Enterprise Java Development on a Budget: Leveraging Java Open Source Technologies

BRIAN SAM-BODDEN AND CHRISTOPHER JUDD

Apress™

Enterprise Java Development on a Budget: Leveraging Java Open Source Technologies
Copyright © 2004 by Brian Sam-Bodden and Christopher Judd

ISBN (pbk): 1-59059-125-9

Printed and bound in the United States of America 10987654321

Technical Reviewer: Jeff Linwood

Editorial Board: Steve Anglin, Dan Appleman, Gary Cornell, James Cox, Tony Davis, John Franklin, Chris Mills, Steve Rycroft, Dominic Shakeshaft, Julian Skinner, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Project Manager: Sofia Marchant

Copy Manager: Nicole LeClerc

Copy Editor: Mark Nigara

Production Manager: Kari Brooks

Production Editor: Janet Vail

Compositor: Susan Glinert

Proofreader: Linda Seifert

Indexer: Rebecca Plunkett

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, e-mail orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, e-mail orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

About the Authors

Brian Sam-Bodden has been working with object technologies for the last nine years with a strong emphasis on the Java platform. He holds dual bachelor degrees from Ohio Wesleyan University in computer science and physics. He currently serves as president and chief software architect for Integrallis Software, LLC (`http://www.integrallis.com`). He has worked as an architect, developer, mentor, and trainer for several Fortune 500 companies in industries such as taxes, insurance, retail sciences, telecommunications, distribution, banking, finance, aviation, and scientific data management. As an independent consultant, Brian has promoted the use of Open Source in the industry by educating his clients on the cost benefit and productivity gains achieved by it. He is a Sun Certified Java programmer, developer, and architect. Brian is a frequent speaker at Java user groups and at conferences throughout the country. You can contact him at `bsbodden@integrallis.com`.

Christopher Judd is the president and primary consultant for Judd Solutions, LLC (`http://www.juddsolutions.com`), and is an international speaker, Open Source evangelist, Central Ohio Java Users Group (`http://www.cojug.org`) board member and JBuilder-certified developer and instructor. He has spent the last nine years developing software in the insurance, retail, government, manufacturing, service, and transportation industries. His current focus is consulting, mentoring, and training with Java, J2EE, J2ME, web services, and related technologies. He also holds a bachelor's degree from Ashland University in computer information systems with minors in accounting and finance. You can contact Chris at `cjudd@juddsolutions.com`.

CHAPTER 6

Data Storage Options

The best way to have a good idea is to have a lot of ideas.[1]

—Linus C. Pauling

DATABASES ARE EVERYWHERE in modern society and represent the backbone of the information age. In a J2EE application, choosing the right database platform can mean the difference between a successful application and a failed one.

How do you go about choosing where to put your data? The criteria for choosing a database are as complex as those for choosing an application server or an operating system. The fact that the largest percent of applications end up using a relational database doesn't automatically mean that it's the best solution to your particular problem. For the majority of corporate applications the data that will be manipulated is already living in a relational database, therefore the choice has already been made for you. But for those applications that you're starting from scratch, there are a lot of ideas you can try. Some of the issues that you should weigh before making a decision about which database to choose for your next J2EE application include the following:

- **Cost:** Proprietary or Open Source? The total cost of ownership (TCO) is what determines the impact on the bottom line. Hidden costs can, depending on your application needs, surpass the initial price tag. Sorting through the licensing and support schemes for proprietary databases can be a daunting experience. Proprietary database vendors have artfully devised myriad schemes based on developer seats, CPUs, concurrent users, and connection modes (intranet and/or Internet) to get the most money out of their customers.

- **Maintenance/Administration:** From installation to day-to-day maintenance, a database administrator (DBA) needs tools to ease the tasks of administration, including security, auditing, backup/restore, replication, recovery, and remote databases. Documentation and support, both traditional and online are important for the daily work of a DBA.

1. Safire, William and Leonard Safir. *Leadership: A Treasury of Great Quotations for Those Who Aspire to Lead* (Galahad Books: September 2000).

- **Performance/Scalability:** Although many performance problems attributed to the database are usually caused by poor database design, poorly structured queries and a lack of indexing (in the case of relational databases), relative database performance is usually more a factor of the architecture and real-time demands of an application. Understanding how your application works with data is the first step toward performance analysis. Scalability on the other hand is a more concrete and quantifiable feature. Do the things that work for a few users work for many? How do you respond to an increase in the demand for resources? Can your database support intelligent partitioning or clustering?

- **Features:** Database vendors compete based on a value-added market for features such as multiuser access, storage transparency, query optimization, transactions and concurrency (locking) controls, stored procedures, XML support, and others.

- **Standards:** Whether you're planning to use a relational database or an object database, a basic adherence to standards can mean the difference between the portability of both data and the code that's manipulating the data. For relational databases and Java the question is to what version of the Structured Query Language (SQL) standard does the database adhere? Is the compliance only for a subset of the SQL features? For object databases, do they follow the Object Data Management Group (ODMG) standards?

- **Channels:** Who is consuming the application's information? Are the target clients web-based, wireless handheld devices, or rich GUI clients? What about connectivity? Are they connected continuously or are their connections intermittent?

- **Productivity:** How would choosing a particular database affect your application development? What's the impact on existing code? What's the learning curve for the technology like? Development man hours can quickly surpass the cost of runtime resources. The combination of documentation, development tools, and an adherence to standards can minimize the risk of switching database technology.

Figure 6-1 shows some of the options available when storing and retrieving data in J2EE.

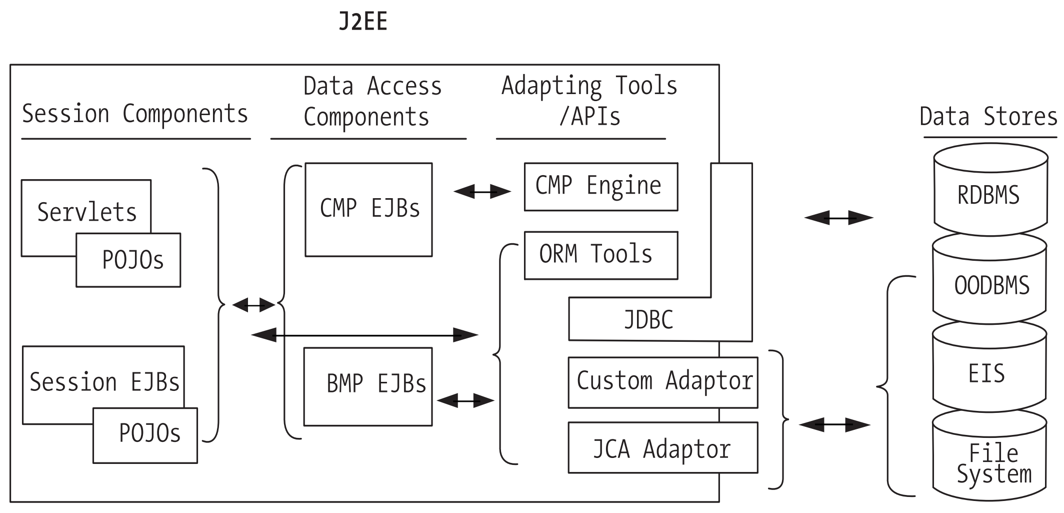


Figure 6-1. Java/J2EE and data storage

Understanding the strengths and weakness of the various choices is a very application-specific task. As you learned in Chapter 2, a solid application design and an understanding of the domain that the data belongs to and the context in which it will be used are the most accurate ways to find requirements for a data-storage technology.

In this chapter some of the open-sourced choices for data storage are covered. In Chapter 7 you'll learn about object-relational mapping (ORM) tools, which can be used to implement your persistence layer when your J2EE applications are confined to work with relational databases.

This chapter begins with some of the pros and cons between choosing a relational or an object database and then moves on to show you some of the choices available from the Open Source community. We use the following four categories of technologies when it comes to storing data:

- Java (embedded) relational database management system (RDBMS)
- Java object-oriented database management system (OODBMS)
- Java XML DBMS (XDBMS)
- Object serialization (including Java Prevalence/Prevayler)

This chapter isn't an attempt to cover all the choices listed in depth, but it's intended as an introduction that should help you gain an understanding of some of the differences between the storage technologies at the conceptual and practical levels, as they apply to the J2EE platform. In particular, this chapter concentrates on the installation, configuration, and usage issues for each tool as they apply to the JBoss application server.

Choosing Between Object and Relational Databases

The relational model proposed by E. F. Codd and the subsequent SQL standard are based on an abstracted model of the data based on the mathematical principles of set theory. In the relational model, data is decoupled from the application logic that uses it. Relational databases provide optimized storage and retrieval of data at the expense of "flattening" the richer semantical connections that the data might have when it's coupled with behavior in the realm of objects. Relational databases are designed with the concept of normalization in mind. Normalization is based on the simple idea that it's more efficient and safer to keep a piece of data in one place only, as proposed by Codd's "Information Rule."

Part of the success of the relational database is due to the SQL standard (an ANSI/ISO standard), which is to an extent the only reason that makes porting an application from one relational database to another a feasible endeavor. Of course, database vendors deviate from the standard in the race to provide market differentiators and value-added features, or when they're simply trying to cover holes left in the standard (such as stored procedures and database triggers).

Comparison of Terms

In the relational model, a relation (a table) is a concept similar to that of an object's class, yet classes can support inheritance and complex composition with statically and dynamically defined datatypes, although, in the relational model, relationships are based on foreign keys. The concept of a tuple (a table row) can be contrasted with an instance of an object, but, although a tuple is a set of values, an object encompasses any type of data and the operations to manipulate it. A column in a database is similar to an object's attribute, but again, in the case of an object, the possible datatypes are only restricted by the programming language base types and the user-defined types.

Relational databases go hand in hand with procedural languages and have been proven to work particularly well when dealing with complex queries, when adding or modifying large volumes of data, or when working with data for which only simple datatypes are required. Relational databases are an obvious choice for data warehousing, and high-volume Online Transaction Processing (OLTP), in which the data is combined and queried in very predictable ways.

Yet for certain domains it has been proven that the relational model falls short, especially when data needs to be manipulated and analyzed in highly complex ways. This is clearly seen in the fields of financial analysis and forecasting, in the chemical and biological sciences, in game theory, network management, process control, computer-aided design/manufacturing (CAD/CAM), and multimedia storage and analysis, among many others. Generally speaking, object-oriented databases are better suited for storing data with complex datatypes and numerous relationships. The differences between the two models are brought to light when you try to use the relational model to store medium to complex object hierarchies given that the concepts of data abstraction, inheritance, and encapsulation can't be easily represented using the relational model.

These differences are the root of the object-relational impedance mismatch that you'll learn how to deal with in Chapter 7. With an object database there's no need for any kind of "mapping" between your objects and their storage format.

The answers to the following questions can guide you in the decision between a relational database and an object-oriented database:

- **Simple data:** If your data has a natural tendency to be organized in table form, then use a relational database.

- **Complex data:** If your data is highly complex and it makes little sense to manipulate it outside of the realm of an object, then use an object database.

- **Transactions:** If you have a high number of concurrent users performing short-lived transactions, then use a relational database.

- **Volume of data:** If you're dealing with large volumes of data that need to be queried in complex ways—often to provide specific pieces of information—and the manipulation of the data is left to the client, then use a relational database.

- **Reporting:** Reporting tools in general use SQL to gather data to produce reports. Ad hoc query tools expose SQL-like constructs so that users can create their own customized reports. Data stored in relational databases is usually easier to manipulate in order to produce tabular reports, therefore, if your business depends on reporting, use a relational database.

- **Legacy concerns:** If your company already has a heavy investment in relational technology, legacy data, legacy applications, and in-house expertise, it's very likely that unless your application can work in isolation you'll have to use a relational database.

NOTE If you're working with an already-designed object model and you're now faced with the decision of what database technology to use, it's important to understand what the guiding forces where when the model was created. You can usually tell this by the granularity of the objects in the model. Typically with data-driven models, you tend to have large, coarse-grained objects that reflect the "normalized" nature of relational databases and look very much like an Entity Relation Diagram (ERD) that has been infused with behavior by the addition of methods. On the other hand, an object-driven model tends to have smaller, finer-grained, more reusable objects that are the result of the process of object-oriented analysis and design (OOAD), as you learned in Chapter 2.

Figure 6-2 shows the relationship between data and query complexity and the choice between a relational and an object database. In Figure 6-2 you can see that as query complexity increases you're better served by a relational database, although increased object model or data complexity calls for an object database. The problem arises when you need a combination of both; for those situations, the safest bet is to use a relational database couple with a strong ORM tool.

There are, however, applications that can benefit from using a combination of both technologies. For example, for data that's only manipulated as objects you can use an object database, though for data used in ad hoc queries, or data that's purely descriptive and doesn't represent the state of an object or for data that is used in objects that are simple data wrappers you can use a relational database. Caching is another area where an in-memory object-database in the middle tier can improve performance without introducing a great deal of complexity. Of course, using both technologies together in certain scenarios requires a tight integration between the relational and object systems, especially if there's a need to share data stored using both technologies. In such cases, issues such as data synchronization and replication become relevant.

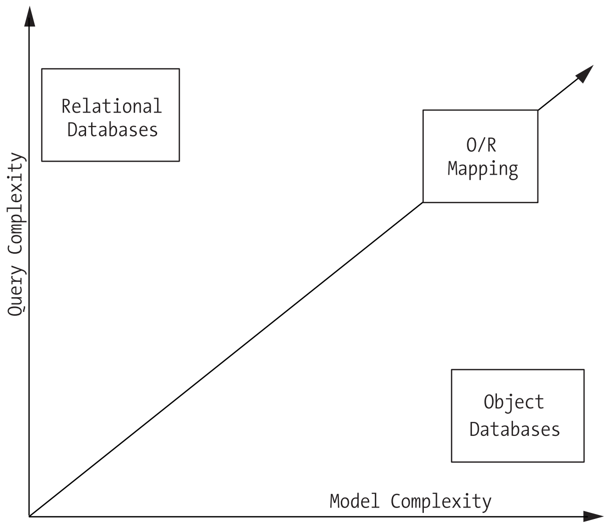


Figure 6-2. Trade-offs between relational and object databases

Relational Database Choices

The relational database is the standard for data storage in the enterprise. Regardless of its advantages or disadvantages, there's a great deal invested in the technology, and products have attained levels of maturity that are deemed to be at the enterprise level. Relational-database technology has proven itself in countless fields. The fact that it's decoupled from the applications has proven to be a blessing when it comes to adapting itself to the changing needs and trends of enterprise computing. From the mainframe, through the client-server, to the world of web services and service-oriented architectures, there's a good chance that you'll be using a relational database in your next project.

When choosing a relational database, take into account the following factors:

- **SQL:** What level of the SQL standard (SQL-92/SQL-99) does it support?
- **Features:** Store procedures, triggers, clustering, and replication available.
- **Optimizations:** Dynamic query optimization, caching.

- **Connectivity:** Level (type) of JDBC driver supported. See sidebar on JDBC drivers.

- **Datatypes:** Are special or custom datatypes supported?

Also, there are other choices that come into play when you use a relational database, because you need to take data from the relational world of tables to the Java world of objects. The strategies available for enterprise applications are as follows:

- **ORM:** Uses a relational database adapted with an ORM tool to turn relational data into objects and back (bridging the so called object-relational impedance mismatch), as shown in Chapter 7.

- **CMP:** Uses a coarser-grained component-based approach, as shown in Chapter 5.

- **JDBC:** Uses the JDBC API in either Session Beans or BMP Entity Beans.

BEST PRACTICE When using a relational database, regardless of the method of access (ORM, CMP) it's a good idea to test, with handwritten JDBC, the queries that will be the bread and butter of your application against several different databases, using different drivers with the help of a qualified DBA. If performance is a concern these numbers can give you a baseline for choosing your database, driver, and mapping/access strategy. Also, by comparing any generated SQL against the hand-optimized queries you can judge the relative efficiency of a tool.

JDBC Drivers

The quality of the connectivity solution you choose can have dramatic effects on application performance, scalability, and reliability. Most database vendors bundle a JDBC driver with their products, and experience has demonstrated that frequently they are subpar to both commercial and open-sourced third-party drivers.

The type of the driver supported by the database is also important. The JDBC specification mandates a set of interfaces to be implemented. How they are implemented is left to the vendors. The different types of implementation are officially categorized as follows:

- **Type 1:** Uses a call to native libraries typically written in a lower-level language like C.
- **Type 2:** Uses a hybrid approach of Java code and native libraries.
- **Type 3:** Uses only Java, but it has to map SQL calls to a vendor-specific protocol using an adaptor on the server side.
- **Type 4:** Pure client-side Java implementation that requires no mapping adaptors.

JDBC Type 4 drivers are typically recommended because they are usually more efficient, more portable, require less maintenance, and have easier installation procedures (no client-side binaries, no server-side adaptors, just put it in the classpath).

Pure Java Databases

Most J2EE servers come with an integrated/embedded pure-Java database. An embedded Java database is normally running in the same Java Virtual Machine (JVM) as the processes using it, therefore it gets a boost in performance by avoiding interprocess communication. Also, most of the time the JDBC driver can intelligently talk to the database without incurring any network overhead. The pure Java databases that are currently available provide a lightweight solution that's SQL compliant and requires a minimal amount of administration. There are a few pure Java-embedded SQL databases that have gained acceptance, among them hsqldb, McKoi SQL, and Axion DB. Table 6-1 shows a comparison of features in all three open-sourced databases.

NOTE This section covers hsqldb and McKoi SQL databases. The Axion DB product is, at the time of this writing, not ready to be used with JBoss.

Table 6-1. Pure Java Databases Feature Matrix

Database	URL	SQL Level
hsqldb	`http://hsqldb.sourceforge.net`	SQL-92 subset
McKoi SQL	`http://mckoi.com/database`	SQL-92 subset
Axion DB	`http://axion.tigris.org/`	SQL-92 subset

hsqldb

The hsqldb database engine is the successor to the now-closed Hypersonic SQL project, and it's the most-used open-sourced Java RDBMS. It's bundled with many Open Source and commercial projects and distributed under an Apache/ BSD-like license. It's a fairly fast and small database.

In Chapter 5 you learned how to set up a JBoss datasource using the embedded version of hsqldb in Server mode. The original version of the TCMS case study system was developed using hsqldb given its availability and its support for a large percent of the SQL-92 standard.

hsqldb Operating Modes

hsqldb can operate in several modes:

- **In-memory:** In this mode hsqldb keeps data in memory only and serves as a relational application cache. The data is never saved to disk, therefore this option is useful for testing scenarios where you would normally need to flush the database after each test. It's also useful for applications that don't need persistent data but want the advantages of SQL in order to manipulate transient data.

- **In-process:** Also referred to as stand-alone mode. In this mode hsqldb writes data to the file system once as part of its shutdown sequence.

Multithreaded	**Transactions**	**JDBC License**	**Client/Server Mode**
No	Yes	HSQL Development Group License (include copyright/ no implied endorsements)	Yes
Yes	Yes	GPL	Yes
No	Yes	Axion License (include copyright/ no implied endorsements)	No

- **Client/Server:** Supports server, web server, and servlet modes. The server mode allows TCP/IP connections to the database and it's the preferred mode for "production" applications using hsqldb in JBoss. The web server mode uses HTTP, effectively serving as a proxy/gateway to go through firewalls or for general connections over the Internet. The servlet mode is similar to the web server mode, but it's encapsulated in a Java servlet (and isn't meant to be used by servlet-based applications necessarily, but it can be deployed in a web container).

By creating appropriate hsqldb datasources in JBoss you can use a combination of the modes to cover your application needs. For example for volatile session information you can use the in-memory mode in combination with the server mode for your enterprise data.

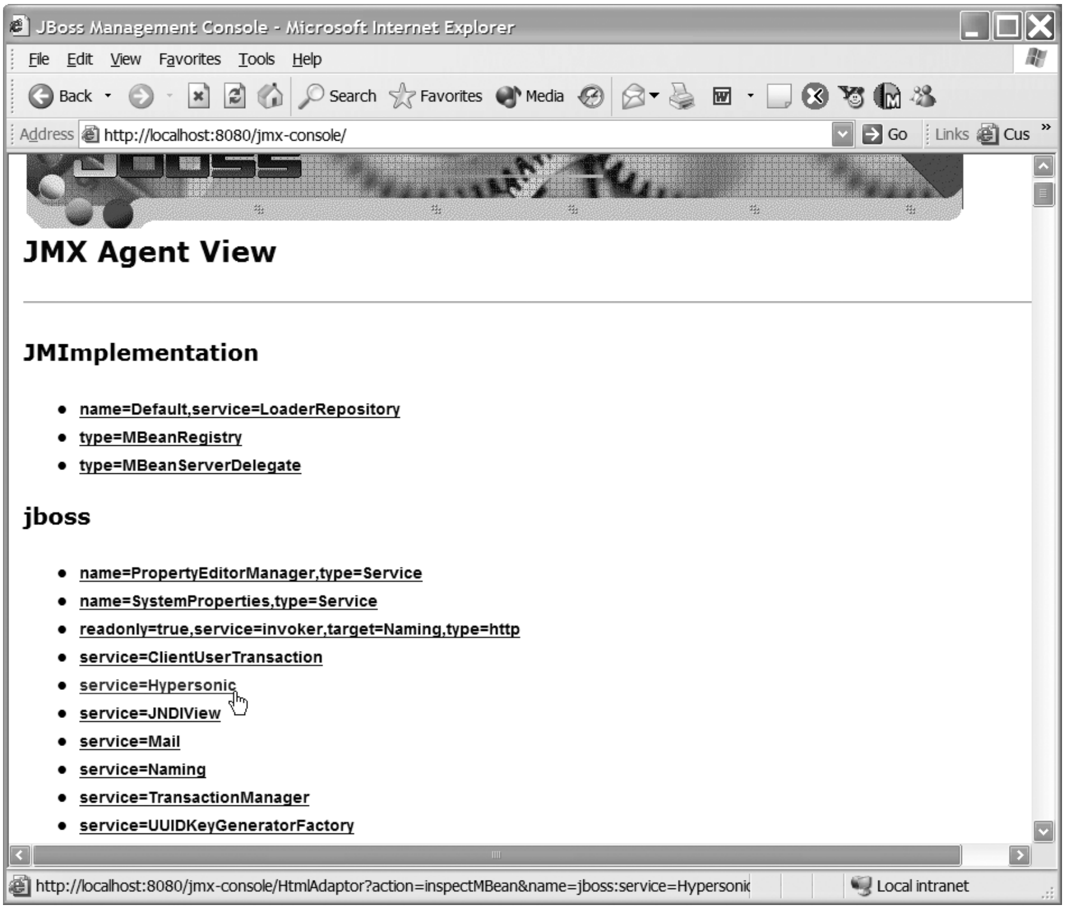
TIP In the JBoss configuration file hsqldb-ds.xml (as well as in the file tcms-ds.xml created in Chapter 5), the hsqldb operating mode is defined by the connection-url element of the local-tx-datasource element. Setting the value to "jdbc:hsqldb:." uses the in-memory mode or a value in the form "jdbc:hsqldb:database" where "database" is the name of the database file for in-process mode.

hsqldb Database Manager

The hsqldb distribution includes a database manager that's a Swing-based application that let's you administer to your databases. Typically, you can start this application by using the following command line (assuming that hsqldb.jar is in your classpath):

```
java org.hsqldb.util.DatabaseManager
```

But because you're running hsqldb embedded in JBoss, you can use the JBoss JMX Admin console to take advantage of the hsqldb MBean integration to launch the application. To accomplish this from the JBoss console (`http://localhost:8080/jmx-console`) find the service named "Hypersonic" as shown in Figure 6-3.


JBoss Management Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:8080/jmx-console/
Go Links Cus

JMX Agent View

JMImplementation

- name=Default,service=LoaderRepository
- type=MBeanRegistry
- type=MBeanServerDelegate

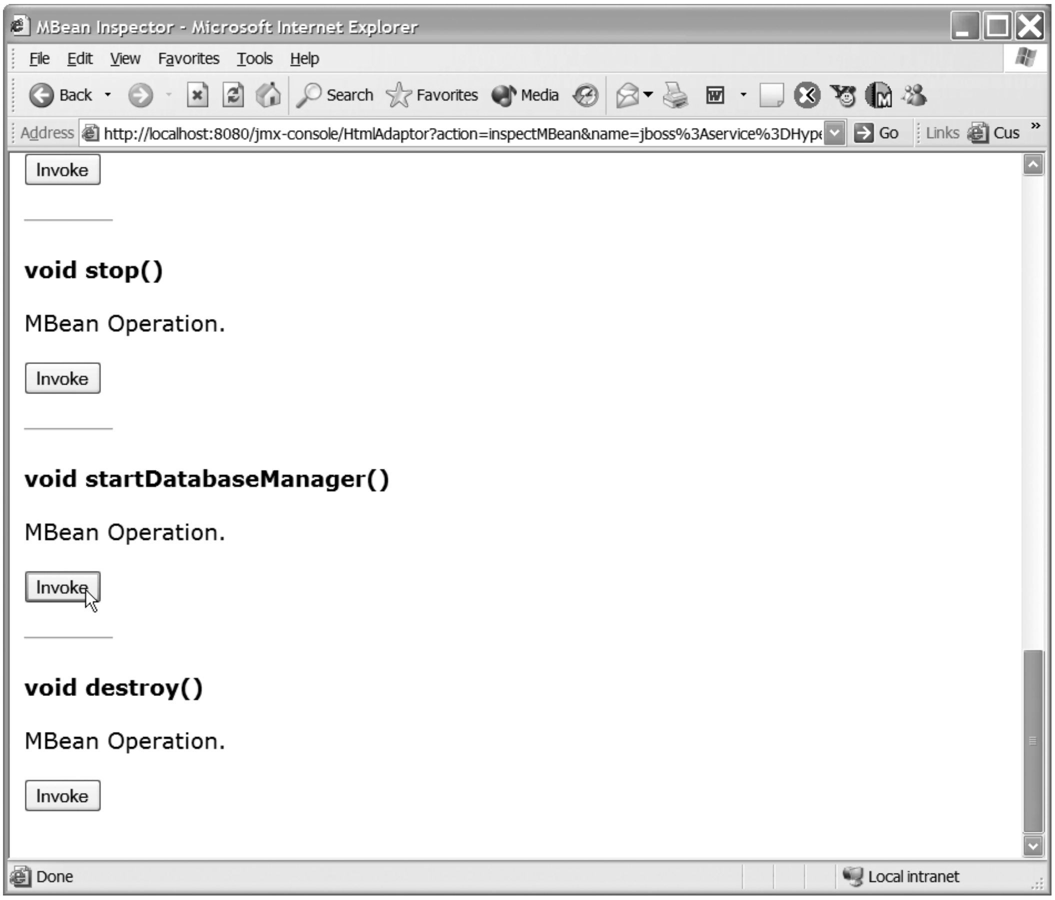
jboss

- name=PropertyEditorManager,type=Service
- name=SystemProperties,type=Service
- readonly=true,service=invoker,target=Naming,type=http
- service=ClientUserTransaction
- service=Hypersonic
- service=JNDIView
- service=Mail
- service=Naming
- service=TransactionManager
- service=UUIDKeyGeneratorFactory

http://localhost:8080/jmx-console/HtmlAdaptor?action=inspectMBean&name=jboss:service=Hypersonic

Figure 6-3. JBoss JMX Admin console

Next, click the link that will take you to the MBean view for the Hypersonic service. Here you can invoke the startDatabaseManager() MBean operation as shown in Figure 6-4.


MBean Inspector - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:8080/jmx-console/HtmlAdaptor?action=inspectMBean&name=jboss%3Aservice%3DHype
Go Links Cus
Done
Local intranet

Invoke

void stop()

MBean Operation.

Invoke

void startDatabaseManager()

MBean Operation.

Invoke

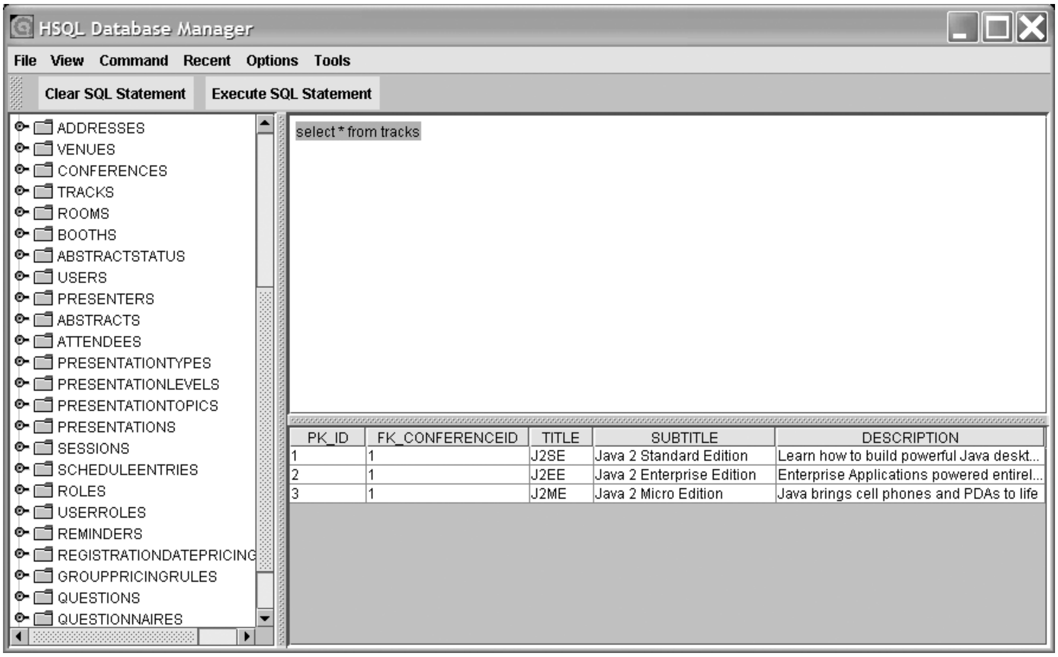
void destroy()

MBean Operation.

Invoke

Figure 6-4. JBoss JMX MBean view for Hypersonic service

Invoking the method should launch the HSQLDB Database Manager, as shown in Figure 6-5, in which a typical SQL query is shown against one of the TCMS tables.



```
select * from tracks
```

PK_ID	FK_CONFERENCEID	TITLE	SUBTITLE	DESCRIPTION
1	1	J2SE	Java 2 Standard Edition	Learn how to build powerful Java deskt...
2	1	J2EE	Java 2 Enterprise Edition	Enterprise Applications powered entirel...
3	1	J2ME	Java 2 Micro Edition	Java brings cell phones and PDAs to life

Figure 6-5. HSQLDB Database Manager

CAUTION When launching the HSQLDB Manager from the JMX Admin console the Database Manager executes in the server process and if the MBean method is invoked several times then there will be as many instances of the Database Manager as the number of times the startDatabaseManager() method was invoked. Therefore it's important to secure the JBoss console because a simple script could easily compromise your server.

McKoi SQL

The McKoi SQL database is another pure-Java RDBMS that, like hsqldb, can be used in embedded stand-alone mode or in client/server mode. One of the features that make the McKoi SQL database more suited for an enterprise production system is that it offers support for the highest level of transaction isolation (TRANSACTION_SERIALIZABLE). Although not as popular as hsqldb, the McKoi database is informally reported to be a stronger database platform both in performance and scalability.

CAUTION Before you consider embedding a Java database such as the McKoi SQL database in your product, it would be wise to seek legal counsel (at least informal) on the intricacies of the GPL and LGPL licenses when it comes to Java applications. This is especially important if you're planning to use the software in a commercial fashion or are planning to distribute it outside of your organization.

Creating a McKoi JBoss Datasource

Datasource deployments in JBoss are handled by the JBoss JCA implementation. Most databases, especially those that are open-sourced, include a custom JCA adapter in the form of a configuration file and one or more MBeans (for other databases you can use one the two generic JDBC JCA adaptors for regular "local" drivers and for XA (2PC) drivers). McKoi DB ships with a JBoss JCA MBean adaptor contributed by Howard Lewis Ship (creator of the Tapestry web framework).

The rest of this section shows you how to configure and deploy the McKoi DB in JBoss.

Make the McKoi JAR File Available to JBoss

The first step is to copy the mckoidb.jar to the tcms/lib directory (or the lib directory of the JBoss server that you're using).

Create a Database

To create a database, change directories to the location of the McKoi distribution and on the command line enter the following command:

```
java -jar mckoidb.jar -create "sa" "admin"
```

This will create a database with the username sa and the password admin. The console output should look like this:

```
Mckoi SQL Database ( 1.0.2 )
Copyright (C) 2000, 2001, 2002, 2003 Diehl and Associates, Inc.  All rights
reserved.
Use: -h for help.

  Mckoi SQL Database comes with ABSOLUTELY NO WARRANTY.
  This is free software, and you are welcome to redistribute it
  under certain conditions.  See LICENSE.txt for details of the
  GPL License.
```

Under the McKoi distribution directory you should now have a subdirectory named data, which contains the newly generated database files. Create a new directory under the tcms/data directory and name it mckoi (you should see an hsqldb directory at that level also). Copy the data directory under the newly created mckoi directory. The resulting directory structure is shown in Figure 6-6. It isn't necessary to copy the log directory because this directory is automatically created.

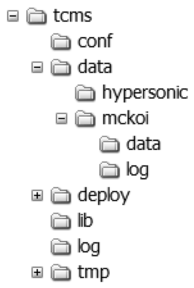


Figure 6-6. McKoi directory in JBoss

The McKoi database uses a properties file called db.conf to control the runtime behavior of the database engine. In the mckoi directory created in the previous step, create a new text file and name it db.conf. The sample db.conf provided here should suffice (notice that the standard port is 9157, if you use any other port number you would have to specify it in the connection URL used in your code or connection configuration files):

```
#######################################################
#
# Configuration options for the Mckoi SQL Database.
#
#######################################################

database_path=./data
log_path=./log
root_path=configuration
jdbc_server_port=9157
ignore_case_for_identifiers=disabled
regex_library=gnu.regexp
data_cache_size=4194304
max_cache_entry_size=8192
maximum_worker_threads=4
debug_log_file=debug.log
debug_level=20
```

Create the Datasource File

The McKoi JCA adaptor is contained in the package net.sf.tapestry.contrib.mckoi, which is part of the current McKoi distribution. This package contains the MBean used to interact with the database.

An XML descriptor with the -ds.xml suffix is needed for deployment so that the JBoss JCA deployer can recognize it. Next, create a file with the contents as shown here and save it as mckoi-ds.xml in any directory. Notice that the connection-url element doesn't specify a port because you're using the default port 9157.

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
    <local-tx-datasource>
        <depends>jboss:service=McKoi</depends>
        <jndi-name>mckoiDS</jndi-name>
        <connection-url>jdbc:mckoi://localhost</connection-url>
        <driver-class>com.mckoi.JDBCDriver</driver-class>
        <user-name/>
        <password/>
        <min-pool-size>5</min-pool-size>
    </local-tx-datasource>
    <mbean code="net.sf.tapestry.contrib.mckoi.McKoiDB"
           name="jboss:service=McKoi">
        <!--
        ConfigPath attribute is relative to the current working directory which
        is the %JBOSS_HOME%/bin directory
        -->
       <attribute name="ConfigPath">../server/tcms/data/mckoi/db.conf</attribute>
    </mbean>
</datasources>
```

Deploy the Datasource

To deploy the datasource simply copy the mckoi-ds.xml file to the tcms/deploy directory. The output on the JBoss console should resemble the following:

```
01:01:21,314 INFO  [MainDeployer] Starting deployment of package:
file:/C:/java/jboss/jboss-3.2.1/server/tcms/deploy/mckoi-ds.xml
01:01:21,344 INFO  [XSLSubDeployer] transformed into doc: [#document: null]
01:01:21,374 INFO  [McKoiDB] Creating
01:01:21,384 INFO  [McKoiDB] Created
01:01:21,384 INFO  [RARDeployment] Creating
01:01:21,384 INFO  [RARDeployment] Created
```

```
01:01:21,384 INFO  [JBossManagedConnectionPool] Creating
01:01:21,384 INFO  [JBossManagedConnectionPool] Created
01:01:21,384 INFO  [TxConnectionManager] Creating
01:01:21,384 INFO  [TxConnectionManager] Created
01:01:21,384 INFO  [McKoiDB] Starting
01:01:21,454 INFO  [McKoiDB] TCP JDBC Server (multi_threaded) on port: 9157
01:01:21,454 INFO  [McKoiDB] Started
01:01:21,454 INFO  [RARDeployment] Starting
01:01:21,464 INFO  [RARDeployment] Started
01:01:21,464 INFO  [JBossManagedConnectionPool] Starting
01:01:21,464 INFO  [JBossManagedConnectionPool] Started
01:01:21,464 INFO  [TxConnectionManager] Starting
01:01:21,474 INFO  [mckoiDS] Bound connection factory for resource adapter for
ConnectionManager 'jboss.jca:service=LocalTxCM
,name=mckoiDS to JNDI name 'java:/mckoiDS'
01:01:21,474 INFO  [TxConnectionManager] Started
01:01:21,474 INFO  [MainDeployer] Deployed package: file:/C:/java/jboss/jboss-
        3.2.1/server/tcms/deploy/mckoi-ds.xml
```

Now you can use the McKoi database in JBoss by using the datasource with the JNDI name mckoiDS. The McKoi JDBC driver is contained in both the mckoidb.jar (along with the database engine), and individually in the mkjdbc.jar, which would be suitable for distribution to external clients.

The McKoi Query Tool

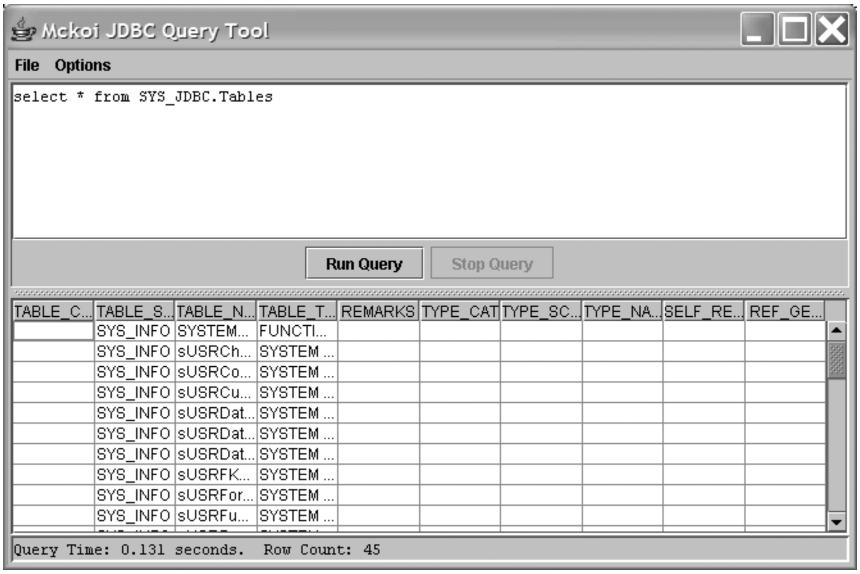
The McKoi query tool is a simple Swing-based tool that manipulates the database. It's included as part of the mckoidb.jar file. To launch the Query Tool type (with JBoss started and the McKoi datasource deployed), enter the following:

```
java -cp mckoidb.jar com.mckoi.tools.JDBCQueryTool
    —url "jdbc:mckoi://localhost" -u "sa" -p "admin"
```

The console will display a short message showing the JDBC driver being used and the URL of the database, as shown here:

```
Using JDBC Driver: com.mckoi.JDBCDriver
Connection established to: jdbc:mckoi:
```

Figure 6-7 shows the McKoi query tool in action.



```
select * from SYS_JDBC.Tables
```

TABLE_C...	TABLE_S...	TABLE_N...	TABLE_T...	REMARKS	TYPE_CAT	TYPE_SC...	TYPE_NA...	SELF_RE...	REF_GE...
	SYS_INFO	SYSTEM...	FUNCTI...						
	SYS_INFO	sUSRCh...	SYSTEM ...						
	SYS_INFO	sUSRCo...	SYSTEM ...						
	SYS_INFO	sUSRCu...	SYSTEM ...						
	SYS_INFO	sUSRDat...	SYSTEM ...						
	SYS_INFO	sUSRDat...	SYSTEM ...						
	SYS_INFO	sUSRDat...	SYSTEM ...						
	SYS_INFO	sUSRFK...	SYSTEM ...						
	SYS_INFO	sUSRFor...	SYSTEM ...						
	SYS_INFO	sUSRFu...	SYSTEM ...						

```
Query Time: 0.131 seconds.  Row Count: 45
```

Figure 6-7. McKoi query tool

NOTE There are other up-and-coming, open-sourced, pure-Java databases, many of which came out of the original Hypersonic project created by Thomas Mueller (like hsqldb). The Axion database, hosted at `http://axion.tigris.org`, doesn't have a formal 1.0 release yet, but it's one project to keep in mind in the near future.

Non-Java Relational Databases and Java

Typically, after the prototyping and development stages of a project you'll have to move to a non-Java database to handle the increasing loads. Many Internet sites looking for an open-sourced alternative have settled on using MySQL (`http://www.mysql.com`) including Yahoo! and NASA. Following behind MySQL is PostgreSQL (`http://www.postgresql.org`), Firebird (`http://firebird.sourceforge.net`), which is a fork of Borland's Interbase 6.0 during its short stint as an open-sourced database. MaxDB (`http://www.mysql.com/products/maxdb`) is an enhanced version of the former SAP DB (SAP AG's Open Source database).

PostgreSQL, Firebird, and MaxDB all cover a broader portion of the SQL-92 standard than MySQL does, and they offer more enterprise features such as stored procedures, triggers, distributed 2-phase commit (2PC) support, among other features. MySQL offers great speed (provides RAM tables), which makes it suitable for e-commerce websites, where most data is read. All four databases are supported by JBoss and provided under the docs/examples/jca directory of the JBoss distribution (3.2.X), where you'll find -ds.xml files for each of them. The procedure for using these databases is similar to the steps taken to enable the use of the McKoi and hsqldb databases.

For a fine-grained, feature-by-feature comparison you can use the MySQL online tool located at `http://www.mysql.com/information/crash-me.php`.

Java Object-Oriented Database Management System

In Chapter 7 we cover some of the intricacies of object-relational mapping. If you work with objects and are using a relational database you'll have to deal with mapping sooner or later. The more complex and rich your object model becomes, the more complex the mapping between it and a relational database will be. The ORM tools covered in Chapter 7, as an implied goal, have to sit between your object model and a relational database by providing your objects with an interface for persistence that, for all intents and purposes, isn't any different from an OODBMS.

In Chapter 7 you'll also read about orthogonal persistence (also called transparent persistence), which is the idea that at the code level, you deal with a system that knows objects only, and that you shouldn't need to be very aware of the fact that some objects are persistent and some others are transient. Although this isn't completely the case with either OODBMS or ORM tools, their APIs provide a natural extension for persistence in an object-oriented environment. ORM tools ease the object-relational impedance mismatch although OODBMS completely make it disappear. Without the layer of convoluted JDBC code embedded in your object models, or the maintenance of OR mappings required for ORM tools you gain the following:

- **Ease of development:** A reduction of complexity in your development process by making object persistence a natural feature of the system.

- **Performance:** A common misconception is that OODBMSs are naturally slower and less scalable that RDBMSs. In reality, given the right application, an OODBMS can be much more efficient than an RDBMS.

- **Integration:** Just like a CORBA ORB can provide access to remote objects, an OODBMS can serve as a repository of active objects in a system composed of many applications.

- **Simplicity:** Complex data is better handled by an object database. Many-to-many relationships are dealt with naturally, without normalization. Traversing an object hierarchy is much easier and efficient than in the relational model.

The relational model is simple and elegant but it's fundamentally different from the object model. As the complexity of applications increase, a good object model calls for fine-grained objects. For such highly complex object models, an OODBMS can provide ease of development (no data mapping), better performance, and scalability. OODBMS can provide features such as fast navigation and retrieval of information, versioning, and support for long transactions.

The ODMG Standard

Interest in OODBMS prompted the formation in 1991 of the Object Database Management Group (ODMG), which comprised most major OODBMS vendors at the time. The last version of the standard was specified in 1999 and it's referred to as ODMG 3.0. The ODMG standard defines an Object Definition Language (ODL), an Object Query Language (OQL), and language mappings. The JDO specification is the replacement or continuation for the Java language mappings work started by the ODMG.

TIP A possible low-risk entry point for an OODBMS in your enterprise application is the storage of session state information. Object databases make excellent middle-tier databases for use by EJBs or servlets. An e-commerce architecture using an object database for session state will likely provide better performance and reliability than if no database or a relational database was used instead.

Ozone Object-Oriented Database

The Ozone Database project (`http://www.ozone-db.org`) is a pure-Java OODBMS distributed under the GNU GPL/LGPL licenses (GPL for the core database engine and LGPL for the access API). The Ozone Database project started as a research project by Falko Braeutigam and is rapidly evolving into a full-fledged OODBMS. Ozone provides a multithreaded, multiuser, transactional, cached and clustered database environment that provides both an ODMG 3.0 interface as well as a native API to store and retrieve Java objects, binary large objects (BLOBs), and XML documents. Ozone integrates with J2EE environments by providing JTA/XA support and JMX-based management of services. It provides fine-grained access rights (at the object level), deadlock recognition, garbage collection, and an advanced collections API with support for lazy loading using dynamic proxies.

Ozone uses a central activation architecture in which objects never physically leave the database but are manipulated by reference with proxy objects using Ozone RMI (Ozone's version of Remote Method Invocation) in conjunction with Java serialization and runtime reflection, which makes persistence as transparent as possible. This architecture is better suited for application where the state of the objects is constantly being modified by client applications because only the data for the specific mutating operation needs to be transported over the wire.

Ozone works by storing the root object (referred to as "named objects") of an object graph, from which you can, by using normal Java constructs, navigate and retrieve related objects. Because Ozone uses Java serialization and RMI, your objects (at least the Ozone-aware implementations) need to implement and extend certain Ozone classes and interfaces. Ozone promotes that persistence logic should be executed in the database server to reduce the network overhead, which is similar to Entity Beans being fronted by a Session Facade. For an object to be stored in Ozone it must provide an interface that extends the Ozone remote interface (org.ozoneDB.OzoneRemote) and an implementation of said interface that extends the Ozone remote object (org.ozoneDB.OzoneObject, which implements java.io.Serializable). After compiling your classes, the Ozone Post Processor (OPP) is used to create stubs for the remote objects. Ozone RMI uses a remote object's stub class as a proxy in clients so that clients can manipulate a particular database object. By working with the remote interface, the code that's manipulating an Ozone object is unaware that it's dealing with a persistent object.

Download Ozone

In this chapter you'll be using version 1.2-alpha of the Ozone DB, which can be obtained from the project's download page at `http://sourceforge.net/projects/ozone` (at the SourceForge project page). The Ozone distribution is available in both source and binary forms. For the JBoss examples in this chapter you'll need the binary distribution contained in the ozone-1.2-alpha-bin.zip file. Download the file and unzip the contents to a suitable location such as c:\java\ozone-1.2.

Embedding Ozone DB as a JBoss Service

For Ozone to work inside of JBoss you need to compile the provided Ozone MBean (org.ozoneDB.embed.jboss.OzoneService) and deploy it as a JBoss service archive (SAR). A JBoss SAR file contains a JBoss service definition (jboss-service.xml) and its associated files.

NOTE A SAR file is a JAR archive with the extension .sar. They are specific to JBoss and aren't part of the J2EE specification.

The Ozone distribution provides a complete Java project that packages and deploys the SAR file to your local JBoss server. This project is contained under the thirdparty\jboss directory of the Ozone distribution.

TIP Under the Ozone JBoss MBean project directory (thirdparty\jboss) the Ozone project distributes some of the JBoss JARs that are needed to compile the MBean. These JARs are located under the thirdparty\jboss\lib directory. We recommend that you overwrite these JARs with the JARs from your JBoss distribution to avoid any class incompatibilities. The JARs that need to be updated are jboss-system.jar, jboss-common.jar, jboss-jmx.jar, and jboss.jar. The first three can be found under the lib directory of the JBoss distribution, and jboss.jar is found in the server/{JBOSS_SERVER}/lib directory (in the case of the TCMS system that directory resolves to server/tcms/lib).

At the root of the thirdparty\jboss directory you'll find an Ant build script. On the command line, use the now familiar projecthelp Ant command-line switch to discover the available Targets, as follows:

```
ant -projecthelp
```

This should produce output similar to the following:

```
Buildfile: build.xml

Main targets:

Other targets:

 compile
 deploy
 package
 prepare
Default target: deploy
```

CAUTION The Ant script for the Ozone JBoss integration assumes that you're using the source distribution of OZONE and therefore looks for the OZONE libraries in the server/build/lib directory. In the binary distribution these files are located under the lib directory at the root of the distribution. Therefore, for the Ant build to work you would need to change the value of the Ant property server.lib.dir from ../../ server/build/lib to ../../lib.

As you can see from the output of the Ant projecthelp command the default target is deployed, which on further examination of the Ant script, uses the JBOSS_HOME environment property to deploy the packaged file to the deploy directory of the "default" JBoss server. If you're using the tcms server (or any server besides "default") you'll have to execute the package target instead and manually copy the file to the deploy directory of the tcms server (or you can change the Ant script to use the "tcms" server rather than the default). To package the OZONE JBoss SAR file type, enter the following:

```
ant package
```

The output of the Ant script should resemble the following:

```
Buildfile: build.xml

prepare:
   [delete] Deleting directory C:\java\ozone-1.2-alpha\thirdparty\jboss\dist
    [mkdir] Created dir: C:\java\ozone-1.2-alpha\thirdparty\jboss\dist
```

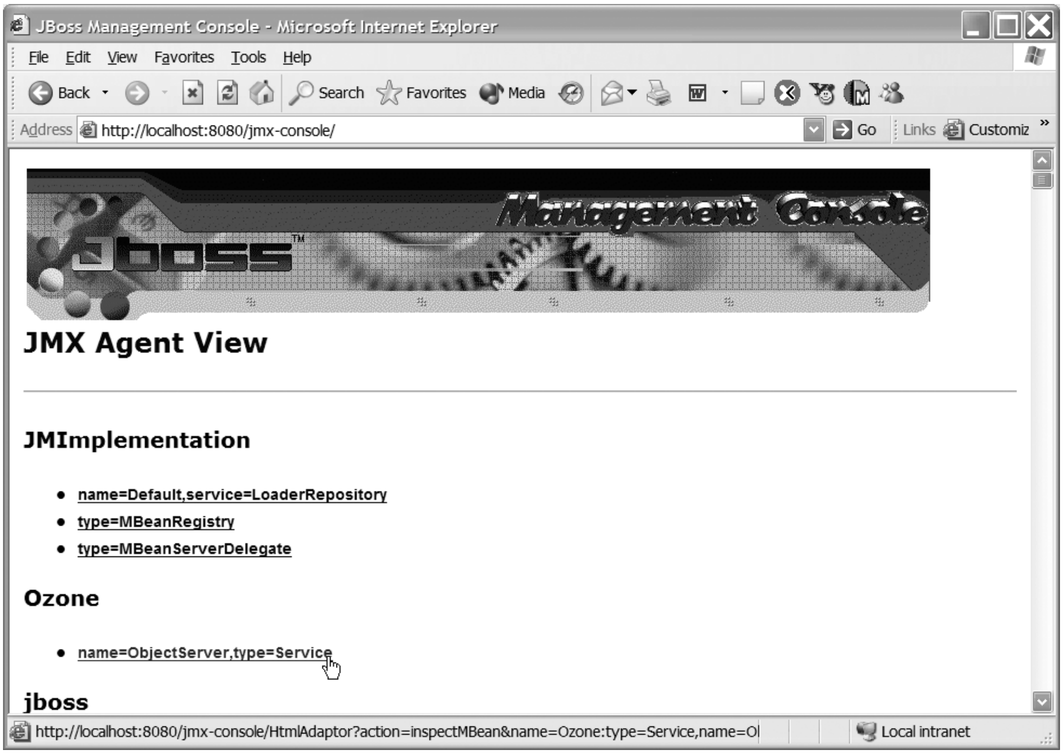
```
compile:
    [javac] Compiling 2 source files to C:\java\ozone-1.2-
alpha\thirdparty\jboss\build

package:
      [jar] Building jar: C:\java\ozone-1.2-
alpha\thirdparty\jboss\dist\ozoneService.sar

BUILD SUCCESSFUL
Total time: 5 seconds
```

After running the Ant script, a directory named dist under the thirdparty\jboss is created. In this directory you'll find the SAR file ozoneService.sar. To deploy the service simply copy the file to the deploy directory of the tcms JBoss server. The output of the deployment should resemble the following:

```
00:59:45,914 INFO  [MainDeployer] Starting deployment of package:
    file:/C:/java/jboss/jboss-3.2.1/server/tcms/deploy/ozoneService.sar
...
00:59:50,070 INFO  [OzoneService] Creating
00:59:50,070 INFO  [OzoneService] Created
00:59:50,120 INFO  [OzoneService] Starting
00:59:50,120 INFO  [OzoneService] Ozone ObjectServer - Starting up...
00:59:50,120 INFO  [OzoneService]
    ** Starting Database in C:\java\jboss\jboss-3.2.1\server\tcms/db/OzoneDB **
00:59:50,320 INFO  [OzoneService]       No DB found, creating new Database...
...
00:59:50,671 INFO  [Env] Ozone version 1.2-alpha
...
00:59:53,765 INFO  [GarbageCollector] startup...
...
00:59:53,765 INFO  [KeyGenerator] startup...
...
00:59:53,785 INFO  [ClassManager] startup...
...
00:59:53,795 INFO  [UserManager] startup...
00:59:53,795 INFO  [UserManager] admin user: Brian Sam-Bodden
...
00:59:53,925 INFO  [TransactionManager] startup...
...
00:59:54,086 INFO  [WizardStore] startup...
...
00:59:54,086 INFO  [WizardStore] checking for pending shadow clusters...
...
```

```
00:59:54,166 INFO  [AdminManager] startup...
...
00:59:54,196 INFO  [AdminManager] No admin object found. Initializing...
...
00:59:54,396 INFO  [OzoneService] ** Database ready **
...
00:59:54,406 INFO  [OzoneService] Started
00:59:54,426 INFO  [MainDeployer] Deployed package:
    file:/C:/java/jboss/jboss-3.2.1/server/tcms/deploy/ozoneService.sar
```

You can check the deployment of the Ozone service by using the JBoss JMX console Agent view as shown in Figure 6-8.


JBoss Management Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:8080/jmx-console/
Go Links Customiz
Management Console
JBoss™

JMX Agent View

JMImplementation

- name=Default,service=LoaderRepository
- type=MBeanRegistry
- type=MBeanServerDelegate

Ozone

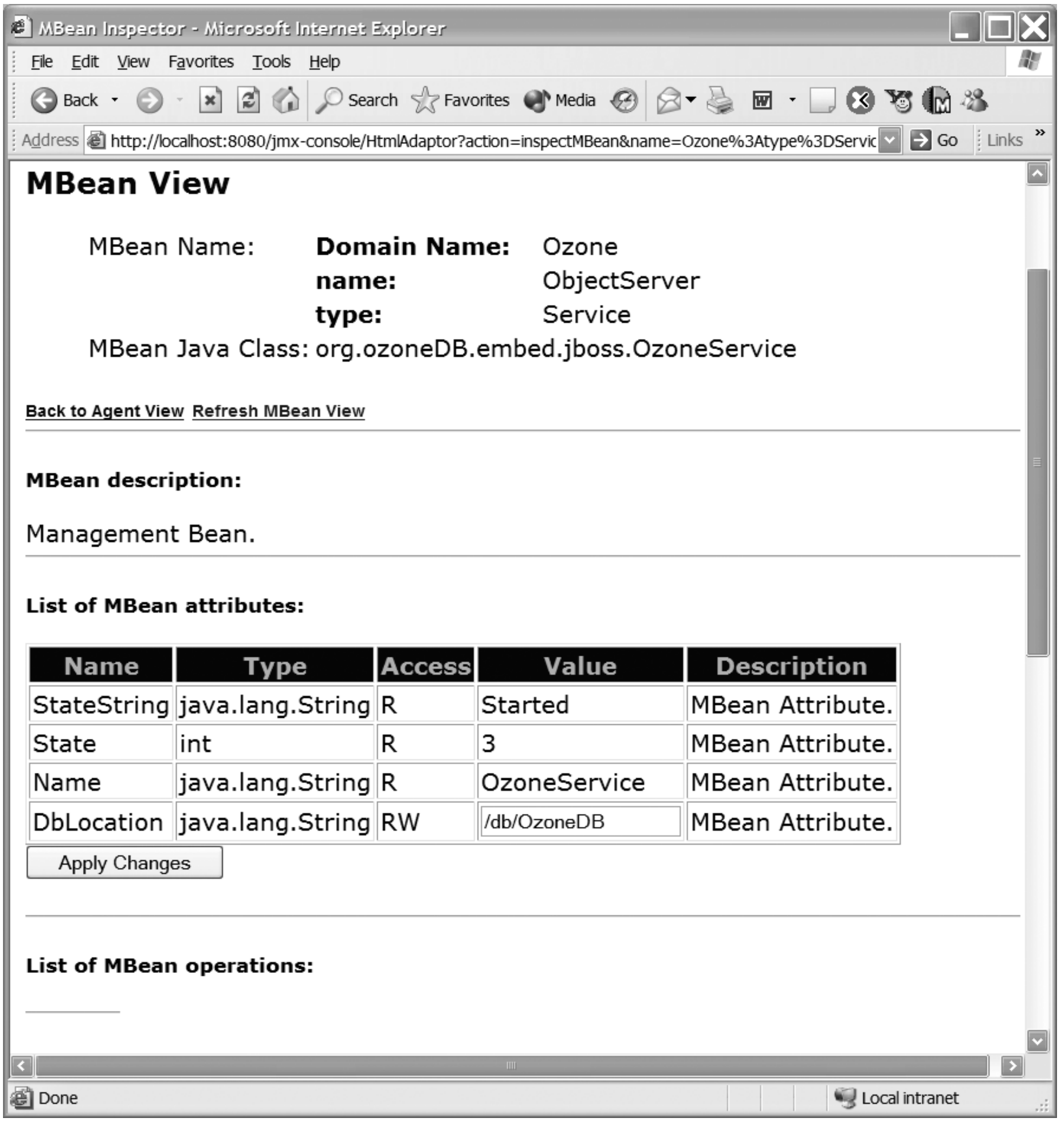
- name=ObjectServer,type=Service

jboss

http://localhost:8080/jmx-console/HtmlAdaptor?action=inspectMBean&name=Ozone:type=Service,name=Ol
Local intranet

Figure 6-8. The Ozone service in the JBoss console

Clicking the hyperlink name=ObjectServer,type=Service will take you to the MBean view where you can start and stop the Ozone service as shown in Figure 6-9, which might come in handy during the testing stages of your application.


MBean Inspector - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:8080/jmx-console/HtmlAdaptor?action=inspectMBean&name=Ozone%3Atype%3DServic
Go Links
Back to Agent View Refresh MBean View
Apply Changes
Done
Local intranet

MBean View

MBean Name:	**Domain Name:**	Ozone
	name:	ObjectServer
	type:	Service
MBean Java Class:	org.ozoneDB.embed.jboss.OzoneService	

MBean description:

Management Bean.

List of MBean attributes:

Name	Type	Access	Value	Description
StateString	java.lang.String	R	Started	MBean Attribute.
State	int	R	3	MBean Attribute.
Name	java.lang.String	R	OzoneService	MBean Attribute.
DbLocation	java.lang.String	RW	/db/OzoneDB	MBean Attribute.

List of MBean operations:

Figure 6-9. The Ozone service MBean view

Exploring the Database with the Ozone AdminGui

Now that you have an Ozone database running you can use the included AdminGui to manage and browse the contents of the database. The AdminGui application is contained in the ozoneAdminGui-0.1.jar JAR file, which is located in the lib directory of the Ozone distribution. A Windows batch file as well as a

UNIX shell script is provided in the bin directory. To launch the application (in Windows) change directories to the Ozone bin directory and enter the following:

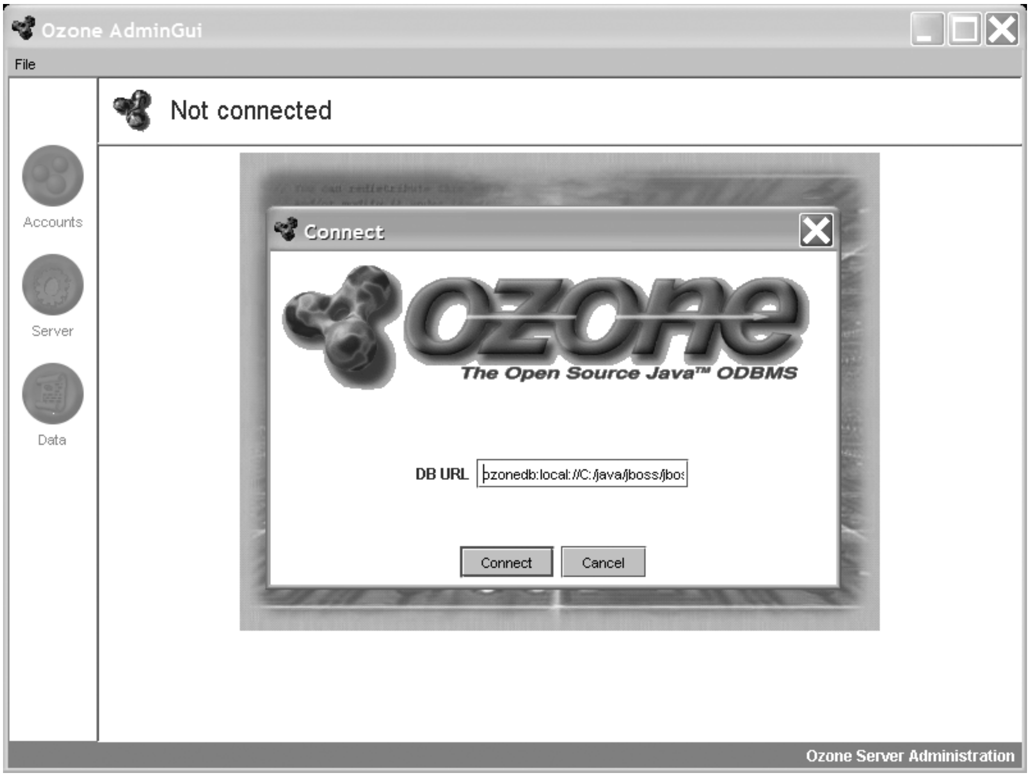
```
ozoneAdminGui
```

On the console you should see the following message:

```
Starting AdminGui ....
```

The AdminGui application should now be running, as shown in Figure 6-10.

CAUTION In the 1.2-alpha distribution of the Ozone DB the batch file AdminGui.bat doesn't work. Instead you can use the following command line at the root of the Ozone distribution directory:
java -cp "lib/ozoneAdminGui-0.1.jar;lib/ozoneServer-1.2-alpha.jar;lib/log4j-1.2.jar" org.ozoneDB.adminGui.main.AdminGui.



DB URL: ozonedb:local://C:/java/jboss/jbo

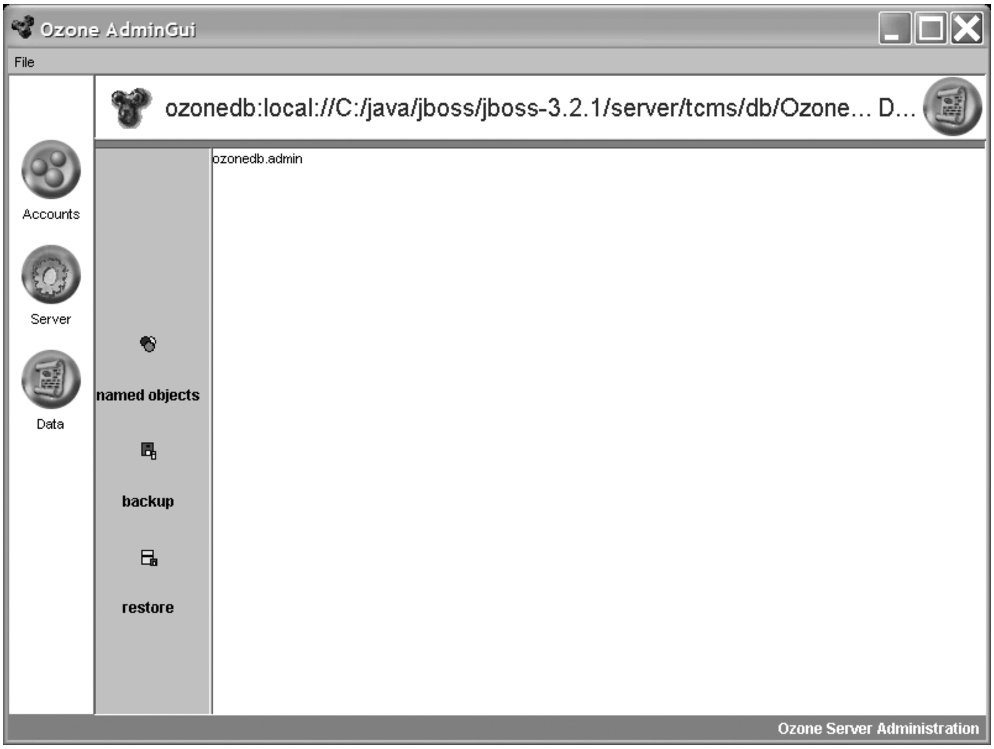
Figure 6-10. The Ozone DB AdminGui tool

The AdminGui connect dialog box has a field where you must enter the database URL, which defaults to a remote Ozone DB running on the local host at the default port (ozonedb:remote://localhost:3333). Because the Ozone DB running under JBoss is running in local mode you would need to change the URL to point to the directory where the DB is running. In the case of the tcms JBoss server at C:/java/jboss/jboss-3.2.1/server/tcms/ the correct local URL would be ozonedb:local://C:/java/jboss/jboss-3.2.1/server/tcms/db/OzoneDB.

From the AdminGui you have three main areas of functionality:

- **Accounts:** Allows you to create database accounts as well as list database groups and database users.
- **Server:** Allows you to monitor transactions on the server, force the Ozone garbage collector, or shut down the database server.
- **Data:** Allows you see a list of all named (root) objects, and provides functions to back up and restore a database to and from a file.

At startup on an empty database, the only named object you should see (by selecting Data ➤ named objects) is the ozonedb.admin object as shown in Figure 6-11.


Ozone AdminGui
File
ozonedb:local://C:/java/jboss/jboss-3.2.1/server/tcms/db/Ozone... D...
Accounts
Server
Data
named objects
backup
restore
Ozone Server Administration

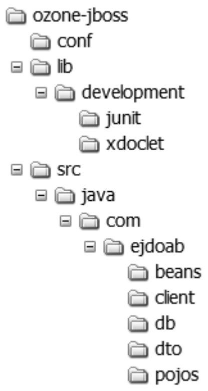
ozonedb.admin

Figure 6-11. Ozone DB named object in an empty database

Session Bean Ozone Project

Now that you have the Ozone DB integrated and deployed as a JBoss service, it's time to write a J2EE application to use with the database. As you did in Chapter 5, you'll use the domain objects Conference and ConferenceTrack. You'll create two plain old Java objects (POJOs), one for Conference and one for Tracks and use Ozone to store them.

Before coding begins you need a suitable project structure. Like the EJB CMP examples shown in Chapter 5, Figure 6.12 shows the directory structure of the Ozone/JBoss project (the complete project is available from the download site).



```
ozone-jboss
  conf
  lib
    development
      junit
      xdoclet
  src
    java
      com
        ejdoab
          beans
          client
          db
          dto
          pojos
```

Figure 6-12. Ozone/JBoss project directory structure

The steps to create a typical Ozone/JBoss application are as follows:

- **Remote interfaces:** Create the remote interfaces for each one of your business objects (which the POJOs will implement). This interface needs to extend the org.ozoneDB.OzoneRemote interface.

- **POJOs:** For each remote interface, create a POJO that implements the remote interface and extends the org.ozoneDB.OzoneObject class.

- **DTOs:** Data transfer objects are a flexible way to provide a "client" view of the database. Well-designed DTOs can provide semantically rich objects as return types for a service, and they can minimize network round-trips and provide client-side validation.

- **DatabaseManager:** The DatabaseManager class represents your persistence logic. It's the only class that will directly deal with Ozone persistence APIs. This class will represent the root of your application's object hierarchy as well as the facade for all database operations.

- **Session Facade:** A stateless Session Bean will provide business logic and will become the database client (just like a CMP EJB is the client to a relational database). All J2EE clients interact with your database objects through the Session Bean Facade, thereby isolating your data and providing a deterministic point of control.

The next few sections will walk you through the development of an Ozone/JBoss sample application.

Creating the Remote Interfaces

For the example you'll create two interfaces in the package com.ejdoab.pojos: Conference and Track. A Conference can contain a collection of Track objects.

The code for the Conference class is shown here:

```
package com.ejdoab.pojos;

import java.util.Date;
import java.util.List;

import org.ozoneDB.OzoneRemote;

public interface Conference extends OzoneRemote {
    // getters
    public String getName();
    public String getDescription();
    public Date getStartDate();
    public Date getEndDate();
    public Date getAbstractSubmissionStartDate();
    public Date getAbstractSubmissionEndDate();
    public List getTracks();
    // setters / mutators
    public void setName(String value); /*update*/
    public void setDescription(String value); /*update*/
    public void setStartDate(Date value); /*update*/
    public void setEndDate(Date value); /*update*/
    public void setAbstractSubmissionStartDate(Date value); /*update*/
    public void setAbstractSubmissionEndDate(Date value); /*update*/
    public void addTrack(Track track); /*update*/
    public void deleteTrack(String name); /*update*/
}
```

The first thing to notice is that every method signature of a method that changes the object is followed by the comment /*update*/. These comments are used by the OPP tool to determine which methods in the generated proxies need to be part of a transaction.

The code for the Track interface is similar to that of the Conference interface, as shown here:

```
package com.ejdoab.pojos;

import org.ozoneDB.OzoneRemote;

public interface Track extends OzoneRemote {
      // getters
    public String getDescription();
    public String getSubTitle();
    public String getTitle();
    // setters
    public void setDescription(String value); /*update*/
    public void setSubTitle(String value); /*update*/
    public void setTitle(String value); /*update*/
}
```

Creating Data Transfer Objects

DTOs will be used as the transport mechanism for data in and out of a Session Facade. Typically, your DTOs will represent a "client" view of the data, which is structured in a way to make the client's work easier and more efficient. In the case of the example at hand the DTOs are merely POJOs with identical method signatures, as the remote interfaces previously defined.

All DTOs for the sample application will be placed in the com.ejdoab.dto package. The code for ConferenceDTO is shown here:

```
package com.ejdoab.dto;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.util.List;

import com.ejdoab.pojos.Conference;
import com.ejdoab.pojos.Track;
```

```
public class ConferenceDTO implements Serializable {
    protected String _name;
    protected String _description;
    protected Date _startDate;
    protected Date _endDate;
    protected Date _abstractSubmissionStartDate;
    protected Date _abstractSubmissionEndDate;
    protected List _tracks;

    //
    // constructors
    //

    private ConferenceDTO() {
        _tracks = new ArrayList();
    }

    public ConferenceDTO(
        String name,
        String description,
        Date startDate,
        Date endDate,
        Date abstractSubmissionStartDate,
        Date abstractSubmissionEndDate) {
        this();
        _name = name;
        _description = description;
        _startDate = startDate;
        _endDate = endDate;
        _abstractSubmissionStartDate = abstractSubmissionStartDate;
        _abstractSubmissionEndDate = abstractSubmissionEndDate;
    }

    public ConferenceDTO(Conference conference) {
        this();
        _name = conference.getName();
        _description = conference.getDescription();
        _startDate = conference.getStartDate();
        _endDate = conference.getEndDate();
        _abstractSubmissionStartDate =
conference.getAbstractSubmissionStartDate();
        _abstractSubmissionEndDate = conference.getAbstractSubmissionEndDate();
        for (Iterator i = conference.getTracks().iterator(); i.hasNext();) {
            Track track = (Track) i.next();
            _tracks.add(new TrackDTO(track));
        }
    }

    //
    // getters
    //

    public String getName() {
        return _name;
    }

    public String getDescription() {
        return _description;
    }

    public Date getStartDate() {
        return _startDate;
    }

    public Date getEndDate() {
        return _endDate;
    }

    public Date getAbstractSubmissionStartDate() {
        return _abstractSubmissionStartDate;
    }

    public Date getAbstractSubmissionEndDate() {
        return _abstractSubmissionEndDate;
    }

    public List getTracks() {
        return _tracks;
    }

    //
    // setters
    //
```

```
public void setName(String value) {
    _name = value;
}

public void setDescription(String value) {
    _description = value;
}

public void setStartDate(Date value) {
    _startDate = value;
}

public void setEndDate(Date value) {
    _endDate = value;
}

public void setAbstractSubmissionStartDate(Date value) {
    _abstractSubmissionStartDate = value;
}

public void setAbstractSubmissionEndDate(Date value) {
    _abstractSubmissionEndDate = value;
}

public void addTrack(TrackDTO track) {
    _tracks.add(track);
}

public void deleteTrack(String name) {
    if (_tracks != null) {
        for (Iterator i = _tracks.iterator(); i.hasNext();) {
            Conference old = (Conference) i.next();
            if (old.getName().equalsIgnoreCase(name)) {
                _tracks.remove(old);
            }
        }
    }
}

//
// utility
//
```

```
    public String toString() {
        return new StringBuffer()
            .append("[Conference] name = ")
            .append(_name != null ? _name : "n/a")
            .append(", description = ")
            .append(_description != null ? _description : "n/a")
            .append(", start date = ")
            .append(_startDate != null ? _startDate.toString() : "n/a")
            .append(", end date = ")
            .append(_endDate != null ? _endDate.toString() : "n/a")
            .append(", # tracks = ")
            .append(_tracks != null ? _tracks.size() : 0)
            .toString();
    }
}
```

As you can see, the signature of ConferenceDTO is identical to that of the remote interface Conference. The DTO also provides a constructor that takes an instance of Conference to populate itself. This is an alternative approach to the one taken in Chapter 5 when you used a DTOFactory to create DTOs from business objects and vice versa. Notice that the ConferenceDTO contains a list of TrackDTOs, thereby mimicking the structure in Conference (and ConferenceImpl, which you'll see in the next section).

The code shown here for the TrackDTO is equally simple:

```
package com.ejdoab.dto;

import java.io.Serializable;

import com.ejdoab.pojos.Track;

public class TrackDTO implements Serializable {

    protected String _title;
    protected String _subTitle;
    protected String _description;
    protected Integer _conferenceId;

    //
    // constructors
    //
```

```
public TrackDTO(String title, String subTitle, String description) {
    _title = title;
    _subTitle = subTitle;
    _description = description;
}

public TrackDTO(Track track) {
    _title = track.getTitle();
    _subTitle = track.getSubTitle();
    _description = track.getDescription();
}

//
// getters
//

public String getDescription() {
    return _description;
}

public String getSubTitle() {
    return _subTitle;
}

public String getTitle() {
    return _title;
}

//
// setters
//

public void setDescription(String value) {
    _description = value;
}

public void setSubTitle(String value) {
    _subTitle = value;
}

public void setTitle(String value) {
    _title = value;
}
```

```
    public String toString() {
        return new StringBuffer()
                    .append("[Track] title = ")
                    .append(_title != null ? _title : "n/a")
                    .append(", subTitle = ")
                    .append(_subTitle != null ? _subTitle : "n/a")
                    .append(", description = ")
                    .append(_description != null ? _description : "n/a")
                    .toString();
    }
}
```

NOTE Notice that the DTO doesn't directly implement the remote interface because that would introduce dependencies on Ozone in the client side and it wouldn't work in the case of using an Ozone LocalDatabase as in the JBoss example. This is because proxies cannot be passed from a LocalDatabase environment to an external client.

Creating the POJOs

The Ozone-ready implementations of the business interfaces extend the org.ozoneDB.OzoneObject class, which makes them database (server-side, persistent) objects. The OzoneObject class provides a default implementation of the OzoneCompatible interface, which binds the object to the database for life-cycle management.

The code for the ConferenceImpl is shown here:

```
package com.ejdoab.pojos;

import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.util.List;

import org.ozoneDB.OzoneObject;

import com.ejdoab.dto.*;
```

```
public class ConferenceImpl extends OzoneObject implements Conference {
    /**
     * set the serialization version to make it compatible
     * with new class versions
     */
    public static final long serialVersionUID = 1L;

    protected String _name;
    protected String _description;
    protected Date _startDate;
    protected Date _endDate;
    protected Date _abstractSubmissionStartDate;
    protected Date _abstractSubmissionEndDate;
    protected List _tracks;

    //
    // constructors
    //

    public ConferenceImpl() {
        _tracks = new ArrayList();
    }

    public ConferenceImpl(
        String name,
        String description,
        Date startDate,
        Date endDate,
        Date abstractSubmissionStartDate,
        Date abstractSubmissionEndDate) {
        this();
        ...
    }

    public ConferenceImpl(ConferenceDTO conference) {
        this();
        _name = conference.getName();
        _description = conference.getDescription();
        _startDate = conference.getStartDate();
        _endDate = conference.getEndDate();
        _abstractSubmissionStartDate =
            conference.getAbstractSubmissionStartDate();
        _abstractSubmissionEndDate = conference.getAbstractSubmissionEndDate();
```

```
            for (Iterator i = conference.getTracks().iterator(); i.hasNext();) {
                TrackDTO track = (TrackDTO) i.next();
                _tracks.add(new TrackImpl(track));
            }
        }

        //
        // getters
        //

        ...

        //
        // setters
        //

        ...

        public void addTrack(Track track) {
            _tracks.add(track);
        }

        public void deleteTrack(String name) {
            if (_tracks != null) {
                for (Iterator i = _tracks.iterator(); i.hasNext();) {
                    Conference old = (Conference) i.next();
                    if (old.getName().equalsIgnoreCase(name)) {
                        _tracks.remove(old);
                    }
                }
            }
        }

        //
        // utility
        //

        public String toString() {
            ...
        }
    }
```

Notice that at the beginning of the class serialVersionUID is set to the value 1L. Because Ozone uses Java serialization, it's important to set the serialVersionUID to a unique value so that the objects stored in the server and those that are being manipulated in the client are compatible (because compiling a Java class generates a new serialVersionUID if one isn't explicitly set).

The addTrack and deleteTrack methods are provided to add and remove a Track object from a given Conference object. Also, a constructor is provided to create a ConferenceImpl, given a ConferenceDTO. The TrackImpl class implementing the Track interface follows the same pattern, except that because it doesn't hold any depended objects it's actually simpler than the ConferenceImpl class, as shown here:

```
package com.ejdoab.pojos;

import org.ozoneDB.OzoneObject;

import com.ejdoab.dto.TrackDTO;

public class TrackImpl extends OzoneObject implements Track {
    /**
     * set the serialization version to make it compatible
     * with new class versions
     */
    public static final long serialVersionUID = 1L;

    protected String _title;
    protected String _subTitle;
    protected String _description;
    protected Integer _conferenceId;

    //
    // constructors
    //

    public TrackImpl() {
    }

    public TrackImpl(String title, String subTitle, String description) {
        ...
    }
    public TrackImpl(TrackDTO track) {
        _title = track.getTitle();
        _subTitle = track.getSubTitle();
        _description = track.getDescription();
    }

    //
    // getters
    //
    ...

    //
    // setters
    //
    ...

    //
    // utility
    //

    public String toString() {
        ...
    }
}
```

Persistence Logic

To manage a collection of Conferences and provide create, read, update, and delete (CRUD) operations you'll need a database object to serve as the entry point for your persistence-logic operations. This object will become the root object of your hierarchy of objects in Ozone (a named object).

Like the two previous database objects (Conference and Track) the ConferencesManager provides a remote interface and an implementation. Because it's a good idea to separate the persistence logic from the business logic, the ConferencesManager interface and implementation are placed in the com.ejdoab.db package. The ConferencesManager will provide the entry point into the database for the Session Facade, as follows:

```
package com.ejdoab.db;

import java.util.Collection;
import org.ozoneDB.OzoneRemote;
import com.ejdoab.dto.ConferenceDTO;

public interface ConferencesManager extends OzoneRemote {
    // getters
    public ConferenceDTO getConferenceByName(String name);
    public Collection getAllConferences();

    // setters / mutators
   public void addOrUpdateConference(ConferenceDTO conference); /*update*/
    public boolean deleteConference(String name); /*update*/
    public boolean deleteAllConferences(); /*update*/
}
```

The ConferencesManager provides several methods for retrieving one or more Conference objects as well as methods for adding and deleting one or all Conferences. Internally, the list of Conferences is kept in a Java List. Notice that the implementation of the ConferencesManager interface enforces a persistence-logic rule by enabling only uniquely named Conferences to be stored. The implementation of the ConferencesManager is shown here:

```
package com.ejdoab.db;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.List;

import org.ozoneDB.OzoneInterface;
import org.ozoneDB.OzoneObject;

import com.ejdoab.dto.ConferenceDTO;
import com.ejdoab.pojos.Conference;
import com.ejdoab.pojos.ConferenceImpl;

public class ConferencesManagerImpl
    extends OzoneObject
    implements ConferencesManager {
```

```
/**
 * set the serialization version to make it compatible
 * with new class versions
 */
public static final long serialVersionUID = 1L;

private List conferences;

//
// factory method
// (to avoid the chicken-egg problem when building the app)
//
public static ConferencesManager create(OzoneInterface db) {
    return (ConferencesManager) db.createObject(
                                    ConferencesManagerImpl.class,
                                    OzoneInterface.Public,
                                    ConferencesManager.class.getName()
                                );
}

//
// constructors
//
public ConferencesManagerImpl() {
    conferences = new ArrayList();
}

//
// getters
//

public ConferenceDTO getConferenceByName(String name) {
    ConferenceDTO result = null;

    for (Iterator i = conferences.iterator(); i.hasNext();) {
        Conference conference = (Conference) i.next();
        if (conference.getName().equalsIgnoreCase(name)) {
            result = new ConferenceDTO(conference);
        }
    }

    return result;
}
```

```
    public Collection getAllConferences() {
        return conferences;
    }

    //
    // setters
    //
    public void addOrUpdateConference(ConferenceDTO conference) {
        if (conference.getName() != null) {
            for (Iterator i = conferences.iterator(); i.hasNext();) {
                Conference old = (Conference) i.next();
                if (old.getName().equalsIgnoreCase(conference.getName())) {
                    conferences.remove(old);
                }
            }
            conferences.add(new ConferenceImpl(conference));
        }
    }

    public void removeConferences() {
        conferences.clear();
    }

    public boolean deleteConference(String name) {
        Conference target = null;
        for (Iterator i = conferences.iterator(); i.hasNext();) {
            Conference conference = (Conference) i.next();
            if (conference.getName().equalsIgnoreCase(name)) {
                target = conference;
            }
        }
        return conferences.remove(target);
    }

    public boolean deleteAllConferences() {
        conferences.clear();
        return conferences.isEmpty();
    }
}
```

The only method of particular interest is the static factory create method, which takes an instance of an OzoneInterface as a parameter. This method's purpose is to construct a ConferencesManager object, which is bound to the database. As mentioned before, the OPP generates a comprehensive factory class, so technically this method isn't required, but in order for the application build process to have a single compilation target, this method should be added here because it will be used in the Session Facade. Otherwise the Session Facade would have to be compiled after the OPP target, which needs to happen before compilation of the classes in the pojos and dto packages.

Creating the Session Facade

To work with the Ozone database you'll use a stateless Session Bean that will provide services to manipulate the Conferences and Track objects in the database using their peer DTOs. The com.ejdoab.beans.ConferenceOzoneFacadeBean will provide the following methods:

- ConferenceDTO getConferenceByName(String name)
- Collection getAllConferences
- void addOrUpdateConference(ConferenceDTO dto)
- boolean deleteConference(String name)
- boolean deleteAllConferences()

As you learned in Chapter 5, you'll use XDoclet to automate the creation of the EJB glue files. A private field will hold an instance of OzoneInterface, which is looked up using JNDI in the Bean's ejbCreate method. The OzoneInterface is then used to find (or create if it doesn't exist) the named object stored under the com.ejdoab.db.ConferencesManager name, as shown here:

```
package com.ejdoab.beans;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

import javax.ejb.CreateException;
import javax.ejb.EJBException;
import javax.ejb.SessionBean;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import org.ozoneDB.OzoneInterface;

import com.ejdoab.db.ConferencesManager;
import com.ejdoab.db.ConferencesManagerImpl;
import com.ejdoab.dto.ConferenceDTO;
import com.ejdoab.pojos.Conference;

/**
 * @ejb.bean
 *      name="ConferenceOzoneFacade"
 *      type="Stateless"
 *      view-type="both"
 *      jndi-name="ejb.ConferenceOzoneFacadeHome"
 *      local-jndi-name="ejb.ConferenceOzoneFacadeLocalHome"
 * @ejb.transaction
 *      type="Required"
 * @ejb.util
 *      generate="physical"
 */
public abstract class ConferenceOzoneFacadeBean implements SessionBean {

    private OzoneInterface db;
    private ConferencesManager conferencesManager;

    //
    // business methods
    //

    /**
     * @ejb.interface-method
     * @ejb.transaction
     *      type="NotSupported"
     */
    public ConferenceDTO getConferenceByName(String name) {
        ConferenceDTO result = null;
        if (conferencesManager != null) {
              result = conferencesManager.getConferenceByName(name);
        }
        return result;
    }
```

```
/**
 * @ejb.interface-method
 * @ejb.transaction
 *      type="NotSupported"
 */
public Collection getAllConferences() {
    List results = Collections.EMPTY_LIST;
    if (conferencesManager != null) {
        Collection allConferences = conferencesManager.getAllConferences();
        if (!allConferences.isEmpty()) {
            results = new ArrayList(allConferences.size());
            for (Iterator iter = allConferences.iterator();
                iter.hasNext();
                ) {
                Conference conference = (Conference) iter.next();
                // return a DTO instead of a proxy object,
                // we don't want clients to have
                // direct access to the database objects
                results.add(new ConferenceDTO(conference));
            }
        }
    }
    return results;
}

/**
 * @ejb.interface-method
 * @ejb.transaction
 *      type="NotSupported"
 */
public void addOrUpdateConference(ConferenceDTO dto) {
    if (conferencesManager != null) {
        conferencesManager.addOrUpdateConference(dto);
    }
}

/**
 * @ejb.interface-method
 * @ejb.transaction
 *      type="NotSupported"
 */
```

```
public boolean deleteConference(String name) {
    boolean result = false;
    if (conferencesManager != null) {
        result = conferencesManager.deleteConference(name);
    }
    return result;
}

/**
 * @ejb.interface-method
 * @ejb.transaction
 *      type="NotSupported"
 */
public boolean deleteAllConferences() {
    boolean result = false;
    if (conferencesManager != null) {
        result = conferencesManager.deleteAllConferences();
    }
    return result;
}

//===========================================
//  EJB callbacks
//===========================================

/**
 * @ejb.create-method
 */
public void ejbCreate() throws CreateException {
    Context context = null;
    // Lookup the Ozone DB Interface
    try
    {
        context = new InitialContext();
        db = (OzoneInterface) new InitialContext()
            .lookup(OzoneInterface.class.getName());
        conferencesManager = (ConferencesManager) db
            .objectForName(ConferencesManager.class.getName());
        if (conferencesManager == null) {
            conferencesManager = ConferencesManagerImpl.create(db);
        }
        } catch (NamingException e) {
            throw new EJBException(e);
        } catch (Exception e) {
            throw new EJBException(e);
        }
    }

    public void ejbActivate() {
        try {
            db = (OzoneInterface) new InitialContext()
                .lookup(OzoneInterface.class.getName());
            db.reloadClasses();
        } catch (Exception e) {
            throw new EJBException(e);
        }
    }

    public void ejbPassivate() {
        db = null;
    }

    public void ejbPostCreate() throws CreateException {}
}
```

Notice that the connection to the database is managed via the EJB callback methods ejbCreate, ejbActivate, and ejbPassivate. In ejbCreate an OzoneInterface (to the database) is looked up using JNDI. The OzoneInterface is used to locate the instance of ConferencesManager, which is bound to the database under the name com.ejdoab.db.ConferencesManager (the result of using ConferencesManager.class.getName method). If the object isn't found then it's created using the factory method in ConferencesManagerImpl. In ejbActivate the ConferenceManager is retrieved again and the classes are reloaded using the reloadClasses method.

The Session Bean business methods in turn use the instance of ConferencesManager to perform their functions.

Putting It All Together with Ant

Let's walk through the Ant build script that will accomplish the build tasks required for the example. The build script is located at the root of the project directory. A sample of the build.properties file shown here has a section that defines the ejb-jar that will be generated. It also has a section for the JBoss-specific settings (refer to Chapter 5 for JBoss configuration instructions), and finally, it has a section that defines the location of the Ozone distribution.

```
// app
jar-name=ozone-jboss-test.jar

// jboss specific
jboss.home=c:/java/jboss/jboss-3.2.1
jboss.server=tcms
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099

// OZONE specific
OZONE_HOME=c:/java/ozone-1.2-alpha
```

The first section of the script deals with loading and configuring the build properties:

```
<?xml version="1.0"?>
<project name="ozone-jboss" default="all" basedir=".">

    <!-- =================================================================== -->
    <!-- Configures Project's Properties                                      -->
    <!-- =================================================================== -->
    <property file="build.properties"/>
    <property name="server-dir"
              location="${jboss.home}/server/${jboss.server}" />
    <property name="server-lib-dir"  location="${server-dir}/lib" />
    <property name="server-conf-dir" location="${server-dir}/conf" />
    <property name="server-client-dir"  location="${jboss.home}/client" />
    <property name="deploy-dir"      location="${server-dir}/deploy" />

    <property name="root" location="${basedir}" />
    <property name="src" location="${root}/src/java" />
    <property name="classes" location="${root}/classes" />
    <property name="generated" location="${root}/generated" />
    <property name="generated-ejb" location="${generated}/ejb-src" />
    <property name="generated-ozone" location="${generated}/ozone-src" />
    <property name="descriptors-ejb" location="${generated}/descriptors/ejb" />
    <property name="dist" location="${root}/dist" />
    <property name="conf" location="${root}/conf" />
    <property name="build" location="${root}/build" />

    <property name="lib" location="lib" />
    <property name="lib-dev" location="${lib}/development" />
```

Next, several path elements are created for all the project dependencies, including Ozone, JBoss, and XDoclet. Notice that throughout the build you make use of the ${jboss.home} and ${OZONE _HOME} properties. Both of these properties are defined in the build.properties file.

```
<!-- =================================================================== -->
<!-- Configures the ClassPath                                            -->
<!-- =================================================================== -->
<path id="class.path">
    <fileset dir="lib">
        <include name="*.jar"/>
    </fileset>
    <fileset dir="${server-lib-dir}">
        <include name="*servlet.jar"/>
    </fileset>
    <fileset dir="${server-client-dir}">
        <include name="jbossall-client.jar"/>
    </fileset>
    <pathelement location="${classes}" />
</path>

<path id="xdoclet.class.path">
    <path refid="class.path"/>
    <fileset dir="${lib-dev}/xdoclet">
        <include name="*.jar"/>
    </fileset>
</path>

<path id="ozone.class.path">
    <path refid="xdoclet.class.path"/>
    <fileset dir="${OZONE_HOME}/lib">
        <include name="*.jar"/>
    </fileset>
</path>
```

The taskdefs for the XDoclet ejbdoclet are loaded and the tasks for preparing and cleaning the project directories are provided. The Ozone distribution provides an Ant task to execute OPP. The task class is OPPTask and it's contained in the org.ozoneDB.tools.OPP directory. In the script you'll load the task under the name oppdoclet, as shown here:

```
<!-- =================================================================== -->
<!-- Declare taskdefs                                                    -->
<!-- =================================================================== -->
<taskdef
    name="ejbdoclet"
    classname="xdoclet.modules.ejb.EjbDocletTask"
    classpathref="xdoclet.class.path"
    />

<taskdef
    name="oppdoclet"
    classname="org.ozoneDB.tools.OPP.OPPTask"
    classpathref="ozone.class.path"
    />

<!-- =================================================================== -->
<!-- Prepares the directory structure                                    -->
<!-- =================================================================== -->
<target name="prepare" description="prepares the project's directories">
    <echo>preparing project's directories...</echo>
    <mkdir dir="${classes}"/>
    <mkdir dir="${generated-ejb}"/>
    <mkdir dir="${generated-ozone}"/>
    <mkdir dir="${descriptors-ejb}"/>
    <mkdir dir="${build}"/>
    <mkdir dir="${dist}"/>
</target>

<!-- =================================================================== -->
<!-- Cleans the directory structure                                      -->
<!-- =================================================================== -->
<target name="clean" description="removes all build products">
    <echo>cleaning...</echo>
    <delete dir="${classes}"/>
    <delete dir="${generated}"/>
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
</target>
```

The generate task uses the loaded XDoclet ejbdoclet to generate the required EJB files from the annotated ConferenceOzoneFacadeBean, as shown here:

```
<!-- =================================================================== -->
<!-- Generate EJB glue files using XDoclet's ejbdoclet                   -->
<!-- =================================================================== -->
<target name="generate" description="uses XDoclet to generate EJB files">
    <echo>generating ejbs glue...</echo>
    <ejbdoclet
        destdir="${generated-ejb}"
        excludedtags="@version,@author,@todo"
        ejbspec="2.0"
        force="true"
        >
        <fileset dir="${src}">
            <include name="**/*Bean.java"/>
        </fileset>

        <utilobject kind="physical" includeGUID="true"/>
        <remoteinterface/>
        <localinterface/>
        <homeinterface/>
        <localhomeinterface/>
        <valueobject/>
        <entitycmp/>
        <session/>
        <entitypk/>
        <utilobject cacheHomes="true" includeGUID="true"/>
        <deploymentdescriptor
            destdir="${descriptors-ejb}"
            validatexml="true"
            />
        <jboss
            version="3.0"
            unauthenticatedPrincipal="nobody"
            xmlencoding="UTF-8"
            destdir="${descriptors-ejb}"
            validatexml="true"
            />
    </ejbdoclet>
</target>
```

The compile target depends on the generate task and it compiles both the files in the ${src} directory as well as the generated EJB files in the directory ${generated-ejb}, as shown here:

```
<!-- =================================================================== -->
<!-- Compiles all the classes                                            -->
<!-- =================================================================== -->
<target name="compile" depends="generate"
    description="compiles all sources">
    <echo>compiling...</echo>
    <javac
        destdir="${classes}"
        classpathref="ozone.class.path"
        debug="on"
        deprecation="on"
        optimize="off"
        >
        <src path="${src}"/>
        <src path="${generated-ejb}"/>
    </javac>
</target>
```

Generating the Proxies with the Ozone Post Processor

The OPP is a postprocessor that creates the actual proxy classes, as well as factory classes, which the database direct clients manipulate. OPP is a code generator and it doesn't perform any kind of bytecode manipulation. OPP inspects your Ozone implementation classes (those extending OzoneObject) and creates a proxy peer class, which is actually what the clients interact with. The methods in these proxy classes invoke the server-side object using Ozone RMI.

The target OPP, which depends on the compile target, uses the oppdoclet task to invoke the OPP processor executable, which is contained in the org.ozoneDB.tools.OPP.OPP class. To learn about the command-line options for OPP at the Ozone distribution bin directory, enter the following:

```
opp
```

This should produce the OPP command-line help as shown here:

```
Ozone Post Processor
usage: opp [-ks] [-st] [-p<pattern>] [-ni] [-nf] [-nc] [-q] [-h] [-o<directory>]
[-odmg] [-ip] class [class]*
   -ks       save the generated resolver files
   -KS       save the generated resolver files; do not invoke compiler
   -st       print stack trace
   -p        regular expression to specify update methods
   -ni       do not search interface code for update methods
   -nf       do not create a Factory class
   -q        supress output of any messages
   -o        output directory
   -s        resolver directory
   -odmg     create proxies for the ozone ODMG interface
   -ip       ignore package names
   -nc       do not create code needed for direct invokes and ClientCacheDatabase
   -version  shows version information
   -h        shows this help
```

The OPP target executes OPP using the loaded task. The cache attribute tells OPP to keep the generated source files, the equivalent of using the –KS switch, as shown in the command-line help. The source element tells OPP where the Java source tree containing the implementation files are located; in this case, it's in the ${src} directory. The output attribute option determines where the generated files are to be placed; in this case, they go in the ${generated-ozone} directory.

```
    <!-- =================================================================== -->
    <!-- Builds the proxies                                                  -->
    <!-- =================================================================== -->
    <target name="OPP" depends="compile">
        <oppdoclet output="${generated-ozone}" cache="true">
            <source dir="${src}">
                <include name="**/*Impl.java"/>
            </source>
            <classpath refid="ozone.class.path"/>
        </oppdoclet>
    </target>
```

The ejb-jar uses the jar task to create a JAR file that can be deployed to JBoss, as follows:

```
<!-- =================================================================== -->
<!-- Package the EJB JAR                                                 -->
<!-- =================================================================== -->
<target name="ejb-jar" depends="OPP"
    description="packages the ejb-jar file">
    <echo>packaging ejb-jar...</echo>
    <jar jarfile="${dist}/${jar-name}">
    <metainf dir="${descriptors-ejb}" includes="*.xml"/>
    <fileset dir="${classes}">
        <include name="com/ejdoab/**/*.class" />
        <exclude name="com/ejdoab/client/*" />
     </fileset>
  </jar>
</target>
```

Finally, a convenience target called deploy is added to perform all of the build tasks and to copy the resulting ejb-jar file to the JBoss deploy directory, as shown here:

```
<!-- =================================================================== -->
<!-- Deploys EJB-JAR                                                     -->
<!-- =================================================================== -->
<target name="deploy" depends="clean,prepare,ejb-jar"
    description="deploys the EJB-JAR file to JBoss">
    <copy file="${dist}/${jar-name}" todir="${deploy-dir}"/>
</target>
```

To execute the build file on a command line, type the following:

```
ant deploy
```

This should produce output similar to what's shown here:

```
Buildfile: build.xml

clean:
     [echo] cleaning...
...
prepare:
     [echo] preparing project's directories...
...
generate:
     [echo] generating ejbs glue...
...
```

```
compile:
     [echo] compiling...
...
OPP:
[oppdoclet] Loader is set
[oppdoclet] Begin build
[oppdoclet] Begin Processing com.ejdoab.db.ConferencesManagerImpl...
[oppdoclet]    Begin Resolving update methods...
[oppdoclet]       No ocd was found!
[oppdoclet]    End Resolving update methods
[oppdoclet]    update method [4]: addOrUpdateConference
[oppdoclet]    update method [4]: deleteAllConferences
[oppdoclet]    update method [4]: deleteConference
[oppdoclet]    Begin Generating factory for:
    com.ejdoab.db.ConferencesManagerImpl...
[oppdoclet]    End Generating factory for: com.ejdoab.db.ConferencesManagerImpl
[oppdoclet]    Begin Generating proxy for:
    com.ejdoab.db.ConferencesManagerImpl...
[oppdoclet]    End Generating proxy for: com.ejdoab.db.ConferencesManagerImpl
[oppdoclet] End Processing com.ejdoab.db.ConferencesManagerImpl
    generated in 0.17 seconds.
[oppdoclet] Generation completed with 0 warnings and 0 errors
[oppdoclet] End build
[oppdoclet] Loader is set
[oppdoclet] Begin build
[oppdoclet] Begin Processing com.ejdoab.pojos.ConferenceImpl...
[oppdoclet]    Begin Resolving update methods...
[oppdoclet]       No ocd was found!
[oppdoclet]    End Resolving update methods
[oppdoclet]    update method [4]: setName
[oppdoclet]    update method [4]: setDescription
[oppdoclet]    update method [4]: setStartDate
[oppdoclet]    update method [4]: setEndDate
[oppdoclet]    update method [4]: setAbstractSubmissionStartDate
[oppdoclet]    update method [4]: setAbstractSubmissionEndDate
[oppdoclet]    update method [4]: addTrack
[oppdoclet]    update method [4]: deleteTrack
[oppdoclet]    Begin Generating factory for: com.ejdoab.pojos.ConferenceImpl...
[oppdoclet]    End Generating factory for: com.ejdoab.pojos.ConferenceImpl
[oppdoclet]    Begin Generating proxy for: com.ejdoab.pojos.ConferenceImpl...
[oppdoclet]    End Generating proxy for: com.ejdoab.pojos.ConferenceImpl
[oppdoclet] End Processing com.ejdoab.pojos.ConferenceImpl
    generated in 0.14 seconds.
[oppdoclet] Generation completed with 0 warnings and 0 errors
```

```
[oppdoclet] End build
[oppdoclet] Loader is set
[oppdoclet] Begin build
[oppdoclet] Begin Processing com.ejdoab.pojos.TrackImpl...
[oppdoclet]    Begin Resolving update methods...
[oppdoclet]       No ocd was found!
[oppdoclet]    End Resolving update methods
[oppdoclet]    update method [4]: setDescription
[oppdoclet]    update method [4]: setSubTitle
[oppdoclet]    update method [4]: setTitle
[oppdoclet]    Begin Generating factory for: com.ejdoab.pojos.TrackImpl...
[oppdoclet]    End Generating factory for: com.ejdoab.pojos.TrackImpl
[oppdoclet]    Begin Generating proxy for: com.ejdoab.pojos.TrackImpl...
[oppdoclet]    End Generating proxy for: com.ejdoab.pojos.TrackImpl
[oppdoclet] End Processing com.ejdoab.pojos.TrackImpl generated in 0.121 seconds.
[oppdoclet] Generation completed with 0 warnings and 0 errors
[oppdoclet] End build

ejb-jar:
     [echo] packaging ejb-jar...
...
deploy:
     [copy] Copying 1 file to C:\java\jboss\jboss-3.2.1\server\tcms\deploy

BUILD SUCCESSFUL
Total time: 11 seconds
```

On the JBoss console you should see the archive being deployed with output similar to the following:

```
...
13:39:50,387 INFO  [MainDeployer] Starting deployment of package:
    file:/C:/java/jboss/jboss-3.2.1/server/tcms/deploy/ozone-jboss-test.jar
13:39:51,038 INFO  [EjbModule] Creating
13:39:51,048 INFO  [EjbModule] Deploying ConferenceOzoneFacade
13:39:51,058 INFO  [StatelessSessionContainer] Creating
13:39:51,068 INFO  [StatelessSessionInstancePool] Creating
13:39:51,068 INFO  [StatelessSessionInstancePool] Created
13:39:51,068 INFO  [StatelessSessionContainer] Created
13:39:51,068 INFO  [EjbModule] Created
13:39:51,078 INFO  [EjbModule] Starting
13:39:51,078 INFO  [StatelessSessionContainer] Starting
13:39:51,118 INFO  [StatelessSessionInstancePool] Starting
13:39:51,118 INFO  [StatelessSessionInstancePool] Started
```

```
13:39:51,118 INFO  [StatelessSessionContainer] Started
13:39:51,118 INFO  [EjbModule] Started
13:39:51,118 INFO  [EJBDeployer] Deployed:
    file:/C:/java/jboss/jboss-3.2.1/server/tcms/deploy/ozone-jboss-test.jar
13:39:51,128 INFO  [MainDeployer] Deployed package:
    file:/C:/java/jboss/jboss-3.2.1/server/tcms/deploy/ozone-jboss-test.jar
...
```

Test Client

Finally, you'll need a test client similar to the one used in Chapter 5. In this client you'll look up the ConferenceOzoneFacade Session Bean and manipulate the Conference objects stored in the database. First, you'll retrieve all existing conferences. Next, you'll search for a Conference object by its name. If it isn't found, then you'll create the Conference object and store it in the database, as follows:

```
package com.ejdoab.client;

...

import com.ejdoab.beans.ConferenceOzoneFacade;
import com.ejdoab.beans.ConferenceOzoneFacadeHome;
import com.ejdoab.dto.ConferenceDTO;
import com.ejdoab.dto.TrackDTO;

/**
 * Simple EJB Test – ConferenceOzoneFacade Test
 */
public class Client {
    private static final String ICF = "org.jnp.interfaces.NamingContextFactory";
    private static final String SERVER_URI = "localhost:1099";
    private static final String PKG_PREFIXES =
        "org.jboss.naming:org.jnp.interfaces";

    public static void main(String args[]) {
        Context ctx;
        ConferenceOzoneFacadeHome confHome;
        ConferenceOzoneFacade conf;
```

```
// initial context JBossNS configuration
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, ICF);
env.put(Context.PROVIDER_URL, SERVER_URI);
env.put(Context.URL_PKG_PREFIXES, PKG_PREFIXES);

try {

    // ----------
    // JNDI Stuff
    // ----------

    ctx = new InitialContext(env);
    // look up the home interface
    System.out.println(
        "[jndi lookup] Looking Up ConferenceOzoneFacade” +
        “ Remote Home Interface" );
    Object obj = ctx.lookup("ejb.ConferenceOzoneFacadeHome");
    // cast and narrow
    confHome = (ConferenceOzoneFacadeHome) PortableRemoteObject
        .narrow( obj, ConferenceOzoneFacadeHome.class);
    conf = confHome.create();

    // ----------
    // Tests
    // ----------

    //
    // getAllConferences
    //
    Collection c = conf.getAllConferences();
    if (!c.isEmpty()) {
        Iterator i = c.iterator();
        System.out.println(
            "[getAllConferences] listing conferences in the database:");
        while (i.hasNext()) {
            ConferenceDTO conference = (ConferenceDTO) i.next();
            System.out.println(conference.toString());
        }
    }
```

```
else {
    System.out.println(
        "[getAllConferences] there are no conferences “
        + “in the database");
}

//
// Find Conference by Name
//
String confName = "Apress OSC";
System.out.println("[getConferenceByName] searching with "
    + confName);

ConferenceDTO conference = null;
try {
    conference = conf.getConferenceByName(confName);
} catch (Exception e) {
    e.printStackTrace();
}

Date today = new Date();

if (conference == null) {
    System.out.println(
        "[getConferenceByName] Conference was not found”
        + “, creating it");
    ConferenceDTO newConf =
        new ConferenceDTO(
            "Apress OSC",
            "Apress Open Source Conference",
            today,
            today,
            today,
            today);
    newConf.addTrack(
        new TrackDTO(
          "J2SE",
          "Java 2 Standard Edition",
          "Learn how to build powerful Java desktop applications"));
```

```
                newConf.addTrack(
                    new TrackDTO(
                      "J2EE",
                      "Java 2 Enterprise Edition",
                     "Enterprise Applications powered entirely by Open Source"));
                newConf.addTrack(
                    new TrackDTO(
                        "J2ME",
                        "Java 2 Micro Edition",
                        "Java brings cell phones and PDAs to life"));

                conf.addOrUpdateConference(newConf);
            }
            else {
                System.out.println(
                    "[getConferenceByName] Conference was found\n" +
                    conference + "\n");
                System.out.println("Here are the tracks:\n");
                Collection tracks = conference.getTracks();
                for (Iterator iter = tracks.iterator(); iter.hasNext();) {
                    TrackDTO track = (TrackDTO) iter.next();
                    System.out.println(track);
                }
            }
        }
        catch (RemoteException re) {
            System.out.println("[rmi] remote exception: " + re.getMessage());
        }
        catch (NamingException ne) {
            System.out.println("[naming] naming exception: " + ne.getMessage());
        }
        catch (CreateException ce) {
            System.out.println("[ejb] create exception: " + ce.getMessage());
        }
    }
}
```

Results

To run the test, as you've done previously with all JBoss test clients, you'll need the jbossall-client.jar in the classpath. Compile the test client class (the included Ant build script should have already compiled the class) and execute it by issuing the following command:

```
java –cp classes;%JBOSS_HOME%\client\jbossall-➥
client.jar;%OZONE_HOME%\lib\ozoneServer-1.2-➥
alpha.jar com.ejdoab.client.Client
```

where JBOSS_HOME refers to the JBoss distribution and OZONE_HOME refers to the Ozone distribution. The results of running the test-client application for the first time are shown here:

```
[jndi lookup] Looking Up ConferenceOzoneFacade Remote Home Interface
[getAllConferences] there are no conferences in the database
[getConferenceByName] searching for Apress OSC
[getConferenceByName] Conference was not found, creating it
```

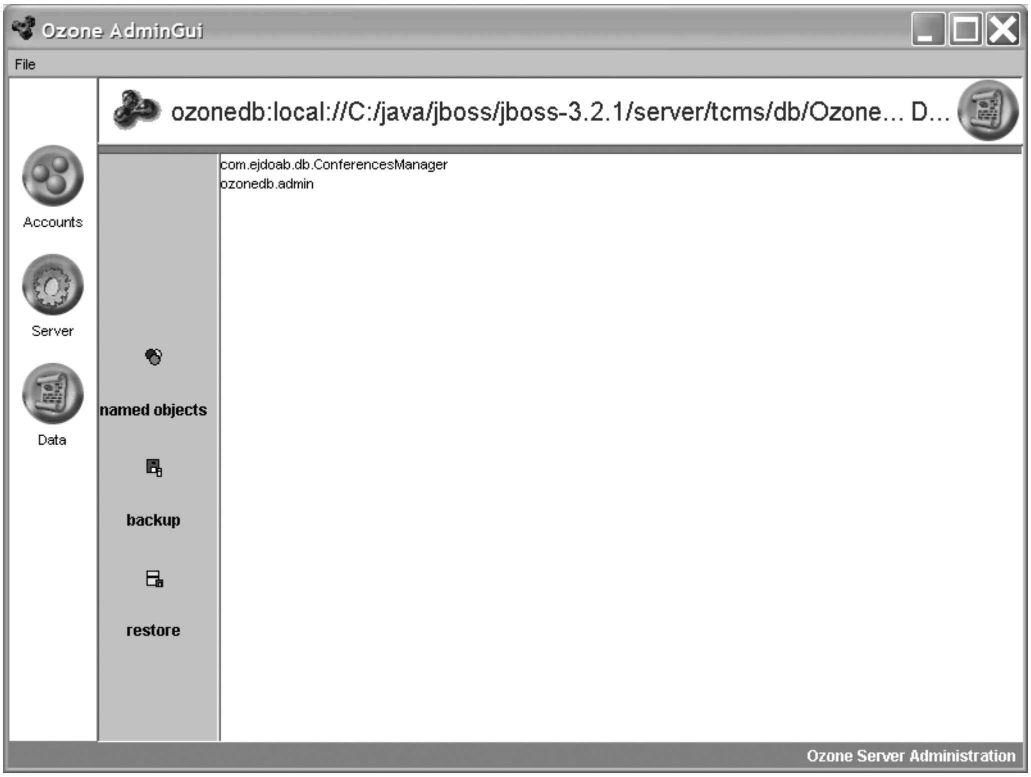
As you can see on the first run, the Conference object isn't found by either of the methods selected, so it's created. On the second run, as you can see in the following output, the newly created Conference object is now found along with its associated Tracks:

```
[jndi lookup] Looking Up ConferenceOzoneFacade Remote Home Interface
[getAllConferences] listing conferences in the database:
[Conference] name = Apress OSC,
             description = Apress Open Source Conference,
             start date = Sun Feb 01 05:34:03 EST 2004,
             end date = Sun Feb 01 05:34:03 EST 2004,
             # tracks = 3
[getConferenceByName] searching with Apress OSC
[getConferenceByName] Conference was found
[Conference] name = Apress OSC,
             description = Apress Open Source Conference,
             start date = Sun Feb 01 05:34:03 EST 2004,
             end date = Sun Feb 01 05:34:03 EST 2004,
             # tracks = 3
```

Here are the tracks:

```
[Track] title = J2SE,
        subTitle = Java 2 Standard Edition,
        description = Learn how to build powerful Java desktop applications
[Track] title = J2EE,
        subTitle = Java 2 Enterprise Edition,
        description = Enterprise Applications powered entirely by Open Source
[Track] title = J2ME,
        subTitle = Java 2 Micro Edition,
        description = Java brings cell phones and PDAs to life
```

You can now use the Ozone AdminGUI tool to browse the database. Figure 6-13 shows the newly created named object ConferencesManager.


Ozone AdminGui
File
ozonedb:local://C:/java/jboss/jboss-3.2.1/server/tcms/db/Ozone... D...
Accounts
Server
Data
named objects
backup
restore
Ozone Server Administration

com.ejdoab.db.ConferencesManager
ozonedb.admin

Figure 6-13. ConferenceManager object on the AdminGui tool

An object-oriented alternative to a relational database, the Ozone DB can handle the complexities of a rich object model. If you're accustomed to working with RMI or EJBs, the Ozone DB will make you feel at home. We expect that in the near future the creation of the remote interface, implementation classes, and OCD files will be automated further with the use of an attribute-oriented programming (AOP) tool such as XDoclet. There are, of course, countless features of the Ozone DB that aren't covered in this chapter. You can learn more about them by visiting the Ozone website (`http:// http://www.ozone-db.org`) or reading through the posts in the Ozone user newsgroup (`comp.java.ozone.user`).

Other Data-Storage Technologies

There are other data-storage technologies worthy of mention that this chapter doesn't cover. This section briefly mentions the choices available and points you to the right places to gather more information.

Java Prevalence

Java Prevalence, as embodied by the Prevayler project (`http://www.prevayler.org`), is a new take on the idea of a totally in-memory database that provides transparent persistence, fault-tolerance, and load-balancing capabilities. The Prevayler project was started by Klaus Wuestefeld, and it's rapidly gaining both supporters and detractors at a very high rate. Prevayler works on the assumption that the available amount of physical memory is sufficient to keep all objects in the system in memory.

Prevayler uses object serialization at set intervals (or on demand) to produce a snapshot of the system. Atomic changes to the state of the working memory are logged as serialized "transactions" before they're applied to the system. If the system crashes, the last snapshot is loaded and any commands logged after the time of the snapshot creation are executed against the snapshot in order to bring the system to its last known state.

For usage purposes, Prevayler works like an object-oriented database except that if you want to mutate the data in a Prevayler system (similar to a named object in Ozone) you must encapsulate the operations that perform the changes in a transaction, which in Prevayler is a serializable class that represents a command object. Each command is applied in a serialized fashion to the system. A Prevayler transaction must be deterministic, meaning that it should always produce the same final state when applied to a Prevayler system in certain state.

Prevayler is a very simple, robust, and fast system that can be used to provide transparent persistence in a J2EE system in certain scenarios. It shares the advantages of object-databases in that it let's you work with POJOs (it's actually one of the least intrusive tools we tested). It can be a great choice for a fault-tolerant in-memory solution or for an object graph that doesn't grow wildly. As mentioned previously the only imposition on the development is in the semantics of the interactions that mutate your prevalent system.

Prevayler can be a great way to prototype your system and gain a baseline for what the performance could be with a complete in-memory solution. Development is fairly simple, it doesn't require any pre- or postprocessing of your classes, bytecode manipulation, or the need to inherit or implement any proprietary classes or interfaces (except for the marker interface java.io.Serializable).

The semantics of a Prevayler transaction are easily applied to a service-oriented architecture.

So, if all of the criteria for using an object-oriented database apply to your system, your object graph can comfortably fit in memory, and if your system has no distribution requirements then Prevayler can be a great choice to provide unparalleled performance to your users.

Native XML Databases

XML is rapidly become the lingua franca for business-to-business transactions, and a new breed of database that's tailored for the efficient storage, retrieval, and management of XML is surfacing. Just as object and relational systems have impedance mismatches, XML, with its document-centric hierarchical paradigm, also has incompatibilities at certain levels with object systems and relational databases. XML databases provide a way to store XML in its natural state. These databases typically create indexes for each document stored, which eases the task of querying and aggregating data.

The XML:DB initiative for XML databases (`http://www.xmldb.org/`) is a group that's looking to standardize the definition of an XML database. It identifies the following areas as examples of applications that could benefit from using a native XML database:

- Corporate information portals
- Membership databases
- Product catalogs
- Parts databases
- Patient-information tracking
- Business-to-business document exchange

A native XML database will ideally enable you to work with native XML technologies and tools such as (in the case of Java) XSLT, XPath, XQuery, JAXP, Xerces/Xalan, JDOM, and XOM. Currently two Open Source implementations exist. They are mature enough (there are many mature commercial XML database such as Software AG's Tamino) and are close to being ready for use in production environments. These are as follows:

- **Xindice** (`http://xml.apache.org/xindice`)**:** The Xindice native XML database (formerly dbXML) is an Apache project that provides a pure-Java XML database that supports the XML:DB API. It uses the XML:DB XUpdate for updates and the W3C's XPath for queries.

- **eXist** (`http://exist.sourceforge.net`)**:** The eXist project provides a native XML database that supports XQuery, autoindexing, and extensions for full-text searching. An XQuery processing servlet (XQueryServlet) can be used in combination with XSLT to generate content such as HTML with simple XQuery files.

XML provides a way for modeling document-based unstructured or semi-structured data that's sometimes difficult to model in the object and relational models. A native XML database can ease the complexity of working with XML by offering a native way to store and work with your XML documents.

Conclusion

In this chapter you've learned that deciding where and how to store your data can be a daunting decision. The storage mechanism you choose will have a deep impact in the way you develop your applications. Although not an in-depth treatise on Java data-storage choices, this chapter should have given you a place to start the quest for the right database for your application.

The lesson of this chapter is that you should understand your application's needs when it comes to storing and manipulating data, and the best way to do this is to prototype and test the main features of your application by using different approaches including CMP EJBs, ORM tools, embedded in-memory SQL databases, OODBMS, and XDBMS.