CompressionSocketFactory
            </invoker-mbean>
            <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-factory>
            <proxy-factory-config>
                <client-interceptors>
                    <home>
<interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
                    </home>
                    <bean>
                        <interceptor>
                            org.jboss.proxy.ejb.StatelessSessionInterceptor
                        </interceptor>
<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
                    </bean>
                </client-interceptors>
            </proxy-factory-config>
        </invoker-proxy-binding>
    </invoker-proxy-bindings>
</jboss>
```

The `StatelessSession` EJB `invoker-bindings` settings specify that the `stateless-compression-invoker` will be used with the home interface bound under the JNDI name `jrmp-compressed/StatelessSession`. The `stateless-compression-invoker` is linked to the custom JRMP invoker we just declared.

The following example, `org.jboss.test.hello` testsuite package, is an example of using the `HttpInvoker` to configure a stateless session bean to use the RMI/HTTP protocol.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss PUBLIC
          "-//JBoss//DTD JBOSS 3.2//EN"
          "http://www.jboss.org/j2ee/dtd/jboss_3_2.dtd">
<jboss>
    <enterprise-beans>
        <session>
            <ejb-name>HelloWorldViaHTTP</ejb-name>
            <jndi-name>helloworld/HelloHTTP</jndi-name>
            <invoker-bindings>
                <invoker>
                    <invoker-proxy-binding-name>
                        stateless-http-invoker
                    </invoker-proxy-binding-name>
                </invoker>
            </invoker-bindings>
        </session>
    </enterprise-beans>
    <invoker-proxy-bindings>
        <!-- A custom invoker for RMI/HTTP -->
        <invoker-proxy-binding>
```

```
            <name>stateless-http-invoker</name>
            <invoker-mbean>jboss:service=invoker,type=http</invoker-mbean>
            <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-factory>
            <proxy-factory-config>
                <client-interceptors>
                    <home>
<interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
                    </home>
                    <bean>
                        <interceptor>
                            org.jboss.proxy.ejb.StatelessSessionInterceptor
                        </interceptor>
<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
                    </bean>
                </client-interceptors>
            </proxy-factory-config>
        </invoker-proxy-binding>
    </invoker-proxy-bindings>
</jboss>
```

Here a custom invoker-proxy-binding named `stateless-http-invoker` is defined. It uses the `HttpInvoker` MBean as the detached invoker. The `jboss:service=invoker,type=http` name is the default name of the `HttpInvoker` MBean as found in the `http-invoker.sar/META-INF/jboss-service.xml` descriptor, and its service descriptor fragment is show here:

```
<!-- The HTTP invoker service configuration -->
<mbean code="org.jboss.invocation.http.server.HttpInvoker"
       name="jboss:service=invoker,type=http">
    <!-- Use a URL of the form
http://<hostname>:8080/invoker/EJBInvokerServlet
         where <hostname> is InetAddress.getHostname value on which the
server
         is running. -->
    <attribute name="InvokerURLPrefix">http://</attribute>
    <attribute
name="InvokerURLSuffix">:8080/invoker/EJBInvokerServlet</attribute>
    <attribute name="UseHostName">true</attribute>
</mbean>
```

The client proxy posts the EJB invocation content to the `EJBInvokerServlet` URL specified in the `HttpInvoker` service configuration.

2.2. The HA JRMPInvoker - Clustered RMI/JRMP Transport

The `org.jboss.invocation.jrmp.server.JRMPInvokerHA` service is an extension of the

JRMPInvoker that is a cluster aware invoker. The JRMPInvokerHA fully supports all of the attributes of the JRMPInvoker. This means that customized bindings of the port, interface and socket transport are available to clustered RMI/JRMP as well. For additional information on the clustering architecture and the implementation of the HA RMI proxies see the JBoss Clustering docs.

2.3. The HA HttpInvoker - Clustered RMI/HTTP Transport

The RMI/HTTP layer allows for software load balancing of the invocations in a clustered environment. An HA capable extension of the HTTP invoker has been added that borrows much of its functionality from the HA-RMI/JRMP clustering.

To enable HA-RMI/HTTP you need to configure the invokers for the EJB container. This is done through either a jboss.xml descriptor, or the standardjboss.xml descriptor. *Example 14.3, "A jboss.xml stateless session configuration for HA-RMI/HTTP"* shows is an example of a stateless session configuration taken from the org.jboss.test.hello testsuite package.

```
<jboss>
    <enterprise-beans>
        <session>
            <ejb-name>HelloWorldViaClusteredHTTP</ejb-name>
            <jndi-name>helloworld/HelloHA-HTTP</jndi-name>
            <invoker-bindings>
                <invoker>
                    <invoker-proxy-binding-name>
                        stateless-httpHA-invoker
                    </invoker-proxy-binding-name>
                </invoker>
            </invoker-bindings>
            <clustered>true</clustered>
        </session>
    </enterprise-beans>
    <invoker-proxy-bindings>
        <invoker-proxy-binding>
            <name>stateless-httpHA-invoker</name>
            <invoker-mbean>jboss:service=invoker,type=httpHA</invoker-mbean>
<proxy-factory>org.jboss.proxy.ejb.ProxyFactoryHA</proxy-factory>
            <proxy-factory-config>
                <client-interceptors>
                    <home>
<interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
                    </home>
                    <bean>
                        <interceptor>
                            org.jboss.proxy.ejb.StatelessSessionInterceptor
                        </interceptor>
<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
<interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
                    </bean>
                </client-interceptors>
```

```
            </proxy-factory-config>
         </invoker-proxy-binding>
    </invoker-proxy-bindings>
</jboss>
```

Example 14.3. A jboss.xml stateless session configuration for HA-RMI/HTTP

The `stateless-httpHA-invoker` invoker-proxy-binding references the `jboss:service=invoker,type=httpHA` invoker service. This service would be configured as shown below.

```
<mbean code="org.jboss.invocation.http.server.HttpInvokerHA"
       name="jboss:service=invoker,type=httpHA">
    <!-- Use a URL of the form
         http://<hostname>:8080/invoker/EJBInvokerHAServlet
         where <hostname> is InetAddress.getHostname value on which the
server
         is running.
    -->
    <attribute name="InvokerURLPrefix">http://</attribute>
    <attribute
name="InvokerURLSuffix">:8080/invoker/EJBInvokerHAServlet</attribute>
    <attribute name="UseHostName">true</attribute>
</mbean>
```

The URL used by the invoker proxy is the `EJBInvokerHAServlet` mapping as deployed on the cluster node. The `HttpInvokerHA` instances across the cluster form a collection of candidate http URLs that are made available to the client side proxy for failover and/or load balancing.

3. The EJB Container

An EJB container is the component that manages a particular class of EJB. In JBoss there is one instance of the `org.jboss.ejb.Container` created for each unique configuration of an EJB that is deployed. The actual object that is instantiated is a subclass of `Container` and the creation of the container instance is managed by the `EJBDeployer` MBean.

3.1. EJBDeployer MBean

The `org.jboss.ejb.EJBDeployer` MBean is responsible for the creation of EJB containers. Given an EJB JAR that is ready for deployment, the `EJBDeployer` will create and initialize the necessary EJB containers, one for each type of EJB. The configurable attributes of the `EJBDeployer` are:

- **VerifyDeployments**: a boolean flag indicating if the EJB verifier should be run. This validates that the EJBs in a deployment unit conform to the EJB 2.1 specification. Setting this to true is useful for ensuring your deployments are valid.
- **VerifierVerbose**: A boolean that controls the verboseness of any verification failures/warnings that result from the verification process.
- **StrictVerifier**: A boolean that enables/disables strict verification. When strict verification is enable an EJB will deploy only if verifier reports no errors.
- **CallByValue**: a boolean flag that indicates call by value semantics should be used by default.
- **ValidateDTDs**: a boolean flag that indicates if the `ejb-jar.xml` and `jboss.xml` descriptors should be validated against their declared DTDs. Setting this to true is useful for ensuring your deployment descriptors are valid.
- **MetricsEnabled**: a boolean flag that controls whether container interceptors marked with an `metricsEnabled=true` attribute should be included in the configuration. This allows one to define a container interceptor configuration that includes metrics type interceptors that can be toggled on and off.
- **WebServiceName**: The JMX ObjectName string of the web service MBean that provides support for the dynamic class loading of EJB classes.
- **TransactionManagerServiceName**: The JMX ObjectName string of the JTA transaction manager service. This must have an attribute named `TransactionManager` that returns that `javax.transaction.TransactionManager` instance.

The deployer contains two central methods: deploy and undeploy. The deploy method takes a URL, which either points to an EJB JAR, or to a directory whose structure is the same as a valid EJB JAR (which is convenient for development purposes). Once a deployment has been made, it can be undeployed by calling undeploy on the same URL. A call to deploy with an already deployed URL will cause an undeploy, followed by deployment of the URL. JBoss has support for full re-deployment of both implementation and interface classes, and will reload any changed classes. This will allow you to develop and update EJBs without ever stopping a running server.

During the deployment of the EJB JAR the `EJBDeployer` and its associated classes perform three main functions, verify the EJBs, create a container for each unique EJB, initialize the container with the deployment configuration information. We will talk about each function in the following sections.

3.1.1. Verifying EJB deployments

When the `VerifyDeployments` attribute of the `EJBDeployer` is true, the deployer performs a verification of EJBs in the deployment. The verification checks that an EJB meets EJB specification compliance. This entails validating that the EJB deployment unit contains the required home and remote, local home and local interfaces. It will also check that the objects appearing in these interfaces are of the proper types and that the required methods are present in the implementation class. This is a useful behavior that is enabled by default since there are

a number of steps that an EJB developer and deployer must perform correctly to construct a proper EJB JAR, and it is easy to make a mistake. The verification stage attempts to catch any errors and fail the deployment with an error that indicates what needs to be corrected.

Probably the most problematic aspect of writing EJBs is the fact that there is a disconnection between the bean implementation and its remote and home interfaces, as well as its deployment descriptor configuration. It is easy to have these separate elements get out of synch. One tool that helps eliminate this problem is XDoclet. It allows you to use custom JavaDoc-like tags in the EJB bean implementation class to generate the related bean interfaces, deployment descriptors and related objects. See the XDoclet home page, *http://sourceforge.net/projects/xdoclet* for additional details.

3.1.2. Deploying EJBs Into Containers

The most important role performed by the `EJBDeployer` is the creation of an EJB container and the deployment of the EJB into the container. The deployment phase consists of iterating over EJBs in an EJB JAR, and extracting the bean classes and their metadata as described by the `ejb-jar.xml` and `jboss.xml` deployment descriptors. For each EJB in the EJB JAR, the following steps are performed:

- Create subclass of `org.jboss.ejb.Container` depending on the type of the EJB: stateless, stateful, BMP entity, CMP entity, or message driven. The container is assigned a unique `ClassLoader` from which it can load local resources. The uniqueness of the `ClassLoader` is also used to isolate the standard `java:comp` JNDI namespace from other J2EE components.
- Set all container configurable attributes from a merge of the `jboss.xml` and `standardjboss.xml` descriptors.
- Create and add the container interceptors as configured for the container.
- Associate the container with an application object. This application object represents a J2EE enterprise application and may contain multiple EJBs and web contexts.

If all EJBs are successfully deployed, the application is started which in turn starts all containers and makes the EJBs available to clients. If any EJB fails to deploy, a deployment exception is thrown and the deployment module is failed.

3.1.3. Container configuration information

JBoss externalizes most if not all of the setup of the EJB containers using an XML file that conforms to the `jboss_4_0.dtd`. The section DTD that relates to container configuration information is shown in *Figure 14.4, “The jboss_4_0 DTD elements related to container configuration.”*.

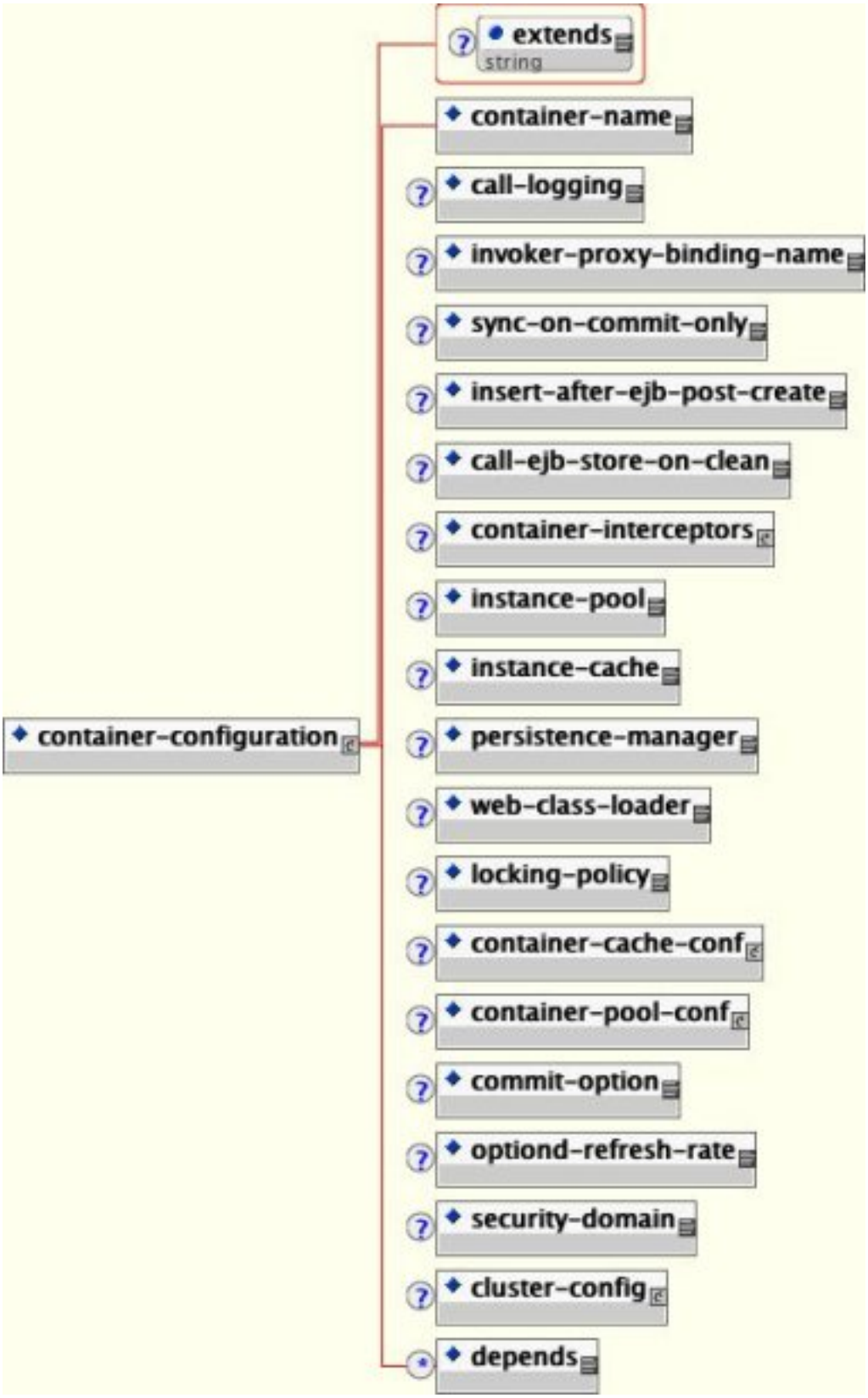


Figure 14.4. The jboss_4_0 DTD elements related to container

configuration.

The `container-configuration` element and its subelements specify container configuration settings for a type of container as given by the `container-name` element. Each configuration specifies information such as the default invoker type, the container interceptor makeup, instance caches/pools and their sizes, persistence manager, security, and so on. Because this is a large amount of information that requires a detailed understanding of the JBoss container architecture, JBoss ships with a standard configuration for the four types of EJBs. This configuration file is called `standardjboss.xml` and it is located in the conf directory of any configuration file set that uses EJBs. The following is a sample of `container-configuration` from `standardjboss.xml`.

```
<container-configuration>
    <container-name>Standard CMP 2.x EntityBean</container-name>
    <call-logging>false</call-logging>
<invoker-proxy-binding-name>entity-rmi-invoker</invoker-proxy-binding-name>
    <sync-on-commit-only>false</sync-on-commit-only>
    <insert-after-ejb-post-create>false</insert-after-ejb-post-create>
    <call-ejb-store-on-clean>true</call-ejb-store-on-clean>
    <container-interceptors>
<interceptor>org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor</interceptor>
        <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>
        <interceptor>org.jboss.ejb.plugins.SecurityInterceptor</interceptor>
        <interceptor>org.jboss.ejb.plugins.TxInterceptorCMT</interceptor>
<interceptor>org.jboss.ejb.plugins.CallValidationInterceptor</interceptor>
        <interceptor metricsEnabled="true">
            org.jboss.ejb.plugins.MetricsInterceptor
        </interceptor>
<interceptor>org.jboss.ejb.plugins.EntityCreationInterceptor</interceptor>
<interceptor>org.jboss.ejb.plugins.EntityLockInterceptor</interceptor>
<interceptor>org.jboss.ejb.plugins.EntityInstanceInterceptor</interceptor>
<interceptor>org.jboss.ejb.plugins.EntityReentranceInterceptor</interceptor>
        <interceptor>
org.jboss.resource.connectionmanager.CachedConnectionInterceptor
        </interceptor>
<interceptor>org.jboss.ejb.plugins.EntitySynchronizationInterceptor</interceptor>
<interceptor>org.jboss.ejb.plugins.cmp.jdbc.JDBCRelationInterceptor</interceptor>
    </container-interceptors>
    <instance-pool>org.jboss.ejb.plugins.EntityInstancePool</instance-pool>
<instance-cache>org.jboss.ejb.plugins.InvalidableEntityInstanceCache</instance-cache>
<persistence-manager>org.jboss.ejb.plugins.cmp.jdbc.JDBCStoreManager</persistence-manager>
<locking-policy>org.jboss.ejb.plugins.lock.QueuedPessimisticEJBLock</locking-policy>
    <container-cache-conf>
<cache-policy>org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy</cache-policy>
        <cache-policy-conf>
            <min-capacity>50</min-capacity>
            <max-capacity>1000000</max-capacity>
            <overager-period>300</overager-period>
            <max-bean-age>600</max-bean-age>
            <resizer-period>400</resizer-period>
            <max-cache-miss-period>60</max-cache-miss-period>
            <min-cache-miss-period>1</min-cache-miss-period>
            <cache-load-factor>0.75</cache-load-factor>
```

```
        </cache-policy-conf>
    </container-cache-conf>
    <container-pool-conf>
        <MaximumSize>100</MaximumSize>
    </container-pool-conf>
    <commit-option>B</commit-option>
</container-configuration>
```

These two examples demonstrate how extensive the container configuration options are. The container configuration information can be specified at two levels. The first is in the `standardjboss.xml` file contained in the configuration file set directory. The second is at the EJB JAR level. By placing a `jboss.xml` file in the EJB JAR `META-INF` directory, you can specify either overrides for container configurations in the `standardjboss.xml` file, or entirely new named container configurations. This provides great flexibility in the configuration of containers. As you have seen, all container configuration attributes have been externalized and as such are easily modifiable. Knowledgeable developers can even implement specialized container components, such as instance pools or caches, and easily integrate them with the standard container configurations to optimize behavior for a particular application or environment.

How an EJB deployment chooses its container configuration is based on the explicit or implict `jboss/enterprise-beans/<type>/configuration-name` element. The `configuration-name` element is a link to a `container-configurations/container-configuration` element. It specifies which container configuration to use for the referring EJB. The link is from a `configuration-name` element to a `container-name` element.

You are able to specify container configurations per class of EJB by including a `container-configuration` element in the EJB definition. Typically one does not define completely new container configurations, although this is supported. The typical usage of a `jboss.xml` level `container-configuration` is to override one or more aspects of a `container-configuration` coming from the `standardjboss.xml` descriptor. This is done by specifying `container-configuration` that references the name of an existing `standardjboss.xmlcontainer-configuration/container-name` as the value for the `container-configuration/extends` attribute. The following example shows an example of defining a new `Secured Stateless SessionBean` configuration that is an extension of the `Standard Stateless SessionBean` configuration.

```
<?xml version="1.0"?>
<jboss>
    <enterprise-beans>
        <session>
            <ejb-name>EchoBean</ejb-name>
            <configuration-name>Secured Stateless
SessionBean</configuration-name>
            <!-- ... -->
        </session>
    </enterprise-beans>
    <container-configurations>
        <container-configuration extends="Standard Stateless SessionBean">
            <container-name>Secured Stateless SessionBean</container-name>
            <!-- Override the container security domain -->
```

```
            <security-domain>java:/jaas/my-security-domain</security-domain>
        </container-configuration>
    </container-configurations>
</jboss>
```

If an EJB does not provide a container configuration specification in the deployment unit EJB JAR, the container factory chooses a container configuration from the `standardjboss.xml` descriptor based on the type of the EJB. So, in reality there is an implicit `configuration-name` element for every type of EJB, and the mappings from the EJB type to default container configuration name are as follows:

- container-managed persistence entity version 2.0 = Standard CMP 2.x EntityBean
- container-managed persistence entity version 1.1 = Standard CMP EntityBean
- bean-managed persistence entity = Standard BMP EntityBean
- stateless session = Standard Stateless SessionBean
- stateful session = Standard Stateful SessionBean
- message driven = Standard Message Driven Bean

It is not necessary to indicate which container configuration an EJB is using if you want to use the default based on the bean type. It probably provides for a more self-contained descriptor to include the `configuration-name` element, but this is purely a matter of style.

Now that you know how to specify which container configuration an EJB is using and can define a deployment unit level override, we now will look at the `container-configuration` child elements in the following sections. A number of the elements specify interface class implementations whose configuration is affected by other elements, so before starting in on the configuration elements you need to understand the `org.jboss.metadata.XmlLoadable` interface.

The `XmlLoadable` interface is a simple interface that consists of a single method. The interface definition is:

```
import org.w3c.dom.Element;
public interface XmlLoadable
{
    public void importXml(Element element) throws Exception;
}
```

Classes implement this interface to allow their configuration to be specified via an XML document fragment. The root element of the document fragment is what would be passed to the `importXml` method. You will see a few examples of this as the container configuration elements are described in the following sections.

3.1.3.1. The container-name element

The `container-name` element specifies a unique name for a given configuration. EJBs link to a particular container configuration by setting their `configuration-name` element to the value of the `container-name` for the container configuration.

3.1.3.2. The call-logging element

The `call-logging` element expects a boolean (true or false) as its value to indicate whether or not the `LogInterceptor` should log method calls to a container. This is somewhat obsolete with the change to log4j, which provides a fine-grained logging API.

3.1.3.3. The invoker-proxy-binding-name element

The `invoker-proxy-binding-name` element specifies the name of the default invoker to use. In the absence of a bean level `invoker-bindings` specification, the `invoker-proxy-binding` whose name matches the `invoker-proxy-binding-name` element value will be used to create home and remote proxies.

3.1.3.4. The sync-on-commit-only element

This configures a performance optimization that will cause entity bean state to be synchronized with the database only at commit time. Normally the state of all the beans in a transaction would need to be synchronized when an finder method is called or when an remove method is called, for example.

3.1.3.5. insert-after-ejb-post-create

This is another entity bean optimization which cause the database insert command for a new entity bean to be delayed until the `ejbPostCreate` method is called. This allows normal CMP fields as well as CMR fields to be set in a single insert, instead of the default insert followed by an update, which allows removes the requirement for relation ship fields to allow null values.

3.1.3.6. call-ejb-store-on-clean

By the specification the container is required to call `ejbStore` method on an entity bean instance when transaction commits even if the instance was not modified in the transaction. Setting this to false will cause JBoss to only call `ejbStore` for dirty objects.

3.1.3.7. The container-interceptors Element

The `container-interceptors` element specifies one or more interceptor elements that are to be configured as the method interceptor chain for the container. The value of the interceptor element is a fully qualified class name of an `org.jboss.ejb.Interceptor` interface implementation. The container interceptors form a `linked-list` structure through which EJB method invocations pass. The first interceptor in the chain is invoked when the `MBeanServer` passes a method invocation to the container. The last interceptor invokes the business method on the bean. We will discuss the `Interceptor` interface latter in this chapter when we talk about

the container plugin framework. Generally, care must be taken when changing an existing standard EJB interceptor configuration as the EJB contract regarding security, transactions, persistence, and thread safety derive from the interceptors.

3.1.3.8. The instance-pool element

The `instance-pool` element specifies the fully qualified class name of an `org.jboss.ejb.InstancePool` interface implementation to use as the container `InstancePool`. We will discuss the InstancePool interface in detail latter in this chapter when we talk about the container plugin framework.

3.1.3.9. The container-pool-conf element

The `container-pool-conf` is passed to the `InstancePool` implementation class given by the `instance-pool` element if it implements the `XmlLoadable` interface. All current JBoss `InstancePool` implementations derive from the `org.jboss.ejb.plugins.AbstractInstancePool` class which provides support for elements shown in *Figure 14.5, "The container-pool-conf element DTD"*.

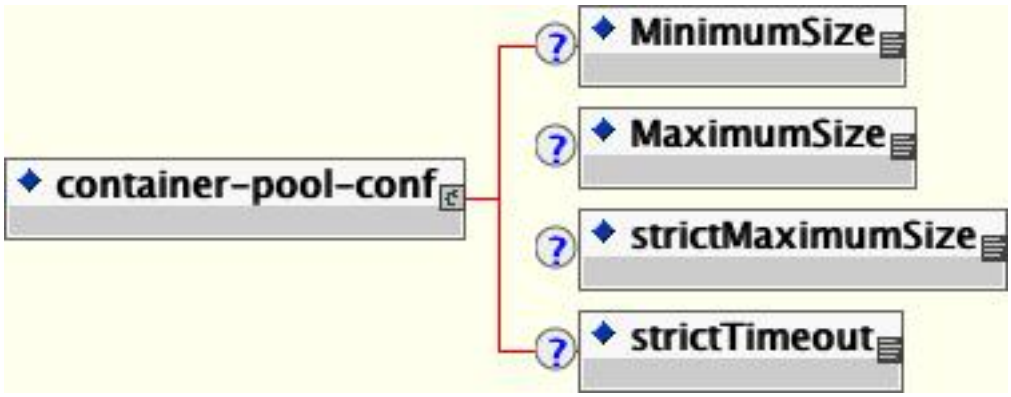


Figure 14.5. The container-pool-conf element DTD

- **MinimumSize**: The `MinimumSize` element gives the minimum number of instances to keep in the pool, although JBoss does not currently seed an `InstancePool` to the `MinimumSize` value.
- **MaximumSize**: The `MaximumSize` specifies the maximum number of pool instances that are allowed. The default use of `MaximumSize` may not be what you expect. The pool `MaximumSize` is the maximum number of EJB instances that are kept available, but additional instances can be created if the number of concurrent requests exceeds the `MaximumSize` value.
- **strictMaximumSize**: If you want to limit the maximum concurrency of an EJB to the pool `MaximumSize`, you need to set the `strictMaximumSize` element to true. When `strictMaximumSize` is true, only `MaximumSize` EJB instances may be active. When there are

MaximumSize active instances, any subsequent requests will be blocked until an instance is freed back to the pool. The default value for strictMaximumSize is false.

- **strictTimeout**: How long a request blocks waiting for an instance pool object is controlled by the strictTimeout element. The strictTimeout defines the time in milliseconds to wait for an instance to be returned to the pool when there are MaximumSize active instances. A value less than or equal to 0 will mean not to wait at all. When a request times out waiting for an instance a java.rmi.ServerException is generated and the call aborted. This is parsed as a Long so the maximum possible wait time is 9,223,372,036,854,775,807 or about 292,471,208 years, and this is the default value.

3.1.3.10. The instance-cache element

The instance-cache element specifies the fully qualified class name of the org.jboss.ejb.InstanceCache interface implementation. This element is only meaningful for entity and stateful session beans as these are the only EJB types that have an associated identity. We will discuss the InstanceCache interface in detail latter in this chapter when we talk about the container plugin framework.

3.1.3.11. The container-cache-conf element

The container-cache-conf element is passed to the InstanceCache implementation if it supports the XmlLoadable interface. All current JBoss InstanceCache implementations derive from the org.jboss.ejb.plugins.AbstractInstanceCache class which provides support for the XmlLoadable interface and uses the cache-policy child element as the fully qualified class name of an org.jboss.util.CachePolicy implementation that is used as the instance cache store. The cache-policy-conf child element is passed to the CachePolicy implementation if it supports the XmlLoadable interface. If it does not, the cache-policy-conf will silently be ignored.

There are two JBoss implementations of CachePolicy used by the standardjboss.xml configuration that support the current array of cache-policy-conf child elements. The classes are org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy and org.jboss.ejb.plugins.LRUStatefulContextCachePolicy. The LRUEnterpriseContextCachePolicy is used by entity bean containers while the LRUStatefulContextCachePolicy is used by stateful session bean containers. Both cache policies support the following cache-policy-conf child elements, shown in *Figure 14.6, "The container-cache-conf element DTD"*.

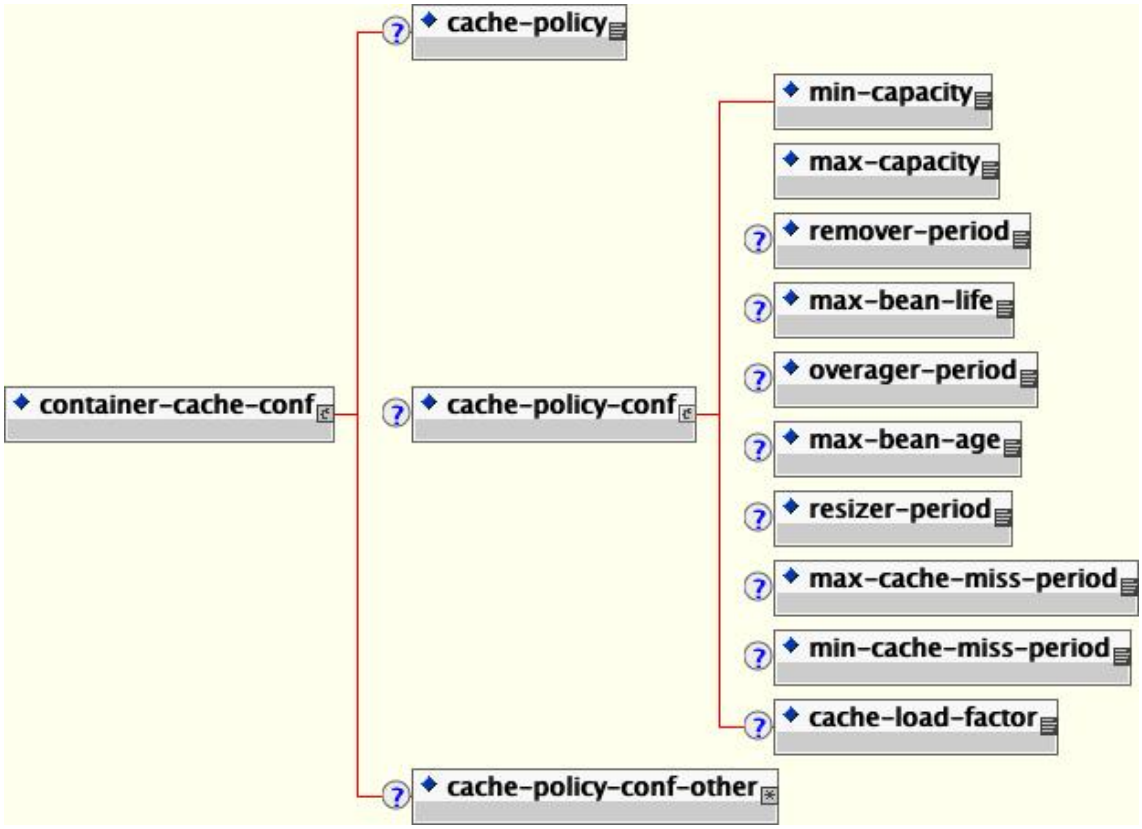


Figure 14.6. The container-cache-conf element DTD

- **min-capacity**: specifies the minimum capacity of this cache
- **max-capacity**: specifies the maximum capacity of the cache, which cannot be less than `min-capacity`.
- **overager-period**: specifies the period in seconds between runs of the overager task. The purpose of the overager task is to see if the cache contains beans with an age greater than the `max-bean-age` element value. Any beans meeting this criterion will be passivated.
- **max-bean-age**: specifies the maximum period of inactivity in seconds a bean can have before it will be passivated by the overager process.
- **resizer-period**: specifies the period in seconds between runs of the resizer task. The purpose of the resizer task is to contract or expand the cache capacity based on the remaining three element values in the following way. When the resizer task executes it checks the current period between cache misses, and if the period is less than the `min-cache-miss-period` value the cache is expanded up to the `max-capacity` value using the `cache-load-factor`. If instead the period between cache misses is greater than the `max-cache-miss-period` value the cache is contracted using the `cache-load-factor`.

- **max-cache-miss-period**: specifies the time period in seconds in which a cache miss should signal that the cache capacity be contracted. It is equivalent to the minimum miss rate that will be tolerated before the cache is contracted.
- **min-cache-miss-period**: specifies the time period in seconds in which a cache miss should signal that the cache capacity be expanded. It is equivalent to the maximum miss rate that will be tolerated before the cache is expanded.
- **cache-load-factor**: specifies the factor by which the cache capacity is contracted and expanded. The factor should be less than 1. When the cache is contracted the capacity is reduced so that the current ratio of beans to cache capacity is equal to the cache-load-factor value. When the cache is expanded the new capacity is determined as `current-capacity * 1/cache-load-factor`. The actual expansion factor may be as high as 2 based on an internal algorithm based on the number of cache misses. The higher the cache miss rate the closer the true expansion factor will be to 2.

The `LRUStatefulContextCachePolicy` also supports the remaining child elements:

- **remover-period**: specifies the period in seconds between runs of the remover task. The remover task removes passivated beans that have not been accessed in more than `max-bean-life` seconds. This task prevents stateful session beans that were not removed by users from filling up the passivation store.
- **max-bean-life**: specifies the maximum period of inactivity in seconds that a bean can exist before being removed from the passivation store.

An alternative cache policy implementation is the `org.jboss.ejb.plugins.NoPassivationCachePolicy` class, which simply never passivates instances. It uses an in-memory `HashMap` implementation that never discards instances unless they are explicitly removed. This class does not support any of the `cache-policy-conf` configuration elements.

3.1.3.12. The persistence-manager element

The `persistence-manager` element value specifies the fully qualified class name of the persistence manager implementation. The type of the implementation depends on the type of EJB. For stateful session beans it must be an implementation of the `org.jboss.ejb.StatefulSessionPersistenceManager` interface. For BMP entity beans it must be an implementation of the `org.jboss.ejb.EntityPersistenceManager` interface, while for CMP entity beans it must be an implementation of the `org.jboss.ejb.EntityPersistenceStore` interface.

3.1.3.13. The web-class-loader Element

The `web-class-loader` element specifies a subclass of `org.jboss.web.WebClassLoader` that is used in conjunction with the `WebService` MBean to allow dynamic loading of resources and classes from deployed ears, EJB JARs and WARs. A `WebClassLoader` is associated with a

`Container` and must have an `org.jboss.mx.loading.UnifiedClassLoader` as its parent. It overrides the `getURLs()` method to return a different set of URLs for remote loading than what is used for local loading.

`WebClaossLoader` has two methods meant to be overridden by subclasses: `getKey()` and `getBytes()`. The latter is a no-op in this implementation and should be overridden by subclasses with bytecode generation ability, such as the classloader used by the iiop module.

`WebClassLoader` subclasses must have a constructor with the same signature as the `WebClassLoader(ObjectName containerName, UnifiedClassLoader parent)` constructor.

3.1.3.14. The locking-policy element

The `locking-policy` element gives the fully qualified class name of the EJB lock implementation to use. This class must implement the `org.jboss.ejb.BeanLock` interface. The current JBoss versions include:

- **org.jboss.ejb.plugins.lock.QueuedPessimisticEJBLock**: an implementation that holds threads awaiting the transactional lock to be freed in a fair FIFO queue. Non-transactional threads are also put into this wait queue as well. This class pops the next waiting transaction from the queue and notifies only those threads waiting associated with that transaction. The `QueuedPessimisticEJBLock` is the current default used by the standard configurations.
- **org.jboss.ejb.plugins.lock.QueuedPessimisticEJBLockNoADE**: This behaves the same as the `QueuedPessimisticEJBLock` except that deadlock detection is disabled.
- **org.jboss.ejb.plugins.lock.SimpleReadWriteEJBLock**: This lock allows multiple read locks concurrently. Once a writer has requested the lock, future read-lock requests whose transactions do not already have the read lock will block until all writers are done; then all the waiting readers will concurrently go (depending on the reentrant setting / methodLock). A reader who promotes gets first crack at the write lock, ahead of other waiting writers. If there is already a reader that is promoting, we throw an inconsistent read exception. Of course, writers have to wait for all read-locks to release before taking the write lock.
- **org.jboss.ejb.plugins.lock.NoLock**: an anti-locking policy used with the instance per transaction container configurations.

Locking and deadlock detection will be discussed in more detail in *Section 4, “Entity Bean Locking and Deadlock Detection”*.

3.1.3.15. The commit-option and optiond-refresh-rate elements

The commit-option value specifies the EJB entity bean persistent storage commit option. It must be one of `A`, `B`, `C` or `D`.

- **A**: the container caches the beans state between transactions. This option assumes that the container is the only user accessing the persistent store. This assumption allows the

container to synchronize the in-memory state from the persistent storage only when absolutely necessary. This occurs before the first business method executes on a found bean or after the bean is passivated and reactivated to serve another business method. This behavior is independent of whether the business method executes inside a transaction context.

- **B**: the container caches the bean state between transactions. However, unlike option `A` the container does not assume exclusive access to the persistent store. Therefore, the container will synchronize the in-memory state at the beginning of each transaction. Thus, business methods executing in a transaction context don't see much benefit from the container caching the bean, whereas business methods executing outside a transaction context (transaction attributes Never, NotSupported or Supports) access the cached (and potentially invalid) state of the bean.
- **C**: the container does not cache bean instances. The in-memory state must be synchronized on every transaction start. For business methods executing outside a transaction the synchronization is still performed, but the `ejbLoad` executes in the same transaction context as that of the caller.
- **D**: is a JBoss-specific commit option which is not described in the EJB specification. It is a lazy read scheme where bean state is cached between transactions as with option `A`, but the state is periodically resynchronized with that of the persistent store. The default time between reloads is 30 seconds, but may configured using the `optiond-refresh-rate` element.

3.1.3.16. The security-domain element

The `security-domain` element specifies the JNDI name of the object that implements the `org.jboss.security.AuthenticationManager` and `org.jboss.security.RealmMapping` interfaces. It is more typical to specify the `security-domain` under the `jboss` root element so that all EJBs in a given deployment are secured in the same manner. However, it is possible to configure the security domain for each bean configuration. The details of the security manager interfaces and configuring the security layer are discussed in *Chapter 8, Security on JBoss*.

3.1.3.17. cluster-config

The `cluster-config` element allows to specify cluster specific settings for all EJBs that use the container configuration. Specification of the cluster configuration may be done at the container configuration level or at the individual EJB deployment level.

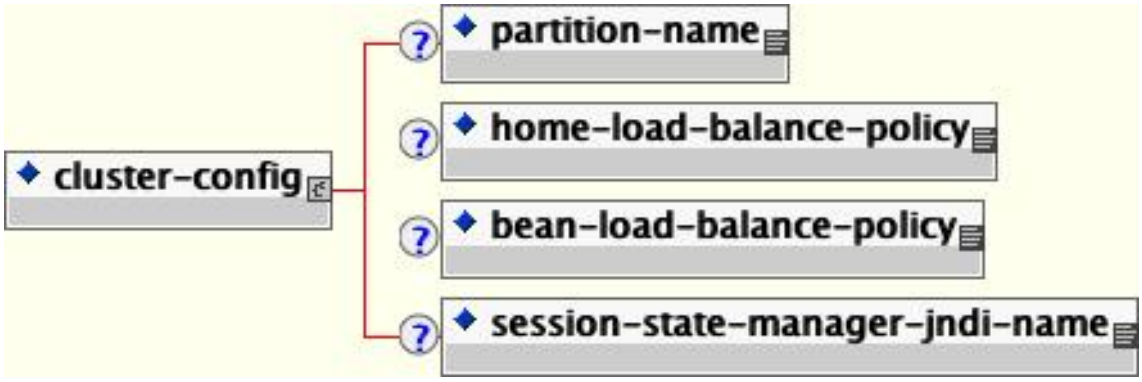


Figure 14.7. The cluster-config and related elements

- **partition-name**: The `partition-name` element indicates where to find the `org.jboss.ha.framework.interfaces.HAPartition` interface to be used by the container to exchange clustering information. This is not the full JNDI name under which `HAPartition` is bound. Rather, it should correspond to the `PartitionName` attribute of the `ClusterPartitionMBean` service that is managing the desired cluster. The actual JNDI name of the `HAPartition` binding will be formed by appending `/HASessionState/` to the partition-name value. The default value is `DefaultPartition`.
- **home-load-balance-policy**: The `home-load-balance-policy` element indicates the Java class name to be used to load balance calls made on the home proxy. The class must implement the `org.jboss.ha.framework.interface.LoadBalancePolicy` interface. The default policy is `org.jboss.ha.framework.interfaces.RoundRobin`.
- **bean-load-balance-policy**: The `bean-load-balance-policy` element indicates the java class name to be used to load balance calls in the bean proxy. The class must implement the `org.jboss.ha.framework.interface.LoadBalancePolicy` interface. For entity beans and stateful session beans, the default is `org.jboss.ha.framework.interfaces.FirstAvailavble`. For stateless session beans, `org.jboss.ha.framework.interfaces.RoundRobin`.
- **session-state-manager-jndi-name**: The `session-state-manager-jndi-name` element indicates the name of the `org.jboss.ha.framework.interfaces.HASessionState` to be used by the container as a backend for state session management in the cluster. Unlike the partition-name element, this is a JNDI name under which the `HASessionState` implementation is bound. The default location used is `/HASessionState/Default`.

3.1.3.18. The depends element

The `depends` element gives a JMX `ObjectName` of a service on which the container or EJB depends. Specification of explicit dependencies on other services avoids having to rely on the deployment order being after the required services are started.

3.2. Container Plug-in Framework

The JBoss EJB container uses a framework pattern that allows one to change implementations of various aspects of the container behavior. The container itself does not perform any significant work other than connecting the various behavioral components together. Implementations of the behavioral components are referred to as plugins, because you can plug in a new implementation by changing a container configuration. Examples of plug-in behavior you may want to change include persistence management, object pooling, object caching, container invokers and interceptors. There are four subclasses of the `org.jboss.ejb.Container` class, each one implementing a particular bean type:

- **org.jboss.ejb.EntityContainer**: handles `javax.ejb.EntityBean` types
- **org.jboss.ejb.StatelessSessionContainer**: handles Stateless `javax.ejb.SessionBean` types
- **org.jboss.ejb.StatefulSessionContainer**: handles Stateful `javax.ejb.SessionBean` types
- **org.jboss.ejb.MessageDrivenContainer** handles `javax.ejb.MessageDrivenBean` types

The EJB containers delegate much of their behavior to components known as container plug-ins. The interfaces that make up the container plugin points include the following:

- org.jboss.ejb.ContainerPlugin
- org.jboss.ejb.ContainerInvoker
- org.jboss.ejb.Interceptor
- org.jboss.ejb.InstancePool
- org.jboss.ejb.InstanceCache
- org.jboss.ejb.EntityPersistanceManager
- org.jboss.ejb.EntityPersistanceStore
- org.jboss.ejb.StatefulSessionPersistenceManager

The container's main responsibility is to manage its plug-ins. This means ensuring that the plug-ins have all the information they need to implement their functionality.

3.2.1. org.jboss.ejb.ContainerPlugin

The `ContainerPlugin` interface is the parent interface of all container plug-in interfaces. It provides a callback that allows a container to provide each of its plug-ins a pointer to the container the plug-in is working on behalf of. The `ContainerPlugin` interface is given below.

```
public interface ContainerPlugin
    extends Service, AllowedOperationsFlags
{
    /** co
     * This callback is set by the container so that the plugin
     * may access its container
     *
```

```
     * @param con the container which owns the plugin
     */
    public void setContainer(Container con);
}
```

Example 14.4. The org.jboss.ejb.ContainerPlugin interface

3.2.2. org.jboss.ejb.Interceptor

The `Interceptor` interface enables one to build a chain of method interceptors through which each EJB method invocation must pass. The `Interceptor` interface is given below.

```
import org.jboss.invocation.Invocation;

public interface Interceptor
    extends ContainerPlugin
{
    public void setNext(Interceptor interceptor);
    public Interceptor getNext();
    public Object invokeHome(Invocation mi) throws Exception;
    public Object invoke(Invocation mi) throws Exception;
}
```

Example 14.5. The org.jboss.ejb.Interceptor interface

All interceptors defined in the container configuration are created and added to the container interceptor chain by the `EJBDeployer`. The last interceptor is not added by the deployer but rather by the container itself because this is the interceptor that interacts with the EJB bean implementation.

The order of the interceptor in the chain is important. The idea behind ordering is that interceptors that are not tied to a particular `EnterpriseContext` instance are positioned before interceptors that interact with caches and pools.

Implementers of the `Interceptor` interface form a linked-list like structure through which the `Invocation` object is passed. The first interceptor in the chain is invoked when an invoker passes a `Invocation` to the container via the JMX bus. The last interceptor invokes the business method on the bean. There are usually on the order of five interceptors in a chain depending on the bean type and container configuration. `Interceptor` semantic complexity ranges from simple to complex. An example of a simple interceptor would be `LoggingInterceptor`, while a complex example is `EntitySynchronizationInterceptor`.

One of the main advantages of an interceptor pattern is flexibility in the arrangement of interceptors. Another advantage is the clear functional distinction between different interceptors.

For example, logic for transaction and security is cleanly separated between the `TXInterceptor` and `SecurityInterceptor` respectively.

If any of the interceptors fail, the call is terminated at that point. This is a fail-quickly type of semantic. For example, if a secured EJB is accessed without proper permissions, the call will fail as the `SecurityInterceptor` before any transactions are started or instances caches are updated.

3.2.3. org.jboss.ejb.InstancePool

An `InstancePool` is used to manage the EJB instances that are not associated with any identity. The pools actually manage subclasses of the `org.jboss.ejb.EnterpriseContext` objects that aggregate unassociated bean instances and related data.

```
public interface InstancePool
    extends ContainerPlugin
{
    /**
     * Get an instance without identity. Can be used
     * by finders and create-methods, or stateless beans
     *
     * @return Context /w instance
     * @exception RemoteException
     */
    public EnterpriseContext get() throws Exception;

    /** Return an anonymous instance after invocation.
     *
     * @param ctx
     */
    public void free(EnterpriseContext ctx);

    /**
     * Discard an anonymous instance after invocation.
     * This is called if the instance should not be reused,
     * perhaps due to some exception being thrown from it.
     *
     * @param ctx
     */
    public void discard(EnterpriseContext ctx);

    /**
     * Return the size of the pool.
     *
     * @return the size of the pool.
     */
    public int getCurrentSize();

    /**
     * Get the maximum size of the pool.
     *
     * @return the size of the pool.
     */
    public int getMaxSize();
```

```
}
```

Example 14.6. The org.jboss.ejb.InstancePool interface

Depending on the configuration, a container may choose to have a certain size of the pool contain recycled instances, or it may choose to instantiate and initialize an instance on demand.

The pool is used by the `InstanceCache` implementation to acquire free instances for activation, and it is used by interceptors to acquire instances to be used for Home interface methods (create and finder calls).

3.2.4. org.jboss.ebj.InstanceCache

The container `InstanceCache` implementation handles all EJB-instances that are in an active state, meaning bean instances that have an identity attached to them. Only entity and stateful session beans are cached, as these are the only bean types that have state between method invocations. The cache key of an entity bean is the bean primary key. The cache key for a stateful session bean is the session id.

```
public interface InstanceCache
    extends ContainerPlugin
{
    /**
     * Gets a bean instance from this cache given the identity.
     * This method may involve activation if the instance is not
     * in the cache.
     * Implementation should have O(1) complexity.
     * This method is never called for stateless session beans.
     *
     * @param id the primary key of the bean
     * @return the EnterpriseContext related to the given id
     * @exception RemoteException in case of illegal calls
     * (concurrent / reentrant), NoSuchObjectException if
     * the bean cannot be found.
     * @see #release
     */
    public EnterpriseContext get(Object id)
        throws RemoteException, NoSuchObjectException;

    /**
     * Inserts an active bean instance after creation or activation.
     * Implementation should guarantee proper locking and O(1) complexity.
     *
     * @param ctx the EnterpriseContext to insert in the cache
     * @see #remove
     */
    public void insert(EnterpriseContext ctx);

    /**
```

```
     * Releases the given bean instance from this cache.
     * This method may passivate the bean to get it out of the cache.
     * Implementation should return almost immediately leaving the
     * passivation to be executed by another thread.
     *
     * @param ctx the EnterpriseContext to release
     * @see #get
     */
    public void release(EnterpriseContext ctx);

    /**
     * Removes a bean instance from this cache given the identity.
     * Implementation should have O(1) complexity and guarantee
     * proper locking.
     *
     * @param id the primary key of the bean
     * @see #insert
     */
    public void remove(Object id);

    /**
     * Checks whether an instance corresponding to a particular
     * id is active
     *
     * @param id the primary key of the bean
     * @see #insert
     */
    public boolean isActive(Object id);
}
```

Example 14.7. The org.jboss.ejb.InstanceCache interface

In addition to managing the list of active instances, the `InstanceCache` is also responsible for activating and passivating instances. If an instance with a given identity is requested, and it is not currently active, the `InstanceCache` must use the `InstancePool` to acquire a free instance, followed by the persistence manager to activate the instance. Similarly, if the `InstanceCache` decides to passivate an active instance, it must call the persistence manager to passivate it and release the instance to the `InstancePool`.

3.2.5. org.jboss.ejb.EntityPersistenceManager

The `EntityPersistenceManager` is responsible for the persistence of EntityBeans. This includes the following:

- Creating an EJB instance in a storage
- Loading the state of a given primary key into an EJB instance
- Storing the state of a given EJB instance
- Removing an EJB instance from storage

- Activating the state of an EJB instance
- Passivating the state of an EJB instance

```
public interface EntityPersistenceManager
    extends ContainerPlugin
{
    /**
     * Returns a new instance of the bean class or a subclass of the
     * bean class.
     *
     * @return the new instance
     */
    Object createBeanClassInstance() throws Exception;

    /**
     * This method is called whenever an entity is to be created. The
     * persistence manager is responsible for calling the ejbCreate method
     * on the instance and to handle the results properly wrt the persistent
     * store.
     *
     * @param m the create method in the home interface that was
     * called
     * @param args any create parameters
     * @param instance the instance being used for this create call
     */
    void createEntity(Method m,
                      Object[] args,
                      EntityEnterpriseContext instance)
        throws Exception;

    /**
     * This method is called whenever an entity is to be created. The
     * persistence manager is responsible for calling the ejbPostCreate
method
     * on the instance and to handle the results properly wrt the persistent
     * store.
     *
     * @param m the create method in the home interface that was
     * called
     * @param args any create parameters
     * @param instance the instance being used for this create call
     */
    void postCreateEntity(Method m,
                          Object[] args,
                          EntityEnterpriseContext instance)
        throws Exception;

    /**
     * This method is called when single entities are to be found. The
     * persistence manager must find out whether the wanted instance is
     * available in the persistence store, and if so it shall use the
     * ContainerInvoker plugin to create an EJBObject to the instance, which
     * is to be returned as result.
     *
     * @param finderMethod the find method in the home interface that was
     * called
```

```
     * @param args any finder parameters
     * @param instance the instance to use for the finder call
     * @return an EJBObject representing the found entity
     */
    Object findEntity(Method finderMethod,
                      Object[] args,
                      EntityEnterpriseContext instance)
        throws Exception;

    /**
     * This method is called when collections of entities are to be
     * found. The persistence manager must find out whether the wanted
     * instances are available in the persistence store, and if so it
     * shall use the ContainerInvoker plugin to create EJBObjects to
     * the instances, which are to be returned as result.
     *
     * @param finderMethod the find method in the home interface that was
     * called
     * @param args any finder parameters
     * @param instance the instance to use for the finder call
     * @return an EJBObject collection representing the found
     * entities
     */
    Collection findEntities(Method finderMethod,
                            Object[] args,
                            EntityEnterpriseContext instance)
                    throws Exception;

    /**
     * This method is called when an entity shall be activated. The
     * persistence manager must call the ejbActivate method on the
     * instance.
     *
     * @param instance the instance to use for the activation
     *
     * @throws RemoteException thrown if some system exception occurs
     */
    void activateEntity(EntityEnterpriseContext instance)
        throws RemoteException;

    /**
     * This method is called whenever an entity shall be load from the
     * underlying storage. The persistence manager must load the state
     * from the underlying storage and then call ejbLoad on the
     * supplied instance.
     *
     * @param instance the instance to synchronize
     *
     * @throws RemoteException thrown if some system exception occurs
     */
    void loadEntity(EntityEnterpriseContext instance)
        throws RemoteException;

    /**
     * This method is used to determine if an entity should be stored.
     *
```

```
     * @param instance the instance to check
     * @return true, if the entity has been modified
     * @throws Exception thrown if some system exception occurs
     */
    boolean isModified(EntityEnterpriseContext instance) throws Exception;

    /**
     * This method is called whenever an entity shall be stored to the
     * underlying storage. The persistence manager must call ejbStore
     * on the supplied instance and then store the state to the
     * underlying storage.
     *
     * @param instance the instance to synchronize
     *
     * @throws RemoteException thrown if some system exception occurs
     */
    void storeEntity(EntityEnterpriseContext instance)
        throws RemoteException;

    /**
     * This method is called when an entity shall be passivate. The
     * persistence manager must call the ejbPassivate method on the
     * instance.
     *
     * @param instance the instance to passivate
     *
     * @throws RemoteException thrown if some system exception occurs
     */
    void passivateEntity(EntityEnterpriseContext instance)
        throws RemoteException;

    /**
     * This method is called when an entity shall be removed from the
     * underlying storage. The persistence manager must call ejbRemove
     * on the instance and then remove its state from the underlying
     * storage.
     *
     * @param instance the instance to remove
     *
     * @throws RemoteException thrown if some system exception occurs
     * @throws RemoveException thrown if the instance could not be removed
     */
    void removeEntity(EntityEnterpriseContext instance)
        throws RemoteException, RemoveException;
}
```

Example 14.8. The org.jboss.ejb.EntityPersistenceManager interface

3.2.6. The org.jboss.ejb.EntityPersistenceStore interface

As per the EJB 2.1 specification, JBoss supports two entity bean persistence semantics:

container managed persistence (CMP) and bean managed persistence (BMP). The CMP implementation uses an implementation of the `org.jboss.ejb.EntityPersistanceStore` interface. By default this is the `org.jboss.ejb.plugins.cmp.jdbc.JDBCStoreManager` which is the entry point for the CMP2 persistence engine. The `EntityPersistanceStore` interface is shown below.

```
public interface EntityPersistenceStore
    extends ContainerPlugin
{
    /**
     * Returns a new instance of the bean class or a subclass of the
     * bean class.
     *
     * @return the new instance
     *
     * @throws Exception
     */
    Object createBeanClassInstance()
        throws Exception;

    /**
     * Initializes the instance context.
     *
     * <p>This method is called before createEntity, and should
     * reset the value of all cmpFields to 0 or null.
     *
     * @param ctx
     *
     * @throws RemoteException
     */
    void initEntity(EntityEnterpriseContext ctx);

    /**
     * This method is called whenever an entity is to be created.  The
     * persistence manager is responsible for handling the results
     * properly wrt the persistent store.
     *
     * @param m the create method in the home interface that was
     * called
     * @param args any create parameters
     * @param instance the instance being used for this create call
     * @return The primary key computed by CMP PM or null for BMP
     *
     * @throws Exception
     */
    Object createEntity(Method m,
                        Object[] args,
                        EntityEnterpriseContext instance)
        throws Exception;

    /**
     * This method is called when single entities are to be found. The
     * persistence manager must find out whether the wanted instance
     * is available in the persistence store, if so it returns the
     * primary key of the object.
```

```
     *
     * @param finderMethod the find method in the home interface that was
     * called
     * @param args any finder parameters
     * @param instance the instance to use for the finder call
     * @return a primary key representing the found entity
     *
     * @throws RemoteException thrown if some system exception occurs
     * @throws FinderException thrown if some heuristic problem occurs
     */
    Object findEntity(Method finderMethod,
                      Object[] args,
                      EntityEnterpriseContext instance)
        throws Exception;

    /**
     * This method is called when collections of entities are to be
     * found. The persistence manager must find out whether the wanted
     * instances are available in the persistence store, and if so it
     * must return a collection of primaryKeys.
     *
     * @param finderMethod the find method in the home interface that was
     * called
     * @param args any finder parameters
     * @param instance the instance to use for the finder call
     * @return an primary key collection representing the found
     * entities
     *
     * @throws RemoteException thrown if some system exception occurs
     * @throws FinderException thrown if some heuristic problem occurs
     */
    Collection findEntities(Method finderMethod,
                            Object[] args,
                            EntityEnterpriseContext instance)
        throws Exception;

    /**
     * This method is called when an entity shall be activated.
     *
     * <p>With the PersistenceManager factorization most EJB
     * calls should not exists However this calls permits us to
     * introduce optimizations in the persistence store. Particularly
     * the context has a "PersistenceContext" that a PersistenceStore
     * can use (JAWS does for smart updates) and this is as good a
     * callback as any other to set it up.
     * @param instance the instance to use for the activation
     *
     * @throws RemoteException thrown if some system exception occurs
     */
    void activateEntity(EntityEnterpriseContext instance)
        throws RemoteException;

    /**
     * This method is called whenever an entity shall be load from the
     * underlying storage. The persistence manager must load the state
     * from the underlying storage and then call ejbLoad on the
```

```
     * supplied instance.
     *
     * @param instance the instance to synchronize
     *
     * @throws RemoteException thrown if some system exception occurs
     */
    void loadEntity(EntityEnterpriseContext instance)
        throws RemoteException;

    /**
     * This method is used to determine if an entity should be stored.
     *
     * @param instance the instance to check
     * @return true, if the entity has been modified
     * @throws Exception thrown if some system exception occurs
     */
    boolean isModified(EntityEnterpriseContext instance)
        throws Exception;

    /**
     * This method is called whenever an entity shall be stored to the
     * underlying storage. The persistence manager must call ejbStore
     * on the supplied instance and then store the state to the
     * underlying storage.
     *
     * @param instance the instance to synchronize
     *
     * @throws RemoteException thrown if some system exception occurs
     */
    void storeEntity(EntityEnterpriseContext instance)
        throws RemoteException;

    /**
     * This method is called when an entity shall be passivate. The
     * persistence manager must call the ejbPassivate method on the
     * instance.
     *
     * <p>See the activate discussion for the reason for
     * exposing EJB callback * calls to the store.
     *
     * @param instance the instance to passivate
     *
     * @throws RemoteException thrown if some system exception occurs
     */
    void passivateEntity(EntityEnterpriseContext instance)
        throws RemoteException;

    /**
     * This method is called when an entity shall be removed from the
     * underlying storage. The persistence manager must call ejbRemove
     * on the instance and then remove its state from the underlying
     * storage.
     *
     * @param instance the instance to remove
     *
     * @throws RemoteException thrown if some system exception occurs
```

```
     * @throws RemoveException thrown if the instance could not be removed
     */
    void removeEntity(EntityEnterpriseContext instance)
        throws RemoteException, RemoveException;
}
```

Example 14.9. The org.jboss.ejb.EntityPersistanceStore interface

The default BMP implementation of the `EntityPersistenceManager` interface is `org.jboss.ejb.plugins.BMPPersistenceManager`. The BMP persistence manager is fairly simple since all persistence logic is in the entity bean itself. The only duty of the persistence manager is to perform container callbacks.

3.2.7. org.jboss.ejb.StatefulSessionPersistenceManager

The `StatefulSessionPersistenceManager` is responsible for the persistence of stateful `SessionBeans`. This includes the following:

- Creating stateful sessions in a storage
- Activating stateful sessions from a storage
- Passivating stateful sessions to a storage
- Removing stateful sessions from a storage

The `StatefulSessionPersistenceManager` interface is shown below.

```
public interface StatefulSessionPersistenceManager
    extends ContainerPlugin
{
    public void createSession(Method m, Object[] args,
                              StatefulSessionEnterpriseContext ctx)
        throws Exception;

    public void activateSession(StatefulSessionEnterpriseContext ctx)
        throws RemoteException;

    public void passivateSession(StatefulSessionEnterpriseContext ctx)
        throws RemoteException;

    public void removeSession(StatefulSessionEnterpriseContext ctx)
        throws RemoteException, RemoveException;

    public void removePassivated(Object key);
}
```

Example 14.10. The org.jboss.ejb.StatefulSessionPersistenceManager

interface

The default implementation of the `StatefulSessionPersistenceManager` interface is `org.jboss.ejb.plugins.StatefulSessionFilePersistenceManager`. As its name implies, `StatefulSessionFilePersistenceManager` utilizes the file system to persist stateful session beans. More specifically, the persistence manager serializes beans in a flat file whose name is composed of the bean name and session id with a `.ser` extension. The persistence manager restores a bean's state during activation and respectively stores its state during passivation from the bean's `.ser` file.

4. Entity Bean Locking and Deadlock Detection

This section provides information on what entity bean locking is and how entity beans are accessed and locked within JBoss. It also describes the problems you may encounter as you use entity beans within your system and how to combat these issues. Deadlocking is formally defined and examined. And, finally, we walk you through how to fine tune your system in terms of entity bean locking.

4.1. Why JBoss Needs Locking

Locking is about protecting the integrity of your data. Sometimes you need to be sure that only one user can update critical data at one time. Sometimes, access to sensitive objects in your system need to be serialized so that data is not corrupted by concurrent reads and writes. Databases traditionally provide this sort of functionality with transactional scopes and table and row locking facilities.

Entity beans are a great way to provide an object-oriented interface to relational data. Beyond that, they can improve performance by taking the load off of the database through caching and delaying updates until absolutely needed so that the database efficiency can be maximized. But, with caching, data integrity is a problem, so some form of application server level locking is needed for entity beans to provide the transaction isolation properties that you are used to with traditional databases.

4.2. Entity Bean Lifecycle

With the default configuration of JBoss there is only one active instance of a given entity bean in memory at one time. This applies for every cache configuration and every type of `commit-option`. The lifecycle for this instance is different for every commit-option though.

- For commit option *A*, this instance is cached and used between transactions.
- For commit option *B*, this instance is cached and used between transactions, but is marked as dirty at the end of a transaction. This means that at the start of a new transaction `ejbLoad` must be called.

- For commit option *C*, this instance is marked as dirty, released from the cache, and marked for passivation at the end of a transaction.
- For commit option *D*, a background refresh thread periodically calls `ejbLoad` on stale beans within the cache. Otherwise, this option works in the same way as *A*.

When a bean is marked for passivation, the bean is placed in a passivation queue. Each entity bean container has a passivation thread that periodically passivates beans that have been placed in the passivation queue. A bean is pulled out of the passivation queue and reused if the application requests access to a bean of the same primary key.

On an exception or transaction rollback, the entity bean instance is thrown out of cache entirely. It is not put into the passivation queue and is not reused by an instance pool. Except for the passivation queue, there is no entity bean instance pooling.

4.3. Default Locking Behavior

Entity bean locking is totally decoupled from the entity bean instance. The logic for locking is totally isolated and managed in a separate lock object. Because there is only one allowed instance of a given entity bean active at one time, JBoss employs two types of locks to ensure data integrity and to conform to the EJB spec.

- **Method Lock**: The method lock ensures that only one thread of execution at a time can invoke on a given Entity Bean. This is required by the EJB spec.
- **Transaction Lock**: A transaction lock ensures that only one transaction at a time has access to a give Entity Bean. This ensures the ACID properties of transactions at the application server level. Since, by default, there is only one active instance of any given Entity Bean at one time, JBoss must protect this instance from dirty reads and dirty writes. So, the default entity bean locking behavior will lock an entity bean within a transaction until it completes. This means that if any method at all is invoked on an entity bean within a transaction, no other transaction can have access to this bean until the holding transaction commits or is rolled back.

4.4. Pluggable Interceptors and Locking Policy

We saw that the basic entity bean lifecycle and behavior is defined by the container configuration defined in `standardjboss.xml` descriptor. Let's look at the `container-interceptors` definition for the *Standard CMP 2.x EntityBean* configuration.

```
<container-interceptors>
<interceptor>org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.SecurityInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.TxInterceptorCMT</interceptor>
<interceptor>org.jboss.ejb.plugins.CallValidationInterceptor</interceptor>
    <interceptor
metricsEnabled="true">org.jboss.ejb.plugins.MetricsInterceptor</interceptor>
```

```
<interceptor>org.jboss.ejb.plugins.EntityCreationInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.EntityLockInterceptor</interceptor>
<interceptor>org.jboss.ejb.plugins.EntityInstanceInterceptor</interceptor>
<interceptor>org.jboss.ejb.plugins.EntityReentranceInterceptor</interceptor>
<interceptor>org.jboss.resource.connectionmanager.CachedConnectionInterceptor</interceptor>
<interceptor>org.jboss.ejb.plugins.EntitySynchronizationInterceptor</interceptor>
<interceptor>org.jboss.ejb.plugins.cmp.jdbc.JDBCRelationInterceptor</interceptor>
</container-interceptors>
```

The interceptors shown above define most of the behavior of the entity bean. Below is an explanation of the interceptors that are relevant to this section.

- **EntityLockInterceptor**: This interceptor's role is to schedule any locks that must be acquired before the invocation is allowed to proceed. This interceptor is very lightweight and delegates all locking behavior to a pluggable locking policy.
- **EntityInstanceInterceptor**: The job of this interceptor is to find the entity bean within the cache or create a new one. This interceptor also ensures that there is only one active instance of a bean in memory at one time.
- **EntitySynchronizationInterceptor**: The role of this interceptor is to synchronize the state of the cache with the underlying storage. It does this with the `ejbLoad` and `ejbStore` semantics of the EJB specification. In the presence of a transaction this is triggered by transaction demarcation. It registers a callback with the underlying transaction monitor through the JTA interfaces. If there is no transaction the policy is to store state upon returning from invocation. The synchronization polices *A*, *B* and *C* of the specification are taken care of here as well as the JBoss specific commit-option *D*.

4.5. Deadlock

Finding deadlock problems and resolving them is the topic of this section. We will describe what deadlocking MBeans, how you can detect it within your application, and how you can resolve deadlocks. Deadlock can occur when two or more threads have locks on shared resources. *Figure 14.8, "Deadlock definition example"* illustrates a simple deadlock scenario. Here, `Thread 1` has the lock for `Bean A`, and `Thread 2` has the lock for `Bean B`. At a later time, `Thread 1` tries to lock `Bean B` and blocks because `Thread 2` has it. Likewise, as `Thread 2` tries to lock A it also blocks because `Thread 1` has the lock. At this point both threads are deadlocked waiting for access to the resource already locked by the other thread.

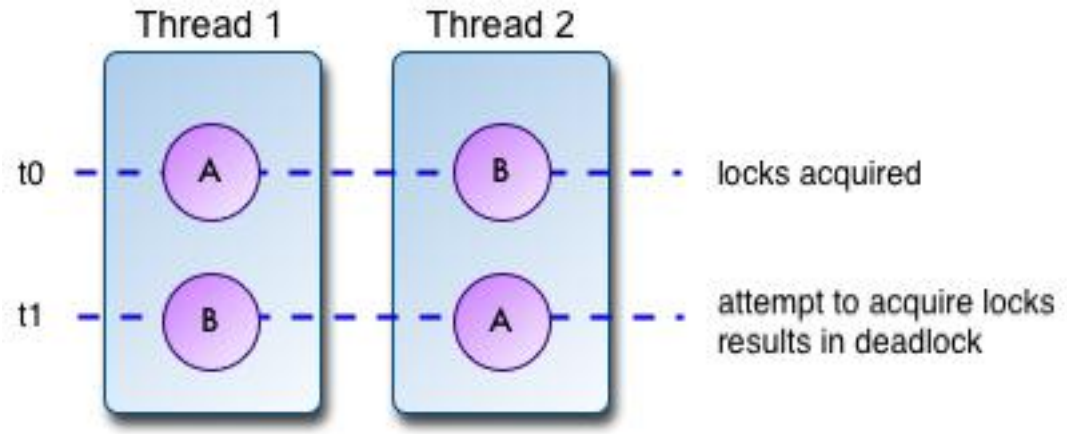


Figure 14.8. Deadlock definition example

The default locking policy of JBoss is to lock an Entity bean when an invocation occurs in the context of a transaction until the transaction completes. Because of this, it is very easy to encounter deadlock if you have long running transactions that access many entity beans, or if you are not careful about ordering the access to them. Various techniques and advanced configurations can be used to avoid deadlocking problems. They are discussed later in this section.

4.5.1. Deadlock Detection

Fortunately, JBoss is able to perform deadlock detection. JBoss holds a global internal graph of waiting transactions and what transactions they are blocking on. Whenever a thread determines that it cannot acquire an entity bean lock, it figures out what transaction currently holds the lock on the bean and add itself to the blocked transaction graph. An example of what the graph may look like is given in *Table 14.1, "An example blocked transaction table"*.

Blocking TX	Tx that holds needed lock
Tx1	Tx2
Tx3	Tx4
Tx4	Tx1

Table 14.1. An example blocked transaction table

Before the thread actually blocks it tries to detect whether there is deadlock problem. It does this by traversing the block transaction graph. As it traverses the graph, it keeps track of what transactions are blocked. If it sees a blocked node more than once in the graph, then it knows there is deadlock and will throw an `ApplicationDeadlockException`. This exception will cause a transaction rollback which will cause all locks that transaction holds to be released.

4.5.2. Catching ApplicationDeadlockException

Since JBoss can detect application deadlock, you should write your application so that it can retry a transaction if the invocation fails because of the `ApplicationDeadlockException`. Unfortunately, this exception can be deeply embedded within a `RemoteException`, so you have to search for it in your catch block. For example:

```
try {
    // ...
} catch (RemoteException ex) {
    Throwable cause = null;
    RemoteException rex = ex;
    while (rex.detail != null) {
        cause = rex.detail;
        if (cause instanceof ApplicationDeadlockException) {
                // ... We have deadlock, force a retry of the transaction.
            break;
        }
        if (cause instanceof RemoteException) {
            rex = (RemoteException)cause;
        }
    }
}
```

4.5.3. Viewing Lock Information

The `EntityLockMonitor` MBean service allows one to view basic locking statistics as well as printing out the state of the transaction locking table. To enable this monitor uncomment its configuration in the `conf/jboss-service.xml`:

```
<mbean code="org.jboss.monitor.EntityLockMonitor"
       name="jboss.monitor:name=EntityLockMonitor"/>
```

The `EntityLockMonitor` has no configurable attributes. It does have the following read-only attributes:

- **MedianWaitTime**: The median value of all times threads had to wait to acquire a lock.
- **AverageContenders**: The ratio of the total number of contentions to the sum of all threads that had to wait for a lock.
- **TotalContentions**: The total number of threads that had to wait to acquire the transaction lock. This happens when a thread attempts to acquire a lock that is associated with another transaction
- **MaxContenders**: The maximum number of threads that were waiting to acquire the transaction lock.

It also has the following operations:

- **clearMonitor**: This operation resets the lock monitor state by zeroing all counters.
- **printLockMonitor**: This operation prints out a table of all EJB locks that lists the `ejbName` of the bean, the total time spent waiting for the lock, the count of times the lock was waited on and the number of transactions that timed out waiting for the lock.

4.6. Advanced Configurations and Optimizations

The default locking behavior of entity beans can cause deadlock. Since access to an entity bean locks the bean into the transaction, this also can present a huge performance/throughput problem for your application. This section walks through various techniques and configurations that you can use to optimize performance and reduce the possibility of deadlock.

4.6.1. Short-lived Transactions

Make your transactions as short-lived and fine-grained as possible. The shorter the transaction you have, the less likelihood you will have concurrent access collisions and your application throughput will go up.

4.6.2. Ordered Access

Ordering the access to your entity beans can help lessen the likelihood of deadlock. This means making sure that the entity beans in your system are always accessed in the same exact order. In most cases, user applications are just too complicated to use this approach and more advanced configurations are needed.

4.6.3. Read-Only Beans

Entity beans can be marked as read-only. When a bean is marked as read-only, it never takes part in a transaction. This means that it is never transactionally locked. Using commit-option *D* with this option is sometimes very useful when your read-only bean's data is sometimes updated by an external source.

To mark a bean as read-only, use the `read-only` flag in the `jboss.xml` deployment descriptor.

```
<jboss>
    <enterprise-beans>
        <entity>
            <ejb-name>MyEntityBean</ejb-name>
            <jndi-name>MyEntityHomeRemote</jndi-name>
            <read-only>True</read-only>
        </entity>
    </enterprise-beans>
</jboss>
```

Example 14.11. Marking an entity bean read-only using jboss.xml

4.6.4. Explicitly Defining Read-Only Methods

After reading and understanding the default locking behavior of entity beans, you're probably wondering, "Why lock the bean if its not modifying the data?" JBoss allows you to define what methods on your entity bean are read only so that it will not lock the bean within the transaction if only these types of methods are called. You can define these read only methods within a `jboss.xml` deployment descriptor. Wildcards are allowed for method names. The following is an example of declaring all getter methods and the `anotherReadOnlyMethod` as read-only.

```
<jboss>
    <enterprise-beans>
        <entity>
            <ejb-name>nextgen.EnterpriseEntity</ejb-name>
            <jndi-name>nextgen.EnterpriseEntity</jndi-name>
            <method-attributes>
                <method>
                    <method-name>get*</method-name>
                    <read-only>true</read-only>
                </method>
                <method>
                    <method-name>anotherReadOnlyMethod</method-name>
                    <read-only>true</read-only>
                </method>
            </method-attributes>
        </entity>
    </enterprise-beans>
</jboss>
```

Example 14.12. Defining entity bean methods as read only

4.6.5. Instance Per Transaction Policy

The Instance Per Transaction policy is an advanced configuration that can totally wipe away deadlock and throughput problems caused by JBoss's default locking policy. The default Entity Bean locking policy is to only allow one active instance of a bean. The Instance Per Transaction policy breaks this requirement by allocating a new instance of a bean per transaction and dropping this instance at the end of the transaction. Because each transaction has its own copy of the bean, there is no need for transaction based locking.

This option does sound great but does have some drawbacks right now. First, the transactional isolation behavior of this option is equivalent to `READ_COMMITTED`. This can create repeatable reads when they are not desired. In other words, a transaction could have a copy of a stale bean. Second, this configuration option currently requires commit-option *B* or *C* which can be a performance drain since an ejbLoad must happen at the beginning of the transaction. But, if

your application currently requires commit-option *B* or *C* anyways, then this is the way to go. The JBoss developers are currently exploring ways to allow commit-option *A* as well (which would allow the use of caching for this option).

JBoss has container configurations named `Instance Per Transaction CMP 2.x EntityBean` and `Instance Per Transaction BMP EntityBean` defined in the standardjboss.xml that implement this locking policy. To use this configuration, you just have to reference the name of the container configuration to use with your bean in the jboss.xml deployment descriptor as show below.

```
<jboss>
    <enterprise-beans>
        <entity>
            <ejb-name>MyCMP2Bean</ejb-name>
            <jndi-name>MyCMP2</jndi-name>
            <configuration-name>
                Instance Per Transaction CMP 2.x EntityBean
            </configuration-name>
        </entity>
        <entity>
            <ejb-name>MyBMPBean</ejb-name>
            <jndi-name>MyBMP</jndi-name>
            <configuration-name>
                Instance Per Transaction BMP EntityBean
            </configuration-name>
        </entity>
    </enterprise-beans>
</jboss>
```

Example 14.13. An example of using the Instance Per Transaction policy.

4.7. Running Within a Cluster

Currently there is no distributed locking capability for entity beans within the cluster. This functionality has been delegated to the database and must be supported by the application developer. For clustered entity beans, it is suggested to use commit-option *B* or *C* in combination with a row locking mechanism. For CMP, there is a row-locking configuration option. This option will use a SQL `select for update` when the bean is loaded from the database. With commit-option *B* or *C*, this implements a transactional lock that can be used across the cluster. For BMP, you must explicitly implement the select for update invocation within the BMP's `ejbLoad` method.

4.8. Troubleshooting

This section will describe some common locking problems and their solution.

4.8.1. Locking Behavior Not Working

Many JBoss users observe that locking does not seem to be working and see concurrent access to their beans, and thus dirty reads. Here are some common reasons for this:

- If you have custom `container-configurations`, make sure you have updated these configurations.
- Make absolutely sure that you have implemented `equals` and `hashCode` correctly from custom/complex primary key classes.
- Make absolutely sure that your custom/complex primary key classes serialize correctly. One common mistake is assuming that member variable initializations will be executed when a primary key is unmarshalled.

4.8.2. IllegalStateException

An IllegalStateException with the message "removing bean lock and it has tx set!" usually means that you have not implemented `equals` and/or `hashCode` correctly for your custom/complex primary key class, or that your primary key class is not implemented correctly for serialization.

4.8.3. Hangs and Transaction Timeouts

One long outstanding bug of JBoss is that on a transaction timeout, that transaction is only marked for a rollback and not actually rolled back. This responsibility is delegated to the invocation thread. This can cause major problems if the invocation thread hangs indefinitely since things like entity bean locks will never be released. The solution to this problem is not a good one. You really just need to avoid doing stuff within a transaction that could hang indefinitely. One common mistake is making connections across the internet or running a web-crawler within a transaction.

5. EJB Timer Configuration

The J2EE timer service allows for any EJB object to register for a timer callback either at a designated time in the future. Timer events can be used for auditing, reporting or other cleanup tasks that need to need to happen at some given time in the future. Timer events are intended to be persistent and should be executed even in the event of a server failure. Coding to EJB timers is a standard part of the J2EE specification, so we won't explore the programming model. We will, instead, look at the configuration of the timer service in JBoss so that you can understand how to make timers work best in your environment

The EJB timer service is configure by several related MBeans in the `ejb-deployer.xml` file. The primary MBean is the `EJBTimerService` MBean.

```
<mbean code="org.jboss.ejb.txtimer.EJBTimerServiceImpl"
name="jboss.ejb:service=EJBTimerService">
    <attribute
name="RetryPolicy">jboss.ejb:service=EJBTimerService,retryPolicy=fixedDelay</attribute>
```

```
    <attribute
name="PersistencePolicy">jboss.ejb:service=EJBTimerService,persistencePolicy=database</attri
    <attribute
name="TimerIdGeneratorClassName">org.jboss.ejb.txtimer.BigIntegerTimerIdGenerator</attribute>
    <attribute
name="TimedObjectInvokerClassName">org.jboss.ejb.txtimer.TimedObjectInvokerImpl</attribute>
</mbean>
```

The `EJBTimerService` has the following configurable attributes:

- **RetryPolicy**: This is name of the MBean that implements the retry policy. The MBean must support the `org.jboss.ejb.txtimer.RetryPolicy interface`. JBoss provides one implementation, `FixedDelayRetryPolicy`, which will be described later.
- **PersistencePolicy**: This is the name of the MBean that implements the the persistence strategy for saving timer events. The MBean must support the `org.jboss.ejb.txtimer.PersistencePolicy` interface. JBoss provides two implementations, NoopPersistencePolicy and DatabasePersistencePolicy, which will be described later.
- **TimerIdGeneratorClassName**: This is the name of a class that provides the timer ID generator strategy. This class must implement the `org.jboss.ejb.txtimer.TimerIdGenerator` interface. JBoss provides the `org.jboss.ejb.txtimer.BigIntegerTimerIdGenerator` implementation.
- **TimedObjectInvokerClassname**: This is the name of a class that provides the timer method invocation strategy. This class must implement the `org.jboss.ejb.txtimer.TimedObjectInvoker` interface. JBoss provides the `org.jboss.ejb.txtimer.TimedObjectInvokerImpl` implementation.

The retry policy MBean definition used is shown here:

```
<mbean code="org.jboss.ejb.txtimer.FixedDelayRetryPolicy"
       name="jboss.ejb:service=EJBTimerService,retryPolicy=fixedDelay">
    <attribute name="Delay">100</attribute>
</mbean>
```

The retry policy takes one configuration value:

- **Delay**: This is the delay (ms) before retrying a failed timer execution. The default delay is 100ms.

If EJB timers do not need to be persisted, the `NoopPersistence` policy can be used. This MBean is commented out by default, but when enabled will look like this:

```
<mbean code="org.jboss.ejb.txtimer.NoopPersistencePolicy"
```

```
        name="jboss.ejb:service=EJBTimerService,persistencePolicy=noop"/>
```

Most applications that use timers will want timers to be persisted. For that the `DatabasePersitencePolicy` MBean should be used.

```
<mbean code="org.jboss.ejb.txtimer.DatabasePersistencePolicy"
       name="jboss.ejb:service=EJBTimerService,persistencePolicy=database">
    <!-- DataSource JNDI name -->
    <depends
optional-attribute-name="DataSource">jboss.jca:service=DataSourceBinding,name=DefaultDS</dep
    <!-- The plugin that handles database persistence -->
    <attribute
name="DatabasePersistencePlugin">org.jboss.ejb.txtimer.GeneralPurposeDatabasePersistencePlug
</mbean>
```

- **DataSource**: This is the MBean for the DataSource that timer data will be written to.
- **DatabasePersistencePlugin**: This is the name of the class the implements the persistence strategy. This should be `org.jboss.ejb.txtimer.GeneralPurposeDatabasePersistencePlugin`.

The CMP Engine

This chapter will explore the use of container managed persistence (CMP) in JBoss. We will assume a basic familiarity the EJB CMP model and focus on the operation of the JBoss CMP engine. Specifically, we will look at how to configure and optimize CMP applications on JBoss. For more introductory coverage of basic CMP concepts, we recommend *Enterprise Java Beans, Fourth Edition* (O'Reilly 2004).

1. Example Code

This chapter is example-driven. We will work with the crime portal application which stores information about imaginary criminal organizations. The data model we will be working with is shown in *Figure 15.1, "The crime portal example classes"*.

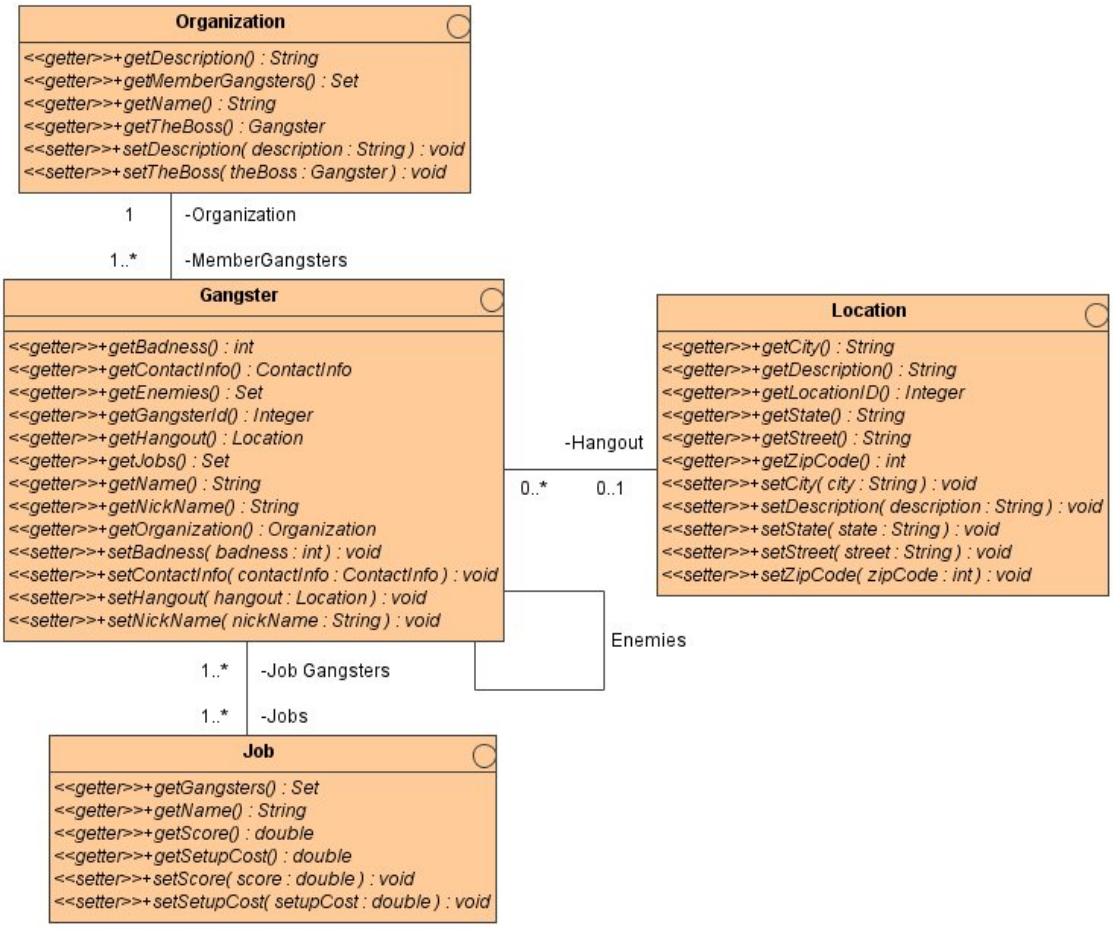


Figure 15.1. The crime portal example classes

The source code for the crime portal is available in the `src/main/org/jboss/cmp2` directory of the example code. To build the example code, run Ant as shown below

```
[examples]$ ant -Dchap=cmp2 config
```

This command builds and deploys the application to the JBoss server. When you start yours JBoss server, or if it is already running, you should see the following deployment messages:

```
15:46:36,704 INFO  [OrganizationBean$Proxy] Creating organization Yakuza,
Japanese Gangsters
15:46:36,790 INFO  [OrganizationBean$Proxy] Creating organization Mafia,
Italian Bad Guys
15:46:36,797 INFO  [OrganizationBean$Proxy] Creating organization Triads,
Kung Fu Movie Extras
15:46:36,877 INFO  [GangsterBean$Proxy] Creating Gangster 0 'Bodyguard'
Yojimbo
15:46:37,003 INFO  [GangsterBean$Proxy] Creating Gangster 1 'Master' Takeshi
15:46:37,021 INFO  [GangsterBean$Proxy] Creating Gangster 2 'Four finger'
Yuriko
15:46:37,040 INFO  [GangsterBean$Proxy] Creating Gangster 3 'Killer' Chow
15:46:37,106 INFO  [GangsterBean$Proxy] Creating Gangster 4 'Lightning'
Shogi
15:46:37,118 INFO  [GangsterBean$Proxy] Creating Gangster 5 'Pizza-Face'
Valentino
15:46:37,133 INFO  [GangsterBean$Proxy] Creating Gangster 6 'Toohless' Toni
15:46:37,208 INFO  [GangsterBean$Proxy] Creating Gangster 7 'Godfather'
Corleone
15:46:37,238 INFO  [JobBean$Proxy] Creating Job 10th Street Jeweler Heist
15:46:37,247 INFO  [JobBean$Proxy] Creating Job The Greate Train Robbery
15:46:37,257 INFO  [JobBean$Proxy] Creating Job Cheap Liquor Snatch and Grab
```

Since the beans in the examples are configured to have their tables removed on undeployment, anytime you restart the JBoss server you need to rerun the config target to reload the example data and re-deploy the application.

1.1. Enabling CMP Debug Logging

In order to get meaningful feedback from the chapter tests, you will want to increase the log level of the CMP subsystem before running running the test. To enable debug logging add the following category to your `log4j.xml` file:

```
<category name="org.jboss.ejb.plugins.cmp">
    <priority value="DEBUG"/>
</category>
```

In addition to this, it is necessary to decrease the threshold on the `CONSOLE` appender to allow debug level messages to be logged to the console. The following changes also need to be applied to the `log4j.xml` file.

```
<appender name="CONSOLE" class="org.apache.log4j.ConsoleAppender">
    <errorHandler class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
    <param name="Target"    value="System.out"/>
```

```
    <param name="Threshold" value="DEBUG" />

    <layout class="org.apache.log4j.PatternLayout">
        <!-- The default pattern: Date Priority [Category] Message\n -->
        <param name="ConversionPattern" value="%d{ABSOLUTE} %-5p [%c{1}]
%m%n"/>
    </layout>
</appender>
```

To see the full workings of the CMP engine you would need to enable the custom `TRACE` level priority on the `org.jboss.ejb.plugins.cmp` category as shown here:

```
<category name="org.jboss.ejb.plugins.cmp">
    <priority value="TRACE" class="org.jboss.logging.XLevel"/>
</category>
```

1.2. Running the examples

The first test target illustrates a number of the customization features that will be discussed throughout this chapter. To run these tests execute the following ant target:

```
[examples]$ ant -Dchap=cmp2 -Dex=test run-example
```

```
22:30:09,862 DEBUG [OrganizationEJB#findByPrimaryKey] Executing SQL: SELECT
t0_OrganizationEJ
B.name FROM ORGANIZATION t0_OrganizationEJB WHERE t0_OrganizationEJB.name=?
22:30:09,927 DEBUG [OrganizationEJB] Executing SQL: SELECT desc, the_boss
FROM ORGANIZATION W
HERE (name=?)
22:30:09,931 DEBUG [OrganizationEJB] load relation SQL: SELECT id FROM
GANGSTER WHERE (organi
zation=?)
22:30:09,947 DEBUG [StatelessSessionContainer] Useless invocation of
remove() for stateless s
ession bean
22:30:10,086 DEBUG [GangsterEJB#findBadDudes_ejbql] Executing SQL: SELECT
t0_g.id FROM GANGST
ER t0_g WHERE (t0_g.badness > ?)
22:30:10,097 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT
t0_GangsterEJB.id FRO
M GANGSTER t0_GangsterEJB WHERE t0_GangsterEJB.id=?
22:30:10,102 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT
t0_GangsterEJB.id FRO
M GANGSTER t0_GangsterEJB WHERE t0_GangsterEJB.id=?
```

These tests exercise various finders, selectors and object to table mapping issues. We will refer to the tests throughout the chapter.

The other main target runs a set of tests to demonstrate the optimized loading configurations presented in *Section 7, "Optimized Loading"*. Now that the logging is setup correctly, the read-ahead tests will display useful information about the queries performed. Note that you do not have to restart the JBoss server for it to recognize the changes to the log4j.xml file, but it may take a minute or so. The following shows the actual execution of the readahead client:

```
[examples]$ ant -Dchap=cmp2 -Dex=readahead run-example
```

When the readahead client is executed, all of the SQL queries executed during the test are displayed in the JBoss server console. The important items of note when analyzing the output are the number of queries executed, the columns selected, and the number of rows loaded. The following shows the read-ahead none portion of the JBoss server console output from readahead:

```
22:44:31,570 INFO  [ReadAheadTest]
############################################################
### read-ahead none
###
22:44:31,582 DEBUG [GangsterEJB#findAll_none] Executing SQL: SELECT t0_g.id
FROM GANGSTER t0_
g ORDER BY t0_g.id ASC
22:44:31,604 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,615 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,622 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,628 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,635 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,644 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,649 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,658 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name,
badness, organization
, hangout FROM GANGSTER WHERE (id=?)
22:44:31,670 INFO  [ReadAheadTest]
###
############################################################
...
```

We will revisit this example and explore the output when we discuss the settings for optimized

loading.

2. The jbosscmp-jdbc Structure

The `jbosscmp-jdbc.xml` descriptor is used to control the behavior of the JBoss engine. This can be done globally through the `conf/standardjbosscmp-jdbc.xml` descriptor found in the server configuration file set, or per EJB JAR deployment via a `META-INF/jbosscmp-jdbc.xml` descriptor.

The DTD for the `jbosscmp-jdbc.xml` descriptor can be found in `JBOSS_DIST/docs/dtd/jbosscmp-jdbc_4_0.dtd`. The public doctype for this DTD is:

```
<!DOCTYPE jbosscmp-jdbc PUBLIC
     "-//JBoss//DTD JBOSSCMP-JDBC 4.0//EN"
     "http://www.jboss.org/j2ee/dtd/jbosscmp-jdbc_4_0.dtd">
```

The top level elements are shown in *Figure 15.2, "The jbosscmp-jdbc content model."*.

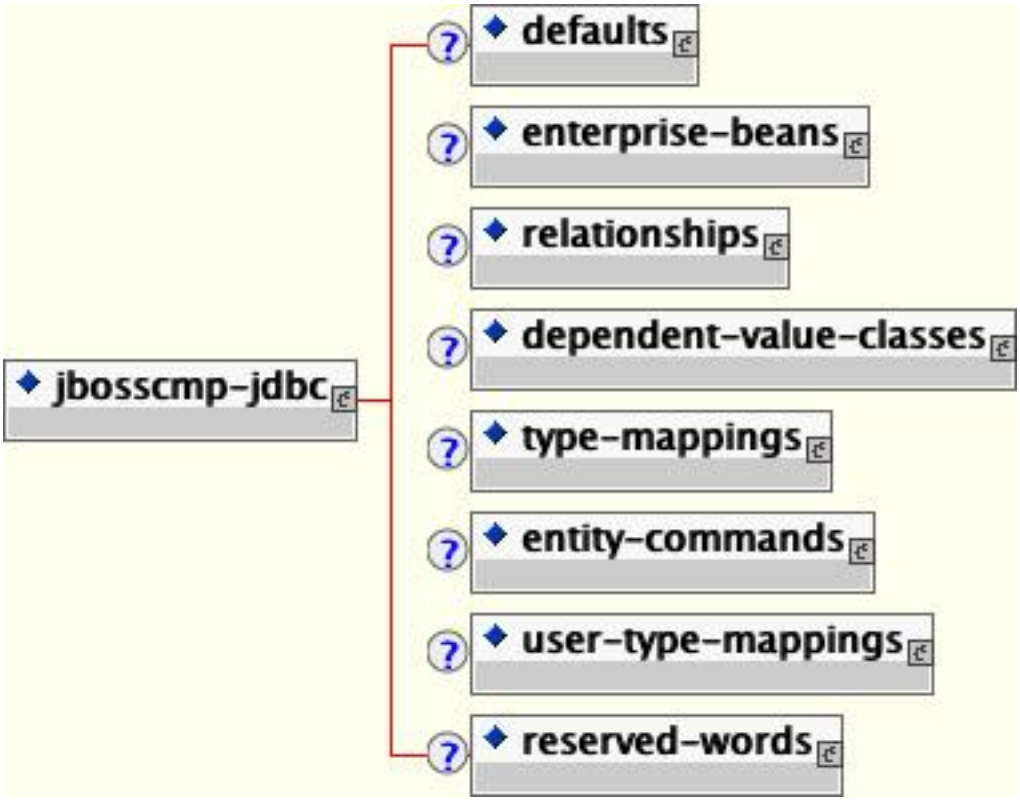


Figure 15.2. The jbosscmp-jdbc content model.

- **defaults**: The defaults section allows for the specification of default behavior/settings for behavior that controls entity beans. Use of this section simplifies the amount of information

needed for the common behaviors found in the entity beans section. See *Section 12, "Defaults"* for the details of the defaults content.

- **enterprise-beans**: The `enterprise-beans` element allows for customization of entity beans defined in the `ejb-jar.xmlenterprise-beans` descriptor. This is described in detail in *Section 3, "Entity Beans"*.
- **relationships**: The `relationships` element allows for the customization of tables and the loading behavior of entity relationships. This is described in detail in *Section 5, "Container Managed Relationships"*.
- **dependent-value-classes**: The `dependent-value-classes` element allows for the customization of the mapping of dependent value classes to tables. Dependent value classes are described in detail in *Section 4.5, "Dependent Value Classes (DVCs)"* (DVCs).
- **type-mappings**: The `type-mappings` element defines the Java to SQL type mappings for a database, along with SQL templates, and function mappings. This is described in detail in *Section 13, "Datasource Customization"*.
- **entity-commands**: The `entity-commands` element allows for the definition of the entity creation command instances that know how to create an entity instance in a persistent store. This is described in detail in *Section 11, "Entity Commands and Primary Key Generation"*.
- **user-type-mappings**: The `user-type-mappings` elements defines a mapping of a user types to a column using a mapper class. A mapper is like a mediator. When storing, it takes an instance of the user type and translates it to a column value. When loading, it takes a column value and translates it to an instance of the user type. Details of the user type mappings are described in *Section 13.4, "User Type Mappings"*.
- **reserved-words**: The `reserved-words` element defines one or more reserved words that should be escaped when generating tables. Each reserved word is specified as the content of a `word` element.

3. Entity Beans

We'll start our look at entity beans in JBoss by examining one of the CMP entity beans in the crime portal. We'll look at the gangster bean, which is implemented as local CMP entity bean. Although JBoss can provide remote entity beans with pass-by-reference semantics for calls in the same VM to get the performance benefit as from local entity beans, the use of local entity beans is strongly encouraged.

We'll start with the required home interface. Since we're only concerned with the CMP fields at this point, we'll show only the methods dealing with the CMP fields.

```
// Gangster Local Home Interface
public interface GangsterHome
    extends EJBLocalHome
{
    Gangster create(Integer id, String name, String nickName)
        throws CreateException;
```

```
    Gangster findByPrimaryKey(Integer id)
        throws FinderException;
}
```

The local interface is what clients will use to talk. Again, it contains only the CMP field accessors.

```
// Gangster Local Interface
public interface Gangster
    extends EJBLocalObject
{
    Integer getGangsterId();

    String getName();

    String getNickName();
    void setNickName(String nickName);

    int getBadness();
    void setBadness(int badness);
}
```

Finally, we have the actual gangster bean. Despite it's size, very little code is actually required. The bulk of the class is the create method.

```
// Gangster Implementation Class
public abstract class GangsterBean
    implements EntityBean
{
     private EntityContext ctx;
     private Category log = Category.getInstance(getClass());
     public Integer ejbCreate(Integer id, String name, String nickName)
         throws CreateException
     {
         log.info("Creating Gangster " + id + " '" + nickName + "' "+ name);
         setGangsterId(id);
         setName(name);
         setNickName(nickName);
         return null;
     }

     public void ejbPostCreate(Integer id, String name, String nickName) {
     }

     // CMP field accessors --------------------------------------------
     public abstract Integer getGangsterId();
     public abstract void setGangsterId(Integer gangsterId);
     public abstract String getName();
     public abstract void setName(String name);
     public abstract String getNickName();
     public abstract void setNickName(String nickName);
     public abstract int getBadness();
     public abstract void setBadness(int badness);
```

```
    public abstract ContactInfo getContactInfo();
    public abstract void setContactInfo(ContactInfo contactInfo);
    //...

    // EJB callbacks ------------------------------------------------
    public void setEntityContext(EntityContext context) { ctx = context; }
    public void unsetEntityContext() { ctx = null; }
    public void ejbActivate() { }
    public void ejbPassivate() { }
    public void ejbRemove() { log.info("Removing " + getName()); }
    public void ejbStore() { }
    public void ejbLoad() { }
}
```

The only thing missing now is the `ejb-jar.xml` deployment descriptor. Although the actual bean class is named `GangsterBean`, we've called the entity `GangsterEJB`.

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar
xmlns="http://java.sun.com/xml/ns/"Whats_new_in_JBoss_4-J2EE_Certification_and_Standards_Com
version="2.1"
    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/ejb-jar_\2_1.xsd">
   <display-name>Crime Portal</display-name>

    <enterprise-beans>
        <entity>
            <display-name>Gangster Entity Bean</display-name>
            <ejb-name>GangsterEJB</ejb-name>
            <local-home>org.jboss.cmp2.crimeportal.GangsterHome</local-home>
            <local>org.jboss.cmp2.crimeportal.Gangster</local>

            <ejb-class>org.jboss.cmp2.crimeportal.GangsterBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-version>2.x</cmp-version>
            <abstract-schema-name>gangster</abstract-schema-name>

            <cmp-field>
                <field-name>gangsterId</field-name>
            </cmp-field>
            <cmp-field>
                <field-name>name</field-name>
            </cmp-field>
            <cmp-field>
                <field-name>nickName</field-name>
            </cmp-field>
            <cmp-field>
                <field-name>badness</field-name>
            </cmp-field>
            <cmp-field>
                <field-name>contactInfo</field-name>
            </cmp-field>
```

```
            <primkey-field>gangsterId</primkey-field>

            <!-- ... -->
        </entity>
    </enterprise-beans>
</ejb-jar>
```

Note that we've specified a CMP version of `2.x` to indicate that this is EJB 2.x CMP entity bean. The abstract schema name was set to `gangster`. That will be important when we look at EJB-QL queries in *Section 6, "Queries"*.

3.1. Entity Mapping

The JBoss configuration for the entity is declared with an `entity` element in the `jbosscmp-jdbc.xml` file. This file is located in the `META-INF` directory of the EJB JAR and contains all of the optional configuration information for configuring the CMP mapping. The `entity` elements for each entity bean are grouped together in the `enterprise-beans` element under the top level `jbosscmp-jdbc` element. A stubbed out entity configuration is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jbosscmp-jdbc PUBLIC
     "-//JBoss//DTD JBOSSCMP-JDBC 3.2//EN"
     "http://www.jboss.org/j2ee/dtd/jbosscmp-jdbc_3_2.dtd">
<jbosscmp-jdbc>
    <defaults>
        <!-- application-wide CMP defaults -->
    </defaults>
    <enterprise-beans>
        <entity>
            <ejb-name>GangsterEJB</ejb-name>
            <!-- overrides to defaults section -->
            <table-name>gangster</table-name>
            <!-- CMP Fields (see CMP-Fields) -->
            <!-- Load Groups (see Load Groups)-->
            <!-- Queries (see Queries) -->
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

The `ejb-name` element is required to match the entity specification here with the one in the `ejb-jar.xml` file. The remainder of the elements specify either overrides the global or application-wide CMP defaults and CMP mapping details specific to the bean. The application defaults come from the `defaults` section of the `jbosscmp-jdbc.xml` file and the global defaults come from the `defaults` section of the `standardjbosscmp-jdbc.xml` file in the `conf` directory for the current server configuration file set. The `defaults` section is discussed in *Section 12, "Defaults"*. *Figure 15.3, "The entity element content model"* shows the full `entity` content model.

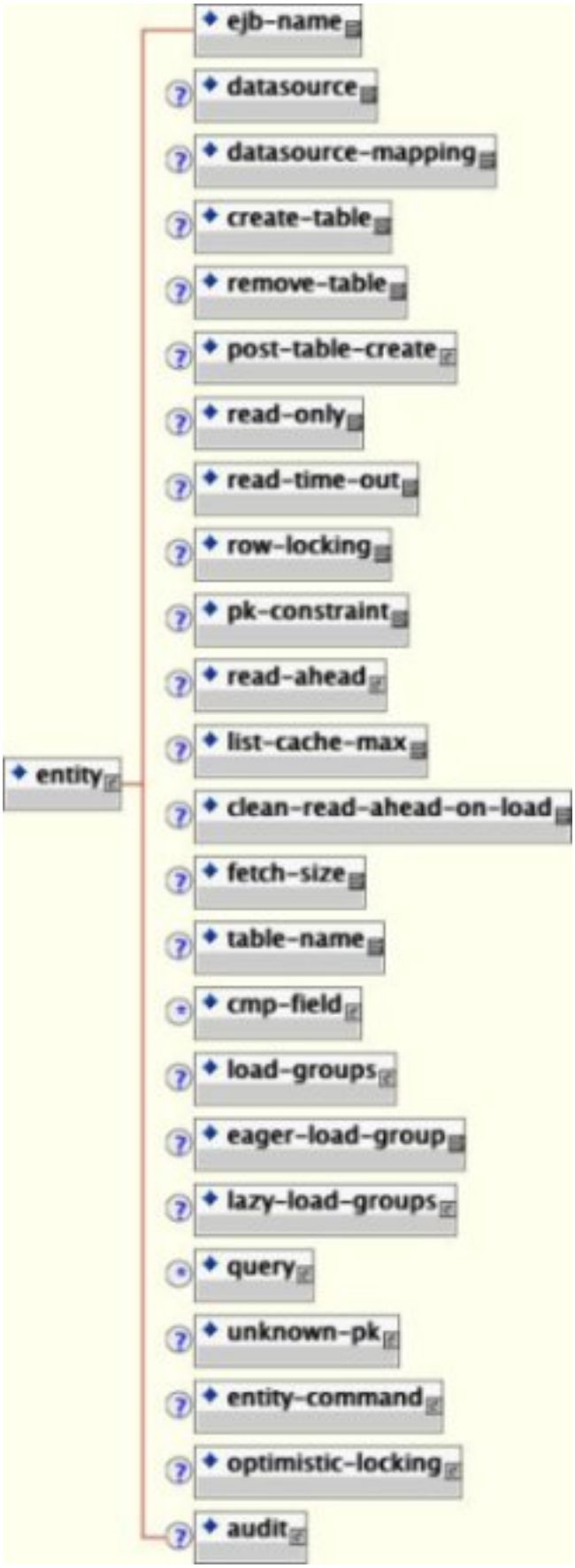

entity
ejb-name
datasource
datasource-mapping
create-table
remove-table
post-table-create
read-only
read-time-out
row-locking
pk-constraint
read-ahead
list-cache-max
clean-read-ahead-on-load
fetch-size
table-name
cmp-field
load-groups
eager-load-group
lazy-load-groups
query
unknown-pk
entity-command
optimistic-locking
audit

Figure 15.3. The entity element content model

A detailed description of each entity element follows:

- **ejb-name**: This required element is the name of the EJB to which this configuration applies. This element must match an `ejb-name` of an entity in the `ejb-jar.xml` file.
- **datasource**: This optional element is the `jndi-name` used to look up the datasource. All database connections used by an entity or relation-table are obtained from the datasource. Having different datasources for entities is not recommended, as it vastly constrains the domain over which finders and ejbSelects can query. The default is `java:/DefaultDS` unless overridden in the defaults section.
- **datasource-mapping**: This optional element specifies the name of the `type-mapping`, which determines how Java types are mapped to SQL types, and how EJB-QL functions are mapped to database specific functions. Type mappings are discussed in *Section 13.3, "Mapping"*. The default is `Hypersonic SQL` unless overridden in the defaults section.
- **create-table**: This optional element when true, specifies that JBoss should attempt to create a table for the entity. When the application is deployed, JBoss checks if a table already exists before creating the table. If a table is found, it is logged, and the table is not created. This option is very useful during the early stages of development when the table structure changes often. The default is false unless overridden in the defaults section.
- **alter-table**: If `create-table` is used to automatically create the schema, `alter-table` can be used to keep the schema current with changes to the entity bean. Alter table will perform the following specific tasks:
  - new fields will be created
  - fields which are no longer used will be removed
  - string fields which are shorter than the declared length will have their length increased to the declared length. (not supported by all databases)
- **remove-table**: This optional element when true, JBoss will attempt to drop the table for each entity and each relation table mapped relationship. When the application is undeployed, JBoss will attempt to drop the table. This option is very useful during the early stages of development when the table structure changes often. The default is false unless overridden in the defaults section.
- **post-table-create**: This optional element specifies an arbitrary SQL statement that should be executed immediately after the database table is created. This command is only executed if `create-table` is true and the table did not previously exist.
- **read-only**: This optional element when true specifies that the bean provider will not be allowed to change the value of any fields. A field that is read-only will not be stored in, or inserted into, the database. If a primary key field is read-only, the create method will throw a `CreateException`. If a set accessor is called on a read-only field, it throws an `EJBException`.

Read-only fields are useful for fields that are filled in by database triggers, such as last update. The `read-only` option can be overridden on a per `cmp-field` basis, and is discussed in *Section 4.3, "Read-only Fields"*. The default is false unless overridden in the `defaults` section.

- **read-time-out**: This optional element is the amount of time in milliseconds that a read on a read-only field is valid. A value of 0 means that the value is always reloaded at the start of a transaction, and a value of -1 means that the value never times out. This option can also be overridden on a per `cmp-field` basis. If `read-only` is false, this value is ignored. The default is -1 unless overridden in the `defaults` section.
- **row-locking**: This optional element if true specifies that JBoss will lock all rows loaded in a transaction. Most databases implement this by using the `SELECT FOR UPDATE` syntax when loading the entity, but the actual syntax is determined by the `row-locking-template` in the datasource-mapping used by this entity. The default is false unless overridden in the `defaults` section.
- **pk-constraint**: This optional element if true specifies that JBoss will add a primary key constraint when creating tables. The default is true unless overridden in the defaults section.
- **read-ahead**: This optional element controls caching of query results and `cmr-fields` for the entity. This option is discussed in *Section 7.3, "Read-ahead"*.
- **fetch-size**: This optional element specifies the number of entities to read in one round-trip to the underlying datastore. The default is 0 unless overridden in the defaults section.
- **list-cache-max**: This optional element specifies the number of read-lists that can be tracked by this entity. This option is discussed in `on-load`. The default is 1000 unless overridden in the defaults section.
- **clean-read-ahead-on-load**: When an entity is loaded from the read ahead cache, JBoss can remove the data used from the read ahead cache. The default is `false`.
- **table-name**: This optional element is the name of the table that will hold data for this entity. Each entity instance will be stored in one row of this table. The default is the `ejb-name`.
- **cmp-field**: The optional element allows one to define how the `ejb-jar.xmlcmp-field` is mapped onto the persistence store. This is discussed in *Section 4, "CMP Fields"*.
- **load-groups**: This optional element specifies one or more groupings of CMP fields to declare load groupings of fields. This is discussed in *Section 7.2, "Load Groups"*.
- **eager-load-groups**: This optional element defines one or more load grouping as eager load groups. This is discussed in *Section 8.2, "Eager-loading Process"*.
- **lazy-load-groups**: This optional element defines one or more load grouping as lazy load groups. This is discussed in *Section 8.3, "Lazy loading Process"*.
- **query**: This optional element specifies the definition of finders and selectors. This is discussed in *Section 6, "Queries"*.

- **unknown-pk**: This optional element allows one to define how an unknown primary key type of `java.lang.Object` maps to the persistent store.
- **entity-command**: This optional element allows one to define the entity creation command instance. Typically this is used to define a custom command instance to allow for primary key generation. This is described in detail in *Section 11, "Entity Commands and Primary Key Generation"*.
- **optimistic-locking**: This optional element defines the strategy to use for optimistic locking. This is described in detail in *Section 10, "Optimistic Locking"*.
- **audit**: This optional element defines the CMP fields that will be audited. This is described in detail in *Section 4.4, "Auditing Entity Access"*.

4. CMP Fields

CMP fields are declared on the bean class as abstract getter and setter methods that follow the JavaBean property accessor conventions. Our gangster bean, for example, has a `getName()` and a `setName()` method for accessing the `name` CMP field. In this section we will look at how the configure these declared CMP fields and control the persistence and behavior.

4.1. CMP Field Declaration

The declaration of a CMP field starts in the `ejb-jar.xml` file. On the gangster bean, for example, the `gangsterId`, `name`, `nickName` and `badness` would be declared in the `ejb-jar.xml` file as follows:

```
<ejb-jar>
  <enterprise-beans>
    <entity>
        <ejb-name>GangsterEJB</ejb-name>
        <cmp-field><field-name>gangsterId</field-name></cmp-field>
        <cmp-field><field-name>name</field-name></cmp-field>
        <cmp-field><field-name>nickName</field-name></cmp-field>
        <cmp-field><field-name>badness</field-name></cmp-field>
    </entity>
  </enterprise-beans>
           </ejb-jar>
```

Note that the J2EE deployment descriptor doesn't declare any object-relational mapping details or other configuration. It is nothing more than a simple declaration of the CMP fields.

4.2. CMP Field Column Mapping

The relational mapping configuration of a CMP field is done in the `jbosscmp-jdbc.xml` file. The structure is similar to the `ejb-jar.xml` with an entity `element` that has `cmp-field` elements under it with the additional configuration details.

The following is shows the basic column name and data type mappings for the gangster bean.

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <table-name>gangster</table-name>

      <cmp-field>
        <field-name>gangsterId</field-name>
        <column-name>id</column-name>
      </cmp-field>
      <cmp-field>
        <field-name>name</field-name>
        <column-name>name</column-name>
        <not-null/>
      </cmp-field>
      <cmp-field>
        <field-name>nickName</field-name>
        <column-name>nick_name</column-name>
        <jdbc-type>VARCHAR</jdbc-type>
        <sql-type>VARCHAR(64)</sql-type>
      </cmp-field>
      <cmp-field>
        <field-name>badness</field-name>
        <column-name>badness</column-name>
      </cmp-field>
    </entity>
  </enterprise-beans>
           </jbosscmp-jdbc>
```

The full content model of the `cmp-field` element of the `jbosscmp-jdbc.xml` is shown below.

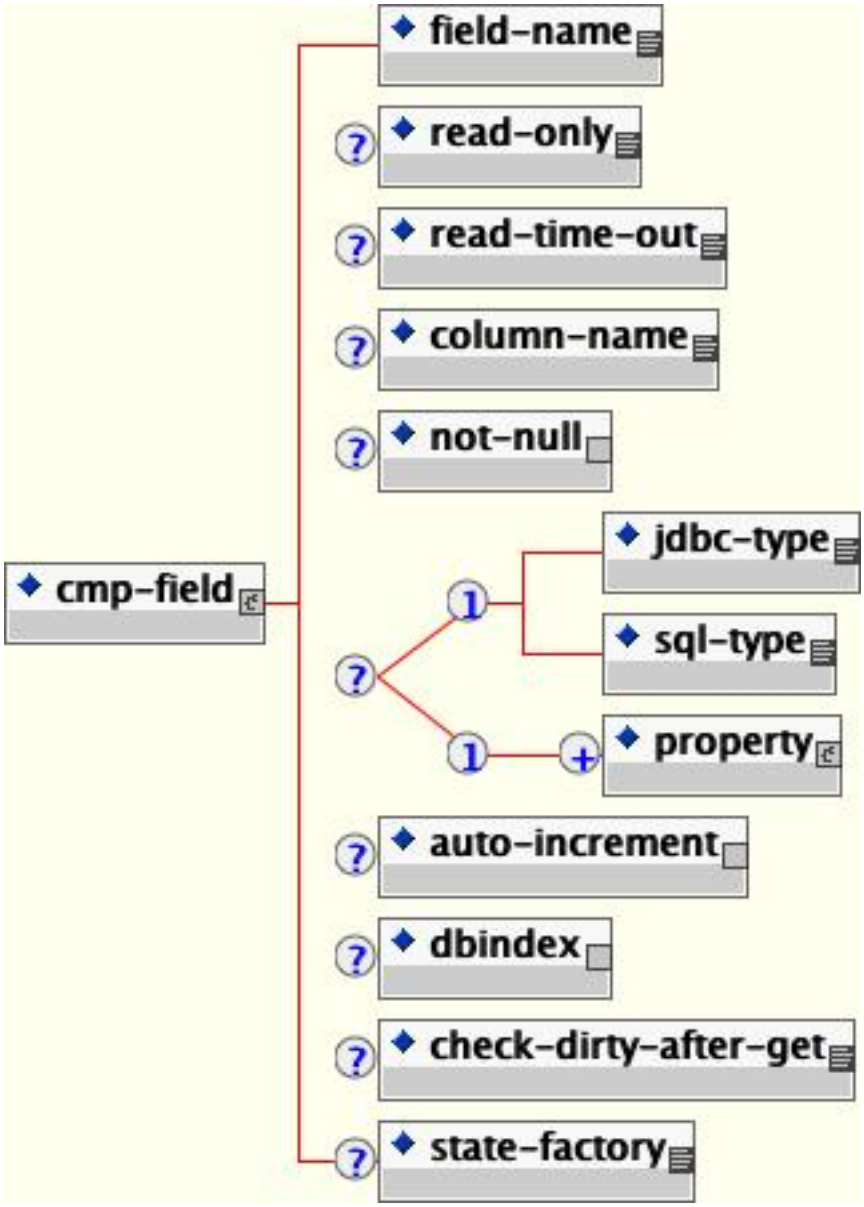


Figure 15.4. The JBoss entity element content model

A detailed description of each element follows:

- **field-name**: This required element is the name of the `cmp-field` that is being configured. It must match the `field-name` element of a `cmp-field` declared for this entity in the `ejb-jar.xml` file.
- **read-only**: This declares that field in question is read-only. This field will not be written to the database by JBoss. Read-only fields are discussed in *Section 4.3, "Read-only Fields"*.
- **read-only-timeout**: This is the time in milliseconds that a read-only field value will be

considered valid.

- **column-name**: This optional element is the name of the column to which the `cmp-field` is mapped. The default is to use the `field-name` value.
- **not-null**: This optional element indicates that JBoss should add a NOT NULL to the end of the column declaration when automatically creating the table for this entity. The default for primary key fields and primitives is not null.
- **jdbc-type**: This is the JDBC type that is used when setting parameters in a JDBC prepared statement or loading data from a JDBC result set. The valid types are defined in `java.sql.Types`. This is only required if `sql-type` is specified. The default JDBC type will be based on the database type in the `datasourcemapping`.
- **sql-type**: This is the SQL type that is used in create table statements for this field. Valid SQL types are only limited by your database vendor. This is only required if `jdbc-type` is specified. The default SQL type will be base on the database type in the `datasourcemapping`
- **property**: This optional element allows one to define how the properties of a dependent value class CMP field should be mapped to the persistent store. This is discussed further in *Section 4.5, "Dependent Value Classes (DVCs)"*.
- **auto-increment**: The presence of this optional field indicates that it is automatically incremented by the database layer. This is used to map a field to a generated column as well as to an externally manipulated column.
- **dbindex**: The presence of this optional field indicates that the server should create an index on the corresponding column in the database. The index name will be fieldname_index.
- **check-dirty-after-get**: This value defaults to false for primitive types and the basic java.lang immutable wrappers (`Integer`, `String`, etc...). For potentially mutable objects, JBoss will mark they field as potentially dirty after a get operation. If the dirty check on an object is too expensive, you can optimize it away by setting `check-dirty-after-get` to false.
- **state-factory**: This specifies class name of a state factory object which can perform dirty checking for this field. State factory classes must implement the `CMPFieldStateFactory` interface.

4.3. Read-only Fields

JBoss allows for read-only CMP fields by setting the `read-only` and `read-time-out` elements in the `cmp-field` declaration. These elements work the same way as they do at the entity level. If a field is read-only, it will never be used in an `INSERT` or `UPDATE` statement. If a primary key field is `read-only`, the create method will throw a `CreateException`. If a set accessor is called for a read-only field, it throws an `EJBException`. Read-only fields are useful for fields that are filled in by database triggers, such as last update. A read-only CMP field declaration example follows:

```
<jbosscmp-jdbc>
```

```
    <enterprise-beans>
        <entity>
            <ejb-name>GangsterEJB</ejb-name>
            <cmp-field>
                <field-name>lastUpdated</field-name>
                <read-only>true</read-only>
                <read-time-out>1000</read-time-out>
            </cmp-field>
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

4.4. Auditing Entity Access

The `audit` element of the entity section allows one to specify how access to and entity bean is audited. This is only allowed when an entity bean is accessed under a security domain so that this is a caller identity established. The content model of the audit element is given *Figure 15.5, "The jbosscmp-jdbc.xml audit element content model"*.

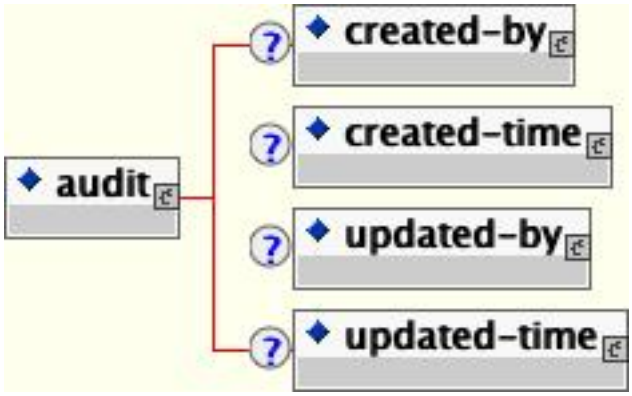


Figure 15.5. The jbosscmp-jdbc.xml audit element content model

- **created-by**: This optional element indicates that the caller who created the entity should be saved to either the indicated `column-name` or cmp `field-name`.
- **created-time**: This optional element indicates that the time of entity creation should be saved to either the indicated `column-name` or cmp `field-name`.
- **updated-by**: This optional element indicates that the caller who last modified the entity should be saved to either the indicated `column-name` or CMP `field-name`.
- **updated-time**: This optional element indicates that the last time of entity modification should be saved to either the indicated `column-name` or CMP `field-name`.

For each element, if a `field-name` is given, the corresponding audit information should be

stored in the specified CMP field of the entity bean being accessed. Note that there does not have to be an corresponding CMP field declared on the entity. In case there are matching field names, you will be able to access audit fields in the application using the corresponding CMP field abstract getters and setters. Otherwise, the audit fields will be created and added to the entity internally. You will be able to access audit information in EJB-QL queries using the audit field names, but not directly through the entity accessors.

If, on the other hand, a `column-name` is specified, the corresponding audit information should be stored in the indicated column of the entity table. If JBoss is creating the table the `jdbc-type` and `sql-type` element can then be used to define the storage type.

The declaration of audit information with given column names is shown below.

```
<jbosscmp-jdbc>
    <enterprise-beans>
        <entity>
            <ejb-name>AuditChangedNamesEJB</ejb-name>
            <table-name>cmp2_audit_changednames</table-name>
            <audit>
                <created-by>
                    <column-name>createdby</column-name>
                </created-by>
                <created-time>
                    <column-name>createdtime</column-name>
                </created-time>
                <updated-by>
                    <column-name>updatedby</column-name></updated-by>
                <updated-time>
                    <column-name>updatedtime</column-name>
                </updated-time>
            </audit>
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

4.5. Dependent Value Classes (DVCs)

A dependent value class (DVC) is a fancy term used to identity any Java class that is the type of a `cmp-field` other than the automatically recognized types core types such as strings and number values. By default, a DVC is serialized, and the serialized form is stored in a single database column. Although not discussed here, there are several known issues with the long-term storage of classes in serialized form.

JBoss also supports the storage of the internal data of a DVC into one or more columns. This is useful for supporting legacy JavaBeans and database structures. It is not uncommon to find a database with a highly flattened structure (e.g., a `PURCHASE_ORDER` table with the fields `SHIP_LINE1`, `SHIP_LINE2`, `SHIP_CITY`, etc. and an additional set of fields for the billing address). Other common database structures include telephone numbers with separate fields for area code, exchange, and extension, or a person's name spread across several fields. With a DVC, multiple columns can be mapped to one logical field.

JBoss requires that a DVC to be mapped must follow the JavaBeans naming specification for simple properties, and that each property to be stored in the database must have both a getter and a setter method. Furthermore, the bean must be serializable and must have a no argument constructor. A property can be any simple type, an unmapped DVC or a mapped DVC, but cannot be an EJB. A DVC mapping is specified in a `dependent-value-class` element within the `dependent-value-classes` element.

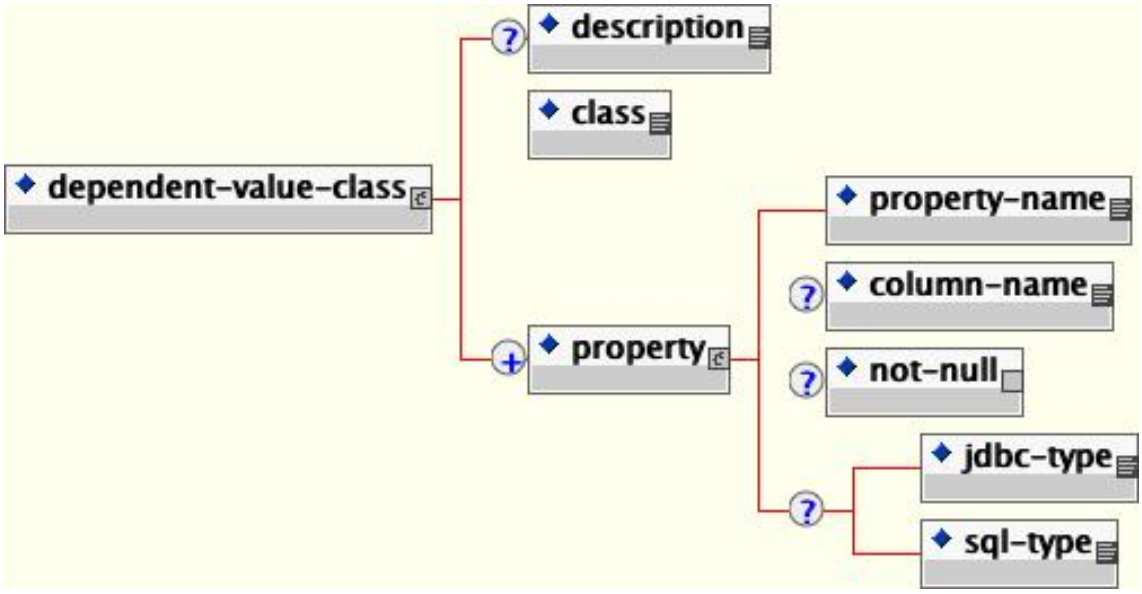


Figure 15.6. The jbosscmp-jdbc dependent-value-class element model.

Here is an example of a simple `ContactInfo` DVC class.

```
public class ContactInfo
    implements Serializable
{
    /** The cell phone number. */
    private PhoneNumber cell;

    /** The pager number. */
    private PhoneNumber pager;

    /** The email address */
    private String email;


    /**
     * Creates empty contact info.
     */
    public ContactInfo() {
    }

    public PhoneNumber getCell() {
        return cell;
    }
```

```
    public void setCell(PhoneNumber cell) {
        this.cell = cell;
    }

    public PhoneNumber getPager() {
        return pager;
    }

    public void setPager(PhoneNumber pager) {
      this.pager = pager;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email.toLowerCase();
    }

    // ... equals, hashCode, toString
}
```

The contact info includes a phone number, which is represented by another DVC class.

```
public class PhoneNumber
    implements Serializable
{
    /** The first three digits of the phone number. */
    private short areaCode;

    /** The middle three digits of the phone number. */
        private short exchange;

    /** The last four digits of the phone number. */
        private short extension;

    // ... getters and setters

    // ... equals, hashCode, toString
}
```

The DVC mappings for these two classes are relatively straight forward.

```
<dependent-value-classes>
    <dependent-value-class>
        <description>A phone number</description>
        <class>org.jboss.cmp2.crimeportal.PhoneNumber</class>
        <property>
            <property-name>areaCode</property-name>
            <column-name>area_code</column-name>
        </property>
```

```
        <property>
            <property-name>exchange</property-name>
            <column-name>exchange</column-name>
        </property>
        <property>
            <property-name>extension</property-name>
            <column-name>extension</column-name>
        </property>
    </dependent-value-class>

    <dependent-value-class>
        <description>General contact info</description>
        <class>org.jboss.cmp2.crimeportal.ContactInfo</class>
        <property>
            <property-name>cell</property-name>
            <column-name>cell</column-name>
        </property>
        <property>
            <property-name>pager</property-name>
            <column-name>pager</column-name>
        </property>
        <property>
            <property-name>email</property-name>
            <column-name>email</column-name>
            <jdbc-type>VARCHAR</jdbc-type>
            <sql-type>VARCHAR(128)</sql-type>
        </property>
    </dependent-value-class>
</dependent-value-classes>
```

Each DVC is declared with a `dependent-value-class` element. A DVC is identified by the Java class type declared in the class element. Each property to be persisted is declared with a property element. This specification is based on the `cmp-field` element, so it should be self-explanatory. This restriction will also be removed in a future release. The current proposal involves storing the primary key fields in the case of a local entity and the entity handle in the case of a remote entity.

The `dependent-value-classes` section defines the internal structure and default mapping of the classes. When JBoss encounters a field that has an unknown type, it searches the list of registered DVCs, and if a DVC is found, it persists this field into a set of columns, otherwise the field is stored in serialized form in a single column. JBoss does not support inheritance of DVCs; therefore, this search is only based on the declared type of the field. A DVC can be constructed from other DVCs, so when JBoss runs into a DVC, it flattens the DVC tree structure into a set of columns. If JBoss finds a DVC circuit during startup, it will throw an `EJBException`. The default column name of a property is the column name of the base `cmp-field` followed by an underscore and then the column name of the property. If the property is a DVC, the process is repeated. For example, a `cmp-field` named `info` that uses the `ContactInfo` DVC would have the following columns:

```
info_cell_area_code
info_cell_exchange
```

```
info_cell_extension
info_pager_area_code
info_pager_exchange
info_pager_extension
info_email
```

The automatically generated column names can quickly become excessively long and awkward. The default mappings of columns can be overridden in the entity element as follows:

```
<jbosscmp-jdbc>
    <enterprise-beans>
        <entity>
            <ejb-name>GangsterEJB</ejb-name>
            <cmp-field>
                <field-name>contactInfo</field-name>
                <property>
                    <property-name>cell.areaCode</property-name>
                    <column-name>cell_area</column-name>
                </property>
                <property>
                    <property-name>cell.exchange</property-name>
                    <column-name>cell_exch</column-name>
                </property>
                <property>
                    <property-name>cell.extension</property-name>
                    <column-name>cell_ext</column-name>
                </property>

                <property>
                    <property-name>pager.areaCode</property-name>
                    <column-name>page_area</column-name>
                </property>
                <property>
                    <property-name>pager.exchange</property-name>
                    <column-name>page_exch</column-name>
                </property>
                <property>
                    <property-name>pager.extension</property-name>
                    <column-name>page_ext</column-name>
                </property>

                <property>
                    <property-name>email</property-name>
                    <column-name>email</column-name>
                    <jdbc-type>VARCHAR</jdbc-type>
                    <sql-type>VARCHAR(128)</sql-type>
                </property>
            </cmp-field>
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

When overriding property info for the entity, you need to refer to the property from a flat

perspective as in `cell.areaCode`.

5. Container Managed Relationships

Container Managed Relationships (CMRs) are a powerful new feature of CMP 2.0. Programmers have been creating relationships between entity objects since EJB 1.0 was introduced (not to mention since the introduction of databases), but before CMP 2.0 the programmer had to write a lot of code for each relationship in order to extract the primary key of the related entity and store it in a pseudo foreign key field. The simplest relationships were tedious to code, and complex relationships with referential integrity required many hours to code. With CMP 2.0 there is no need to code relationships by hand. The container can manage one-to-one, one-to-many and many-to-many relationships, with referential integrity. One restriction with CMRs is that they are only defined between local interfaces. This means that a relationship cannot be created between two entities in separate applications, even in the same application server.

There are two basic steps to create a container managed relationship: create the `cmr-field` abstract accessors and declare the relationship in the `ejb-jar.xml` file. The following two sections describe these steps.

5.1. CMR-Field Abstract Accessors

CMR-Field abstract accessors have the same signatures as `cmp-fields`, except that single-valued relationships must return the local interface of the related entity, and multi-valued relationships can only return a `java.util.Collection` (or `java.util.Set`) object. For example, to declare a one-to-many relationship between organization and gangster, we declare the relationship from organization to gangster in the `OrganizationBean` class:

```
public abstract class OrganizationBean
    implements EntityBean
{
    public abstract Set getMemberGangsters();
    public abstract void setMemberGangsters(Set gangsters);
}
```

We also can declare the relationship from gangster to organization in the `GangsterBean` class:

```
public abstract class GangsterBean
    implements EntityBean
{
    public abstract Organization getOrganization();
    public abstract void setOrganization(Organization org);
}
```

Although each bean declared a CMR field, only one of the two beans in a relationship must have a set of accessors. As with CMP fields, a CMR field is required to have both a getter and a setter method.

5.2. Relationship Declaration

The declaration of relationships in the `ejb-jar.xml` file is complicated and error prone. Although we recommend using a tool like XDoclet to manage the deployment descriptors for CMR fields, it's still important to understand how the descriptor works. The following illustrates the declaration of the organization/gangster relationship:

```
<ejb-jar>
    <relationships>
        <ejb-relation>
            <ejb-relation-name>Organization-Gangster</ejb-relation-name>
            <ejb-relationship-role>
                <ejb-relationship-role-name>org-has-gangsters
</ejb-relationship-role-name>
                <multiplicity>One</multiplicity>
                <relationship-role-source>
                    <ejb-name>OrganizationEJB</ejb-name>
                </relationship-role-source>
                <cmr-field>
                    <cmr-field-name>memberGangsters</cmr-field-name>
                    <cmr-field-type>java.util.Set</cmr-field-type>
                </cmr-field>
            </ejb-relationship-role>
            <ejb-relationship-role>
                <ejb-relationship-role-name>
                    gangster-belongs-to-org
                </ejb-relationship-role-name>
                <multiplicity>Many</multiplicity>
                <cascade-delete/>
                <relationship-role-source>
                    <ejb-name>GangsterEJB</ejb-name>
                </relationship-role-source>
                <cmr-field>
                    <cmr-field-name>organization</cmr-field-name>
                </cmr-field>
            </ejb-relationship-role>
        </ejb-relation>
    </relationships>
</ejb-jar>
```

As you can see, each relationship is declared with an `ejb-relation` element within the top level `relationships` element. The relation is given a name in the `ejb-relation-name` element. This is important because we will need to refer to the role by name in the `jbosscmp-jdbc.xml` file. Each `ejb-relation` contains two `ejb-relationship-role` elements (one for each side of the relationship). The `ejb-relationship-role` tags are as follows:

- **ejb-relationshiprole-name**: This optional element is used to identify the role and match the database mapping the `jbosscmp-jdbc.xml` file. The relationship role names for each side of a relationship must be different.
- **multiplicity**: This indicates the multiplicity of this side of the relationship. The valid values are

`One` or `Many`. In this example, the multiplicity of the organization is `One` and the multiplicity of the gangster is `Many` because the relationship is from one organization to many gangsters. Note, as with all XML elements, this element is case sensitive.

- **cascade-delete**: When this optional element is present, JBoss will delete the child entity when the parent entity is deleted. Cascade deletion is only allowed for a role where the other side of the relationship has a multiplicity of one. The default is to not cascade delete.
- **relationship-role-source**
  - **ejb-name**: This required element gives the name of the entity that has the role.
- **cmr-field**
  - **cmr-field-name**: This is the name of the CMR field of the entity has one, if it has one.
  - **cmr-field-type**: This is the type of the CMR field, if the field is a collection type. It must be `java.util.Collection` or `java.util.Set`.

After adding the CMR field abstract accessors and declaring the relationship, the relationship should be functional. The next section discusses the database mapping of the relationship.

5.3. Relationship Mapping

Relationships can be mapped using either a foreign key or a separate relation table. One-to-one and one-to-many relationships use the foreign key mapping style by default, and many-to-many relationships use only the relation table mapping style. The mapping of a relationship is declared in the `relationships` section of the `jbosscmp-jdbc.xml` descriptor via `ejb-relation` elements. Relationships are identified by the `ejb-relation-name` from the `ejb-jar.xml` file. The `jbosscmp-jdbc.xmlejb-relation` element content model is shown in *Figure 15.7, “The jbosscmp-jdbc.xml ejb-relation element content model”*.

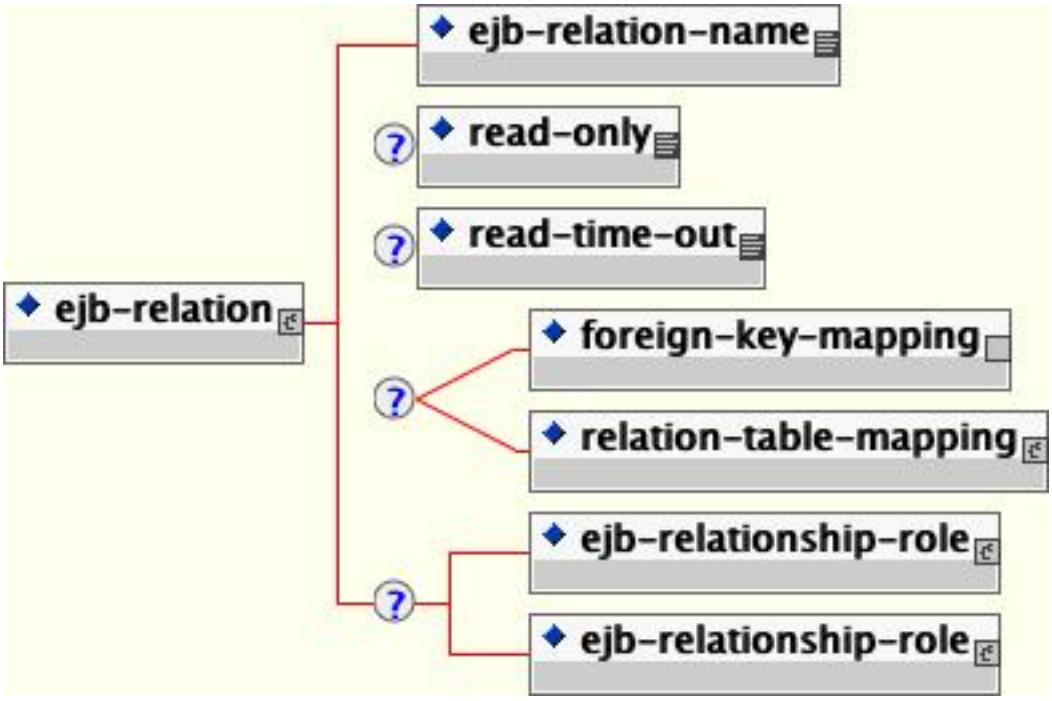


Figure 15.7. The jbosscmp-jdbc.xml ejb-relation element content model

The basic template of the relationship mapping declaration for `Organization-Gangster` relationship follows:

```
<jbosscmp-jdbc>
    <relationships>
        <ejb-relation>
            <ejb-relation-name>Organization-Gangster</ejb-relation-name>
            <foreign-key-mapping/>
            <ejb-relationship-role>
<ejb-relationship-role-name>org-has-gangsters</ejb-relationship-role-name>
                <key-fields>
                    <key-field>
                        <field-name>name</field-name>
                        <column-name>organization</column-name>
                    </key-field>
                </key-fields>
            </ejb-relationship-role>
            <ejb-relationship-role>
<ejb-relationship-role-name>gangster-belongs-to-org</ejb-relationship-role-name>
                <key-fields/>
            </ejb-relationship-role>
        </ejb-relation>
    </relationships>
</jbosscmp-jdbc>
```

After the `ejb-relation-name` of the relationship being mapped is declared, the relationship can be declared as read only using the `read-only` and `read-time-out` elements. They have the

same semantics as their counterparts in the entity element.

The `ejb-relation` element must contain either a `foreign-key-mapping` element or a `relation-table-mapping` element, which are described in *Section 5.3.2, "Foreign Key Mapping"* and *Section 5.3.3, "Relation table Mapping"*. This element may also contain a pair of `ejb-relationship-role` elements as described in the following section.

5.3.1. Relationship Role Mapping

Each of the two `ejb-relationship-role` elements contains mapping information specific to an entity in the relationship. The content model of the `ejb-relationship-role` element is shown in *Figure 15.8, "The jbosscmp-jdbc ejb-relationship-role element content model"* .

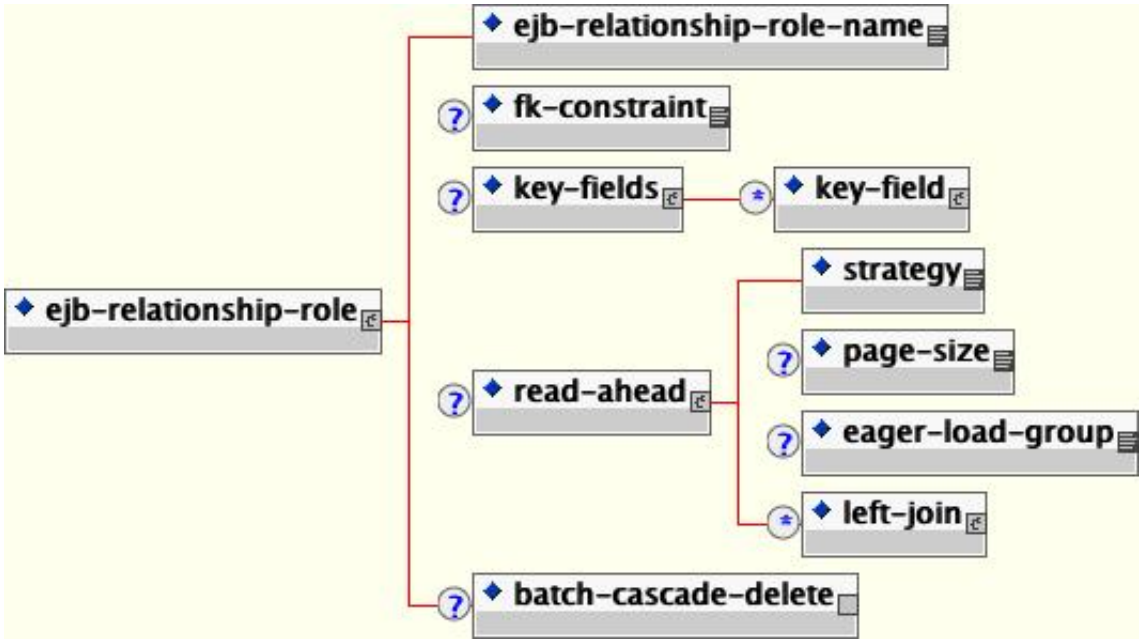


Figure 15.8. The jbosscmp-jdbc ejb-relationship-role element content model

A detailed description of the main elements follows:

- **ejb-relationship-role-name**: This required element gives the name of the role to which this configuration applies. It must match the name of one of the roles declared for this relationship in the `ejb-jar.xml` file.
- **fk-constraint**: This optional element is a true/false value that indicates whether JBoss should add a foreign key constraint to the tables for this side of the relationship. JBoss will only add generate the constraint if both the primary table and the related table were created by JBoss during deployment.
- **key-fields**: This optional element specifies the mapping of the primary key fields of the

current entity, whether it is mapped in the relation table or in the related object. The `key-fields` element must contain a `key-field` element for each primary key field of the current entity. The `key-fields` element can be empty if no foreign key mapping is needed for this side of the relation. An example of this would be the many side of a one-to-many relationship. The details of this element are described below.

- **read-ahead**: This optional element controls the caching of this relationship. This option is discussed in *Section 8.3.1, "Relationships"*.
- **batch-cascade-delete**: This indicates that a cascade delete on this relationship should be performed with a single SQL statement. This requires that the relationship be marked as `batch-delete` in the `ejb-jar.xml`.

As noted above, the `key-fields` element contains a `key-field` for each primary key field of the current entity. The `key-field` element uses the same syntax as the `cmp-field` element of the entity, except that `key-field` does not support the `not-null` option. Key fields of a `relation-table` are automatically not null, because they are the primary key of the table. On the other hand, foreign key fields must be nullable by default. This is because the CMP specification requires an insert into the database after the `ejbCreate` method and an update to it after to pick up CMR changes made in `ejbPostCreate`. Since the EJB specification does not allow a relationship to be modified until `ejbPostCreate`, a foreign key will be initially set to null. There is a similar problem with removal. You can change this insert behavior using the `jboss.xmlinsert-after-ejb-post-create` container configuration flag. The following example illustrates the creation of a new bean configuration that uses `insert-after-ejb-post-create` by default.

```
<jboss>
    <!-- ... -->
    <container-configurations>
        <container-configuration extends="Standard CMP 2.x EntityBean">
            <container-name>INSERT after ejbPostCreate
Container</container-name>
            <insert-after-ejb-post-create>true</insert-after-ejb-post-create>
        </container-configuration>
    </container-configurations>
</jboss>
```

An alternate means of working around the non-null foreign key issue is to map the foreign key elements onto non-null CMP fields. In this case you simply populate the foreign key fields in `ejbCreate` using the associated CMP field setters.

The content model of the key-fields element is *Figure 15.9, "The jbosscmp-jdbc key-fields element content model"*.

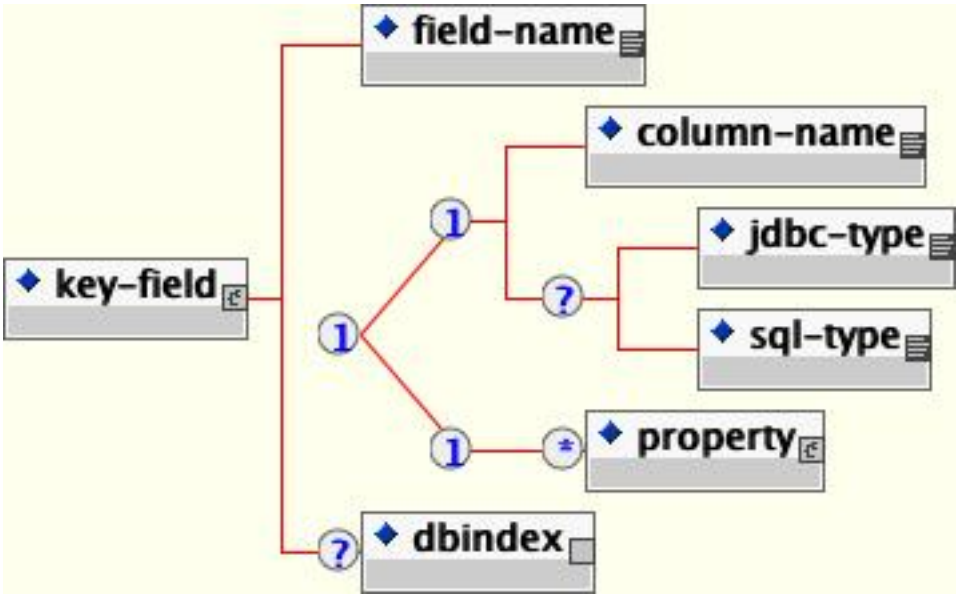


Figure 15.9. The jbosscmp-jdbc key-fields element content model

A detailed description of the elements contained in the `key-field` element follows:

- **field-name**: This required element identifies the field to which this mapping applies. This name must match a primary key field of the current entity.
- **column-name**: Use this element to specify the column name in which this primary key field will be stored. If this is relationship uses `foreign-key-mapping`, this column will be added to the table for the related entity. If this relationship uses `relation-table-mapping`, this column is added to the `relation-table`. This element is not allowed for mapped dependent value class; instead use the property element.
- **jdbc-type**: This is the JDBC type that is used when setting parameters in a JDBC `PreparedStatement` or loading data from a JDBC ResultSet. The valid types are defined in `java.sql.Types`.
- **sql-type**: This is the SQL type that is used in create table statements for this field. Valid types are only limited by your database vendor.
- **property**: Use this element for to specify the mapping of a primary key field which is a dependent value class.
- **dbindex**: The presence of this optional field indicates that the server should create an index on the corresponding column in the database, and the index name will be `fieldname_index`.

5.3.2. Foreign Key Mapping

Foreign key mapping is the most common mapping style for one-to-one and one-to-many

relationships, but is not allowed for many-to many relationships. The foreign key mapping element is simply declared by adding an empty foreign `key-mapping` element to the `ejb-relation` element.

As noted in the previous section, with a foreign key mapping the `key-fields` declared in the `ejb-relationship-role` are added to the table of the related entity. If the `key-fields` element is empty, a foreign key will not be created for the entity. In a one-to-many relationship, the many side (`Gangster` in the example) must have an empty `key-fields` element, and the one side (`Organization` in the example) must have a `key-fields` mapping. In one-to-one relationships, one or both roles can have foreign keys.

The foreign key mapping is not dependent on the direction of the relationship. This means that in a one-to-one unidirectional relationship (only one side has an accessor) one or both roles can still have foreign keys. The complete foreign key mapping for the `Organization-Gangster` relationship is shown below with the foreign key elements highlighted in bold:

```
<jbosscmp-jdbc>
    <relationships>
        <ejb-relation>
            <ejb-relation-name>Organization-Gangster</ejb-relation-name>
            <foreign-key-mapping/>
            <ejb-relationship-role>
<ejb-relationship-role-name>org-has-gangsters</ejb-relationship-role-name>
                <key-fields> <key-field> <field-name>name</field-name>
<column-name>organization</column-name> </key-field> </key-fields>
            </ejb-relationship-role>
            <ejb-relationship-role>
<ejb-relationship-role-name>gangster-belongs-to-org</ejb-relationship-role-name>
                <key-fields/>
            </ejb-relationship-role>
        </ejb-relation>
    </relationships>
</jbosscmp-jdbc>
```

5.3.3. Relation table Mapping

Relation table mapping is less common for one-to-one and one-to-many relationships, but is the only mapping style allowed for many-to-many relationships. Relation table mapping is defined using the `relation-table-mapping` element, the content model of which is shown below.

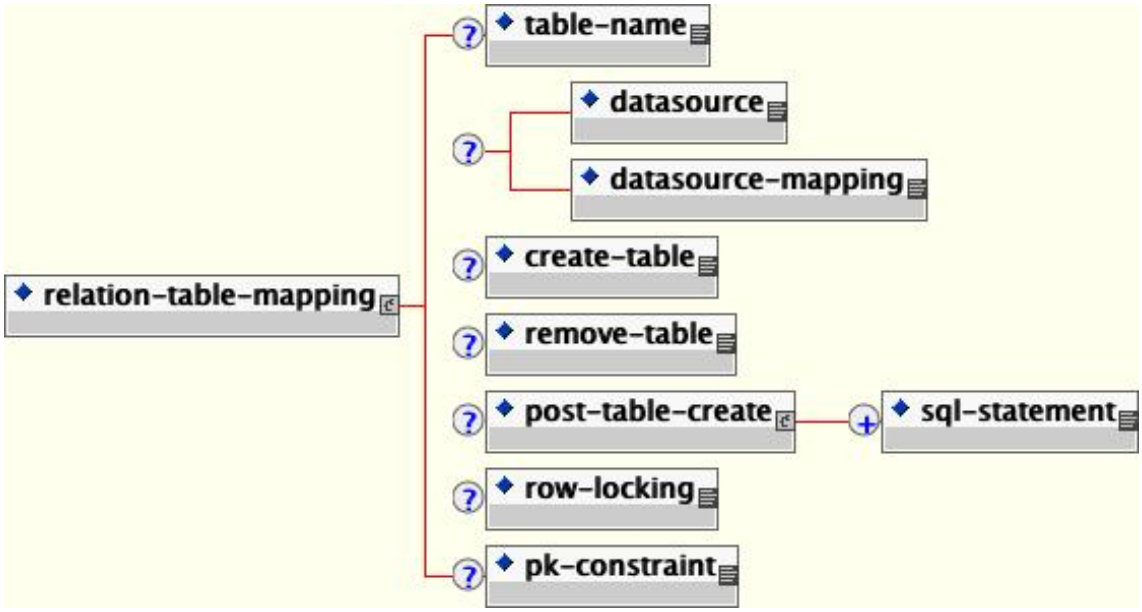


Figure 15.10. The jbosscmp-jdbc relation-table-mapping element content model

The relation-table-mapping for the `Gangster-Job` relationship is shown in with table mapping elements highlighted in bold:

```
<jbosscmp-jdbc>
    <relationships>
        <ejb-relation>
            <ejb-relation-name>Gangster-Jobs</ejb-relation-name>
            <relation-table-mapping>
                <table-name>gangster_job</table-name>
            </relation-table-mapping>
            <ejb-relationship-role>
<ejb-relationship-role-name>gangster-has-jobs</ejb-relationship-role-name>
                <key-fields>
                    <key-field>
                        <field-name>gangsterId</field-name>
                        <column-name>gangster</column-name>
                    </key-field>
                </key-fields>
            </ejb-relationship-role>
            <ejb-relationship-role>
<ejb-relationship-role-name>job-has-gangsters</ejb-relationship-role-name>
                <key-fields>
                    <key-field>
                        <field-name>name</field-name>
                        <column-name>job</column-name>
                    </key-field>
                </key-fields>
            </ejb-relationship-role>
        </ejb-relation>
```

```
    </relationships>
</jbosscmp-jdbc>
```

Example 15.1. The jbosscmp-jdbc.xml Relation-table Mapping

The `relation-table-mapping` element contains a subset of the options available in the `entity` element. A detailed description of these elements is reproduced here for convenience:

- **table-name**: This optional element gives the name of the table that will hold data for this relationship. The default table name is based on the entity and `cmr-field` names.
- **datasource**: This optional element gives the `jndi-name` used to look up the datasource. All database connections are obtained from the datasource. Having different datasources for entities is not recommended, as it vastly constrains the domain over which finders and `ejbSelect`s can query.
- **datasourcemapping**: This optional element allows one to specify the name of the `type-mapping` to use.
- **create-table**: This optional element if true indicates JBoss should attempt to create a table for the relationship. When the application is deployed, JBoss checks if a table already exists before creating the table. If a table is found, it is logged, and the table is not created. This option is very useful during the early stages of development when the table structure changes often.
- **post-table-create**: This optional element specifies an arbitrary SQL statement that should be executed immediately after the database table is created. This command is only executed if `create-table` is true and the table did not previously exist.
- **remove-table**: This optional element if true indicates JBoss should attempt to drop the `relation-table` when the application is undeployed. This option is very useful during the early stages of development when the table structure changes often.
- **row-locking**: This optional element if true indicates JBoss should lock all rows loaded in a transaction. Most databases implement this by using the `SELECT FOR UPDATE` syntax when loading the entity, but the actual syntax is determined by the `row-locking-template` in the `datasource-mapping` used by this entity.
- **pk-constraint**: This optional element if true indicates JBoss should add a primary key constraint when creating tables.

6. Queries

Entity beans allow for two types of queries: finders and selects. A finder provides queries on an

entity bean to clients of the bean. The select method is designed to provide private query statements to an entity implementation. Unlike finders, which are restricted to only return entities of the same type as the home interface on which they are defined, select methods can return any entity type or just one field of the entity. EJB-QL is the query language used to specify finders and select methods in a platform independent way.

6.1. Finder and select Declaration

The declaration of finders has not changed in CMP 2.0. Finders are still declared in the home interface (local or remote) of the entity. Finders defined on the local home interface do not throw a RemoteException. The following code declares the `findBadDudes_ejbql` finder on the `GangsterHome` interface. The `ejbql` suffix here is not required. It is simply a naming convention used here to differentiate the different types of query specifications we will be looking at.

```
public interface GangsterHome
    extends EJBLocalHome
{
    Collection findBadDudes_ejbql(int badness) throws FinderException;
}
```

Select methods are declared in the entity implementation class, and must be public and abstract just like CMP and CMR field abstract accessors and must throw a `FinderException`. The following code declares an select method:

```
public abstract class GangsterBean
    implements EntityBean
{
    public abstract Set ejbSelectBoss_ejbql(String name)
        throws FinderException;
}
```

6.2. EJB-QL Declaration

Every select or finder method (except `findByPrimaryKey`) must have an EJB-QL query defined in the `ejb-jar.xml` file. The EJB-QL query is declared in a query element, which is contained in the entity element. The following are the declarations for `findBadDudes_ejbql` and `ejbSelectBoss_ejbql` queries:

```
<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>GangsterEJB</ejb-name>
            <!-- ... -->
            <query>
                <query-method>
                    <method-name>findBadDudes_ejbql</method-name>
                    <method-params>
                        <method-param>int</method-param>
                    </method-params>
```

```
                </query-method>
                <ejb-ql><![CDATA[
                 SELECT OBJECT(g) FROM gangster g WHERE g.badness > ?1
                 ]]></ejb-ql>
            </query>
            <query>
                <query-method>
                    <method-name>ejbSelectBoss_ejbql</method-name>
                    <method-params>
                        <method-param>java.lang.String</method-param>
                    </method-params>
                </query-method>
                <ejb-ql><![CDATA[
                 SELECT DISTINCT underling.organization.theBoss FROM
gangster underling WHERE underling.name = ?1 OR underling.nickName = ?1
                 ]]></ejb-ql>
            </query>
        </entity>
    </enterprise-beans>
</ejb-jar>
```

EJB-QL is similar to SQL but has some surprising differences. The following are some important things to note about EJB-QL:

- EJB-QL is a typed language, meaning that it only allows comparison of like types (i.e., strings can only be compared with strings).
- In an equals comparison a variable (single valued path) must be on the left hand side. Some examples follow:

```
g.hangout.state = 'CA' Legal
'CA' = g.shippingAddress.state NOT Legal
'CA' = 'CA' NOT Legal
(r.amountPaid * .01) > 300 NOT Legal
r.amountPaid > (300 / .01) Legal
```

- Parameters use a base 1 index like java.sql.PreparedStatement.
- Parameters are only allowed on the right hand side of a comparison. For example:

```
gangster.hangout.state = ?1 Legal
?1 = gangster.hangout.state NOT Legal
```

6.3. Overriding the EJB-QL to SQL Mapping

The EJB-QL query can be overridden in the `jbosscmp-jdbc.xml` file. The finder or select is still required to have an EJB-QL declaration, but the `ejb-ql` element can be left empty. Currently

the SQL can be overridden with JBossQL, DynamicQL, DeclaredSQL or a BMP style custom `ejbFind` method. All EJB-QL overrides are non-standard extensions to the EJB specification, so use of these extensions will limit portability of your application. All of the EJB-QL overrides, except for BMP custom finders, are declared using a `query` element in the jbosscmp-jdbc.xml file. Tthe content model is shown in *Figure 15.11, "The jbosscmp-jdbc query element content model"*.

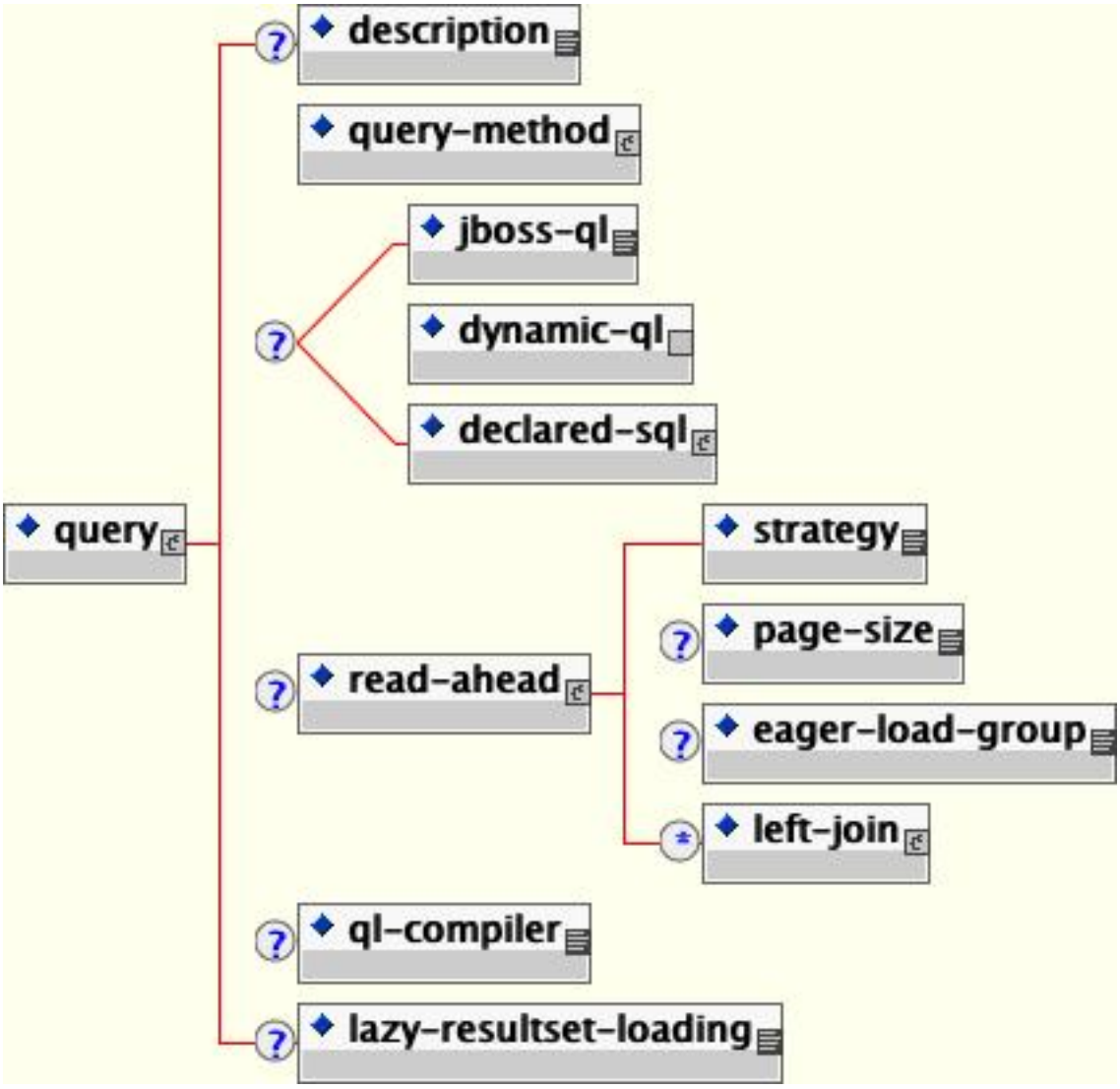


Figure 15.11. The jbosscmp-jdbc query element content model

- **description**: An optional description for the query.
- **query-method**: This required element specifies the query method that being configured. This must match a `query-method` declared for this entity in the `ejb-jar.xml` file.
- **jboss-ql**: This is a JBossQL query to use in place of the EJB-QL query. JBossQL is

discussed in *Section 6.4, "JBossQL"*.

- **dynamic-ql**: This indicated that the method is a dynamic query method and not an EJB-QL query. Dynamic queries are discussed in *Section 6.5, "DynamicQL"*.
- **declared-sql**: This query uses declared SQL in place of the EJB-QL query. Declared SQL is discussed in *Section 6.6, "DeclaredSQL"*.
- **read-ahead**: This optional element allows one to optimize the loading of additional fields for use with the entities referenced by the query. This is discussed in detail in *Section 7, "Optimized Loading"*.

6.4. JBossQL

JBossQL is a superset of EJB-QL that is designed to address some of the inadequacies of EJB-QL. In addition to a more flexible syntax, new functions, key words, and clauses have been added to JBossQL. At the time of this writing, JBossQL includes support for an `ORDER BY`, `OFFSET` and `LIMIT` clauses, parameters in the `IN` and `LIKE` operators, the `COUNT`, `MAX`, `MIN`, `AVG`, `SUM`, `UCASE` and `LCASE` functions. Queries can also include functions in the `SELECT` clause for select methods.

JBossQL is declared in the `jbosscmp-jdbc.xml` file with a `jboss-ql` element containing the JBossQL query. The following example provides an example JBossQL declaration.

```
<jbosscmp-jdbc>
    <enterprise-beans>
        <entity>
            <ejb-name>GangsterEJB</ejb-name>
            <query>
                <query-method>
                    <method-name>findBadDudes_jbossql</method-name>
                    <method-params>
                        <method-param>int</method-param>
                    </method-params>
                </query-method>
                <jboss-ql><![CDATA[ SELECT OBJECT(g) FROM gangster g WHERE
g.badness > ?1 ORDER BY g.badness DESC ]]></jboss-ql>
            </query>
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

The corresponding generated SQL is straightforward.

```
SELECT t0_g.id
    FROM gangster t0_g
    WHERE t0_g.badness > ?
    ORDER BY t0_g.badness DESC
```

Another capability of JBossQL is the ability to retrieve finder results in blocks using the `LIMIT`

and `OFFSET` functions. For example, to iterate through the large number of jobs performed, the following `findManyJobs_jbossql` finder may be defined.

```
<jbosscmp-jdbc>
    <enterprise-beans>
        <entity>
            <ejb-name>GangsterEJB</ejb-name>
            <query>
                <query-method>
                    <method-name>findManyJobs_jbossql</method-name>
                    <method-params>
                        <method-param>int</method-param>
                    </method-params>
                    <method-params>
                        <method-param>int</method-param>
                    </method-params>
                </query-method>
                <jboss-ql><![CDATA[ SELECT OBJECT(j) FROM jobs j OFFSET ?1
LIMIT ?2 ]]></jboss-ql>
            </query>
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

6.5. DynamicQL

DynamicQL allows the runtime generation and execution of JBossQL queries. A DynamicQL query method is an abstract method that takes a JBossQL query and the query arguments as parameters. JBoss compiles the JBossQL and executes the generated SQL. The following generates a JBossQL query that selects all the gangsters that have a hangout in any state in the states set:

```
public abstract class GangsterBean
    implements EntityBean
{
    public Set ejbHomeSelectInStates(Set states)
        throws FinderException
    {
        // generate JBossQL query
        StringBuffer jbossQl = new StringBuffer();
        jbossQl.append("SELECT OBJECT(g) ");
        jbossQl.append("FROM gangster g ");
        jbossQl.append("WHERE g.hangout.state IN (");

        for (int i = 0; i < states.size(); i++) {
            if (i > 0) {
                jbossQl.append(", ");
            }

            jbossQl.append("?").append(i+1);
        }

            jbossQl.append(") ORDER BY g.name");
```

```
        // pack arguments into an Object[]
        Object[] args = states.toArray(new Object[states.size()]);

        // call dynamic-ql query
        return ejbSelectGeneric(jbossQl.toString(), args);
    }
}
```

The DynamicQL select method may have any valid select method name, but the method must always take a string and an object array as parameters. DynamicQL is declared in the `jbosscmp-jdbc.xml` file with an empty `dynamic-ql` element. The following is the declaration for `ejbSelectGeneric`.

```
<jbosscmp-jdbc>
    <enterprise-beans>
        <entity>
            <ejb-name>GangsterEJB</ejb-name>
            <query>
                <query-method>
                    <method-name>ejbSelectGeneric</method-name>
                    <method-params>
                        <method-param>java.lang.String</method-param>
                        <method-param>java.lang.Object[]</method-param>
                    </method-params>
                </query-method>
                <dynamic-ql/>
            </query>
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

6.6. DeclaredSQL

DeclaredSQL is based on the legacy JAWS CMP 1.1 engine finder declaration, but has been updated for CMP 2.0. Commonly this declaration is used to limit a query with a `WHERE` clause that cannot be represented in q EJB-QL or JBossQL. The content model for the declared-sql element is given in *Figure 15.12, “The jbosscmp-jdbc declared-sql element content model.>”*.

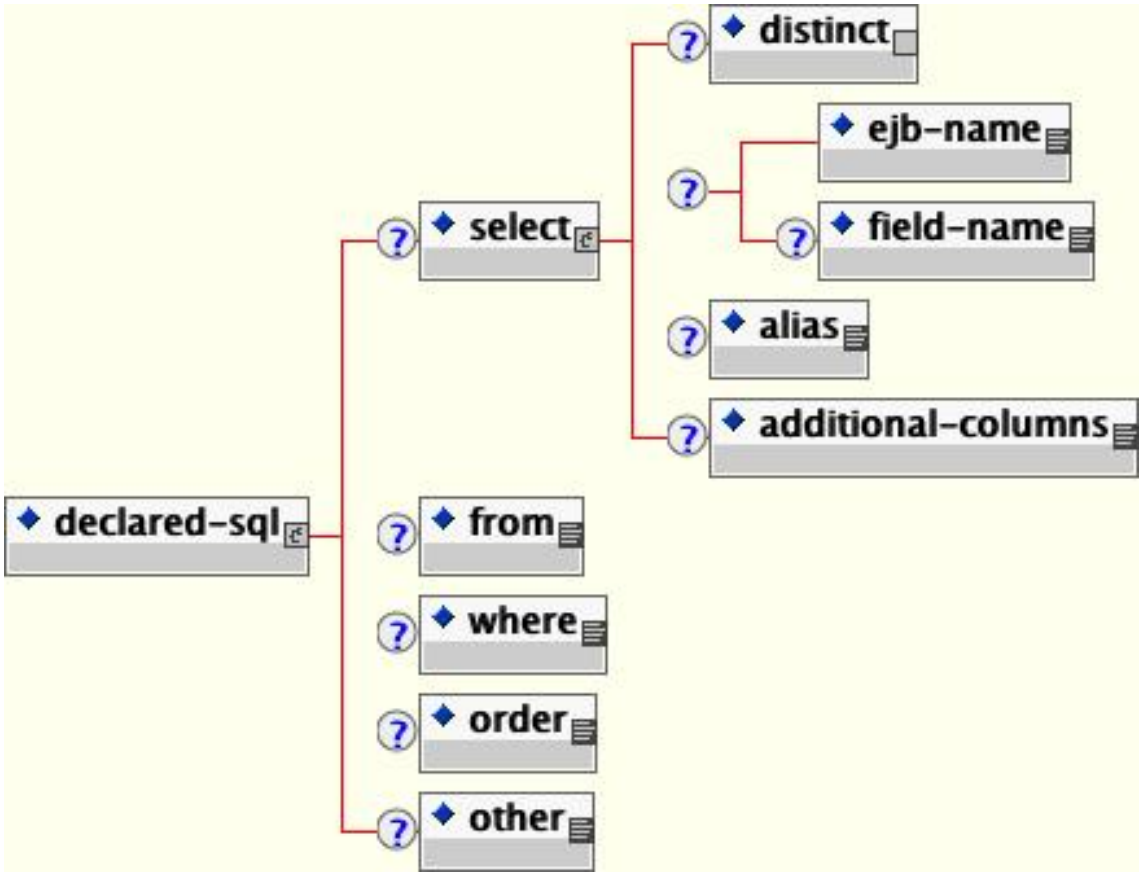


Figure 15.12. The jbosscmp-jdbc declared-sql element content model.>

- **select**: The `select` element specifies what is to be selected and consists of the following elements:
  - **distinct**: If this empty element is present, JBoss will add the `DISTINCT` keyword to the generated `SELECT` clause. The default is to use `DISTINCT` if method returns a `java.util.Set`
  - **ejb-name**: This is the `ejb-name` of the entity that will be selected. This is only required if the query is for a select method.
  - **field-name**: This is the name of the CMP field that will be selected from the specified entity. The default is to select entire entity.
  - **alias**: This specifies the alias that will be used for the main select table. The default is to use the `ejb-name`.
  - **additional-columns**: Declares other columns to be selected to satisfy ordering by arbitrary columns with finders or to facilitate aggregate functions in selects.

- **from**: The `from` element declares additional SQL to append to the generated `FROM` clause.
- **where**: The `where` element declares the `WHERE` clause for the query.
- **order**: The `order` element declares the `ORDER` clause for the query.
- **other**: The `other` element declares additional SQL that is appended to the end of the query.

The following is an example DeclaredSQL declaration.

```
<jbosscmp-jdbc>
    <enterprise-beans>
        <entity>
            <ejb-name>GangsterEJB</ejb-name>
            <query>
                <query-method>
                    <method-name>findBadDudes_declaredsql</method-name>
                    <method-params>
                        <method-param>int</method-param>
                    </method-params>
                </query-method>
                <declared-sql>
                    <where><![CDATA[ badness > {0} ]]></where>
                    <order><![CDATA[ badness DESC ]]></order>
                </declared-sql>
            </query>
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

The generated SQL would be:

```
SELECT id
FROM gangster
WHERE badness > ?
ORDER BY badness DESC
```

As you can see, JBoss generates the `SELECT` and `FROM` clauses necessary to select the primary key for this entity. If desired an additional `FROM` clause can be specified that is appended to the end of the automatically generated `FROM` clause. The following is example DeclaredSQL declaration with an additional `FROM` clause.

```
<jbosscmp-jdbc>
    <enterprise-beans>
        <entity>
            <ejb-name>GangsterEJB</ejb-name>
            <query>
                <query-method>
                    <method-name>ejbSelectBoss_declaredsql</method-name>
                    <method-params>
                        <method-param>java.lang.String</method-param>
                    </method-params>
```

```
                </query-method>
                <declared-sql>
                    <select>
                        <distinct/>
                        <ejb-name>GangsterEJB</ejb-name>
                        <alias>boss</alias>
                    </select>
                    <from><![CDATA[, gangster g, organization o]]></from>
                    <where><![CDATA[
                     (LCASE(g.name) = {0} OR LCASE(g.nick_name) = {0}) AND
                     g.organization = o.name AND o.the_boss = boss.id
                     ]]></where>
                </declared-sql>
            </query>
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

The generated SQL would be:

```
SELECT DISTINCT boss.id
    FROM gangster boss, gangster g, organization o
    WHERE (LCASE(g.name) = ? OR LCASE(g.nick_name) = ?) AND
          g.organization = o.name AND o.the_boss = boss.id
```

Notice that the `FROM` clause starts with a comma. This is because the container appends the declared `FROM` clause to the end of the generated `FROM` clause. It is also possible for the `FROM` clause to start with a SQL `JOIN` statement. Since this is a select method, it must have a `select` element to declare the entity that will be selected. Note that an alias is also declared for the query. If an alias is not declared, the `table-name` is used as the alias, resulting in a `SELECT` clause with the `table_name.field_name` style column declarations. Not all database vendors support the that syntax, so the declaration of an alias is preferred. The optional empty `distinct` element causes the `SELECT` clause to use the `SELECT DISTINCT` declaration. The DeclaredSQL declaration can also be used in select methods to select a CMP field.

Now we well see an example which overrides a select to return all of the zip codes an `Organization` operates in.

```
<jbosscmp-jdbc>
    <enterprise-beans>
        <entity>
            <ejb-name>OrganizationEJB</ejb-name>
            <query>
                <query-method>
<method-name>ejbSelectOperatingZipCodes_declaredsql</method-name>
                    <method-params>
                        <method-param>java.lang.String</method-param>
                    </method-params>
                </query-method>
                <declared-sql> <select> <distinct/>
<ejb-name>LocationEJB</ejb-name> <field-name>zipCode</field-name>
```

```
<alias>hangout</alias> </select> <from><![CDATA[ , organization o, gangster
g ]]></from> <where><![CDATA[ LCASE(o.name) = {0} AND o.name =
g.organization AND g.hangout = hangout.id ]]></where> <order><![CDATA[
hangout.zip ]]></order> </declared-sql>
            </query>
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

The corresponding SQL would be:

```
SELECT DISTINCT hangout.zip
    FROM location hangout, organization o, gangster g
    WHERE LCASE(o.name) = ? AND o.name = g.organization AND g.hangout =
hangout.id
                ORDER BY hangout.zip
```

6.6.1. Parameters

DeclaredSQL uses a completely new parameter handling system, which supports entity and DVC parameters. Parameters are enclosed in curly brackets and use a zero-based index, which is different from the one-based EJB-QL parameters. There are three categories of parameters: simple, DVC, and entity.

- **simple**: A simple parameter can be of any type except for a known (mapped) DVC or an entity. A simple parameter only contains the argument number, such as `{0}`. When a simple parameter is set, the JDBC type used to set the parameter is determined by the `datasourcemapping` for the entity. An unknown DVC is serialized and then set as a parameter. Note that most databases do not support the use of a BLOB value in a WHERE clause.
- **DVC**: A DVC parameter can be any known (mapped) DVC. A DVC parameter must be dereferenced down to a simple property (one that is not another DVC). For example, if we had a CVS property of type `ContactInfo`, valid parameter declarations would be `{0.email}` and `{0.cell.areaCode}` but not `{0.cell}`. The JDBC type used to set a parameter is based on the class type of the property and the `datasourcemapping` of the entity. The JDBC type used to set the parameter is the JDBC type that is declared for that property in the `dependent-value-class` element.
- **entity**: An entity parameter can be any entity in the application. An entity parameter must be dereferenced down to a simple primary key field or simple property of a DVC primary key field. For example, if we had a parameter of type `Gangster`, a valid parameter declaration would be `{0.gangsterId}`. If we had some entity with a primary key field named info of type `ContactInfo`, a `valid parameter` declaration would be `{0.info.cell.areaCode}`. Only fields that are members of the primary key of the entity can be dereferenced (this restriction may be removed in later versions). The JDBC type used to set the parameter is the JDBC type that is declared for that field in the entity declaration.

6.7. EJBQL 2.1 and SQL92 queries

The default query compiler doesn't fully support EJB-QL 2.1 or the SQL92 standard. If you need either of these functions, you can replace the query compiler. The default compiler is specified in `standardjbosscmp-jdbc.xml`.

```
<defaults>
    ...
<ql-compiler>org.jboss.ejb.plugins.cmp.jdbc.JDBCEJBQLCompiler</ql-compiler>
    ...
</defaults>
```

To use the SQL92 compiler, simply specify the SQL92 compiler in `ql-compiler` element.

```
<defaults>
    ...
<ql-compiler>org.jboss.ejb.plugins.cmp.jdbc.EJBQLToSQL92Compiler</ql-compiler>
    ...
</defaults>
```

This changes the query compiler for all beans in the entire system. You can also specify the ql-compiler for each element in `jbosscmp-jdbc.xml`. Here is an example using one of our earlier queries.

```
<query>
    <query-method>
        <method-name>findBadDudes_ejbql</method-name>
        <method-params>
            lt;method-param>int</method-param>
        </method-params>
    </query-method>
    <ejb-ql><![CDATA[
        SELECT OBJECT(g)
        FROM gangster g
        WHERE g.badness > ?1]]>
    </ejb-ql>
    <ql-compiler>org.jboss.ejb.plugins.cmp.jdbc.EJBQLToSQL92Compiler</ql-compiler>
</query>
```

One important limitation of SQL92 query compiler is that it always selects all the fields of an entity regardless the `read-ahead` strategy in use. For example, if a query is configured with the `on-loadread-ahead` strategy, the first query will include all the fields, not just primary key fields but only the primary key fields will be read from the `ResultSet`. Then, on load, other fields will be actually loaded into the read-ahead cache. The `on-findread-ahead` with the default load group `*` works as expected.

6.8. BMP Custom Finders

JBoss also supports bean managed persistence custom finders. If a custom finder method

matches a finder declared in the home or local home interface, JBoss will always call the custom finder over any other implementation declared in the `ejb-jar.xml` or `jbosscmp-jdbc.xml` files. The following simple example finds the entities by a collection of primary keys:

```
public abstract class GangsterBean
    implements EntityBean
{
    public Collection ejbFindByPrimaryKeys(Collection keys)
    {
        return keys;
    }
}
```

This is a very useful finder because it quickly coverts primary keys into real Entity objects without contacting the database. One drawback is that it can create an Entity object with a primary key that does not exist in the database. If any method is invoked on the bad Entity, a NoSuchEntityException will be thrown. Another drawback is that the resulting entity bean violates the EJB specification in that it implements a finder, and the JBoss EJB verifier will fail the deployment of such an entity unless the StrictVerifier attribute is set to false.

7. Optimized Loading

The goal of optimized loading is to load the smallest amount of data required to complete a transaction in the fewest number of queries. The tuning of JBoss depends on a detailed knowledge of the loading process. This section describes the internals of the JBoss loading process and its configuration. Tuning of the loading process really requires a holistic understanding of the loading system, so this chapter may have to be read more than once.

7.1. Loading Scenario

The easiest way to investigate the loading process is to look at a usage scenario. The most common scenario is to locate a collection of entities and iterate over the results performing some operation. The following example generates an html table containing all of the gangsters:

```
public String createGangsterHtmlTable_none()
    throws FinderException
{
    StringBuffer table = new StringBuffer();
    table.append("<table>");

    Collection gangsters = gangsterHome.findAll_none();
    for (Iterator iter = gangsters.iterator(); iter.hasNext();) {
        Gangster gangster = (Gangster) iter.next();
        table.append("<tr>");
        table.append("<td>").append(gangster.getName());
        table.append("</td>");
        table.append("<td>").append(gangster.getNickName());
        table.append("</td>");
        table.append("<td>").append(gangster.getBadness());
```

```
            table.append("</td>");
            table.append("</tr>");
        }

        return table.toString();
    }
```

Assume this code is called within a single transaction and all optimized loading has been disabled. At the `findAll_none` call, JBoss will execute the following query:

```
SELECT t0_g.id
    FROM gangster t0_g
    ORDER BY t0_g.id ASC
```

Then as each of the eight gangster in the sample database is accessed, JBoss will execute the following eight queries:

```
SELECT name, nick_name, badness, hangout, organization
  FROM gangster WHERE (id=0)
SELECT name, nick_name, badness, hangout, organization
  FROM gangster WHERE (id=1)
SELECT name, nick_name, badness, hangout, organization
  FROM gangster WHERE (id=2)
SELECT name, nick_name, badness, hangout, organization
  FROM gangster WHERE (id=3)
SELECT name, nick_name, badness, hangout, organization
  FROM gangster WHERE (id=4)
SELECT name, nick_name, badness, hangout, organization
  FROM gangster WHERE (id=5)
SELECT name, nick_name, badness, hangout, organization
  FROM gangster WHERE (id=6)
SELECT name, nick_name, badness, hangout, organization
  FROM gangster WHERE (id=7)
```

There are two problems with this scenario. First, an excessive number of queries are executed because JBoss executes one query for the `findAll` and one query to access each element found. The reason for this behavior has to do with the handling of query results inside the JBoss container. Although it appears that the actual entity beans selected are returned when a query is executed, JBoss really only returns the primary keys of the matching entities, and does not load the entity until a method is invoked on it. This is known as the *n+1* problem and is addressed with the read-ahead strategies described in the following sections.

Second, the values of unused fields are loaded needlessly. JBoss loads the `hangout` and `organization` fields, which are never accessed. (we have disabled the complex `contactInfo` field for the sake of clarity)

The following table shows the execution of the queries:

id	name	nick_name	badness	hangout	organization
0	Yojimbo	Bodyguard	7	0	Yakuza
1	Takeshi	Master	10	1	Yakuza
2	Yuriko	Four finger	4	2	Yakuza
3	Chow	Killer	9	3	Triads
4	Shogi	Lightning	8	4	Triads
5	Valentino	Pizza-Face	4	5	Mafia
6	Toni	Toothless	2	6	Mafia
7	Corleone	Godfather	6	7	Mafia

Table 15.1. Unoptimized Query Execution

7.2. Load Groups

The configuration and optimization of the loading system begins with the declaration of named load groups in the entity. A load group contains the names of CMP fields and CMR Fields that have a foreign key (e.g., `Gangster` in the Organization-Gangster example) that will be loaded in a single operation. An example configuration is shown below:

```
<jbosscmp-jdbc>
    <enterprise-beans>
        <entity>
            <ejb-name>GangsterEJB</ejb-name>
            <!-- ... -->
            <load-groups>
                <load-group>
                    <load-group-name>basic</load-group-name>
                    <field-name>name</field-name>
                    <field-name>nickName</field-name>
                    <field-name>badness</field-name>
                </load-group>
                <load-group>
                    <load-group-name>contact info</load-group-name>
                    <field-name>nickName</field-name>
                    <field-name>contactInfo</field-name>
                    <field-name>hangout</field-name>
                </load-group>
            </load-groups>
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

In this example, two load groups are declared: `basic` and `contact info`. Note that the load groups do not need to be mutually exclusive. For example, both of the load groups contain the `nickName` field. In addition to the declared load groups, JBoss automatically adds a group named `*` (the star group) that contains every CMP field and CMR field with a foreign key in the

entity.

7.3. Read-ahead

Optimized loading in JBoss is called read-ahead. This refers to the technique of reading the row for an entity being loaded, as well as the next several rows; hence the term read-ahead. JBoss implements two main strategies (`on-find` and `on-load`) to optimize the loading problem identified in the previous section. The extra data loaded during read-ahead is not immediately associated with an entity object in memory, as entities are not materialized in JBoss until actually accessed. Instead, it is stored in the preload cache where it remains until it is loaded into an entity or the end of the transaction occurs. The following sections describe the read-ahead strategies.

7.3.1. on-find

The `on-find` strategy reads additional columns when the query is invoked. If the query is `on-find` optimized, JBoss will execute the following query when the query is executed.

```
SELECT t0_g.id, t0_g.name, t0_g.nick_name, t0_g.badness
    FROM gangster t0_g
    ORDER BY t0_g.id ASC
```

All of the required data would be in the preload cache, so no additional queries would need to be executed while iterating through the query results. This strategy is effective for queries that return a small amount of data, but it becomes very inefficient when trying to load a large result set into memory. The following table shows the execution of this query:

id	name	nick_name	badness	hangout	organization
0	Yojimbo	Bodyguard	7	0	Yakuza
1	Takeshi	Master	10	1	Yakuza
2	Yuriko	Four finger	4	2	Yakuza
3	Chow	Killer	9	3	Triads
4	Shogi	Lightning	8	4	Triads
5	Valentino	Pizza-Face	4	5	Mafia
6	Toni	Toothless	2	6	Mafia
7	Corleone	Godfather	6	7	Mafia

Table 15.2. on-find Optimized Query Execution

The `read-ahead` strategy and `load-group` for a query is defined in the `query` element. If a `read-ahead` strategy is not declared in the `query` element, the strategy declared in the `entity` element or `defaults` element is used. The `on-find` configuration follows:

```
<jbosscmp-jdbc>
```

```
    <enterprise-beans>
        <entity>
            <ejb-name>GangsterEJB</ejb-name>
            <!--...-->
            <query>
                <query-method>
                    <method-name>findAll_onfind</method-name>
                    <method-params/>
                </query-method>
                <jboss-ql><![CDATA[
                 SELECT OBJECT(g)
                 FROM gangster g
                 ORDER BY g.gangsterId
                 ]]></jboss-ql>
                <read-ahead>
                    <strategy>on-find</strategy>
                    <page-size>4</page-size>
                    <eager-load-group>basic</eager-load-group>
                </read-ahead>
            </query>
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

One problem with the `on-find` strategy is that it must load additional data for every entity selected. Commonly in web applications only a fixed number of results are rendered on a page. Since the preloaded data is only valid for the length of the transaction, and a transaction is limited to a single web HTTP hit, most of the preloaded data is not used. The `on-load` strategy discussed in the next section does not suffer from this problem.

7.3.1.1. Left join read ahead

Left join read ahead is an enhanced `on-findread-ahead` strategy. It allows you to preload in one SQL query not only fields from the base instance but also related instances which can be reached from the base instance by CMR navigation. There are no limitation for the depth of CMR navigations. There are also no limitations for cardinality of CMR fields used in navigation and relationship type mapping, i.e. both foreign key and relation-table mapping styles are supported. Let's look at some examples. Entity and relationship declarations can be found below.

7.3.1.2. D#findByPrimaryKey

Suppose we have an entity `D`. A typical SQL query generated for the `findByPrimaryKey` would look like this:

```
SELECT t0_D.id, t0_D.name FROM D t0_D WHERE t0_D.id=?
```

Suppose that while executing `findByPrimaryKey` we also want to preload two collection-valued CMR fields `bs` and `cs`.

```
<query>
    <query-method>
        <method-name>findByPrimaryKey</method-name>
        <method-params>
            <method-param>java.lang.Long</method-param>
        </method-params>
    </query-method>
    <jboss-ql><![CDATA[SELECT OBJECT(o) FROM D AS o WHERE o.id =
?1]]></jboss-ql>
    <read-ahead>
        <strategy>on-find</strategy>
        <page-size>4</page-size>
        <eager-load-group>basic</eager-load-group>
        <left-join cmr-field="bs" eager-load-group="basic"/>
        <left-join cmr-field="cs" eager-load-group="basic"/>
    </read-ahead>
</query>
```

The `left-join` declares the relations to be eager loaded. The generated SQL would look like this:

```
SELECT t0_D.id, t0_D.name,
       t1_D_bs.id, t1_D_bs.name,
       t2_D_cs.id, t2_D_cs.name
  FROM D t0_D
       LEFT OUTER JOIN B t1_D_bs ON t0_D.id=t1_D_bs.D_FK
       LEFT OUTER JOIN C t2_D_cs ON t0_D.id=t2_D_cs.D_FK
 WHERE t0_D.id=?
```

For the `D` with the specific id we preload all its related `B`'s and `C`'s and can access those instance loading them from the read ahead cache, not from the database.

7.3.1.3. D#findAll

In the same way, we could optimize the `findAll` method on `D` selects all the `D`'s. A normal findAll query would look like this:

```
SELECT DISTINCT t0_o.id, t0_o.name FROM D t0_o ORDER BY t0_o.id DESC
```

To preload the relations, we simply need to add the `left-join` elements to the query.

```
<query>
    <query-method>
        <method-name>findAll</method-name>
    </query-method>
    <jboss-ql><![CDATA[SELECT DISTINCT OBJECT(o) FROM D AS o ORDER BY o.id
DESC]]></jboss-ql>
    <read-ahead>
        <strategy>on-find</strategy>
        <page-size>4</page-size>
```

```
        <eager-load-group>basic</eager-load-group>
        <left-join cmr-field="bs" eager-load-group="basic"/>
        <left-join cmr-field="cs" eager-load-group="basic"/>
    </read-ahead>
</query>
```

And here is the generated SQL:

```
SELECT DISTINCT t0_o.id, t0_o.name,
                t1_o_bs.id, t1_o_bs.name,
                t2_o_cs.id, t2_o_cs.name
  FROM D t0_o
       LEFT OUTER JOIN B t1_o_bs ON t0_o.id=t1_o_bs.D_FK
       LEFT OUTER JOIN C t2_o_cs ON t0_o.id=t2_o_cs.D_FK
 ORDER BY t0_o.id DESC
```

Now the simple `findAll` query now preloads the related `B` and `C` objects for each `D` object.

7.3.1.4. A#findAll

Now let's look at a more complex configuration. Here we want to preload instance `A` along with several relations.

- its parent (self-relation) reached from `A` with CMR field `parent`
- the `B` reached from `A` with CMR field `b`, and the related `C` reached from `B` with CMR field `c`
- `B` reached from `A` but this time with CMR field `b2` and related to it `C` reached from B with CMR field c.

For reference, the standard query would be:

```
SELECT t0_o.id, t0_o.name FROM A t0_o ORDER BY t0_o.id DESC FOR UPDATE
```

The following metadata describes our preloading plan.

```
<query>
    <query-method>
        <method-name>findAll</method-name>
    </query-method>
    <jboss-ql><![CDATA[SELECT OBJECT(o) FROM A AS o ORDER BY o.id
DESC]]></jboss-ql>
    <read-ahead>
        <strategy>on-find</strategy>
        <page-size>4</page-size>
        <eager-load-group>basic</eager-load-group>
        <left-join cmr-field="parent" eager-load-group="basic"/>
        <left-join cmr-field="b" eager-load-group="basic">
```

```
            <left-join cmr-field="c" eager-load-group="basic"/>
        </left-join>
        <left-join cmr-field="b2" eager-load-group="basic">
            <left-join cmr-field="c" eager-load-group="basic"/>
        </left-join>
    </read-ahead>
</query>
```

The SQL query generated would be:

```
SELECT t0_o.id, t0_o.name,
       t1_o_parent.id, t1_o_parent.name,
       t2_o_b.id, t2_o_b.name,
       t3_o_b_c.id, t3_o_b_c.name,
       t4_o_b2.id, t4_o_b2.name,
       t5_o_b2_c.id, t5_o_b2_c.name
  FROM A t0_o
       LEFT OUTER JOIN A t1_o_parent ON t0_o.PARENT=t1_o_parent.id
       LEFT OUTER JOIN B t2_o_b ON t0_o.B_FK=t2_o_b.id
       LEFT OUTER JOIN C t3_o_b_c ON t2_o_b.C_FK=t3_o_b_c.id
       LEFT OUTER JOIN B t4_o_b2 ON t0_o.B2_FK=t4_o_b2.id
       LEFT OUTER JOIN C t5_o_b2_c ON t4_o_b2.C_FK=t5_o_b2_c.id
 ORDER BY t0_o.id DESC FOR UPDATE
```

With this configuration, you can navigate CMRs from any found instance of A without an additional database load.

7.3.1.5. A#findMeParentGrandParent

Here is another example of self-relation. Suppose, we want to write a method that would preload an instance, its parent, grand-parent and its grand-grand-parent in one query. To do this, we would used nested `left-join` declaration.

```
<query>
    <query-method>
        <method-name>findMeParentGrandParent</method-name>
        <method-params>
            <method-param>java.lang.Long</method-param>
        </method-params>
    </query-method>
    <jboss-ql><![CDATA[SELECT OBJECT(o) FROM A AS o WHERE o.id =
?1]]></jboss-ql>
    <read-ahead>
        <strategy>on-find</strategy>
        <page-size>4</page-size>
        <eager-load-group>*</eager-load-group>
        <left-join cmr-field="parent" eager-load-group="basic">
            <left-join cmr-field="parent" eager-load-group="basic">
                <left-join cmr-field="parent" eager-load-group="basic"/>
            </left-join>
        </left-join>
    </read-ahead>
```

```
</query>
```

The generated SQL would be:

```
SELECT t0_o.id, t0_o.name, t0_o.secondName, t0_o.B_FK, t0_o.B2_FK,
t0_o.PARENT,
       t1_o_parent.id, t1_o_parent.name,
       t2_o_parent_parent.id, t2_o_parent_parent.name,
       t3_o_parent_parent_parent.id, t3_o_parent_parent_parent.name
  FROM A t0_o
       LEFT OUTER JOIN A t1_o_parent ON t0_o.PARENT=t1_o_parent.id
       LEFT OUTER JOIN A t2_o_parent_parent ON
t1_o_parent.PARENT=t2_o_parent_parent.id
       LEFT OUTER JOIN A t3_o_parent_parent_parent
            ON t2_o_parent_parent.PARENT=t3_o_parent_parent_parent.id
 WHERE (t0_o.id = ?) FOR UPDATE
```

Note, if we remove `left-join` metadata we will have only

```
SELECT t0_o.id, t0_o.name, t0_o.secondName, t0_o.B2_FK, t0_o.PARENT FOR
UPDATE
```

7.3.2. on-load

The `on-load` strategy block-loads additional data for several entities when an entity is loaded, starting with the requested entity and the next several entities in the order they were selected. This strategy is based on the theory that the results of a find or select will be accessed in forward order. When a query is executed, JBoss stores the order of the entities found in the list cache. Later, when one of the entities is loaded, JBoss uses this list to determine the block of entities to load. The number of lists stored in the cache is specified with the `list-cachemax` element of the entity. This strategy is also used when faulting in data not loaded in the `on-find` strategy.

As with the `on-find` strategy, `on-load` is declared in the `read-ahead` element. The `on-load` configuration for this example is shown below.

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <!-- ... -->
      <query>
        <query-method>
          <method-name>findAll_onload</method-name>
          <method-params/>
        </query-method>
        <jboss-ql><![CDATA[
             SELECT OBJECT(g)
             FROM gangster g
```

```
            ORDER BY g.gangsterId
            ]]></jboss-ql>
        <read-ahead>
          <strategy>on-load</strategy>
          <page-size>4</page-size>
          <eager-load-group>basic</eager-load-group>
        </read-ahead>
      </query>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

With this strategy, the query for the finder method in remains unchanged.

```
SELECT t0_g.id
       FROM gangster t0_g
       ORDER BY t0_g.id ASC
```

However, the data will be loaded differently as we iterate through the result set. For a page size of four, JBoss will only need to execute the following two queries to load the `name`, `nickName` and `badness` fields for the entities:

```
SELECT id, name, nick_name, badness
       FROM gangster
       WHERE (id=0) OR (id=1) OR (id=2) OR (id=3)
SELECT id, name, nick_name, badness
       FROM gangster
       WHERE (id=4) OR (id=5) OR (id=6) OR (id=7)
```

The following table shows the execution of these queries:

id	name	nick_name	badness	hangout	organization
0	Yojimbo	Bodyguard	7	0	Yakuza
1	Takeshi	Master	10	1	Yakuza
2	Yuriko	Four finger	4	2	Yakuza
3	Chow	Killer	9	3	Triads
4	Shogi	Lightning	8	4	Triads
5	Valentino	Pizza-Face	4	5	Mafia
6	Toni	Toothless	2	6	Mafia
7	Corleone	Godfather	6	7	Mafia

Table 15.3. on-load Optimized Query Execution

7.3.3. none

The `none` strategy is really an anti-strategy. This strategy causes the system to fall back to the default lazy-load code, and specifically does not read-ahead any data or remember the order of the found entities. This results in the queries and performance shown at the beginning of this chapter. The none strategy is declared with a read-ahead element. If the `read-ahead` element contains a `page-size` element or `eager-load-group`, it is ignored. The none strategy is declared the following example.

```
<jbosscmp-jdbc>
    <enterprise-beans>
        <entity>
            <ejb-name>GangsterEJB</ejb-name>
            <!-- ... -->
            <query>
                <query-method>
                    <method-name>findAll_none</method-name>
                    <method-params/>
                </query-method>
                <jboss-ql><![CDATA[
                 SELECT OBJECT(g)
                 FROM gangster g
                 ORDER BY g.gangsterId
                 ]]></jboss-ql>
                <read-ahead>
                    <strategy>none</strategy>
                </read-ahead>
            </query>
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

8. Loading Process

In the previous section several steps use the phrase "when the entity is loaded." This was intentionally left vague because the commit option specified for the entity and the current state of the transaction determine when an entity is loaded. The following section describes the commit options and the loading processes.

8.1. Commit Options

Central to the loading process are the commit options, which control when the data for an entity expires. JBoss supports four commit options `A`, `B`, `C` and `D`. The first three are described in the Enterprise JavaBeans Specification, but the last one is specific to JBoss. A detailed description of each commit option follows:

- **A**: JBoss assumes it is the sole user of the database; therefore, JBoss can cache the current value of an entity between transactions, which can result is substantial performance gains. As

a result of this assumption, no data managed by JBoss can be changed outside of JBoss. For example, changing data in another program or with the use of direct JDBC (even within JBoss) will result in an inconsistent database state.

- **B**: JBoss assumes that there is more than one user of the database but keeps the context information about entities between transactions. This context information is used for optimizing loading of the entity. This is the default commit option.
- **C**: JBoss discards all entity context information at the end of the transaction.
- **D**: This is a JBoss specific commit option. This option is similar to commit option `A`, except that the data only remains valid for a specified amount of time.

The commit option is declared in the `jboss.xml` file. For a detailed description of this file see *Chapter 14, EJBs on JBoss*. The following example changes the commit option to `A` for all entity beans in the application:

```
<jboss>
    <container-configurations>
        <container-configuration>
            <container-name>Standard CMP 2.x EntityBean</container-name>
            <commit-option>A</commit-option>
        </container-configuration>
    </container-configurations>
</jboss>
```

Example 15.2. The jboss.xml Commit Option Declaration

8.2. Eager-loading Process

When an entity is loaded, JBoss must determine the fields that need to be loaded. By default, JBoss will use the `eager-load-group` of the last query that selected this entity. If the entity has not been selected in a query, or the last query used the `none` read-ahead strategy, JBoss will use the default `eager-load-group` declared for the entity. In the following example configuration, the `basic` load group is set as the default `eager-load-group` for the gangster entity bean:

```
<jbosscmp-jdbc>
    <enterprise-beans>
        <entity>
            <ejb-name>GangsterEJB</ejb-name>
            <!-- ... -->
            <load-groups>
                <load-group>
                    <load-group-name>most</load-group-name>
                    <field-name>name</field-name>
                    <field-name>nickName</field-name>
                    <field-name>badness</field-name>
```

```
                    <field-name>hangout</field-name>
                    <field-name>organization</field-name>
                </load-group>
            </load-groups>
            <eager-load-group>most</eager-load-group>
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

The eager loading process is initiated the first time a method is called on an entity in a transaction. A detailed description of the load process follows:

1. If the entity context is still valid, no loading is necessary, and therefore the loading process is done. The entity context will be valid when using commit option `A`, or when using commit option `D`, and the data has not timed out.
2. Any residual data in the entity context is flushed. This assures that old data does not bleed into the new load.
3. The primary key value is injected back into the primary key fields. The primary key object is actually independent of the fields and needs to be reloaded after the flush in step 2.
4. All data in the preload cache for this entity is loaded into the fields.
5. JBoss determines the additional fields that still need to be loaded. Normally the fields to load are determined by the eager-load group of the entity, but can be overridden if the entity was located using a query or CMR field with an `on-find` or `on-load` read ahead strategy. If all of the fields have already been loaded, the load process skips to step 7.
6. A query is executed to select the necessary column. If this entity is using the `on-load` strategy, a page of data is loaded as described in *Section 7.3.2, "on-load"*. The data for the current entity is stored in the context and the data for the other entities is stored in the preload cache.
7. The `ejbLoad` method of the entity is called.

8.3. Lazy loading Process

Lazy loading is the other half of eager loading. If a field is not eager loaded, it must be lazy loaded. When an access to an unloaded field of a bean is made, JBoss loads the field and all the fields of any `lazy-load-group` the field belong to. JBoss performs a set join and then removes any field that is already loaded. An example configuration is shown below.

```
<jbosscmp-jdbc>
    <enterprise-beans>
        <entity>
            <ejb-name>GangsterEJB</ejb-name>
            <!-- ... -->
            <load-groups>
```

```
                <load-group>
                    <load-group-name>basic</load-group-name>
                    <field-name>name</field-name>
                    <field-name>nickName</field-name>
                    <field-name>badness</field-name>
                </load-group>
                <load-group>
                    <load-group-name>contact info</load-group-name>
                    <field-name>nickName</field-name>
                    <field-name>contactInfo</field-name>
                    <field-name>hangout</field-name>
                </load-group>
            </load-groups>
            <!-- ... -->
            <lazy-load-groups>
                <load-group-name>basic</load-group-name>
                <load-group-name>contact info</load-group-name>
            </lazy-load-groups>
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

When the bean provider calls `getName()` with this configuration, JBoss loads `name`, `nickName` and `badness`, assuming they are not already loaded. When the bean provider calls `getNickName()`, the `name`, `nickName`, `badness`, `contactInfo`, and `hangout` are loaded. A detailed description of the lazy loading process follows:

1. All data in the preload cache for this entity is loaded into the fields.
2. If the field value was loaded by the preload cache the lazy load process is finished.
3. JBoss finds all of the lazy load groups that contain this field, performs a set join on the groups, and removes any field that has already been loaded.
4. A query is executed to select the necessary columns. As in the basic load process, JBoss may load a block of entities. The data for the current entity is stored in the context and the data for the other entities is stored in the preload cache.

8.3.1. Relationships

Relationships are a special case in lazy loading because a CMR field is both a field and query. As a field it can be `on-load` block loaded, meaning the value of the currently sought entity and the values of the CMR field for the next several entities are loaded. As a query, the field values of the related entity can be preloaded using `on-find`.

Again, the easiest way to investigate the loading is to look at a usage scenario. In this example, an HTML table is generated containing each gangster and their hangout. The example code follows:

```
public String createGangsterHangoutHtmlTable()
    throws FinderException
{
    StringBuffer table = new StringBuffer();
    table.append("<table>");
    Collection gangsters = gangsterHome.findAll_onfind();
    for (Iterator iter = gangsters.iterator(); iter.hasNext(); ) {
        Gangster gangster = (Gangster)iter.next();

        Location hangout = gangster.getHangout();
        table.append("<tr>");
        table.append("<td>").append(gangster.getName());
        table.append("</td>");
        table.append("<td>").append(gangster.getNickName());
        table.append("</td>");
        table.append("<td>").append(gangster.getBadness());
        table.append("</td>");
        table.append("<td>").append(hangout.getCity());
        table.append("</td>");
        table.append("<td>").append(hangout.getState());
        table.append("</td>");
        table.append("<td>").append(hangout.getZipCode());
        table.append("</td>");
        table.append("</tr>");
    }

    table.append("</table>");return table.toString();
}
```

Example 15.3. Relationship Lazy Loading Example Code

For this example, the configuration of the gangster's `findAll_onfind` query is unchanged from the `on-find` section. The configuration of the `Location` entity and `Gangster-Hangout` relationship follows:

```
<jbosscmp-jdbc>
    <enterprise-beans>
        <entity>
            <ejb-name>LocationEJB</ejb-name>
            <load-groups>
                <load-group>
                    <load-group-name>quick info</load-group-name>
                    <field-name>city</field-name>
                    <field-name>state</field-name>
                    <field-name>zipCode</field-name>
                </load-group>
            </load-groups>
            <eager-load-group/>
        </entity>
    </enterprise-beans>
    <relationships>
```

```
        <ejb-relation>
            <ejb-relation-name>Gangster-Hangout</ejb-relation-name>
            <foreign-key-mapping/>
            <ejb-relationship-role>
                <ejb-relationship-role-name>
                    gangster-has-a-hangout
                </ejb-relationship-role-name>
                <key-fields/>
                <read-ahead>
                    <strategy>on-find</strategy>
                    <page-size>4</page-size>
                    <eager-load-group>quick info</eager-load-group>
                </read-ahead>
            </ejb-relationship-role>
            <ejb-relationship-role>
                <ejb-relationship-role-name>
                    hangout-for-a-gangster
                </ejb-relationship-role-name>
                <key-fields>
                    <key-field>
                        <field-name>locationID</field-name>
                        <column-name>hangout</column-name>
                    </key-field>
                </key-filaelds>
            </ejb-relationship-role>
        </ejb-relation>
    </relationships>
</jbosscmp-jdbc>
```

Example 15.4. The jbosscmp-jdbc.xml Relationship Lazy Loading Configuration

JBoss will execute the following query for the finder:

```
SELECT t0_g.id, t0_g.name, t0_g.nick_name, t0_g.badness
    FROM gangster t0_g
    ORDER BY t0_g.id ASC
```

Then when the hangout is accessed, JBoss executes the following two queries to load the `city`, `state`, and `zip` fields of the hangout:

```
SELECT gangster.id, gangster.hangout,
       location.city, location.st, location.zip
    FROM gangster, location
    WHERE (gangster.hangout=location.id) AND
          ((gangster.id=0) OR (gangster.id=1) OR
          (gangster.id=2) OR (gangster.id=3))
SELECT gangster.id, gangster.hangout,
       location.city, location.st, location.zip
    FROM gangster, location
```

```
    WHERE (gangster.hangout=location.id) AND
          ((gangster.id=4) OR (gangster.id=5) OR
          (gangster.id=6) OR (gangster.id=7))
```

The following table shows the execution of the queries:

id	name	nick_name	badness	hangout	id	city	st	zip
0	Yojimbo	Bodyguard	7	0	0	San Fran	CA	94108
1	Takeshi	Master	10	1	1	San Fran	CA	94133
2	Yuriko	Four finger	4	2	2	San Fran	CA	94133
3	Chow	Killer	9	3	3	San Fran	CA	94133
4	Shogi	Lightning	8	4	4	San Fran	CA	94133
5	Valentino	Pizza-Face	4	5	5	New York	NY	10017
6	Toni	Toothless	2	6	6	Chicago	IL	60661
7	Corleone	Godfather	6	7	7	Las Vegas	NV	89109

Table 15.4. on-find Optimized Relationship Query Execution

8.4. Lazy loading result sets

By default, when a multi-object finder or select method is executed the JDBC result set is read to the end immediately. The client receives a collection of `EJBLocalObject` or CMP field values which it can then iterate through. For big result sets this approach is not efficient. In some cases it is better to delay reading the next row in the result set until the client tries to read the corresponding value from the collection. You can get this behavior for a query using the `lazy-resultset-loading` element.

```
<query>
    <query-method>
        <method-name>findAll</method-name>
    </query-method>
    <jboss-ql><![CDATA[select object(o) from A o]]></jboss-ql>
    <lazy-resultset-loading>true</lazy-resultset-loading>
</query>
```

The are some issues you should be aware of when using lazy result set loading. Special care should be taken when working with a `Collection` associated with a lazily loaded result set. The first call to `iterator()` returns a special `Iterator` that reads from the `ResultSet`. Until this `Iterator` has been exhausted, subsequent calls to `iterator()` or calls to the `add()` method will result in an exception. The `remove()` and `size()` methods work as would be expected.

9. Transactions

All of the examples presented in this chapter have been defined to run in a transaction. Transaction granularity is a dominating factor in optimized loading because transactions define the lifetime of preloaded data. If the transaction completes, commits, or rolls back, the data in the preload cache is lost. This can result in a severe negative performance impact.

The performance impact of running without a transaction will be demonstrated with an example that uses an `on-find` optimized query that selects the first four gangsters (to keep the result set small), and it is executed without a wrapper transaction. The example code follows:

```
public String createGangsterHtmlTable_no_tx() throws FinderException
{
    StringBuffer table = new StringBuffer();
    table.append("<table>");

    Collection gangsters = gangsterHome.findFour();
    for(Iterator iter = gangsters.iterator(); iter.hasNext(); ) {
        Gangster gangster = (Gangster)iter.next();
        table.append("<tr>");
        table.append("<td>").append(gangster.getName());
        table.append("</td>");
        table.append("<td>").append(gangster.getNickName());
        table.append("</td>");
        table.append("<td>").append(gangster.getBadness());
        table.append("</td>");
        table.append("</tr>");
    }

    table.append("</table>");
    return table.toString();
}
```

The finder results in the following query being executed:

```
SELECT t0_g.id, t0_g.name, t0_g.nick_name, t0_g.badness
  FROM gangster t0_g
  WHERE t0_g.id < 4
  ORDER BY t0_g.id ASC
```

Normally this would be the only query executed, but since this code is not running in a transaction, all of the preloaded data is thrown away as soon as finder returns. Then when the CMP field is accessed JBoss executes the following four queries (one for each loop):

```
SELECT id, name, nick_name, badness
  FROM gangster
  WHERE (id=0) OR (id=1) OR (id=2) OR (id=3)
SELECT id, name, nick_name, badness
  FROM gangster
  WHERE (id=1) OR (id=2) OR (id=3)
SELECT id, name, nick_name, badness
  FROM gangster
  WHERE (id=2) OR (id=3)
SELECT name, nick_name, badness
  FROM gangster
  WHERE (id=3)
```

It's actually worse than this. JBoss executes each of these queries three times; once for each CMP field that is accessed. This is because the preloaded values are discarded between the CMP field accessor calls.

The following figure shows the execution of the queries:

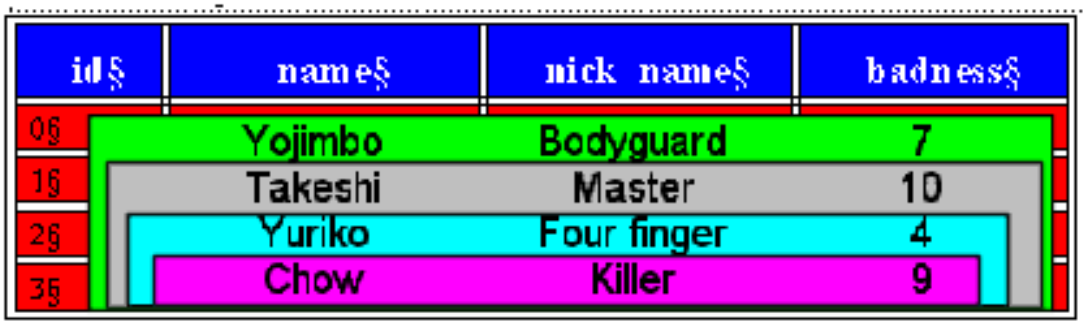


Figure 15.13. No Transaction on-find optimized query execution

This performance is much worse than read ahead none because of the amount of data loaded from the database. The number of rows loaded is determined by the following equation:

$$n + n - 1 + n - 2 + \ldots + 1 + = \frac{n \bullet (n+1)}{2} = O(n^2)$$

This all happens because the transaction in the example is bounded by a single call on the entity. This brings up the important question "How do I run my code in a transaction?" The answer depends on where the code runs. If it runs in an EJB (session, entity, or message driven), the method must be marked with the `Required` or `RequiresNewtrans-attribute` in the `assembly-descriptor`. If the code is not running in an EJB, a user transaction is necessary. The following code wraps a call to the declared method with a user transaction:

```
public String createGangsterHtmlTable_with_tx()
    throws FinderException
{
    UserTransaction tx = null;
    try {
        InitialContext ctx = new InitialContext();
        tx = (UserTransaction) ctx.lookup("UserTransaction");
        tx.begin();

        String table = createGangsterHtmlTable_no_tx();

        if (tx.getStatus() == Status.STATUS_ACTIVE) {
                tx.commit();
        }
            return table;
    } catch (Exception e) {
        try {
            if (tx != null) tx.rollback();
        } catch (SystemException unused) {
            // eat the exception we are exceptioning out anyway
        }
        if (e instanceof FinderException) {
                throw (FinderException) e;
        }
        if (e instanceof RuntimeException) {
                throw (RuntimeException) e;
        }

        throw new EJBException(e);
    }
}
```

10. Optimistic Locking

JBoss has supports for optimistic locking of entity beans. Optimistic locking allows multiple instances of the same entity bean to be active simultaneously. Consistency is enforced based on the optimistic locking policy choice. The optimistic locking policy choice defines the set of fields that are used in the commit time write of modified data to the database. The optimistic consistency check asserts that the values of the chosen set of fields has the same values in the database as existed when the current transaction was started. This is done using a `select for UPDATE WHERE ...` statement that contains the value assertions.

You specify the optimistic locking policy choice using an `optimistic-locking` element in the `jbosscmp-jdbc.xml` descriptor. The content model of the `optimistic-locking` element is shown below and the description of the elements follows.

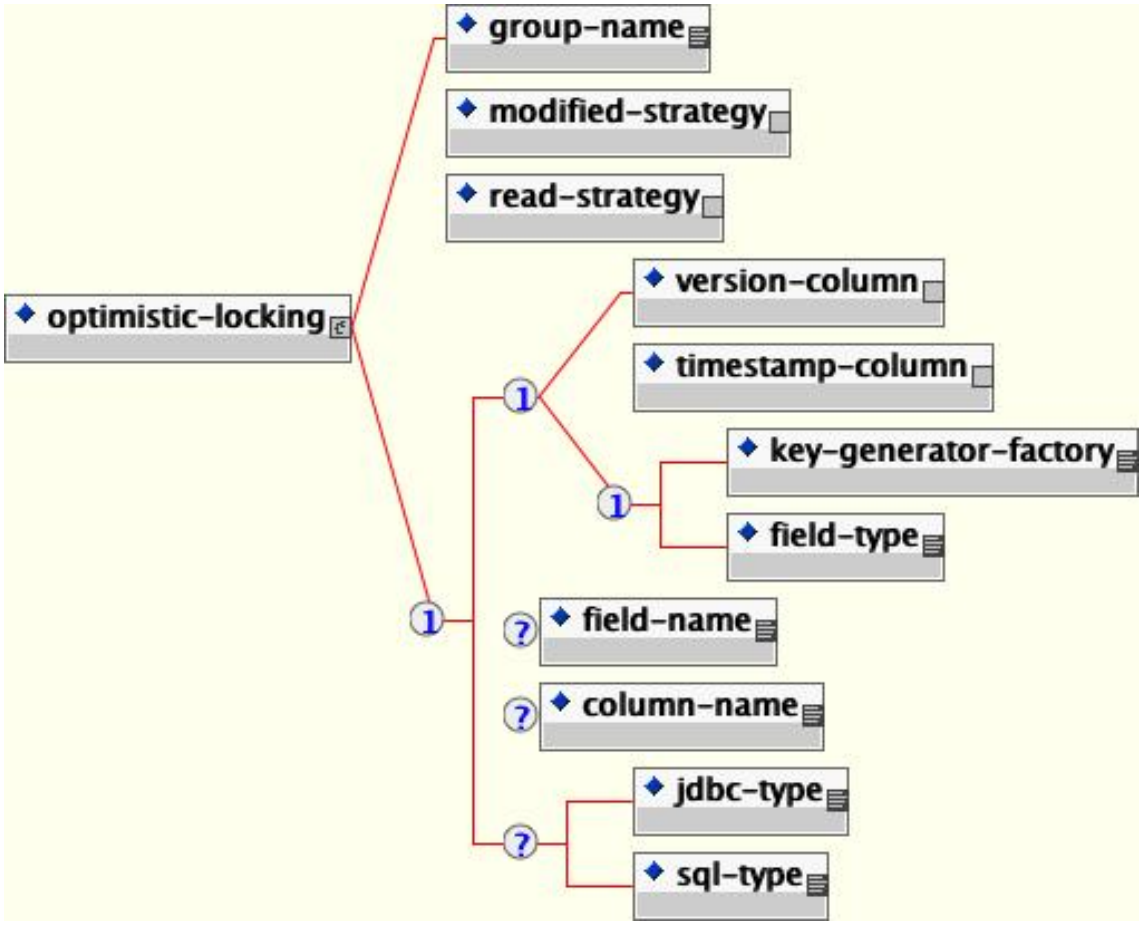


Figure 15.14. The jbosscmp-jdbc optimistic-locking element content model

- **group-name**: This element specifies that optimistic locking is based on the fields of a `load-group`. This value of this element must match one of the entity's `load-group-name`. The fields in this group will be used for optimistic locking.
- **modified-strategy**: This element specifies that optimistic locking is based on the modified fields. This strategy implies that the fields that were modified during transaction will be used for optimistic locking.
- **read-strategy**: This element specifies that optimistic locking is based on the fields read. This strategy implies that the fields that were read/changed in the transaction will be used for optimistic locking.
- **version-column**: This element specifies that optimistic locking is based on a version column strategy. Specifying this element will add an additional version field of type `java.lang.Long` to the entity bean for optimistic locking. Each update of the entity will increase the value of this field. The `field-name` element allows for the specification of the name of the CMP field

while the `column-name` element allows for the specification of the corresponding table column.

- **timestamp-column**: This element specifies that optimistic locking is based on a timestamp column strategy. Specifying this element will add an additional version field of type `java.util.Date` to the entity bean for optimistic locking. Each update of the entity will set the value of this field to the current time. The `field-name` element allows for the specification of the name of the CMP field while the `column-name` element allows for the specification of the corresponding table column.

- **key-generator-factory**: This element specifies that optimistic locking is based on key generation. The value of the element is the JNDI name of a `org.jboss.ejb.plugins.keygenerator.KeyGeneratorFactory` implementation. Specifying this element will add an additional version field to the entity bean for optimistic locking. The type of the field must be specified via the `field-type` element. Each update of the entity will update the key field by obtaining a new value from the key generator. The `field-name` element allows for the specification of the name of the CMP field while the `column-name` element allows for the specification of the corresponding table column.

A sample `jbosscmp-jdbc.xml` descriptor illustrating all of the optimistic locking strategies is given below.

```
<!DOCTYPE jbosscmp-jdbc PUBLIC
    "-//JBoss//DTD JBOSSCMP-JDBC 3.2//EN"
    "http://www.jboss.org/j2ee/dtd/jbosscmp-jdbc_3_2.dtd">
<jbosscmp-jdbc>
    <defaults>
        <datasource>java:/DefaultDS</datasource>
        <datasource-mapping>Hypersonic SQL</datasource-mapping>
    </defaults>
    <enterprise-beans>
        <entity>
            <ejb-name>EntityGroupLocking</ejb-name>
            <create-table>true</create-table>
            <remove-table>true</remove-table>
            <table-name>entitygrouplocking</table-name>
            <cmp-field>
                <field-name>dateField</field-name>
            </cmp-field>
            <cmp-field>
                <field-name>integerField</field-name>
            </cmp-field>
            <cmp-field>
                <field-name>stringField</field-name>
            </cmp-field>
            <load-groups>
                <load-group>
                    <load-group-name>string</load-group-name>
                    <field-name>stringField</field-name>
                </load-group>
                <load-group>
                    <load-group-name>all</load-group-name>
                    <field-name>stringField</field-name>
```

```
                <field-name>dateField</field-name>
            </load-group>
        </load-groups>
        <optimistic-locking>
            <group-name>string</group-name>
        </optimistic-locking>
    </entity>
    <entity>
        <ejb-name>EntityModifiedLocking</ejb-name>
        <create-table>true</create-table>
        <remove-table>true</remove-table>
        <table-name>entitymodifiedlocking</table-name>
        <cmp-field>
            <field-name>dateField</field-name>
        </cmp-field>
        <cmp-field>
            <field-name>integerField</field-name>
        </cmp-field>
        <cmp-field>
            <field-name>stringField</field-name>
        </cmp-field>
        <optimistic-locking>
            <modified-strategy/>
        </optimistic-locking>
    </entity>
    <entity>
        <ejb-name>EntityReadLocking</ejb-name>
        <create-table>true</create-table>
        <remove-table>true</remove-table>
        <table-name>entityreadlocking</table-name>
        <cmp-field>
            <field-name>dateField</field-name>
        </cmp-field>
        <cmp-field>
            <field-name>integerField</field-name>
        </cmp-field>
        <cmp-field>
            <field-name>stringField</field-name>
        </cmp-field>
        <optimistic-locking>
            <read-strategy/>
        </optimistic-locking>
    </entity>
    <entity>
        <ejb-name>EntityVersionLocking</ejb-name>
        <create-table>true</create-table>
        <remove-table>true</remove-table>
        <table-name>entityversionlocking</table-name>
        <cmp-field>
            <field-name>dateField</field-name>
        </cmp-field>
        <cmp-field>
            <field-name>integerField</field-name>
        </cmp-field>
        <cmp-field>
            <field-name>stringField</field-name>
        </cmp-field>
```

```
            <optimistic-locking>
                <version-column/>
                <field-name>versionField</field-name>
                <column-name>ol_version</column-name>
                <jdbc-type>INTEGER</jdbc-type>
                <sql-type>INTEGER(5)</sql-type>
            </optimistic-locking>
        </entity>
        <entity>
            <ejb-name>EntityTimestampLocking</ejb-name>
            <create-table>true</create-table>
            <remove-table>true</remove-table>
            <table-name>entitytimestamplocking</table-name>
            <cmp-field>
                <field-name>dateField</field-name>
            </cmp-field>
            <cmp-field>
                <field-name>integerField</field-name>
            </cmp-field>
            <cmp-field>
                <field-name>stringField</field-name>
            </cmp-field>
            <optimistic-locking>
                <timestamp-column/>
                <field-name>versionField</field-name>
                <column-name>ol_timestamp</column-name>
                <jdbc-type>TIMESTAMP</jdbc-type>
                <sql-type>DATETIME</sql-type>
            </optimistic-locking>
        </entity>
        <entity>
            <ejb-name>EntityKeyGeneratorLocking</ejb-name>
            <create-table>true</create-table>
            <remove-table>true</remove-table>
            <table-name>entitykeygenlocking</table-name>
            <cmp-field>
                <field-name>dateField</field-name>
            </cmp-field>
            <cmp-field>
                <field-name>integerField</field-name>
            </cmp-field>
            <cmp-field>
                <field-name>stringField</field-name>
            </cmp-field>
            <optimistic-locking>
<key-generator-factory>UUIDKeyGeneratorFactory</key-generator-factory>
                <field-type>java.lang.String</field-type>
                <field-name>uuidField</field-name>
                <column-name>ol_uuid</column-name>
                <jdbc-type>VARCHAR</jdbc-type>
                <sql-type>VARCHAR(32)</sql-type>
            </optimistic-locking>
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

11. Entity Commands and Primary Key Generation

Support for primary key generation outside of the entity bean is available through custom implementations of the entity creation command objects used to insert entities into a persistent store. The list of available commands is specified in entity-commands element of the `jbosscmp-jdbc.xml` descriptor. The default `entity-command` may be specified in the `jbosscmp-jdbc.xml` in defaults element. Each entity element can override the `entity-command` in defaults by specifying its own `entity-command`. The content model of the `entity-commands` and child elements is given below.

Figure 15.15. The jbosscmp-jdbc.xml entity-commands element model

Each `entity-command` element specifies an entity generation implementation. The `name` attribute specifies a name that allows the command defined in an `entity-commands` section to be referenced in the defaults and entity elements. The `class` attribute specifies the implementation of the `org.jboss.ejb.plugins.cmp.jdbc.JDBCCreateEntityCommand` that supports the key generation. Database vendor specific commands typically subclass the `org.jboss.ejb.plugins.cmp.jdbc.JDBCIdentityColumnCreateCommand` if the database generates the primary key as a side effect of doing an insert, or the `org.jboss.ejb.plugins.cmp.jdbc.JDBCInsertPKCreateCommand` if the command must insert the generated key.

The optional `attribute` element(s) allows for the specification of arbitrary name/value property pairs that will be available to the entity command implementation class. The `attribute` element has a required `name` attribute that specifies the name property, and the `attribute` element content is the value of the property. The attribute values are accessible through the `org.jboss.ejb.plugins.cmp.jdbc.metadata.JDBCEntityCommandMetaData.getAttribute`(String) method.

11.1. Existing Entity Commands

The following are the current `entity-command` definitions found in the `standardjbosscmp-jdbc.xml` descriptor:

- **default**: (`org.jboss.ejb.plugins.cmp.jdbc.JDBCCreateEntityCommand`) The `JDBCCreateEntityCommand` is the default entity creation as it is the `entity-command` referenced in the `standardjbosscmp-jdbc.xml` defaults element. This entity-command executes an `INSERT INTO` query using the assigned primary key value.
- **no-select-before-insert**: (`org.jboss.ejb.plugins.cmp.jdbc.JDBCCreateEntityCommand`)

This is a variation on `default` that skips select before insert by specifying an attribute `name="SQLExceptionProcessor"` that points to the `jboss.jdbc:service=SQLExceptionProcessor` service. The `SQLExceptionProcessor` service provides a `boolean isDuplicateKey(SQLException e)` operation that allows a for determination of any unique constraint violation.

- **pk-sql** (`org.jboss.ejb.plugins.cmp.jdbc.JDBCPkSqlCreateCommand`) The `JDBCPkSqlCreateCommand` executes an `INSERT INTO` query statement provided by the `pk-sql` attribute to obtain the next primary key value. Its primary target usage are databases with sequence support.

- **mysql-get-generated-keys**: (`org.jboss.ejb.plugins.cmp.jdbc.mysql.JDBCMySQLCreateCommand`) The `JDBCMySQLCreateCommand` executes an `INSERT INTO` query using the `getGeneratedKeys` method from MySQL native `java.sql.Statement` interface implementation to fetch the generated key.

- **oracle-sequence**: (`org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCOracleCreateCommand`) The `JDBCOracleCreateCommand` is a create command for use with Oracle that uses a sequence in conjunction with a `RETURNING` clause to generate keys in a single statement. It has a required `sequence` element that specifies the name of the sequence column.

- **hsqldb-fetch-key**: (`org.jboss.ejb.plugins.cmp.jdbc.hsqldb.JDBCHsqldbCreateCommand`) The `JDBCHsqldbCreateCommand` executes an `INSERT INTO` query after executing a `CALL IDENTITY()` statement to fetch the generated key.

- **sybase-fetch-key**: (`org.jboss.ejb.plugins.cmp.jdbc.sybase.JDBCSybaseCreateCommand`) The `JDBCSybaseCreateCommand` executes an `INSERT` INTO query after executing a `SELECT @@IDENTITY` statement to fetch the generated key.

- **mssql-fetch-key**: (`org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCSQLServerCreateCommand`) The `JDBCSQLServerCreateCommand` for Microsoft SQL Server that uses the value from an `IDENTITY` columns. By default uses `SELECT SCOPE_IDENTITY()` to reduce the impact of triggers; can be overridden with `pk-sql` attribute e.g. for V7.

- **informix-serial**: (`org.jboss.ejb.plugins.cmp.jdbc.informix.JDBCInformixCreateCommand`) The `JDBCInformixCreateCommand` executes an `INSERT` INTO query after using the `getSerial` method from Informix native `java.sql.Statement` interface implementation to fetch the generated key.

- **postgresql-fetch-seq**: (`org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCPostgreSQLCreateCommand`) The `JDBCPostgreSQLCreateCommand` for PostgreSQL that fetches the current value of the

sequence. The optional `sequence` attribute can be used to change the name of the sequence, with the default being `table_pkColumn_seq`.

- **key-generator**: (`org.jboss.ejb.plugins.cmp.jdbc.JDBCKeyGeneratorCreateCommand`) The `JDBCKeyGeneratorCreateCommand` executes an `INSERT INTO` query after obtaining a value for the primary key from the key generator referenced by the `key-generator-factory`. The `key-generator-factory` attribute must provide the name of a JNDI binding of the `org.jboss.ejb.plugins.keygenerator.KeyGeneratorFactory` implementation.

- **get-generated-keys**: (org.jboss.ejb.plugins.cmp.jdbc.jdbc3.JDBCGetGeneratedKeysCreateCommand) The `JDBCGetGeneratedKeysCreateCommand` executes an `INSERT INTO` query using a statement built using the JDBC3 `prepareStatement(String, Statement.RETURN_GENERATED_KEYS)` that has the capability to retrieve the auto-generated key. The generated key is obtained by calling the `PreparedStatement.getGeneratedKeys` method. Since this requires JDBC3 support it is only available in JDK1.4.1+ with a supporting JDBC driver.

An example configuration using the `hsqldb-fetch-keyentity-command` with the generated key mapped to a known primary key `cmp-field` is shown below.

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>LocationEJB</ejb-name>
      <pk-constraint>false</pk-constraint>
      <table-name>location</table-name>

      <cmp-field>
        <field-name>locationID</field-name>
        <column-name>id</column-name>
        <auto-increment/>
      </cmp-field>
      <!-- ... -->
      <entity-command name="hsqldb-fetch-key"/>

    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

An alternate example using an unknown primary key without an explicit `cmp-field` is shown below.

```
<jbosscmp-jdbc>
    <enterprise-beans>
        <entity>
            <ejb-name>LocationEJB</ejb-name>
            <pk-constraint>false</pk-constraint>
            <table-name>location</table-name>
            <unknown-pk>
                <unknown-pk-class>java.lang.Integer</unknown-pk-class>
                <field-name>locationID</field-name>
```

```
                <column-name>id</column-name>
                <jdbc-type>INTEGER</jdbc-type>
                <sql-type>INTEGER</sql-type>
                <auto-increment/>
            </unknown-pk>
            <!--...-->
            <entity-command name="hsqldb-fetch-key"/>
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

12. Defaults

JBoss global defaults are defined in the `standardjbosscmp-jdbc.xml` file of the `server/<server-name>/conf/` directory. Each application can override the global defaults in the `jbosscmp-jdbc.xml` file. The default options are contained in a defaults element of the configuration file, and the content model is shown below.

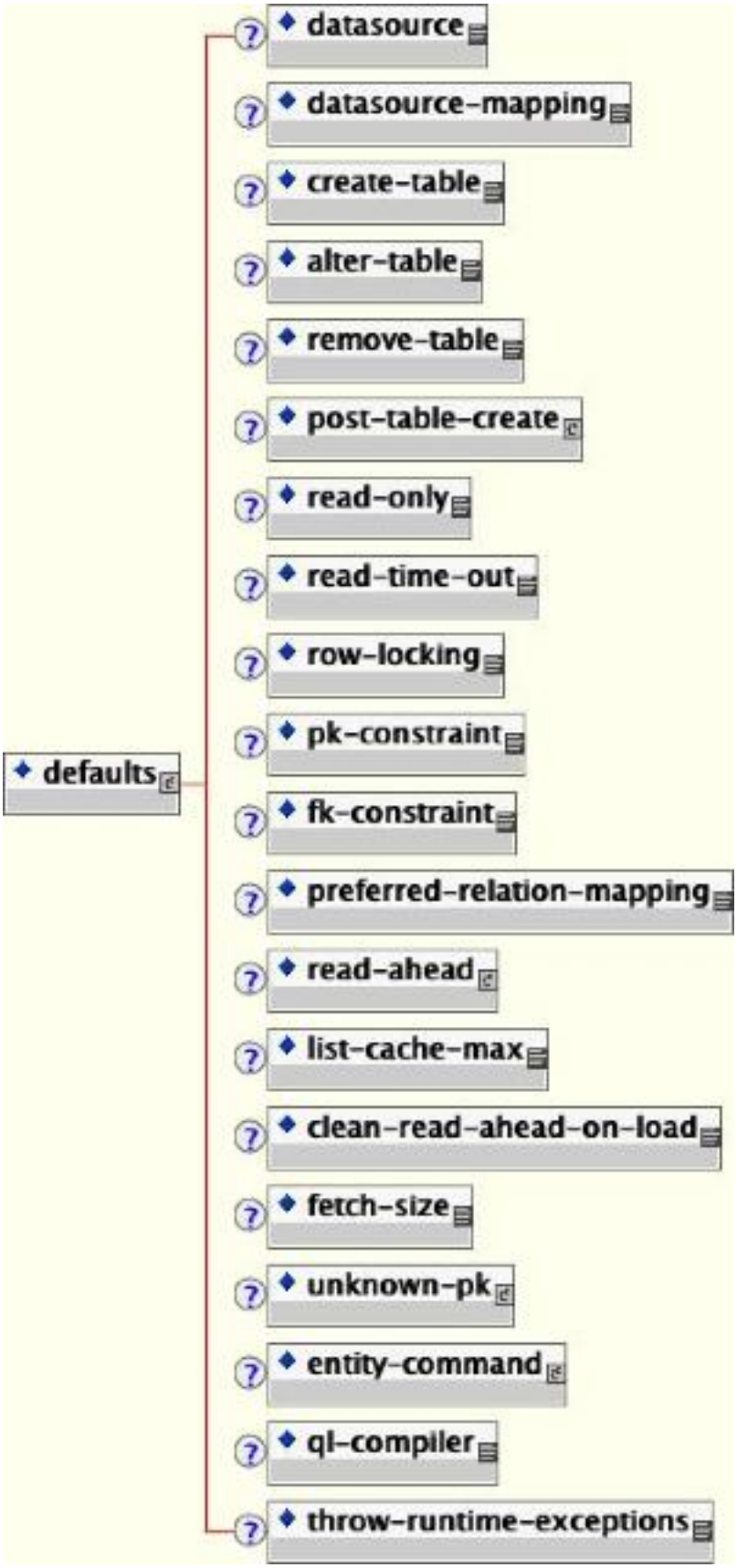

defaults
datasource
datasource-mapping
create-table
alter-table
remove-table
post-table-create
read-only
read-time-out
row-locking
pk-constraint
fk-constraint
preferred-relation-mapping
read-ahead
list-cache-max
clean-read-ahead-on-load
fetch-size
unknown-pk
entity-command
ql-compiler
throw-runtime-exceptions

Figure 15.16. The jbosscmp-jdbc.xml defaults content model

An example of the defaults section follows:

```
<jbosscmp-jdbc>
    <defaults>
        <datasource>java:/DefaultDS</datasource>
        <datasource-mapping>Hypersonic SQL</datasource-mapping>
        <create-table>true</create-table>
        <remove-table>false</remove-table>
        <read-only>false</read-only>
        <read-time-out>300000</read-time-out>
        <pk-constraint>true</pk-constraint>
        <fk-constraint>false</fk-constraint>
        <row-locking>false</row-locking>
        <preferred-relation-mapping>foreign-key</preferred-relation-mapping>
        <read-ahead>
            <strategy>on-load</strategy>
            <page-size>1000</page-size>
            <eager-load-group>*</eager-load-group>
        </read-ahead>
        <list-cache-max>1000</list-cache-max>
    </defaults>
</jbosscmp-jdbc>
```

12.1. A sample jbosscmp-jdbc.xml defaults declaration

Each option can apply to entities, relationships, or both, and can be overridden in the specific entity or relationship. A detailed description of each option follows:

- **datasource**: This optional element is the `jndi-name` used to look up the datasource. All database connections used by an entity or `relation-table` are obtained from the datasource. Having different datasources for entities is not recommended, as it vastly constrains the domain over which finders and ejbSelects can query.
- **datasource-mapping**: This optional element specifies the name of the `type-mapping`, which determines how Java types are mapped to SQL types, and how EJB-QL functions are mapped to database specific functions. Type mappings are discussed in *Section 13.3, "Mapping"*.
- **create-table**: This optional element when true, specifies that JBoss should attempt to create a table for the entity. When the application is deployed, JBoss checks if a table already exists before creating the table. If a table is found, it is logged, and the table is not created. This option is very useful during the early stages of development when the table structure changes often. The default is false.
- **alter-table**: If `create-table` is used to automatically create the schema, `alter-table` can

be used to keep the schema current with changes to the entity bean. Alter table will perform the following specific tasks:
  - new fields will be created
  - fields which are no longer used will be removed
  - string fields which are shorter than the declared length will have their length increased to the declared length. (not supported by all databases)

- **remove-table**: This optional element when true, JBoss will attempt to drop the table for each entity and each relation table mapped relationship. When the application is undeployed, JBoss will attempt to drop the table. This option is very useful during the early stages of development when the table structure changes often. The default is false.

- **read-only**: This optional element when true specifies that the bean provider will not be allowed to change the value of any fields. A field that is read-only will not be stored in, or inserted into, the database. If a primary key field is read-only, the create method will throw a `CreateException`. If a set accessor is called on a `read-only` field, it throws an `EJBException`. Read only fields are useful for fields that are filled in by database triggers, such as last update. The `read-only` option can be overridden on a per field basis. The default is false.

- **read-time-out**: This optional element is the amount of time in milliseconds that a read on a read only field is valid. A value of 0 means that the value is always reloaded at the start of a transaction, and a value of -1 means that the value never times out. This option can also be overridden on a per CMP field basis. If `read-only` is false, this value is ignored. The default is -1.

- **row-locking**: This optional element if true specifies that JBoss will lock all rows loaded in a transaction. Most databases implement this by using the `SELECT FOR UPDATE` syntax when loading the entity, but the actual syntax is determined by the `row-locking-template` in the `datasource-mapping` used by this entity. The default is false.

- **pk-constraint**: This optional element if true specifies that JBoss will add a primary key constraint when creating tables. The default is true.

- **preferred-relation-mapping**: This optional element specifies the preferred mapping style for relationships. The `preferred-relation-mapping` element must be either `foreign-key` or `relation-table`.

- **read-ahead**: This optional element controls caching of query results and CMR fields for the entity. This option is discussed in *Section 7.3, "Read-ahead"*.

- **list-cache-max**: This optional element specifies the number of `read-lists` that can be tracked by this entity. This option is discussed in *Section 7.3.2, "on-load"*. The default is 1000.

- **clean-read-ahead-on-load**: When an entity is loaded from the read ahead cache, JBoss can remove the data used from the read ahead cache. The default is `false`.

- **fetch-size**: This optional element specifies the number of entities to read in one round-trip to the underlying datastore. The default is 0.

- **unknown-pk**: This optional element allows one to define the default mapping of an unknown primary key type of `java.lang.Object` maps to the persistent store.
- **entity-command**: This optional element allows one to define the default command for entity creation. This is described in detail in *Section 11, "Entity Commands and Primary Key Generation"*.
- **ql-compiler**: This optional elements allows a replacement query compiler to be specified. Alternate query compilers were discussed in *Section 6.7, "EJBQL 2.1 and SQL92 queries"*.
- **throw-runtime-exceptions**: This attribute, if set to true, indicates that an error in connecting to the database should be seen in the application as runtime `EJBException` rather than as a checked exception.

13. Datasource Customization

JBoss includes predefined type-mappings for many databases including: Cloudscape, DB2, DB2/400, Hypersonic SQL, InformixDB, InterBase, MS SQLSERVER, MS SQLSERVER2000, mySQL, Oracle7, Oracle8, Oracle9i, PointBase, PostgreSQL, PostgreSQL 7.2, SapDB, SOLID, and Sybase. If you do not like the supplied mapping, or a mapping is not supplied for your database, you will have to define a new mapping. If you find an error in one of the supplied mappings, or if you create a new mapping for a new database, please consider posting a patch at the JBoss project page on SourceForge.

13.1. Type Mapping

Customization of a database is done through the `type-mapping` section of the `jbosscmp-jdbc.xml` descriptor. The content model for the type-mapping element is given in *Figure 15.17, "The jbosscmp-jdbc type-mapping element content model."*.

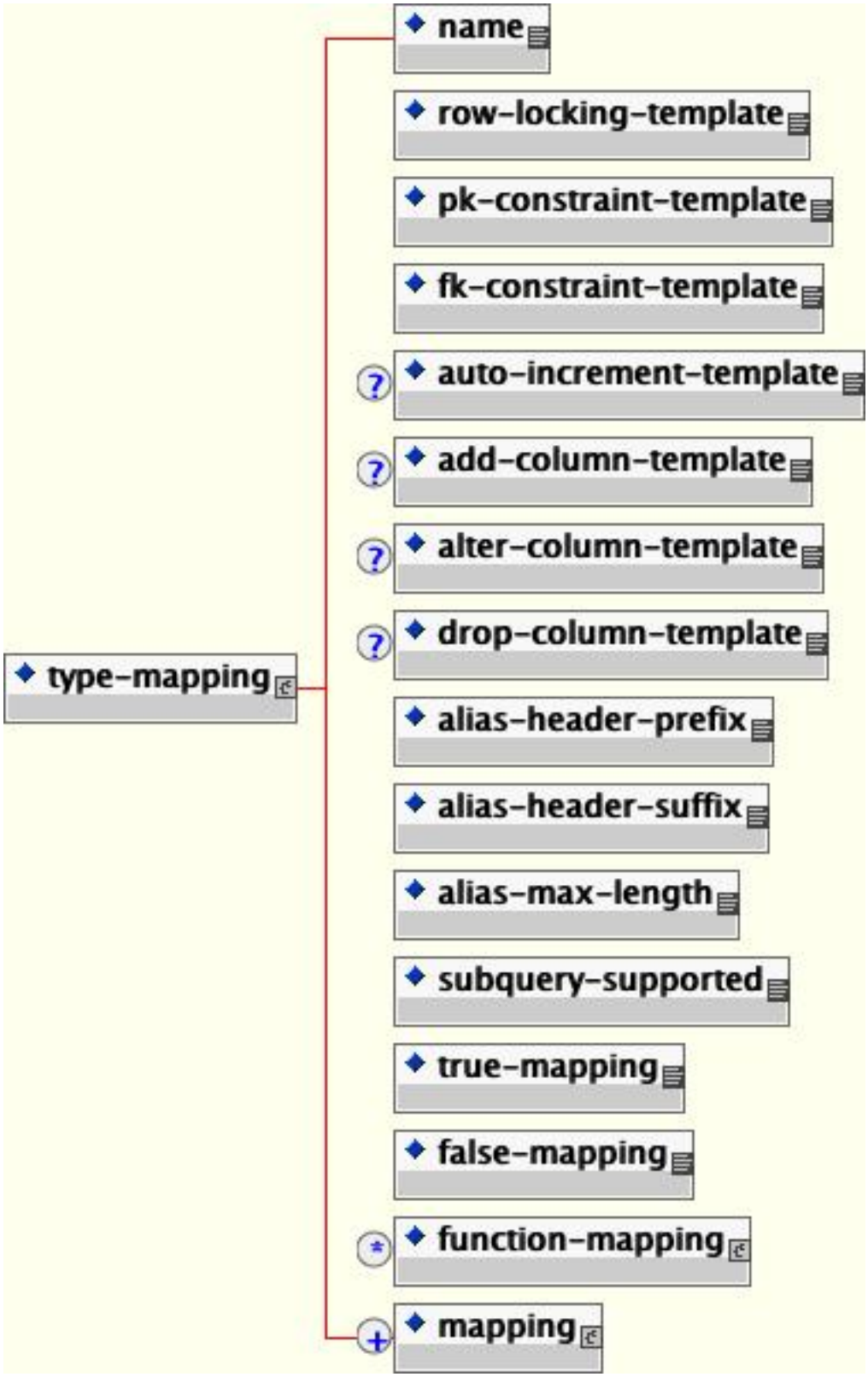


Figure 15.17. The jbosscmp-jdbc type-mapping element content model.

The elements are:

- **name**: This required element provides the name identifying the database customization. It is used to refer to the mapping by the `datasource-mapping` elements found in defaults and entity.
- **row-locking-template**: This required element gives the `PreparedStatement` template used to create a row lock on the selected rows. The template must support three arguments:

  1. the select clause
  2. the from clause. The order of the tables is currently not guaranteed
  3. the where clause

  If row locking is not supported in select statement this element should be empty. The most common form of row locking is select for update as in: `SELECT ?1 FROM ?2 WHERE ?3 FOR UPDATE`.
- **pk-constraint-template**: This required element gives the `PreparedStatement` template used to create a primary key constraint in the create table statement. The template must support two arguments
  1. Primary key constraint name; which is always `pk_{table-name}`
  2. Comma separated list of primary key column names

  If a primary key constraint clause is not supported in a create table statement this element should be empty. The most common form of a primary key constraint is: `CONSTRAINT ?1 PRIMARY KEY (?2)`
- **fk-constraint-template**: This is the template used to create a foreign key constraint in separate statement. The template must support five arguments:
  1. Table name
  2. Foreign key constraint name; which is always `fk_{table-name}_{cmr-field-name}`
  3. Comma separated list of foreign key column names
  4. References table name
  5. Comma separated list of the referenced primary key column names

  If the datasource does not support foreign key constraints this element should be empty. The most common form of a foreign key constraint is: `ALTER TABLE ?1 ADD CONSTRAINT ?2 FOREIGN KEY (?3) REFERENCES ?4 (?5)`.
- **auto-increment-template**: This declares the SQL template for specifying auto increment columns.
- **add-column-template**: When `alter-table` is true, this SQL template specifies the syntax for adding a column to an existing table. The default value is `ALTER TABLE ?1 ADD ?2 ?3`. The parameters are:
  1. the table name

2. the column name
3. the column type

- **alter-column-template**: When `alter-table` is true, this SQL template specifies the syntax for dropping a column to from an existing table. The default value is `ALTER TABLE ?1 ALTER ?2 TYPE ?3`. The parameters are:
  1. the table name
  2. the column name
  3. the column type

- **drop-column-template**: When `alter-table` is true, this SQL template specifies the syntax for dropping a column to from an existing table. The default value is `ALTER TABLE ?1 DROP ?2`. The parameters are:
  1. the table name
  2. the column name

- **alias-header-prefix**: This required element gives the prefix used in creating the alias header. An alias header is prepended to a generated table alias by the EJB-QL compiler to prevent name collisions. The alias header is constructed as follows: alias-header-prefix + int_counter + alias-header-suffix. An example alias header would be `t0_` for an alias-header-prefix of "`t`" and an alias-header-suffix of "_".

- **alias-header-suffix**: This required element gives the suffix portion of the generated alias header.

- **alias-max-length**: This required element gives the maximum allowed length for the generated alias header.

- **subquery-supported**: This required element specifies if this `type-mapping` subqueries as either true or false. Some EJB-QL operators are mapped to exists subqueries. If `subquery-supported` is false, the EJB-QL compiler will use a left join and is null.

- **true-mapping**: This required element defines *true* identity in EJB-QL queries. Examples include `TRUE`, `1`, and `(1=1)`.

- **false-mapping**: This required element defines *false* identity in EJB-QL queries. Examples include `FALSE`, `0`, and `(1=0)`.

- **function-mapping**: This optional element specifies one or more the mappings from an EJB-QL function to an SQL implementation. See *Section 13.2, "Function Mapping"* for the details.

- **mapping**: This required element specifies the mappings from a Java type to the corresponding JDBC and SQL type. See *Section 13.3, "Mapping"* for the details.

13.2. Function Mapping

The function-mapping element model is show below.

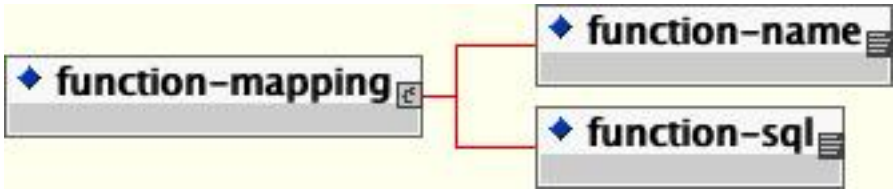


Figure 15.18. The jbosscmp-jdbc function-mapping element content model

The allowed child elements are:

- **function-name**: This required element gives the EJB-QL function name, e.g., `concat`, `substring`.
- **function-sql**: This required element gives the SQL for the function as appropriate for the underlying database. Examples for a `concat` function include: `(?1 || ?2)`, `concat(?1, ?2)`, `(?1 + ?2)`.

13.3. Mapping

A `type-mapping` is simply a set of mappings between Java class types and database types. A set of type mappings is defined by a set of `mapping` elements, the content model for which is shown in *Figure 15.19, "The jbosscmp-jdbc mapping element content model."*.

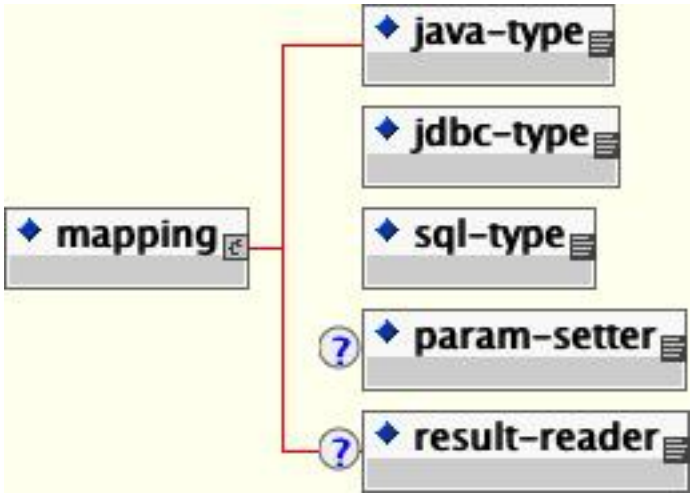


Figure 15.19. The jbosscmp-jdbc mapping element content model.

If JBoss cannot find a mapping for a type, it will serialize the object and use the `java.lang.Object` mapping. The following describes the three child elements of the mapping element:

- **java-type**: This required element gives the fully qualified name of the Java class to be mapped. If the class is a primitive wrapper class such as `java.lang.Short`, the mapping also applies to the primitive type.
- **jdbc-type**: This required element gives the JDBC type that is used when setting parameters in a JDBC `PreparedStatement` or loading data from a JDBC `ResultSet`. The valid types are defined in `java.sql.Types`.
- **sql-type**: This required element gives the SQL type that is used in create table statements. Valid types are only limited by your database vendor.
- **param-setter**: This optional element specifies the fully qualified name of the `JDBCParameterSetter` implementation for this mapping.
- **result-reader**: This option element specifies the fully qualified name of the `JDBCResultSetReader` implementation for this mapping.

An example mapping element for a short in Oracle9i is shown below.

```
<jbosscmp-jdbc>
    <type-mappings>
        <type-mapping>
            <name>Oracle9i</name>
            <!--...-->
            <mapping>
                <java-type>java.lang.Short</java-type>
                <jdbc-type>NUMERIC</jdbc-type>
                <sql-type>NUMBER(5)</sql-type>
            </mapping>
        </type-mapping>
    </type-mappings>
</jbosscmp-jdbc>
```

13.4. User Type Mappings

User type mappings allow one to map from JDBC column types to custom CMP fields types by specifying an instance of `org.jboss.ejb.plugins.cmp.jdbc.Mapper` interface, the definition of which is shown below.

```
public interface Mapper
{
    /**
     * This method is called when CMP field is stored.
     * @param fieldValue - CMP field value
     * @return column value.
     */
    Object toColumnValue(Object fieldValue);

    /**
     * This method is called when CMP field is loaded.
     * @param columnValue - loaded column value.
     * @return CMP field value.
```

```
     */
    Object toFieldValue(Object columnValue);
}
```

A prototypical use case is the mapping of an integer type to its type-safe Java enumeration instance. The content model of the `user-type-mappings` element consists of one or more `user-type-mapping` elements, the content model of which is shown in *Figure 15.20, "The user-type-mapping content model >"*.

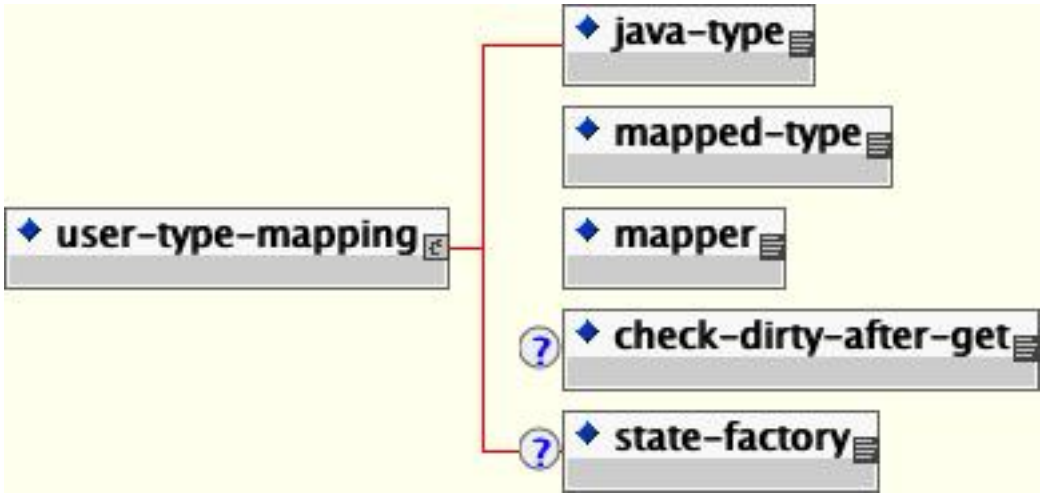


Figure 15.20. The user-type-mapping content model >

- **java-type**: the fully qualified name of the CMP field type in the mapping.
- **mapped-type**: the fully qualified name of the database type in the mapping.
- **mapper**: the fully qualified name of the `Mapper` interface implementation that handles the conversion between the `java-type` and `mapped-type`.

Appendix A. Book Example Installation

The book comes with the source code for the examples discussed in the book. You can download the examples zip file from *here* [http://www.redhat.com/docs/jboss]. Unzipping the example code archive creates a JBoss `jboss4guide` directory that contains an `examples` subdirectory. This is the `examples` directory referred to by the book.

The only customization needed before the examples may be used is to set the location of the JBoss server distribution. This may be done by editing the `examples/build.xml` file and changing the `jboss.dist` property value. This is shown in bold below:

```
<project name="JBoss book examples" default="build-all" basedir=".">
    <!-- Allow override from local properties file -->
    <property file="ant.properties"/>

    <!-- Override with your JBoss server bundle dist location -->
    <property name="jboss.dist"        value="/tmp/jboss-4.2"/>
    <property name="jboss.deploy.conf" value="default"/>
    ...
```

or by creating an `.ant.properties` file in the examples directory that contains a definition for the `jboss.dist` property. For example:

```
jboss.dist=/usr/local/jboss/jboss-4.2
```

Part of the verification process validates that the version you are running the examples against matches what the book examples were tested against. If you have a problem running the examples first look for the output of the validate target such as the following:

```
validate:
     [java] ImplementationTitle: JBoss [Zion]
     [java] ImplementationVendor: JBoss Inc.
     [java] ImplementationVersion: 4.2.0 (build: CVSTag=JBoss_4_2_0
date=200704070944)
     [java] SpecificationTitle: JBoss
     [java] SpecificationVendor: JBoss (http://www.jboss.org/)
     [java] SpecificationVersion: 4.2.0
     [java] JBoss version is: 4.2.0
```

Appendix B. Use Alternative Databases with JBoss AS

1. How to Use Alternative Databases

JBoss utilizes the Hypersonic database as its default database. While this is good for development and prototyping, you or your company will probably require another database to be used for production. This chapter covers configuring JBoss AS to use alternative databases. We cover the procedures for all officially supported databases on the JBoss Enterprise Application Platform 4.2. They include: MySQL 5.0, PostgreSQL 8.1, Oracle 9i and 10g R2, DB2 7.2 and 8, Sybase ASE 12.5, as well as MS SQL 2005.

Please note that in this chapter, we explain how to use alternative databases to support all services in JBoss AS. This includes all the system level services such as EJB and JMS. For individual applications (e.g., WAR or EAR) deployed in JBoss AS, you can still use any backend database by setting up the appropriate data source connection as described in *Section 3, "Configuring JDBC DataSources"*.

We assume that you have already installed the external database server, and have it running. You should create an empty database named `jboss`, accessible via the username / password pair `jbossuser / jbosspass`. The `jboss` database is used to store JBoss AS internal data -- JBoss AS will automatically create tables and data in it.

2. Install JDBC Drivers

For the JBoss Application Server and our applications to use the external database, we also need to install the database's JDBC driver. The JDBC driver is a JAR file, which you'll need to copy into your JBoss AS's `jboss-as/server/production/lib` directory. Replace `production` with the server configuration you are using if needed. This file is loaded when JBoss starts up. So if you have the JBoss AS running, you'll need to shut down and restart. The availability of JDBC drivers for different databases are as follows.

- MySQL JDBC drivers can be obtained from *http://dev.mysql.com/downloads/connector/j/5.0.html*. The download contains documentation, etc. for the JDBC connector, but you really only need the `mysql-connector-java-5.0.4-bin.jar` file to get MySQL to work with and be used by JBoss AS.
- PostgreSQL JDBC drivers can be obtained from: *http://jdbc.postgresql.org/*. For 8.2.3 version, we need the JDBC3 driver `8.2-504 JDBC 3`. The download is just the JDBC driver `postgresql-8.2-504.jdbc3.jar` file.
- Oracle thin JDBC drivers can be obtained from: *http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html* for your Oracle DB

versions.

- IBM DB2 JDBC drivers can be downloaded from the IBM web site *http://www-306.ibm.com/software/data/db2/java/*.
- Sybase JDBC drivers can be downloaded from the Sybase jConnect product page *http://www.sybase.com/products/allproductsa-z/softwaredeveloperkit/jconnect*
- MS SQL Server JDBC drivers can be downloaded from the MSDN web site *http://msdn.microsoft.com/data/jdbc/*.

2.1. Special notes on Sybase

Some of the services in JBoss uses null values for the default tables that are created. Sybase Adaptive Server should be configured to allow nulls by default.

```
sp_dboption db_name, "allow nulls by default", true
```

Refer the sybase manuals for more options.

Enable JAVA services. To use any java service like JMS, CMP, timers etc. configured with Sybase, java should be enabled on Sybase Adaptive Server. To do this use:

```
sp_configure "enable java",1
```

Refer the sybase manuals for more information.

If java is not enabled you might see this exception being thrown when you try to use any of the above services.

```
com.sybase.jdbc2.jdbc.SybSQLException: Cannot run this command because Java
services are not enabled. A user with System Administrator (SA) role must
reconfigure the system to enable Java
```

CMP Configuration. To use Container Managed Persistence for user defined Java objects with Sybase Adaptive Server Enterprise the java classes should be installed in the database. The system table 'sysxtypes' contains one row for each extended, Java-SQL datatype.This table is only used for Adaptive Servers enabled for Java. Install java classes using the installjava program.

```
installjava -f <jar-file-name> -S<sybase-server> -U<super-user>
-P<super-pass> -D<db-name>
```

Refer the installjava manual in Sybase for more options.

Installing Java Classes

1. You have to be a super-user with required privileges to install java classes.
2. The jar file you are trying to install should be created without compression.
3. Java classes that you install and use in the server must be compiled with JDK 1.2.2. If you compile a class with a later JDK, you will be able to install it in the server using the installjava utility, but you will get a java.lang.ClassFormatError exception when you attempt to use the class. This is because Sybase Adaptive Server uses an older JVM internally, and hence requires the java classes to be compiled with the same.

3. Creating a DataSource for the External Database

JBoss AS connects to relational databases via datasources. These datasource definitions can be found in the `jboss-as/server/production/deploy` directory. The datasource definitions are deployable just like WAR and EAR files. The datasource files can be recognized by looking for the XML files that end in `*-ds.xml`.

The datasource definition files for all supported external databases can be found in the `jboss-as/docs/examples/jca` directory.

- MySQL: `mysql-ds.xml`
- PostgreSQL: `postgres-ds.xml`
- Oracle: `oracle-ds.xml`
- DB2: `db2-ds.xml`
- Sybase: `sybase-ds.xml`
- MS SQL Server: `mssql-ds.xml`

The following code snippet shows the `mysql-ds.xml` file as an example. All the other `*-ds.xml` files are very similiar. You will need to change the `connection-url`, as well as the `user-name` / `password`, to fit your own database server installation.

```
<datasources>
  <local-tx-datasource>
    <jndi-name>MySqlDS</jndi-name>
    <connection-url>jdbc:mysql://localhost:3306/jboss</connection-url>
```

```
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <user-name>jbossuser</user-name>
    <password>jbosspass</password>
    <exception-sorter-class-name>
org.jboss.resource.adapter.jdbc.vendor.MySQLExceptionSorter
                </exception-sorter-class-name>
    <!-- should only be used on drivers after 3.22.1 with "ping" support
    <valid-connection-checker-class-name>
org.jboss.resource.adapter.jdbc.vendor.MySQLValidConnectionChecker
                </valid-connection-checker-class-name>
    -->
    <!-- sql to call when connection is created
    <new-connection-sql>some arbitrary sql</new-connection-sql>
      -->
    <!-- sql to call on an existing pooled connection when it is obtained
from pool -
                MySQLValidConnectionChecker is preferred for newer drivers
    <check-valid-connection-sql>some arbitrary
sql</check-valid-connection-sql>
      -->

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml
(optional) -->
    <metadata>
       <type-mapping>mySQL</type-mapping>
    </metadata>
  </local-tx-datasource>

</datasources>
```

Once you customized the `*-ds.xml` file to connect to your external database, you need to copy it to the `jboss-as/server/production/deploy` directory. The database connection is now available through the JNDI name specified in the `*-ds.xml` file.

4. Change Database for the JMS Services

The JMS service in the JBoss AS uses relational databases to persist its messages. For improved performance, we should change the JMS service to take advantage of the external database. To do that, we need to replace the file `jboss-as/server/production/deploy/jms-singleton/hsqldb-jdbc2-service.xml` with a file in `jboss-as/docs/examples/jms/` depending on your external database. Notice that if you are using the `default` server profile, the file path is `jboss-as/server/default/deploy/jms/hsqldb-jdbc2-service.xml`.

- MySQL: `mysql-jdbc2-service.xml`
- PostgreSQL: `postgres-jdbc2-service.xml`

- Oracle: `oracle-jdbc2-service.xml`
- DB2: `db2-jdbc2-service.xml`
- Sybase: `sybase-jdbc2-service.xml`
- MS SQL Server: `mssql-jdbc2-service.xml`

What about the hsqldb-jdbc-state-service.xml file?

Despite its name, the `hsqldb-jdbc-state-service.xml` file applies to all databases. So, there is no need to use a special `jdbc-state-service.xml` for each database.

5. Support Foreign Keys in CMP Services

Next, we need to go change the `jboss-as/server/production/conf/standardjbosscmp-jdbc.xml` file so that the `fk-constraint` property is `true`. That is needed for all external databases we support on the JBoss Enterprise Application Platform. This file configures the database connection settings for the EJB2 CMP beans deployed in the JBoss AS.

```
<fk-constraint>true</fk-constraint>
```

6. Specify Database Dialect for Java Persistence API

The Java Persistence API (JPA) entity manager can save EJB3 entity beans to any backend database. Hibernate provides the JPA implementation in JBoss AS. In order for Hibernate to work correctly with alternative databases, we recommend you configure the database dialect in the `jboss-as/server/production/deploy/ejb3.deployer/META-INF/persistence.properties` file. You need to un-comment the `hibernate.dialect` property and change its value to the following based on the database you setup.

- Oracle 9i: org.hibernate.dialect.Oracle9iDialect
- Oracle 10g: org.hibernate.dialect.Oracle10gDialect
- Microsoft SQL Server 2005: org.hibernate.dialect.SQLServerDialect

- PostgresSQL 8.1: org.hibernate.dialect.PostgreSQLDialect
- MySQL 5.0: org.hibernate.dialect.MySQL5Dialect
- DB2 8.0: org.hibernate.dialect.DB2Dialect
- Sybase ASE 12.5: org.hibernate.dialect.SybaseDialect

DB2 7.2 with Universal JDBC Driver (Type 4)

Large Objects (LOBs) are supported only with DB2 Version 8 servers and above with the universal JDBC driver. Hence JMS services which stores messages as BLOBS and Timer services which uses BLOB fields for storing objects do not work with the JDBC Type 4 driver and DB2 7.2.

DB2 7.2 with JDBC Type 2 driver

All JBoss services work with the JDBC Type 2 driver and DB2 Version 7.2 servers.

7. Change Other JBoss AS Services to Use the External Database

Besides JMS, CMP, and JPA, we still need to hook up the rest of JBoss services with the external database. There are two ways to do it. One is easy but inflexible. The other is flexible but requires more steps. Now, let's discuss those two approaches respectively.

7.1. The Easy Way

The easy way is just to change the JNDI name for the external database to `DefaultDS`. Most JBoss services are hard-wired to use the `DefaultDS` by default. So, by changing the datasource name, we do not need to change the configuration for each service individually.

To change the JNDI name, just open the `*-ds.xml` file for your external database, and change the value of the `jndi-name` property to `DefaultDS`. For instance, in `mysql-ds.xml`, you'd change `MySqlDS` to `DefaultDS` and so on. You will need to remove the `jboss-as/server/production/deploy/hsqldb-ds.xml` file after you are done to avoid duplicated `DefaultDS` definition.

In the `jms/*-jdbc2-service.xml` file, you should also change the datasource name in the `depends` tag for the `PersistenceManagers` MBean to `DefaultDS`. For instance, for `mysql-jdbc2-service.xml` file, we change the `MySqlDS` to `DefaultDS`.

```
... ...
<mbean code="org.jboss.mq.pm.jdbc2.PersistenceManager"
       name="jboss.mq:service=PersistenceManager">
  <depends optional-attribute-name="ConnectionManager">
    jboss.jca:service=DataSourceBinding,name=DefaultDS
  </depends>
  ... ...
```

7.2. The More Flexible Way

Changing the external datasource to `DefaultDS` is convenient. But if you have applications that assume the `DefaultDS` always points to the factory-default HSQL DB, that approach could break your application. Also, changing `DefaultDS` destination forces all JBoss services to use the external database. What if you want to use the external database only on some services?

A safer and more flexible way to hook up JBoss AS services with the external datasource is to manually change the `DefaultDS` in all standard JBoss services to the datasource JNDI name defined in your `*-ds.xml` file (e.g., the `MySqlDS` in `mysql-ds.xml` etc.). Below is a complete list of files that contain `DefaultDS`. You can update them all to use the external database on all JBoss services or update some of them to use different combination of datasources for different services.

- `jboss-as/server/production/conf/login-config.xml`: This file is used in Java EE container managed security services.
- `jboss-as/server/production/conf/standardjbosscmp-jdbc.xml`: This file configures the CMP beans in the EJB container.
- `jboss-as/server/production/deploy/ejb-deployer.xml`: This file configures the JBoss EJB deployer.
- `jboss-as/server/production/deploy/schedule-manager-service.xml`: This file configures the EJB timer services.
- `jboss-as/server/production/deploy/snmp-adaptor.sar/attributes.xml`: This file is used by the SNMP service.
- `jboss-as/server/production/deploy/juddi-service.sar/META-INF/jboss-service.xml`: This file configures the UUDI service.
- `jboss-as/server/production/deploy/juddi-service.sar/juddi.war/WEB-INF/jboss-web.xml`: This file configures the UUDI service.
- `jboss-as/server/production/deploy/juddi-service.sar/juddi.war/WEB-INF/juddi.properties`:

This file configures the UUDI service.

- `jboss-as/server/production/deploy/uuid-key-generator.sar/META-INF/jboss-service.xml`: This file configures the UUDI service.
- `jboss-as/server/production/jms/hsqldb-jdbc-state-service.xml` and `jboss-as/server/all/deploy-hasingleton/jms/hsqldb-jdbc-state-service.xml`: Those files configure the JMS persistence service as we discussed earlier.

8. A Special Note About Oracle DataBases

In our setup discussed in this chapter, we rely on the JBoss AS to automatically create needed tables in the external database upon server startup. That works most of the time. But for databases like Oracle, there might be some minor issues if you try to use the same database server to back more than one JBoss AS instance.

The Oracle database creates tables of the form `schemaname.tablename`. The `TIMERS` and `HILOSEQUENCES` tables needed by JBoss AS would not get created on a schema if the table already exists on a different schema. To work around this issue, you need to edit the `jboss-as/server/production/deploy/ejb-deployer.xml` file to change the table name from `TIMERS` to something like `schemaname2.tablename`.

```
... ...
  <mbean code="org.jboss.ejb.txtimer.DatabasePersistencePolicy"
name="jboss.ejb:service=EJBTimerService,persistencePolicy=database">
    <!-- DataSourceBinding ObjectName -->
    <depends optional-attribute-name="DataSource">
      jboss.jca:service=DataSourceBinding,name=DefaultDS
    </depends>
    <!-- The plugin that handles database persistence -->
    <attribute name="DatabasePersistencePlugin">
      org.jboss.ejb.txtimer.GeneralPurposeDatabasePersistencePlugin
    </attribute>
    <!-- The timers table name -->
    <attribute name="TimersTable">TIMERS</attribute>
  </mbean>
```

Similarly, you need to change the `jboss-as/server/production/deploy/uuid-key-generator.sar/META-INF/jboss-service.xml` file to change the table name from `HILOSEQUENCES` to something like `schemaname2.tablename` as well.

```
... ...
  <!-- HiLoKeyGeneratorFactory -->
  <mbean
```

```
code="org.jboss.ejb.plugins.keygenerator.hilo.HiLoKeyGeneratorFactory"
         name="jboss:service=KeyGeneratorFactory,type=HiLo">

     <depends>jboss:service=TransactionManager</depends>

     <!-- Attributes common to HiLo factory instances -->

     <!-- DataSource JNDI name -->
     <depends
optional-attribute-name="DataSource">jboss.jca:service=DataSourceBinding,name=DefaultDS</depe

     <!-- table name -->
     <attribute name="TableName">HILOSEQUENCES</attribute>

     ... ...
```