Edition

1

Date: 2002-02-06, 6:59:46 PM

DAIN SUNDSTROM

The JBoss Group

JBossCMP Workbook

DAIN SUNDSTROM, AND THE JBOSS GROUP

JBossCMP Workbook

2520 Sharondale Dr.
Atlanta, GA 30305 USA
sales@jbossgroup.com

Table of Content

Table of Listings

Table of Figures

Preface

Forward

JBossCMP is a powerful persistence engine compliant with the EJB 2.0 CMP 2.0 specification. CMP 2.0 builds on the framework of CMP 1.0 by adding container managed relationships and a common entity query language EJB-QL. CMP 2.0 does not necessarily simplify the coding and declaration of relationships and queries, but rather is a standard which facilities with portability.

Intended Audience

This workbook explains how to configure JBossCMP for CMP 2.0. Specifically, it includes an introduction to each feature, along with its configuration, and a guide to specifying the database mapping of container managed data. Although the general reader my find this intellectually stimulating, the configuration of JBossCMP is not required to simply deploy and run an EJB 2.0 application; therefore, this workbook is intended primarily for those interested in using advanced features or specifying an exact database mapping, both of which require further configuration.

This workbook assumes that the reader is familiar with Java, EJB and JBoss. It does not assume familiarity with JAWS, the JBoss CMP 1.1 persistence engine, although such familiarity would be helpful. For those without CMP experience, general CMP 2.0 coding and declaration are covered, although this workbook is by no means a complete introduction to CMP.

Organization

Each chapter of this workbook covers a specific feature of CMP 2.0. The first section of a chapter quickly introduces the feature, describes the java code required, and explains the declaration of the element in the ejb-jar.xml file. The remaining sections describe JBossCMP features and configuration. A short description of each chapter follows:

- **Chapter 1, Setup.** This chapter explains the setup of configuration files relevant to JBossCMP.

- **Chapter 2, Entity**. This chapter explains the configuration of entities with the exception of cmp-fields, cmr-fields, and queries, which are described in separate chapters.
- **Chapter 3, Container Managed Persistent Fields**. This chapter explains the configuration of cmp-fields and focuses on the new features such as eager/lazy loading, read-only fields, and dependent value classes.
- **Chapter 4, Container Managed Relationships**. This chapter explains container managed relationships and the configuration of relationships for JBossCMP. The chapter focuses on the specification of the relationship database mapping.
- **Chapter 5, Queries**. This chapter explains the declaration of queries for finder and ejbSelect methods, and how to override the EJB-QL to SQL mapping.
- **Appendix A, Defaults**. This chapter explains the configuration of JBossCMP default options.
- **Appendix B, Datasource Customization**. This chapter explains the configuration of a datasource

About the Authors

Dain Sundstrom, is an independent consultant in Minneapolis, MN (USA). He graduated from the University of Minnesota with a Bachelors of Science in Computer Science in 1996, and shortly thereafter was introduced to distributed computing (Entera DCE) while working at United Health Care. After venturing into the crazy world of Internet start-ups, he has finally decided to settle down as an independent consultant. During a bragging session at JavaOne 2001, Dain claimed "it wouldn't be that difficult to implement the new CMP 2.0 spec," to which Rickard Öberg responded obligingly "then why don't you do it?" Four months later (working 60+ hours a week), Dain still claimed "it wasn't that difficult."

JBoss Group LLC, headed by Marc Fleury, is composed of over 1000 developers worldwide who are working to deliver a full range of J2EE tools, making JBoss the premier Enterprise Java application server for the Java 2 Enterprise Edition platform.

JBoss is an Open Source, standards-compliant, J2EE application server implemented in 100% Pure Java. The JBoss/Server and complement of products are delivered under a public license. With 50,000 downloads per month, JBoss is the most downloaded J2EE based server in the industry.

Chapter 1

1. Setup

This chapter explains the setup of configuration files relevant to JBossCMP.

CMP 2.0 is a core feature of JBoss 3.0, and is the default persistence manager for EJB 2.0 applications. No action beyond the basic JBoss installation (see JBoss documentation) is required to use CMP 2.0, but there are some details to note when creating a new EJB 2.0 application or when upgrading an EJB 1.1 application.

When JBoss deploys an EJB jar file, it uses the DOCTYPE of `ejb-jar.xml` deployment descriptor to determine the version of the EJB jar. The correct DOCTYPE for the EJB 2.0 deployment descriptor is given in Listing 1-1.

Listing 1-1, The XML document type for the EJB2.0 ejb-jar.xml deployment descriptor.

```
<!DOCTYPE ejb-jar PUBLIC
      "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
      "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
```

If the public id of the DOCTYPE starts with "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0" JBossCMP will default to use the "Standard CMP 2.x EntityBean" configuration in the `standardjboss.xml` file. If you have an application which used a custom entity bean configuration, and you are upgrading to EJB 2.0, you must change the `persistence-manager` and add the new interceptors (see the "Standard CMP 2.x EntityBean" configuration in the `standardjboss.xml` file for details).

No further configuration is necessary to deploy and run you EJB 2.0 application successfully. The next sections will detail how to specify database behavior and the exact database mapping.

The optional configuration of JBossCMP will be familiar to users of JAWS, the default persistence engine in JBoss 2.x. The configuration information for JBossCMP is contained in the `jbosscmp-jdbc.xml` located in the META-INF directory of the EJB jar.

Chapter

2

2. Entity

This chapter explains the configuration of entities with the exception of cmp-fields, cmr-fields, and queries, which are described in separate chapters.

Although several new features have been added and there have been major changes to cmp-fields and finders, the basic entity bean structure has not changed much in CMP 2.0.

Entity Classes

A new feature of EJB 2.0 is the addition of local interfaces. Local interfaces are conceptually the same thing as the remote interface and home interface (sometimes referred to as the remote home), except that local interfaces are only accessible in the same Java VM. This allows local interfaces to use pass by reference semantics, removing the overhead associated with serializing and deserializing every method parameter. Local interfaces are not unique to CMP and are not discussed in this workbook. The simplified code for a User entity follows in Listing 2-1:

Listing 2-1, Sample code for a User entity

```
// User Remote Home Interface
public interface UserHome extends EJBHome {
   public User create(String userId)
      throws RemoteException, CreateException;
   public User findByPrimaryKey(String userId)
      throws RemoteException, FinderException;
   ...
}
```

```
// User Remote Interface
public interface User extends EJBObject {
   public String getUserId() throws RemoteException;
   ...
}

// User Local Home Interface
// Note: local interfaces don't throw RemoteException
public interface UserLocalHome extends EJBLocalHome {
   public UserLocal create(String userId) throws CreateException;
   public UserLocal findByPrimaryKey(String userId) throws FinderException;
   ...
}

// User Local Interface
public interface UserLocal extends EJBLocalObject {
   public String getUserId();
   ...
}

public abstract class UserBean implements EntityBean {
   transient private EntityContext ctx;

   public String ejbCreate(String userId) {
      setUserId(userId);
      return null;
   }

   public void ejbPostCreate(String userId) { }
```

```
   // new cmp-field abstract accessor  (see cmp-field chapter)
   public abstract String getUserId();
   public abstract void setUserId(String userId);

   public void setEntityContext(EntityContext ctx) { this.ctx = ctx; }

   public void unsetEntityContext() { this.ctx = null; }

   public void ejbActivate() { }
   public void ejbPassivate() { }
   public void ejbLoad() { }
   public void ejbStore() { }
   public void ejbRemove() { }
}
```

Entity Declaration

The declaration of an entity in the `ejb-jar.xml` file has not changed much in CMP 2.0. The complete declaration of the `UserEJB` with the new elements flagged is as shown in Listing 2-2.

Listing 2-2, The ejb-jar.xml descriptor for the User entity bean of Listing 2-1.

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC
   "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
   "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">

<ejb-jar>
   <enterprise-beans>
      <entity>
         <description>User</description>
```

```
         <ejb-name>UserEJB</ejb-name>
         <home>org.jboss.docs.cmp2.commerce.UserHome</home>
         <remote>org.jboss.docs.cmp2.commerce.User</remote>

         <!-- new -->
         <local-home>org.jboss.docs.cmp2.commerce.UserLocalHome</local-home>
         <local>org.jboss.docs.cmp2.commerce.UserLocal</local>

         <ejb-class>org.jboss.docs.cmp2.commerce.UserBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>java.lang.String</prim-key-class>
         <reentrant>False</reentrant>

         <!-- new -->
         <cmp-version>2.x</cmp-version>
         <abstract-schema-name>user</abstract-schema-name>

         <cmp-field><field-name>userId</field-name></cmp-field>
         <primkey-field>userId</primkey-field>
      </entity>
   </enterprise-beans>
</ejb-jar>
```

The `local-home` and `local` elements are equivalent to the `home` and `remote` elements. The `cmp-version` element is new and can be either 1.x or 2.x, which is the default. This element was added so 1.x and 2.x entities could be mixed in the same application. The abstract-schema-name element is also new and is used to identify this entity type in EJB-QL queries, which are discussed in "Chapter 5, Queries".

Entity Mapping

The JBossCMP configuration for the entity is declared with an `entity` element in `jbosscmp-jdbc.xml` file. The entity elements are grouped together in the `enterprise-`

beans element under the top-level jboss-cmp element. An example entity configuration is shown in Listing 2-3.

Listing 2-3, An example jbosscmp-jdbc.xml descriptor.

```
<jbosscmp-jdbc>
   <enterprise-beans>
      <entity>
         <ejb-name>UserEJB</ejb-name>
         <table-name>USER</table-name>
         <datasource>java:/DefaultDS</datasource>
         <type-mapping>Hypersonic SQL</type-mapping>
         <debug>true</debug>
         <create-table>true</create-table>
         <remove-table>false</remove-table>
         <read-only>false</read-only>
         <time-out>300</time-out>
         <select-for-update>false</select-for-update>
         <pk-constraint>false</pk-constraint>
         <read-ahead>true</read-ahead>
         ...
      </entity>
   </enterprise-beans>
</jbosscmp-jdbc>
```

All of the elements except for ejb-name, which is used to match the configuration to an entity declared in the ejb-jar.xml file, are optional. Unless noted otherwise, the default values come from the defaults section, which is discussed in "Appendix A, Defaults". A detailed description of each option is as follows:

- **ejb-name**: This is the name of the EJB for which this configuration applies. This element is required and must match an ejb-name of the entity in the ejb-jar.xml file.

- **table-name**: This 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 value is the `ejb-name`.

- **datasource**: This is the jndi-name used to lookup 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.

- **type-mapping**: This 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-mapping is discussed in Appendix B.

- **debug**: This option controls the amount of information that is written to the log file. If true, all sql will be written to the log before it is executed.

- **create-table**: If true, JBossCMP will attempt to create a table for the entity. When the application is deployed, JBossCMP 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.

- **remove-table**: If true, JBossCMP will attempt to drop the table for each entity and each relation-table mapped relationship. When the application is undeployed, JBossCMP will attempt to drop the table. This option is very useful during the early stages of development when the table structure changes often.

- **read-only**: Specifies that the bean provider cannot change the value of any fields. A field that is read-only will not be stored in, or inserted into the database. If a pk 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. The read-only option can be overridden on a per cmp-field basis, which is discussed in "Chapter 3, Container Managed Persistent Fields".

- **time-out**: Specifies the amount of time in ms that a read on a read-only field is valid. A value of 0 means that the value doesn't time out. This option can also be overridden on a per cmp-field basis.

- **select-for-update**: Specifies that JBossCMP should use the `SELECT FOR UPDATE` syntax when loading the entity. The select for update syntax is only supported in some databases and is useful for locking data in the datasource.

- **pk-constraint**: Specifies that JBossCMP should add a primary key constraint when creating tables.
- **read-ahead**: Controls caching of query results for the entity. This option is discussed in "Chapter 5, Queries"

Chapter 3

3. Container Managed Persistent Fields

This chapter explains the configuration of cmp-fields and focuses on the new features such as eager/lazy loading, read-only fields, and dependent value classes.

Although CMP fields have not changed in CMP 2.0 with regards to functionality, they are no longer declared using fields in the bean implementation class. In CMP 2.0, cmp-fields are not directly accessible, rather the value is set and retrieved from abstract accessor functions. This small change in field declaration enables an EJB container to implement new advanced features such as eager/lazy loading and read-only fields.

Abstract Accessors

Each CMP field is declared in the bean implementation class of the entity with a set of abstract accessor methods. Abstract accessors are similar to JavaBean property accessors, except no implementation is given. For example, the following declares the orderStatus cmp field on the order entity:

Listing 3-1, An example abstract accessor declaration in the CMP OrderBean implementation.

```
public abstract class OrderBean implements EntityBean {

   public abstract String getOrderStatus();

   public abstract void setOrderStatus(String orderStatus);


   ...
}
```

Each cmp-field is required to have both a getter and a setter method and each accessor method must be declared public abstract.

CMP-Field Declaration

The declaration of a CMP field in the ejb-jar.xml file has not changed at all in EJB 2.0. For example, to declare the orderStatus field defined above you would add the following to the ejb-jar.xml file:

Listing 3-2, An example jbosscmp-jdbc.xml descriptor illustrating the orderStatus cmp-field declaration.

```
<jbosscmp-jdbc>
   <enterprise-beans>
      <entity>
         <ejb-name>OrderEJB</ejb-name>
       ...
         <cmp-field><field-name>orderStatus</field-name></cmp-field>
      </entity>
   </enterprise-beans>
</jbosscmp-jdbc>
```

The basic configuration of cmp-fields has not changed in JBossCMP, but new options have been added to control the loading of field data from the database.

CMP-Field Column Mapping

The mapping of a cmp-field to a database column is declared in a cmp-field element within the entity. A example cmp-field mapping follows:

Listing 3-3, An example jbosscmp-jdbc.xml descriptor illustrating the orderNumber and orderStatus cmp-fields to database column mapping.

```
<jbosscmp-jdbc>
   <enterprise-beans>
      <entity>
         <ejb-name>OrderEJB</ejb-name>
         <table-name>ORDER_DATA</table-name>
         <cmp-field>
            <field-name>orderNumber</field-name>
            <column-name>ORDER_NUMBER</column-name>
```

```
         </cmp-field>
         <cmp-field>
            <field-name>orderStatus</field-name>
            <column-name>ORDER_STATUS</column-name>
            <jdbc-type>VARCHAR</jdbc-type>
            <sql-type>VARCHAR(50)</sql-type>
         </cmp-field>
      </entity>
   </enterprise-beans>
</jbosscmp-jdbc>
```

In the cmp-field element you can control the name and datatype of the column. A detailed description of the options follows:

- **field-name**: The name of the cmp-field that is being configured. This must match the name of a cmp-field declared for this entity in the ejb-jar.xml file.
- **column-name**: The name of the column to which this cmp-field is mapped. The default is the field-name.
- **jdbc-type**: This is the JDBC type that is used when setting parameters in a JDBC PreparedStatement or loading data from a JDBC ResultSet for this cmp-field. The valid types are defined in java.sql.Types. Note: if this option is specified sql-type must also be specified. The default value is determined by the type-mapping for this entity.
- **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. Note: if this option is specified jdbc-type must also be specified. The default value is determined by the type-mapping for this entity.

Eager and Lazy Loading

One of the most important changes in CMP 2.0 is the change from using class fields for cmp-fields to abstract accessor methods. In CMP 1.x, the container could not know which fields were required in a transaction, so the container had to eager-load every field when loading the bean. In CMP 2.x, the container creates the implementation for the abstract accessors, so the container can know when the data for a field is required. JBossCMP can be configured to eager-load only some of the fields when loading an entity. The cmp-field to be eager-loaded are declared in an eager-load element in the entity element. An example follows:

Listing 3-4, An example jbosscmp-jdbc.xml descriptor illustrating the eager-load element.

```
<jbosscmp-jdbc>
   <enterprise-beans>
      <entity>
         <ejb-name>ProductEJB</ejb-name>
       ...

         <eager-load>
            <field-name>name</field-name>
            <field-name>type</field-name>
         </eager-load>
      </entity>
   </enterprise-beans>
</jbosscmp-jdbc>
```

In the Listing 3-4 example, when the ProductEJB is loaded it will only load the name and type fields. If an eager-load element is not present in the entity declaration, JBossCMP will eager-load all fields. If the eager-load element contains no field-name elements, no fields will be eager-loaded.

Lazy-loading is the other half of eager-loading. If a field is not eager-loaded it must be lazy-loaded. Lazy-loaded fields are organized into groups. When the bean accesses an unloaded field, JBossCMP loads the field and any field in any group of which the unloaded field is a member. The JBossCMP performs a set join and then removes any field that is already loaded. The lazy-load-groups element should follow the eager-load element in the entity. An example is:

Listing 3-5, An example jbosscmp-jdbc.xml descriptor illustrating the lazy-load-groups element.

```
<jbosscmp-jdbc>
   <enterprise-beans>
      <entity>
         <ejb-name>ProductEJB</ejb-name>
         ...
```

```
         <lazy-load-groups>
            <lazy-load-group>
               <desription>pricing info</desription>
               <field-name>unit</field-name>
               <field-name>costPerUnit</field-name>
               <field-name>weight</field-name>
            </lazy-load-group>
            <lazy-load-group>
               <desription>shipping info</desription>
               <field-name>weight</field-name>
               <field-name>length</field-name>
               <field-name>girth</field-name>
            </lazy-load-group>
         </lazy-load-groups>
      </entity>
   </enterprise-beans>
</jbosscmp-jdbc>
```

In the above example, when the bean provider calls getUnit(), JBossCMP loads unit, costPerUnit and weight (assuming they are not already loaded). When the bean provider calls getWeight(), unit, costPerUnit, weight, length, and girth.

Read-only Fields

Another benefit of abstract accessors for cmp fields is the ability to have read-only fields. JAWS supported read-only with time-out for entities. The problem with CMP 1.x was the bean provider could always set a field on a read-only entity, and there was nothing the container could do. With CMP 2.x the container provides the implementation for the accessor, and therefore can throw an exception when the bean provider attempts to set the value of a read-only bean.

In JBossCMP this feature has been extended to the field level with the addition of the read-only and time-out elements to the cmp-field element. These elements work the same way as they do at the entity level. If a field is read-only, it 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 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. An read-only cmp-field declaration example follows:

Listing 3-6, An example jbosscmp-jdbc.xml descriptor illustrating the read-only element.

```
<jbosscmp-jdbc>
   <defaults>
      ...
   </defaults>

   <enterprise-beans>
      <entity>
         <ejb-name>OrderEJB</ejb-name>
         <!-- default overrides -->

         <cmp-field>
            <field-name>lastUpdated</field-name>
            <read-only>true</read-only>

            <!-- optional, in ms -->
            <time-out>1000</time-out>
         </cmp-field>
      </entity>
   </enterprise-beans>
</jbosscmp-jdbc>
```

Dependent Value Classes (DVC)

A Dependent Value Class (DVC) is defined as any java class that is the type of a CMP fields, other then the automatically recognized types. See section 10.3.3 of EJB2.0 final draft for further requirements. By default a DVC is serialized, and the serialized form 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. JBossCMP 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. For example, an order table with the fields ship_line1, ship_line2, ship_city..., and an additional set of fields for the billing address. Another not uncommon database structure has telephone numbers with area code in a separate field from base phone number or a person's name spread across several fields. With a dependent value class multiple fields can be mapped to one logical JavaBean.

> Note: Dependent value classes are not the same thing as dependent value objects that were added in EJB 2.0 Proposed Final Draft 1 and subsequently removed in EJB 2.0 Proposed Final Draft 2.

A DVC to be mapped must follow the JavaBeans naming specification in that each property that will be stored in the database must have both a getter and a setter. The bean must be serializable and must have a no-arg constructor. The properties can be any simple type, an unmapped DVC or a mapped DVC, but it cannot be an EJB.

As an example, consider the declaration of an address DVC and a credit card DVC as shown in Listing 3-7.

Listing 3-7, An example jbosscmp-jdbc.xml descriptor illustrating an address and credit card dependent value classes.

```
<jbosscmp-jdbc>
   ...
   <dependent-value-classes>
      <dependent-value-class>
         <description>Formal Name</description>
         <class>org.jboss.docs.cmp2.commerce.FormalName</class>
         <property>
            <property-name>first</property-name>
            <column-name>FIRST</column-name>
         </property>
         <property>
            <property-name>mi</property-name>
            <column-name>MI</column-name>
         </property>
         <property>
```

```
            <property-name>last</property-name>
            <column-name>LAST</column-name>
         </property>
      </dependent-value-class>
      <dependent-value-class>
         <description>Credit Card</description>
         <class>org.jboss.docs.cmp2.commerce.Card</class>
         <property>
            <property-name>type</property-name>
            <column-name>TYPE</column-name>
         </property>
         <property>
            <!-- type FormalName -->
            <property-name>cardHolder</property-name>
            <column-name>CARD_HOLDER</column-name>
         </property>
         <property>
            <property-name>billingZip</property-name>
            <column-name>BILLING_ZIP</column-name>
         </property>
         <property>
            <property-name>cardNumber</property-name>
            <column-name>CARD_NUMBER</column-name>
         </property>
      </dependent-value-class>
   </dependent-value-classes>
</jbosscmp-jdbc>
```

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.

The dependent-value-classes section defines the internal structure and default mapping of the classes. When JBossCMP encounters a field that has an unknown type, it searches the list of registered DVC, and if a DVC is found, it persist this field into a set of columns, otherwise the field is stored in serialized form in single column. A DVC can be constructed from other DVC, so when JBossCMP runs into a DVC, it flattens the DVC tree structure it into a set of columns. If the JBossCMP finds a DVC circuit, 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 property column name. If the property is a DVC, the process it repeated. For example, a cmp-field of type Card from above and named cc will have the following columns:

```
cc_TYPE
cc_CARD_HOLDER_FIRST
cc_CARD_HOLDER_MI
cc_CARD_HOLDER_LAST
cc_BILLING_ZIP
cc_CARD_NUMBER
```

The default mappings of columns can be overridden in the entity element as shown in Listing 3-8.

Listing 3-8, An example jbosscmp-jdbc.xml descriptor illustrating how to override the credit card dependent value class cmp-field to database column mappings.

```
<jbosscmp-jdbc>
   <enterprise-beans>
      <entity>
         <ejb-name>OrderEJB</ejb-name>
         ...

         <cmp-field>
            <field-name>creditCard</field-name>
            <column-name>CC</column-name>

            <property>
               <property-name>cardHolder.first</property-name>
```

```
               <column-name>CC_FIRST_NAME</column-name>
               <jdbc-type>VARCHAR</jdbc-type>
               <sql-type>VARCHAR(100)</sql-type>
            </property>
            <property>
               <property-name>cardHolder.mi</property-name>
               <column-name>CC_MI</column-name>
            </property>
            <property>
               <property-name>cardHolder.last</property-name>
               <column-name>CC_LAST_NAME</column-name>
            </property>
            <!-- the rest of the properties can be default -->
         </cmp-field>
   </dependent-value-classes>
</jbosscmp-jdbc>
```

In the Listing 3-8 example, when overriding property info for the entity, you need to refer to the property from a flat perspective as in `cardHolder.first` above.

Chapter 4

4. Container Managed Relationships

This chapter explains container managed relationships and the configuration of relationships for JBossCMP. The chapter focuses on the specification of the relationship database mapping.

Container Managed Relationships 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 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 distributed objects.

There are two basic steps to create a container managed relationship: create the CMR abstract accessors and declare the relationship in the `ejb-jar.xml` file. The following two sections describe these steps.

CMR Abstract Accessors

CMR abstract accessors have the same signatures as cmp-fields, except that multi-valued relationships can only return a java.util.Collection (or java.util.Set) object. At least one of the two entities in a relationship must have a `cmr-field` abstract accessor. For example, to declare a one-to-many relationship between Order and LineItem, add the following to the OrderBean class:

```
public abstract class OrderBean implements EntityBean {
   ...
```

```
   public abstract Set getLineItems();
   public abstract void setLineItems(Set lineItems);
}
```

and add the following to the LineItem bean class:

```
public abstract class LineItemBean implements EntityBean {
   ...
   public abstract Order getOrder();
   public abstract void setOrder(Order order);
}
```

Although, in the above example, each bean declared a cmr-field, only one of the two beans in a relationship must have a cmr-field. As with cmp-fields, a cmr-field is required to have both a getter and a setter method.

Relationship Declaration

Relationships are declared with an `ejb-relation` element in the top-level `relationships` element of the `ejb-jar.xml` file. An `ejb-relation` is composed of two `ejb-relationship-role` elements. An `ejb-relationship-role` must always have a `relationship-role-source` and a `multiplicity`. The `ejb-relation` can have an `ejb-relation-name` element, but if it does, the name must be unique in the application. The `ejb-relation-name` is used to match a relationship mapping in the `jbosscmp-jdbc.xml` file. A `relationship-role` may also have a name, but unlike the `ejb-relation-name` the `ejb-relationship-role-name` element is only require to be unique within the relationship. The `relationship-role-source` element contains the `ejb-name` of the entity that has this role. A role `multiplicity` can be either One or Many. The `multiplicity` is Many if the other side of the relationship has a `cmr-field` that is collection valued (i.e., has the type java.util.Collection or java.util.Set), otherwise it has a `multiplicity` of One. If the entity has a `cmr-field` for the relationship, the cmr-field must be declared in the `ejb-relationship-role` element. Additionally, if the `cmr-field` is collection valued, the `cmr-field-type` must be declared. Finally, a role can be set to `cascade-delete`. Cascade deletion is only allowed for a role where the other side of the relation has a multiplicity of one. If `cascade-delete` is enabled for the role, when the parent related entity is deleted the child entity is also deleted. An example declaration for the relationship between Order and LineItem is shown in Listing 4-1.

Listing 4-1, An example of declaring EJB relationships using the ejb-jar.xml descriptor relationships element.

```
<relationships>
   <ejb-relation>
      <ejb-relation-name>Order-LineItem</ejb-relation-name>
      <ejb-relationship-role>

         <!-- exdented to fit on a printed page -->
<ejb-relationship-role-name>order-has-lineitems</ejb-relationship-role-name>

         <multiplicity>One</multiplicity>
         <relationship-role-source>
            <ejb-name>OrderEJB</ejb-name>
         </relationship-role-source>
         <cmr-field>
            <cmr-field-name>lineItems</cmr-field-name>
            <cmr-field-type>java.util.Collection</cmr-field-type>
         </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>

         <!-- exdented to fit on a printed page -->
  <ejb-relationship-role-name>lineitem-belongsto-order</ejb-relationship-role-
name>
         <multiplicity>Many</multiplicity>
         <cascade-delete/>
         <relationship-role-source>
            <ejb-name>LineItemEJB</ejb-name>
         </relationship-role-source>
         <cmr-field>
            <cmr-field-name>order</cmr-field-name>
         </cmr-field>
      </ejb-relationship-role>
```

```
   </ejb-relation>
</relationships>
```

After completing these two steps, the relationship should be functional. For more information on relationships, see section 10.3 of the EJB 2.0 Final Draft. The next section discusses the database mapping of the relationship.

Relationship Mapping

Relationships can be mapped either a using a foreign-key or separate relationship table. One-to-one and one-to-many use the foreign-key mapping style by default, and many-to-many can only use the relation table mapping style. The mapping style for a relationship is declared in the relationships section of the `jbosscmp-jdbc.xml` file. Relationships are identified by the `ejb-relation-name` from the `ejb-jar.xml` file. The beginning of the relationship mapping declaration for Order-LineItem is shown in Listing 4-2.

Listing 4-2, An example jbosscmp-jdbc.xml descriptor illustrating the use of the ejb-relation-name to match the Order-LineItem relationship specified in Listing 4-1.

```
<relationships>
   <ejb-relation>
      <ejb-relation-name>Order-LineItem</ejb-relation-name>
      ...
   </ejb-relation>
</relationships>
```

Foreign-key Mapping

Foreign-key mapping is the most common mapping style for one-to-one and one-to-many relationships. The complete foreign-key mapping for Order-LineItem is given in Listing 4-3.

Listing 4-3, The complete jbosscmp-jdbc.xml descriptor relationships section for the Order-LineItem relationship specified in Listing 4-1.

```
<relationships>
   <ejb-relation>
      <ejb-relation-name>Order-LineItem</ejb-relation-name>
      <foreign-key-mapping>
```

```
         <ejb-relationship-role>

            <!-- exdented to fit on a printed page -->
<ejb-relationship-role-name>order-has-lineitems</ejb-relationship-role-name>

            <foreign-key-fields/>
         </ejb-relationship-role>

         <ejb-relationship-role>

            <!-- exdented to fit on a printed page -->
<ejb-relationship-role-name>lineitem-belongsto-order</ejb-relationship-role-name>

            <foreign-key-fields>
               <foreign-key-field>
                  <field-name>ordernumber</field-name>
                  <column-name>ORDER_NUMBER</column-name>
               </foreign-key-field>
            </foreign-key-fields>
         </ejb-relationship-role>

      </foreign-key-mapping>
   </ejb-relation>
</relationships>
```

As you can see, the foreign-key mapping style is declared by adding the `foreign-key-mapping` element to the relationship. This element contains the mapping information for the two individual roles. If exact field level mapping is not desired, just leave the `foreign-key-mapping` element empty and JBossCMP will automatically generate the proper `foreign-key` fields. Each role is identified by `ejb-relationship-role-name`, and each `ejb-relationship-role` can contain a `foreign-key-fields` element. If no `foreign key fields` are desired for the role, this element is left empty, as in `order-has-lineitems`. Otherwise, the `foreign-key-fields` element contains one or more

foreign-key-field elements. The foreign-key-field element uses the same syntax as the cmp-field element of the entity. The only important note is the field-name element must match the field-name of one of the primary key fields of the related element. This is similar to the REFERENCES clause of a SQL foreign key declaration. In the example above, the lineitem role has a foreign-key for the ordernumber of the order entity.

The foreign-key mapping style is only allowed for one-to-one and one-to-many relationships. In one-to-one relationships one or both roles can have foreign keys. In one-to-many relationships only the many side of the relationship can have foreign keys.

The foreign-key mapping is not dependent on the directionality 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.

If the related entity uses a DVC for a primary key, each property of the primary key must be mapped individually.

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. The complete relation table mapping for Product-ProductCategory is shown in Listing 4-4.

Listing 4-4, The complete jbosscmp-jdbc.xml descriptor relationships section for the Product-ProductCategoryrelationship.

```
<relationships>
   <ejb-relation>
      <ejb-relation-name>Product-ProductCategory</ejb-relation-name>
      <table-mapping>
         <table-name>PRODUCT_PRODUCT_CATEGORY</table-name>
         <create-table>true</create-table>
         <remove-table>true</remove-table>
         <ejb-relationship-role>

            <!-- exdented to fit on a printed page -->
<ejb-relationship-role-name>product-have-productCategories</ejb-relationship-role-
name>
```

```
            <table-key-fields>
               <table-key-field>
                  <field-name>id</field-name>
                  <column-name>PRODUCT_ID</column-name>
               </table-key-field>
            </table-key-fields>
         </ejb-relationship-role>
         <ejb-relationship-role>

            <!-- exdented to fit on a printed page -->
<ejb-relationship-role-name>productCategories-have-product</ejb-relationship-role-
name>

            <table-key-fields>
               <table-key-field>
                  <field-name>id</field-name>
                  <column-name>PRODUCT_CATEGORY_ID</column-name>
               </table-key-field>
            </table-key-fields>
         </ejb-relationship-role>
      </table-mapping>
   </ejb-relation>
</relationships>
```

The `table-mapping` element is quite similar to the `foreign-key-mapping` element, and only the differences will be documented. In the `table-mapping` element the `table-name` can be declared along with the other table options. The table options are the same table options available for entities. Roles in a `table-mapping` have `table-key-fields` like the `foreign-key-mapping` has `foreign-key` fields. Unlike a `foreign-key` field which map the primary keys of the related role, `table-key-fields` map the primary key of the current role. For example, the product role has a `PRODUCT_ID` `table-key-field` and the `ProductCategory` role has a `PRODUCT_CATEGORY_ID`. Table mappings must map the

primary keys of both entities in the relationship, which is different from the `foreign-key mapping`, where only one role must have a mapping.

Chapter 5

5. Queries

This chapter explains the declaration of queries for finder and ejbSelect methods, and how to override the EJB-QL to SQL mapping.

Another powerful new feature of CMP 2.0 is the introduction of the EJB Query Language (EJB-QL) and ejbSelect methods. In CMP 1.1, every EJB container had a different way to specify finders, and this was a serious threat to J2EE portability. In CMP 2.0, EJB-QL was created to specify finders and ejbSelect methods in a platform independent way. The ejbSelect 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, ejbSelect methods can return any entity type or just one field of the entity.

EJB-QL is beyond the scope of this (spectacular) workbook, so only the basic method coding and query declaration will be covered here. For more information, see the **EJB 2.0 Final Draft** chapter 11 or one of the many excellent articles written on CMP 2.0.

Finder and ejbSelect 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 code given in Listing 5-1 illustrates a local home finder declaration.

Listing 5-1, A sample local home interface finder method.

```
public interface OrderHome extends EJBLocalHome {

   ...

   public Collection findWithStatus(String status) throws FinderException;

}
```

The ejbSelect methods are declared in the entity implementation class, and must be public abstract just like cmp and cmr accessors. Select methods must be declared to throw a `FinderException`, but not a `RemoteException`. The code given in Listing 5-2 illustrates an ejbSelect method declaration.

Listing 5-2, A sample ejbSelect declaration.

```
public abstract class OrderBean implements EntityBean {
   ...
   public abstract Set ejbSelectOrdersShippedToCA() throws FinderException;
}
```

EJB-QL Declaration

The EJB 2.0 specification requires that every finder or ejbSelect method have an EJB-QL query defined in the `ejb-jar.xml` file. The EJB-QL is declared in a `query` element, which is contained in the `entity` element. Listing 5-3 gives the declarations for the methods declared in Listing 5-1 and Listing 5-2.

Listing 5-3, The ejb-jar.xml descriptor query elements for the methods declared in Listing 5-1 and Listing 5-2.

```
<ejb-jar>
   <enterprise-beans>
      <entity>
         <ejb-name>OrderEJB</ejb-name>
         ...
         <query>
            <description>Find all orders with the specified status</description>
            <query-method>
               <method-name>findWithStatus</method-name>
               <method-params>
                  <method-param>java.lang.String</method-param>
               </method-params>
            </query-method>
            <ejb-ql>
```

```
                SELECT DISTINCT OBJECT(o)
                FROM Order o
                WHERE o.status=?1
            </ejb-ql>
        </query>
        <query>
            <description>Search for all orders shipped to CA</description>
            <query-method>
                <method-name>ejbSelectOrdersShippedToCA</method-name>
                <method-params>
                </method-params>
            </query-method>
            <ejb-ql>
                SELECT OBJECT(o)
                FROM Order o
                WHERE o.shippingAddress.state = 'CA'
            </ejb-ql>
        </query>
      </entity>
   </enterprise-beans>
</ejb-jar>
```

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).
- EJB-QL is similar to SQL but has some surprising differences. For example, in an equals comparison a variable must be on the left hand side. Examples include:

```
o.shippingAddress.state = 'CA'   Legal
'CA' = o.shippingAddress.state   NOT Legal
'CA' = 'CA'                      NOT Legal
```

- Parameters use a base 1 index like java.sql.PreparedStatement.

Caution: The IS EMPTY operator is not available for MySQL and mSQL, but the IS NOT EMPTY operator is available. This is because MySQL and mSQL do not support the EXISTS clause. In the future it will be possible to map IS EMPTY to a LEFT OUTER JOIN which MySQL supports. Unfortunately mSQL does not support either of these possible mappings, so mSQL will not be able to use IS EMPTY.

EJB-QL to SQL Mapping

The EJB-QL to SQL mapping can be overridden in the `jbosscmp-jdbc.xml` file. The finder or ejbSelect is required to have a query method in the `ejb-jar.xml` file, but the SQL generated can be completely changed. Currently the SQL can only be overridden with declared-sql or a bean managed persistence style ejbFind method. In the future several additional override styles will be implemented, and all will use the following structure:

```
<entity>
   ...

   <query>
      <description>Find all orders with the specified status</description>
      <query-method>
         <method-name>findWithStatus</method-name>
         <method-params>
            <method-param>java.lang.String</method-param>
         </method-params>
      </query-method>
      <!-- override here -->
   </query>
</entity>
```

Note: All EJB-QL overrides are non-standard extensions to the EJB 2.0 specification, so use of these extensions will limit portability of your application.

Declared SQL

Declared SQL is implemented similar to the JAWS finder declaration, except the from and where clauses have been broken up. The `declared-sql` that is equivalent to the findWithStatus above follows:

```
<entity>
   <ejb-name>OrderEJB</ejb-name>
   ...
   <query>
      <description>Find all orders with the specified status</description>
      <query-method>
         <method-name>findWithStatus</method-name>
         <method-params>
            <method-param>java.lang.String</method-param>
         </method-params>
      </query-method>
      <declared-sql>
         <where>STATUS={0}</where>
         <order>ORDER_NUMBER</order>
      </declared-sql>
   </query>
</entity>
```

JBossCMP will generate the select and from clauses necessary to select the primary key for this entity. A from clause can be specified that is appended to the end of the automatically generated from sql. The following example uses a from clause to find all orders shipped to the 12345 zip code:

```
<entity>
   <ejb-name>OrderEJB</ejb-name>
   ...
   <query>
      <description>Search for all orders shipped to the 12345 zip
code</description>
```

```
      <query-method>
         <method-name>ejbSelectOrdersShippedToCA</method-name>
         <method-params>
         </method-params>
      </query-method>
      <declared-sql>
         <from>, address AS a</from>
         <where>order.shippingAddress_id = a.id AND a.zip = 12345</where>
      </declared-sql>
   </query>
</entity>
```

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 common for the from clause to start with a join statement. When a from clause is declared the select clause is generated with the `table_name.field_name` style column declarations, so do not begin a from clause with an AS statement.

The `declared-sql` declaration can also be used for ejbSelect methods. The declaration is exactly the same with the addition of a select clause. The following is an example:

```
<entity>
   <ejb-name>OrderEJB</ejb-name>
   ...

   <query>
      <description>Search all states orders have been shipped to</description>
      <query-method>
         <method-name>ejbSelectOrderShipToStates</method-name>
         <method-params>
         </method-params>
      </query-method>
      <declared-sql>
         <select>
```

```
            <distinct/>
            <ejb-name>FullAddressEJB</ejb-name>
            <field-name>state</field-name>
         </select>
         <from>, OrderEJB as o</from>
         <where>o.shippingAddress_id=FullAddressEJB.id</where>
      </declared-sql>
   </query>
</entity>
```

As stated before the SQL for the select and from clauses is generated, but the objects being selected are declared. The following describes each element of the select clause:

- **distinct**: An optional element that tells JBossCMP to generate a select clause that starts with `SELECT DISTINCT`. If the ejbSelect method returns a `java.util.Set`, this option is automatically set.
- **ejb-name**: This is the `ejb-name` of the entity to select or the name of the entity from which a field will be selected. This element is required.
- **field-name**: This is the name of the `cmp-field` that will be selected from the specified entity. This element is optional.

Parameters

JBossCMP uses a completely new parameter handling system, which supports entity and DVC parameters. Parameters are enclosed in curly brackets and use zero base, which is different from the base one EJB-QL parameters. There are three classes of parameters: simple, DVC, and entity:

- 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 type-mapping for the entity. An unknown DVC is serialized and then set as a parameter.
- 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 property of type `Card` from the section called “Dependent Value Classes (DVC)”, valid parameter declarations would be `{0.cardNumber}` and

`{0.billingAddress.zip}` but not `{0.billingAddress}`. The JDBC type used to set a parameter is based on the class type of the property and the type-mapping of the 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 Order, a valid parameter declaration would be `{0.orderNumber}`. If we had some entity with a primary key field `cc` of type `Card`, a valid parameter declaration would be `{0.cc.billingAddress.zip}`. 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.

Custom Finders

JBossCMP continues the tradition of JAWS in supporting bean managed persistence custom finders. If a custom finder matches a finder declared in the home or local home interface, JBossCMP will always call the custom finder over any other implementation declared in the `ejb-jar.xml` or `jbosscmp-jdbc.xml` files. An example custom finder method follows:

```
public abstract class OrderBean implements EntityBean {
   public Collection ejbFindWithStatus(String status) {
      // return a collection containing primary keys of orders
      return someCollection;
   }
}
```

Read-ahead

The read-ahead option is really an entity level option, but it is discussed here as it only applies to queries. In general, a query that returns entities only returns the primary keys of the entities. It is not until the actual entity is used that the rest of the data for the entity is loaded. A problem arises when the bean provider attempts to access every entity in the collection. In this situation the container would access the database once for the initial query and once for each entity, a highly inefficient process. This is referred to as the 'n+1' problem.

There are several possible solutions for this problem and each has its benefits and drawbacks. JBossCMP uses a plugable strategy engine for these solutions. Currently JBossCMP only supports one strategy, on-load but others are planned. Read-ahead can be

declared in the defaults section and in the entity section. Examples of both read-ahead declaration styles follow:

```
<!-- simple style -->
<ejb-jar>
   <enterprise-beans>
      <entity>
         <description>Models an Order</description>
       ...
         <read-ahead>true<read-ahead>
      </entity>
   </enterprise-beans>
</ejb-jar>

<!-- full style -->
<ejb-jar>
   <enterprise-beans>
      <entity>
         <description>Models an Order</description>
         ...
         <read-ahead>
            <strategy>on-load</strategy>
            <limit>255</limit>
            <cache-size>1000</cache-size>
         </read-ahead>
      </entity>
   </enterprise-beans>
</ejb-jar>
```

A description of each element in the full style follows:

- **strategy**: Selects the strategy to use. Allowed values are on-load and none, which disable read-ahead caching.

- **limit**: Specifies the number of entities that will be read in a single load query. This is effectively the page size.
- **cache-size**: Specifies the number of simultaneous queries that can be served by the cache for this entity.

Appendix

A

Appendix A, Defaults

JBossCMP global defaults are defined in the `standardjbosscmp-jdbc.xml` file of the `conf/<config-name>` directory file in the JBoss 3.0 distribution. 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. An example of the defaults section follows:

```
<jbosscmp-jdbc>
   <defaults>
      <datasource>java:/DefaultDS</datasource>
      <type-mapping>Hypersonic SQL</type-mapping>
      <debug>true</debug>

      <create-table>true</create-table>
      <remove-table>false</remove-table>
      <read-only>false</read-only>
      <time-out>300</time-out>
      <select-for-update>false</select-for-update>
      <preferred-relation-mapping>foreign-key</preferred-relation-mapping>
      <pk-constraint>false</pk-constraint>
      <read-ahead>true</read-ahead>
   </defaults>
</jbosscmp-jdbc>
```

The options can apply to entities, relationships, or both, and can be overridden in the specific entity or relationship. A detailed description of each option is as follows:

- **datasource**: This is the jndi-name used to lookup the datasource. All database connections used by an entity or relation table are obtained from the datasource. This option is applicable to entities and relation tables. Using multiple datasources in an application is not recommended, as it vastly constrains the domain over which finders and ejbSelects can query.

- **type-mapping**: This specifies the name of the default type-mapping. Type-mapping determines how Java types are mapped to sql types, and how EJB-QL functions are mapped to database specific functions. This option applies to entities, and is discussed in "Appendix B, Datasource Customization".

- **debug**: This option controls the amount of information that is written to the log file. If true, all sql will be written to the log before it is executed. This option applies to entities.

- **create-table**: If true, JBossCMP will attempt to create a table for each entity and each relation-table mapped relationship. When the application is deployed, JBossCMP 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. This option applies to entities and relation tables.

- **remove-table**: If true, JBossCMP will attempt to drop the for each entity and each relation-table mapped relationship. When the application is undeployed, JBossCMP will attempt to drop the table. This option is very useful during the early stages of development when the table structure changes often. This option can be overridden on a per entity or per relation-table basis.

- **read-only**: Specifies that by default no field can by changed by the bean provider. A field that is read-only will not be stored in, or inserted into the database. If a pk 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. This option applies to entities and cmp fields.

- **time-out**: Specifies the amount of time in ms that a read on a read-only field is valid. A value of 0 means that the value doesn't time out. This option applies to entities and cmp fields.

- **select-for-update**: Specifies that JBossCMP should use the `SELECT FOR UPDATE` syntax when loading entities. The select for update syntax is only supported in some databases and is useful for locking data in the datasource. This option applies to entities.

- **preferred-relation-mapping**: Is used to determine the default mapping for relationships. The valid options are `foreign-key` and `relation-table`. This option applies to relationships.
- **pk-constraint**: Specifies that JBossCMP should add a primary key constraint when creating tables. This option applies to entities.
- **read-ahead**: Controls caching of query results for entities. This option applies to entities.

Appendix

B

Appendix B, Datasource Customization

This chapter explains the configuration of a datasource.

JBossCMP includes predefined type-mappings for the following databases: InterBase, DB2, Oracle8, Oracle7, Sybase, PostgreSQL, InstantDB, Hypersonic SQL, PointBase, SOLID, mySQL, MS SQLSERVER, MS SQLSERVER2000, DB2/400, SapDB, Cloudscape, and InformixDB. If you don't 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 it to the jboss-dev list.

Type Mapping

A type-mapping is simply a set of mappings between Java class types and database types. The following is the current mapping for a short in for Oracle.

```
<jbosscmp-jdbc>
   <type-mapping>
      <name>Oracle8</name>
      <mapping>
         <java-type>java.lang.Short</java-type>
         <jdbc-type>NUMERIC</jdbc-type>
         <sql-type>NUMBER(5)</sql-type>
      </mapping>
      ...
   </type-mapping>
</jbosscmp-jdbc>
```

If JBossCMP can not find a mapping for a type it will serialize the object and use the java.lang.Object mapping. The following describes the three elements of the `mapping` element:

- **java-type**: This is the fully qualified name of the Java class to be mapped. If the class is a primitive wrapper class such as `java.lang.Short`, then the mapping also applies to the primitive type.
- **jdbc-type**: This is the jdbc type that is used when setting parameters in a JDBC `PreparedStatement` or loading data from a JDBC `ResultSet` for this field. 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 sql-types are only limited by your database vendor.

Function Mapping

EJB-QL contains six functions: `ABS`, `CONCAT`, `SUBSTRING`, `LOCATE`, `LENGTH`, and `SQRT`. By default these functions are mapped to JDBC sql extension scalar functions. For example, `CONCAT('Hot', 'Java')` would map to `{fn concat('Hot', 'Java')}`. Several of the major database vendors don't support this style of functions in an effort to lock users into their database. The mapping for these functions can be overridden by adding function-mapping elements to the `type-mapping` element. The following is an example of the concat function mapping for Oracle.

```
<type-mapping>
      <name>Oracle8</name>
      <function-mapping>
            <function-name>concat</function-name>
            <function-sql>(?1 || ?2)</function-sql>
      </function-mapping>
</type-mapping>
```

Index