Getting Started with JBoss

J2EE applications on the JBoss 3.2.x Server

Luke Taylor and The JBoss Group

© JBoss inc, 2004, all rights reserved. The license given with the downloaded version of the book is a single user license. Redistribution of this document is explicitely forbiden without the prior written consent of JBoss inc.

Contents

Preface v

: Foreword - v
: Target Audience - v
: What this Book Covers - vi
: About the Authors - vi

CHAPTER 1 Getting Started 1

1.1: Downloading and Installing JBoss - 1
1.2: Starting and Stopping the Server - 2
  Running as a Service - 4

CHAPTER 2 The JBoss Server – A Quick Tour 5

2.1: Server Structure - 5
  Main Directories - 5
  Server Configurations - 7
2.2: Basic Configuration Issues - 9
  Core Services - 9
    Logging Service - 9
    Security Service - 10
  Additional Services - 11
2.3: The Web Container – Tomcat - 12

CHAPTER 3 About the Example Applications 14

3.1: The J2EE Tutorial - 14
  What's Different? - 15
    Container-Specific Deployment Descriptors - 15
    Database Changes - 16
    Security Configuration - 16
3.2: J2EE in the Real World - 16

CHAPTER 4 The Duke's Bank Application 18

4.1: Building the Application - 18

Preparing the Files -19
Compiling the Java Source -19
Package the EJBs -20
Package the WAR File. -20
Package the Java Client -20
Assembling the EAR -20
The Database -21
Enabling the HSQL MBean and TCP/IP Connections - 21
Creating the Database Schema - 22
The HSQL Database Manager Tool - 23
Deploying the Application -24
4.2: JNDI and Java Clients -25
The *jndi.properties* File -25
4.3: Security -26
Configuring a Security Domain -26
UsersRolesLoginModule Files- -27
The J2EE Security Model -28
Authentication- 28
Access Control (Authorization)- 29
Application JNDI Information in the JMX Console -29

CHAPTER 5 JMS and Message-Driven Beans **31**

5.1: Building the Example -32
Compiling and Packaging the MDB and Client -32
Specifying the Source Queue for the MDB - 32
5.2: Deploying and Running the Example -32
Running the Client -33
5.3: Managing JMS Destinations -33
The *jbossmq-destinations-service.xml* File -34
Using the DestinationManager from the JMX Console -34
Administering Destinations -34

CHAPTER 6 Container-Managed Persistence **36**

6.1: Building the Example -36
Compiling the Code -37
Packaging the Jars -37
6.2: Deploying and Running the Application -37
Running the Client -39
6.3: CMP Customization -39
XDoclet -41

CHAPTER 7 *Web Services with JBoss.Net* 42

7.1: JBoss.net -42
7.2: Duke’s Bank as a Web Service -43
The Web Service Archive (WSR) File -43
Building and Deploying the WSR File -44
Running the Client -45
Net Traffic Analysis -45

CHAPTER 8 *Using other Databases* 48

8.1: DataSource Configuration -48
JDBC-Wrapper Resource Adapters -48
DataSource Configuration Files -49
8.2: Examples -49
Using MySQL as the Default DataSource -49
Creating a Database and User - 50
Installing the JDBC Driver and Deploying the DataSource - 51
Testing the MySQL DataSource - 51
Setting up an XADataSource with Oracle 9i -52
Padding Xid Values for Oracle Compatibility - 52
Installing the JDBC Driver and Deploying the DataSource - 53
Testing the Oracle DataSource - 54

CHAPTER 9 *Security Configuration* 56

9.1: Security Using a Database -56
9.2: Using Password Hashing -58

Preface

Foreword

JBoss started out as an EJB container and has evolved over several years into a fully fledged application server. While the architecture has grown to support many new software technologies and additional features, there has always been an emphasis on the implementation of the J2EE standards, regardless of whether official certification has been achieved or not.

For the foreseeable future, JBoss will continue to be – first and foremost – a J2EE application server.

Target Audience

The main aim of this book is to get you up and running with JBoss as quickly as possible. We will use Sun's J2EE 1.3 tutorial examples where possible to illustrate the deployment and configuration of J2EE applications in JBoss. While the book is not intended to teach you J2EE, we will be covering the subject from quite a basic standpoint so it will still be useful if you are new to J2EE. If you would like to use JBoss to run the standard Sun J2EE tutorials then this is the book for you. It should ideally be read in parallel with the tutorial texts.

What this Book Covers

The scope of this book is using J2EE 1.3 on the JBoss 3.2.x series. At the time of writing, the latest release is version 3.2.3. You should use this version or later with the examples.

We will cover downloading and installation and see how to start JBoss. Then we'll have a quick tour of the server directory structure and layout, the key configuration files and services.

Moving on to the examples, we'll look at how to deploy the "Duke's Bank" application from the Sun J2EE Tutorial. This will let you see JBoss in action as quickly as possible and also gives you a chance to get some practical experience of simple configuration and deployment issues. Further chapters cover other J2EE topics which aren't used in Duke's Bank – JMS Messaging (and Message-Driven Beans) and container-managed persistence (CMP). These also make use of the J2EE tutorial examples.

There is a separate chapter on web services. We work through how to expose EJB methods from the Duke's Bank application through web servies and then call them with a Java SOAP client.

Configuration of databases is an important issue and this is covered in "Using other Databases" on page 48. We also work through some step-by-step examples.

In "Security Configuration" on page 56 we look at some more advanced security configuration options.

Suggestions for additional topics are always welcome.

About the Authors

Luke Taylor is an independent consultant based in Glasgow, Scotland. He obtained a Ph.D. in theoretical nuclear physics from Glasgow University and subsequently worked in London in software development and as a consultant specializing in Java, CORBA, and security technologies. He founded the company Monkey Machine (**http://www.monkeymachine.ltd.uk**) which offers services primarily in Java and J2EE with a focus on open source implementations such as JBoss.

CHAPTER 1 Getting Started

1.1. Downloading and Installing JBoss

There are two ways you can get a copy of JBoss; you can either download a binary distribution or you can obtain the latest version directly from the source repository using cvs and build it yourself. This is straightforward enough, but unless you need the latest code for a specific reason then you should probably stick to the pre-packaged versions, at least to begin with.

You can download the latest version from the JBoss web site

http://www.jboss.org

At the time of writing, the latest stable release is version 3.2.3. The binary versions are available as either zip or tar.gz files – the contents are the same so grab whichever one is most convenient for the platform you're running on. Once it's downloaded, unpack the archive to a suitable location on your machine. It should all unpack into a single directory named "jboss-" with a version-number suffix. Make sure you don't use a directory which has any spaces in the name (such as the "Program Files" directory on Windows) as this may cause problems. There are no additional installation steps needed before you can get started.

1.2. Starting and Stopping the Server

First make sure you have an up-to-date version of Java on your machine. You need the JDK, not just the JRE. You should also make sure the JAVA_HOME environment variable is set to point to your JDK installation[1].

Now try running the server: you'll find a bin directory inside the main JBoss directory which contains various scripts. Execute the “run” script (run.bat if you're on Windows, run.sh if you're on Linux or another Unix-like system). You should then see the log messages from all the JBoss components as they are deployed and started up. The last message (obviously with different values for the time and start-up speed) should be:

```
00:23:38,718 INFO  [Server] JBoss (MX MicroKernel) [3.2.3 (build: CVSTag=JBoss_3_2_3
date=200311301445)] Started in 26s:593ms
```

To get a live view of the running server, point your browser at the URL

http://localhost:8080/jmx-console[2].

You should see something similar to Figure 1.1. This is the JBoss Management Console which provides a raw view of the JMX MBeans which make up the server[3]. You don't really need to know much about these to begin with, but they can provide a lot of information about the running server and allow you to modify its configuration, start and stop components and so on.

For example, find the “service=JNDIView” link and click on this. This particular MBean provides a service to allow you to view the structure of the JNDI namespaces within the server. Now find the operation called “list” near the bottom of the MBean view page and click the “invoke”. The operation

1. This is required so that the tools.jar file, which contains the javac compiler classes, can be located. Javac is needed for compiling JSPs.
2. Note that by default the web container runs on port 8080, so make sure you don't have anything else already on your machine using that port. Also, there won’t be a default web application deployed at the root context, so browsing to `http://localhost:8080` will produce a “HTTP Status 500” error from Tomcat. On some machines, the name “localhost” won’t resolve properly and you should use the local loopback address “127.0.0.1” instead.
3. The Java Management Extensions (JMX) framework is a key part of the JBoss architecture. The instrumentable components it defines are called MBeans (“Managed Beans”).

returns a view of the current names bound into the JNDI tree – very useful when you start deploying your own applications and want to know why you can't resolve a particular EJB name.

FIGURE 1.1. View of the JMX Management Console Web Application

Have a look at some of the other MBeans and their listed operations, and try changing some of the configuration attributes and see what happens. None of the changes made through the console are persistent; the original configuration will be reloaded when you restart JBoss so you can experiment freely and shouldn't be able to do any permanent damage.

To stop the server, you can type Ctrl-C or you can run the shutdown script from the bin directory. Alternatively, you can use the management console (look for “type=Server” under the section “jboss.system” and invoke the “shutdown” operation).

1.2.1. Running as a Service

In a real deployment scenario, you won’t want to stop and start JBoss manually but will want it to run in the background as a service or daemon when the machine is booted up. The details of how to do this will vary between platforms and will require some system administration knowledge and root privileges.

On Linux or other Unix-like systems, you will have to install a startup script (or get your system administrator to do it). There is an example in the JBoss *bin* directory called *jboss_init_redhat.sh* which you can modify and use.

On a Windows system, you can use a utility like Javaservice which is freely available from

http://www.alexandriasc.com/software/JavaService/index.html.

CHAPTER 2

The JBoss Server – A Quick Tour

2.1. Server Structure

Now that you've downloaded your copy of JBoss and have run the server for the first time, the first thing you will want to know is how the contents are laid out and what goes where. At first glance there seems to be a lot of stuff in there and it's not obvious what you need to look at and what you can safely forget about (at least to begin with) so we'll explore the server directory structure, locations of the key configuration files, log files, deployment and so on. It's worth familiarising yourself with the layout at this stage as it will help you understand the JBoss service architecture and you'll know your way around when it comes to deploying your own applications.

2.1.1. Main Directories

The binary distribution unpacks into a top-level *JBoss-3.2.3* directory. Throughout the book, we will refer to this as the JBOSS_DIST directory. There are four sub-directories immediately below this:

- *bin* – contains various scripts and associated files. We've already seen the "run" script which starts JBoss.

- *client* – stores configuration and jar files which may be needed by a Java client application or an external web container. You can select archives as required or use *jbossall-client.jar*.
- *docs* – contains the XML DTDs used in JBoss for reference (these are also a useful source of documentation on JBoss configuration specifics). There are also example JCA[1] configuration files for setting up datasources for different databases (such as MySQL, Oracle, Postgres)[2].
- *lib* – jar files which are needed to run the JBoss microkernel. You should never add any of your own jar files here.
- *server* – each of the subdirectories in here is a different server configuration. The configuration is selected by passing the option "-c <*config name*>" to the run script. We'll look at these next.

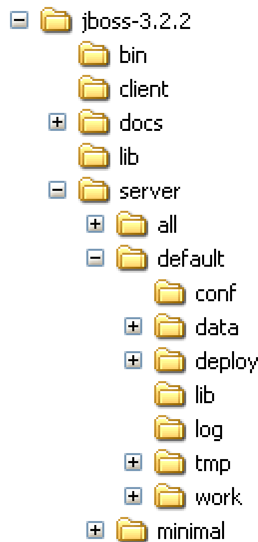

FIGURE 2.1. JBoss Directory Structure

1. J2EE Connector Architecture – provides a standard for providing connectivity between application servers and existing Enterprise Information Systems (EIS).
2. JBoss comes with an embedded instance of the free Hypersonic database and there is a corresponding datasource set up in the default configuration. If you want to use another database then you have to add the appropriate JCA configuration information. We'll see how to do this later.

2.1.2. Server Configurations

Fundamentally, the JBoss architecture consists of a JMX MBean server instance (the “microkernel”) and a set of pluggable component services – the JMX MBeans. This makes it easy to assemble different configurations and gives you the flexibility to tailor them to meet your requirements. You don’t have to run a large, monolithic server all the time; you can remove the components you don’t need (which can also reduce the server startup time considerably) and you can also integrate additional services into JBoss by writing your own MBeans. You certainly don’t need to do this to be able to run standard J2EE applications though – everything you need is already there. You don’t need a detailed understanding of JMX either but it’s worth keeping a picture of this basic architecture in mind as it is central to the way JBoss works.

Within the *server* directory, there are three example configurations: *all*, *default* and *minimal*, each of which installs a different set of services. Not surprisingly, the default configuration is used if you don’t pass any parameters to the run script, so that’s the one we were running in the previous chapter. It contains everything you need to run a stand-alone J2EE server. The other two are

- *minimal* – the bare minimum required to start JBoss. It starts the logging service, a JNDI server and a URL deployment scanner to find new deployments. This is what you would use if you want to use JMX/JBoss to start your own services without anything else from J2EE. This is just the bare server – there is no web container, no EJB or JMS.
- *all* – starts all the available services. This includes the RMI/IIOP and clustering services and the web-services deployer which aren’t loaded in the default configuration.

You can add your own configurations too. The best way to do this is to copy an existing one that is closest to your needs and modify the contents. For example, if you weren’t interested in using messaging, you could copy the “default” directory, renaming it as “myconfig”, remove the *jms* subdirectory and then start JBoss with the command

```
run -c myconfig
```

Whichever server configuration you’re using, the corresponding directory effectively *is* the server while JBoss is running. It contains all the code and configuration information for the MBeans, it’s where the log output goes and it’s where you deploy your applications. Let’s take a look at the contents of the default directory. If you haven’t tried running the server yet, then do so now, as some of the sub-directories are only created if JBoss has previously been started. The full directory structure is shown in Figure 2.1 . The sub-directories are:

- *conf* – contains the *jboss-service.xml* file which specifies the core services. Also used for additional configuration files for these services.
- *data* – this is where the embedded Hypersonic database instance stores its data. It is also used by JBossMQ (the JBoss implementation of JMS) to store messages on disk.

- *deploy* – you deploy your application code (jar, war and ear files) by dropping them in here. It is also used for hot-deployable services (those which can be added to or removed from the running server) and for deploying JCA resource adapters[3]. That's why there's a lot of stuff in there already – in particular you'll notice the jmx-console application (an unpacked war file) which we were using earlier. The directory is constantly scanned for updates and any modified components will be re-deployed automatically. We'll look at deployment in more detail later.
- *lib* – jar files needed by this server configuration. You can add required library files here for JDBC drivers etc.
- *log* – this is where the logging information goes. JBoss uses the Jakarta log4j package for logging and you can also use it directly in your own applications from within the server.
- *tmp* – used by the deployer for temporary storage of unpacked applications etc.
- *work* – used by Tomcat for compilation of JSPs.

The *data, log, tmp* and *work* directories are created by JBoss so won't exist until you've run the server at least once.

We've touched briefly on the issue of hot-deployment of services in JBoss so let's have a look at a practical example of this before we go on to look at server configuration issues in more detail. Start JBoss if it isn't already running and take a look in the *deploy* directory again (make sure you're looking at the one in the default configuration directory). Remove the *mail-service.xml* file and watch the output from the server:

```
18:20:51,312 INFO  [MainDeployer] Undeploying file:/F:/servers/jboss-3.2.2/server/default/deploy/
mail-service.xml
18:20:51,312 INFO  [MailService] Stopping18:20:51,312 INFO  [MailService] Mail service 'java:/Mail'
removed from JNDI
18:20:51,312 INFO  [MailService] Stopped
18:20:51,312 INFO  [MailService] Destroying
18:20:51,312 INFO  [MailService] Destroyed
18:20:51,312 INFO  [DeploymentInfo] Cleaned Deployment: file:/F:/servers/jboss-3.2.2/server/
default/tmp/deploy/tmp32144mail-service.xml
18:20:51,328 INFO  [MainDeployer] Undeployed file:/F:/servers/jboss-3.2.2/server/default/deploy/
mail-service.xml
```

Then replace the file and watch the JBoss re-install the service: hot-deployment in action.

3. The J2EE Connector Architecture defines the Resource Adapter Archive (RAR) file – used for storing JCA implementations for a particular resource.

2.2. Basic Configuration Issues

Now that we've examined the layout of the JBoss server, we'll take a look at some of the main configuration files and what they're used for, again relative to the *default* configuration directory.

2.2.1. Core Services

The core services which are started first are specified in the *conf/jboss-service.xml* file. If you have a look at this file in an editor you'll see MBeans for various services including logging, security, JNDI (and the JNDIView service which we saw earlier). You can try commenting out the entry for the JNDI-View service like so:

```
<!--
    <mbean code="org.jboss.naming.JNDIView" name="jboss:service=JNDIView">
    </mbean>
-->
```

If you then restart JBoss, you'll see that the JNDIView service no longer appears in the management console listing. In practice, you should rarely, if ever, need to modify this file, though there is nothing to stop you adding extra MBean entries in here if you want to. The alternative is to use a separate file in the deploy directory and your service will then also be hot-deployable.

2.2.1.1 Logging Service

We mentioned already that *log4j* is used for logging. If you're not familiar with this package and would like to use it in your applications, you should read more about it on the Jakarta web site. JBoss uses an XML configuration file to set up *log4j*. You can find this file in the *conf* directory. It defines a set of "appenders" for logging[4]. By default, JBoss produces output to both the console and a log file (stored in the log directory). The logging level on the console is INFO whereas the file contains all logging. So if things are going wrong and there doesn't seem to be any useful information in the console, always check the log file to see if there are any debug messages which might help you track down the problem. You may also have to boost the logging limits set for individual categories. For example you will see further down the *log4j.xml* file you may see the entry

```
<!-- Limit JBoss categories to INFO -->
<category name="org.jboss">
  <priority value="INFO"/>
</category>
```

4. "appender" is a log4j term. It specifies a particular output logging destination, what categories of messages should go there, the message format and the level of filtering (DEBUG, WARN, INFO etc.) which should be applied.

which limits the level to INFO for all JBoss classes (apart from those which have more specific overrides provided). If you change this to DEBUG it will produce a lot more logging output.

The file appender is set up to produce a new log file every day, so it doesn't produce a one every time you restart the server and it won't write to a single file indefinitely. The current log file is called *server.log*. Older files have the date they were written added to the name. You will notice that the *log* directory also contains HTTP request logs which are produced by the web container.

As another example, let's say you wanted to set the output from the container-managed persistence engine to DEBUG level and to redirect it to a separate file, called *cmp.log,* in order to analyze the generated SQL commands. You would add the following code to the *log4j.xml* file:

```
<appender name="CMP" class="org.jboss.logging.appender.RollingFileAppender">
  <errorHandler class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
  <param name="File" value="${jboss.server.home.dir}/log/cmp.log"/>
  <param name="Append" value="false"/>
  <param name="MaxFileSize" value="500KB"/>
  <param name="MaxBackupIndex" value="1"/>

  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%d %-5p [%c] %m%n"/>
  </layout>
</appender>

<category name="org.jboss.ejb.plugins.cmp">
  <priority value="DEBUG" />
  <appender-ref ref="CMP"/>
</category>
```

which creates a new file appender and specifies that it should be used by the logger (or "category") for the package `org.jboss.ejb.plugins.cmp`. This will be useful when we come to look at CMP (See "Container-Managed Persistence" on page 36.). Full documentation on using log4j can be found at

http://jakarta.apache.org/log4j.

2.2.1.2 Security Service

The security domain information is stored in the file *login-config.xml* a list of named security domains, each of which specifies a number of JAAS[5] login modules which are used for authentication purposes in that domain. When you want to use security in an application, you specify the name of the domain you want to use in the application's JBoss-specific deployment descriptors, *jboss.xml* and/or *jboss-web.xml*.

5. The Java Authentication and Authorization Service. JBoss uses JAAS to provide pluggable authentication modules. You can use the ones that are provided or write your own if have more specific requirements.

2.2.2. Additional Services

The non-core, hot-deployable services are added to the *deploy* directory. They can be either XML descriptor files (called *<name>-service.xml*) or JBoss “Service Archive” (SAR) files. SARs contain both the XML descriptor and additional resources which the service requires (e.g. classes, library jar files or other archives), all packaged up a single archive.

We’ll go through the deploy directory in the default configuration and identify the contents. This is really just for the sake of completeness, so you can skip this section unless you’d like to know more about the what the existing MBean components are for. In the default configuration deploy directory, you’ll find the following files and sub-directories:

- *http-invoker.sar* – provides RMI/HTTP access for MBeans and EJBs.
- *jbossweb-tomcat41.sar* – an expanded SAR file containing the embedded Tomcat service. This provides the standard web container within JBoss.
- *jms* – JMS-specific services grouped together in a subdirectory.
- *jmx-console.war* – the management console web application which we used in the previous chapter.
- *jmx-invoker-adaptor-server.sar* – provide remote access to the JMX MBean server.
- *management* – sub-directory containing alternative management services, including an improved web console. Currently still in development.
- *cache-invalidation-service.xml* – allows customized control of the EJB cache via JMS.
- *hsqldb-ds.xml* – sets up the embedded Hypersonic database service and the default data source.
- *jboss-jca.sar* – the JBoss JCA implementation. Allows the deployment of JCA resource adaptors within JBoss.
- *jboss-local-jdbc.rar* and *jboss-xa-jdbc.rar* – these are JCA resource adapters to integrate JDBC drivers which support DataSource and XADataSource respectively but for which there is no proprietary JCA implementation.
- *mail-service.xml* – allows applications and services to use JavaMail from within JBoss. Must be configured with relevant mail server information.
- *properties-service.xml* – amongst other things, allows the setting of global system properties (as returned by `System.getProperties`).
- *schedule-manager-service.xml* and *scheduler-service.xml* – task scheduling service.
- *snmp-adaptor.sar* – JMX to SNMP adaptor.
- *sqlexception-service.xml* – provides a means of identifying non-fatal SQL exceptions for a given JDBC driver.
- *transaction-service.xml* – together with the MBeans in *conf/jboss-service.xml,* sets up the JBoss transaction manager and associated services.

- *user-service.xml* – a place to add your own MBeans.
- *uuid-key-generator.sar* – generates unique UUID-based keys.

The files in the *jms* subdirectory are all specific to JMS messaging. Many of them are "invocation layers" which define the transport protocols over which the message transfer takes place. Additional files are:

- *hsqldb-jdbc2-service.xml* – implements caching and persistence using the embedded HSQL database. Also contains the DestinationManager MBean which is the core service for the JMS implementation.
- *jbossmq-destinations-service.xml* – sets up standard JMS Topics and Queues which are used by the JBoss test suite.
- *jbossmq-service.xml* – additional services for JMS, including the interceptor configuration.
- *jms-ra.rar* – resource adapter to allow JMS connection factories to be handled by JCA.
- *jms-ds.xml* – sets up JBoss Messaging as the default JMS provider and supplies JCA configuration information to integrate the JMS resource adapter with JBoss JCA[6].

More detailed information on all these services can be found in "*JBoss Administration and Development*" which also provides comprehensive information on server internals and the implementation of services such as JTA and the J2EE Connector Architecture (JCA).

2.3. The Web Container – Tomcat

JBoss now comes with Tomcat 4.1.x as the default web container. The embedded Tomcat service is the expanded SAR *jbossweb-tomcat41.sar* in the deploy directory. All the necessary jar files needed by Tomcat can be found in there, as well as a *web.xml* file which provides a default configuration set for web applications. If you are already familiar with configuring Tomcat, have a look at the *META-INF/jboss-service.xml* file. Within the MBean declaration for the Tomcat service you will find an element

```
<attribute name="Config"> .... </attribute>
```

which contains a subset of the standard Tomcat format configuration information. As it stands, this includes setting up the HTTP connector on the default port 8080, an AJP connector on port 8009 (can be used if you want to connect via a web server such as Apache) and an example of how to configure an SSL connector (commented out by default).

6. Although the "*-ds*" suffix is used, it doesn't apply only to DataSource configuration but can be used to configure any resource adapter for use with JBoss JCA. The `<adapter-display-name>` element links the information in the JBoss descriptor to a specific resource adapter.

You shouldn't need to modify any of this other than for advanced use. If you've used Tomcat before as a stand-alone server you should be aware that things are a bit different when using the embedded service. JBoss is in charge and you shouldn't need to access the Tomcat directory at all – web applications are deployed by putting them in the JBoss deploy directory and logging output from Tomcat (both internal and access logs) can be found in the JBoss log directory.

CHAPTER 3

About the Example Applications

3.1. The J2EE Tutorial

We will make use of the example applications provided by Sun in the J2EE tutorial, in particular the "Duke's Bank" application. You can find the tutorial on-line at

http://java.sun.com/j2ee/tutorial/1_3-fcs/index.html

You should read the getting started information there and download the example code from

http://java.sun.com/j2ee/1.3/download.html#tutorial

Duke's Bank also makes some use of the Jakarta "Struts" framework which you can get from **http://jakarta.apache.org/struts**.

We will look at how to run the code in JBoss, supplementing the tutorial where necessary with JBoss-specific configuration information and deployment descriptors. While you're online, make sure you've downloaded the additional code that comes with this document – the file should be a zip archive called *jbossj2ee-src.zip*. You should be able to get it from

http://www.jboss.org/docs/jbossj2ee-src.zip

The tutorial uses the Apache "ant" build tool, which you should download and install[1]. Ant is almost universally used in Java projects these days so if you aren't already familiar with its use then we recommend you spend some time reading the documentation that comes with it and learning the basics of Ant build files. The default file name is *build.xml* ant it contains a set of "targets" which you can use to perform automated tasks in your project. Usually all you will have to do is run the "ant" command in the directory which contains the build file. The default target in the file will perform the necessary work.

The tutorial explains how to run the applications with the J2EE SDK Reference Implementation server. Our aim will be to deploy them in JBoss.

3.1.1. What's Different?

J2EE technologies are designed so that the code is independent of the server in which the application is deployed. The deployment descriptors for EJBs and web applications (*ejb-jar.xml* and *web.xml*, respectively) are also standard and do not change between different J2EE containers. However, there are still one or two things that need to be done in order to move the application to JBoss. In particular, we have to supply JBoss-specific descriptors and make sure that the database scripts will work.

3.1.1.1 Container-Specific Deployment Descriptors

Container-specific information is usually contained in extra XML descriptors which map logical information used in the application (such as JNDI names and security role names) to actual values which are used in the server. JBoss uses separate files for the EJB and web modules of an application, called *jboss.xml* and *jboss-web.xml,* respectively. There is also a client version of these files which fulfils the same role in a Java client, in combination with the J2EE *application-client.xml* descriptor[2]. If container-managed persistence (CMP) is being used for entity EJBs, it is also possible to configure the JBoss persistence engine through the *jbosscmp-jdbc.xml* file.

The J2EE SDK refers to these as "runtime descriptors" and defines all the information under one XML DTD. The files are all called *sun-j2ee-ri.xml* once they have been added to the packaged archives by the build process.

1. You can get an up-to-date copy of Ant from **http://ant.apache.org/.** Make sure you are using version 1.5.4 or later.
2. Support for the J2EE application client framework was introduced in JBoss 3.2.3

3.1.1.2 Database Changes

The J2EE SDK comes with the Cloudscape database and this is used throughout the tutorials. We will be using the Hypersonic database which runs as an embedded service within JBoss.

In a real-world situation, porting an application to a different databases is rarely straightforward, especially if proprietary features such as sequences, stored procedures and non-standard SQL are used. For these simple applications, though it is relatively easy. When we look at the Duke's Bank application in the next chapter, you will see that there are only a few minor syntax changes required in the database scripts.

We'll look at how to configure JBoss to use a different database in "Using other Databases" on page 48.

3.1.1.3 Security Configuration

J2EE defines how you specify security constraints within your application, but doesn't say how the authentication and access control mechanisms are actually implemented by the server or how they are configured. As we mentioned earlier, JBoss uses JAAS to provide a pluggable means of incorporating different security technologies in your applications. It also comes with a set of standard modules for the use of file, database and LDAP-based security information. We'll start out using file-based information as this is the simplest approach.

3.2. J2EE in the Real World

The examples here are only intended to get you up and running with JBoss and to help you familiarise yourself with the basics. The applications definitely aren't intended to reflect how you should go about writing production J2EE software – indeed there is a lot of differing opinion on this subject. Many people disagree on the use of EJBs for example, particularly the use of entity beans; the use of bean-managed persistence is especially controversial yet is convenient for examples. There is also endless debate about the use of different web technologies (it would be safe to say that not everyone loves JSPs) and the numerous different "Model-2" frameworks that are out there. Struts was one of the first and is probably the best known but is not without its critics. We've provided some sources at the end of this chapter which you can check out for more information.

Similarly we wouldn't necessarily recommend that you set up your projects using the same structure as the examples. We've stuck to the simple layout of the originals but in practice you may want to do things differently. For a start you'll need to include test code which will often mean writing tests using the JUnit test framework or one of its close relations. You'll also need a means of running it as part of your build. Ant can help you here as it has tasks which are used to run JUnit tests.

If you're starting out, your best bet is probably to look at some existing open-source projects and see how they are structured, and then pick something appropriate for your project. Alternatively you might want to look at a tool like Maven

http://maven.apache.org

which attempts to go beyond Ant and provide a standardized framework for building and testing projects.

Finally, we hope you'll realise that there's a lot more depth to JBoss than we can hope to cover here and once you've worked your way through this basic introduction, we hope you'll be eager to learn more. JBoss is also a continually evolving project with lots of plans for the future. So keep an eye on the bleeding-edge version, even if you're running all your production applications on the stable 3.2.x series.

TABLE 1. Further Information Sources

"JBoss Admin. and Development Guide" (Scott Stark et al.) – comprehensive JBoss documentation covering advanced JBoss topics.	**http://www.jboss.org/docs/index**
"JBoss Clustering" (Sacha Labourey and Bill Burke) – how to run clustered JBoss servers for performance and high availability.	**http://www.jboss.org/docs/index**
JBoss Workbook for "Enterprise Java Beans – 3rd Edition"	**http://www.oreilly.com/catalog/entjbeans3/workbooks/index.html**
"Mastering EJB" (Ed. Roman et al.) – free PDF of book covering EJB 2.0 specification. Very pro-EJB.	**http://www.theserverside.com/books/masteringEJB/index.jsp**
"Expert One-on-One: J2EE Design and Development" (Rod Johnson) – in-depth discussion of J2EE in real-world projects.	**http://www.wiley.com/WileyCDA/WileyTitle/productCd-0764543857.html**

CHAPTER 4

The Duke's Bank Application

One of the first thing you'll want to do once you've got a copy of JBoss is find out how to get an application up and running and see what's involved. So we'll do just that with the Duke's Bank example from the J2EE tutorial.

Duke's Bank demonstrates a selection of J2EE technologies working together to implement a simple on-line banking application. It uses EJBs, web components (JSPs and servlets) and uses a database to store the information. The persistence is bean-managed, with the entity beans containing the SQL statements which are used to manipulate the data.

We won't look in detail at its functionality or comment on the implementation but will concentrate on a step-by-step guide to building and deploying it in JBoss.

4.1. Building the Application

You should already have downloaded the J2EE 1.3 tutorial files and the examples which contain Duke's Bank, as described in "The J2EE Tutorial" on page 14. Make sure you have the 1.3 tutorial files and *not* the 1.4 ones, which contain the same examples but with a different directory layout.

The application also makes use of the Struts web framework so you must download this too. You can get it from

http://jakarta.apache.org/struts .

If you are following the tutorial instructions to build it for the reference implementation, these were written for Struts 1.0 which is now out of date. It will work with version 1.1 but you must also add the extra jar files supplied with Struts to the application, along with the *struts.jar* file.

We'll go through building and deploying the application first and then look at things in a bit more detail.

4.1.1. Preparing the Files

You should be able to obtain the supplementary JBoss files from the same place as this document – the file should be a zip archive called *jbossj2ee-src.zip*. Download this and unpack it into the *j2eetutorial* directory, adding to the existing tutorial files. All the Duke's Bank code is in a *bank* subdirectory and you should find a *jboss-build.xml* file sitting in there if you've unpacked the files correctly. This is our ant build script for the JBoss[1] version of the application. The targets you'll find in it are pretty similar to the original ones.

Download the struts distribution, as above, and copy the *struts.jar, struts-logic.tld* and the supporting jakarta-commons jars (all those prefixed with "*commons-*") to *bank/jar.*

In the *j2eetutorial* directory you should find a file called "*build.properties*". Edit this to set the *jboss.home* property to the full path to your JBoss 3.2.x installation[2]. The build process makes use of the jar files and utilities that come with JBoss so it needs to know where to find them. If you've unpacked JBoss 3.2.3 to the "C:" drive on a windows machine, you would set it to

```
# Set the path to the JBoss directory containing the JBoss application server
# (This is the one containing directories like "bin", "client" etc.)

jboss.home=C:/jboss-3.2.3
```

4.1.2. Compiling the Java Source

At the command line, change to the *j2eetutorial/bank* directory. All the build commands will be run from here. Compilation should be pretty straightforward – just type the command:

```
ant -f jboss-build.xml compile
```

1. Rather than just overrating the existing *build.xml* file, we've used a different name from the default. This means that ant must now be run as "ant -f jboss-build.xml".
2. i.e. the JBOSS_DIST directory (See "Main Directories" on page 5.)

which runs the "compile" target in the build script. If there aren't any errors, you should find a newly created *build* directory with the class files in it.

4.1.3. Package the EJBs

The application has three separate EJB jars: *account-ejb.jar*, *customer-ejb.jar* and *tx-ejb.jar*. Each contains the code and descriptors (*ejb-jar.xml* and *jboss.xml*) for the corresponding entity bean and an associated "controller" session bean which the clients interact with (it is generally considered a bad idea for clients to talk directly to entity beans). Executing the command

```
ant -f jboss-build.xml package-ejb
```

should create them (in the *jar* directory).

4.1.4. Package the WAR File.

Next target is the web application which provides the front end to allow users to interact with the business components (the EJBs). The web source (JSPs, images etc.) is contained in the *src/web* directory and is added unmodified to the archive. The ant WAR task also adds a *WEB-INF* directory which contains the files which aren't meant to be directly accessed by a web browser but are still part of the web application. These include the deployment descriptors (*web.xml* and *jboss-web.xml*), class files, (e.g. servlets and EJB interfaces) and extra jars and JSP tag-library descriptors required by the web application (the Struts files in this example). The command to build the web client WAR file is

```
ant -f jboss-build.xml package-web
```

4.1.5. Package the Java Client

In addition to the web interface, there is a standalone Java client for administering customers and accounts. You can build it using the command

```
ant -f jboss-build.xml package-client
```

It contains the *application-client.xml* and *jboss-client.xml* descriptors as well as the client *jndi.properties* file. The client jar will also be included as an additional module in the EAR file and the server.

4.1.6. Assembling the EAR

The EAR file is the complete application, containing the three EJB modules and the web module. It must also contain an additional descriptor called *application.xml.* It is also possible to deploy EJBs and web application modules individually but the EAR provides a convenient single unit.

```
ant -f jboss-build.xml assemble-app
```

should produce the final file *JBossDukesBank.ear.*

4.1.7. The Database

Before we can deploy the application, we need a viable database for it to run against. If you are writing an application which uses container-managed EJB persistence, you can configure the engine to create the tables for you at deployment, but otherwise you have to have a set of scripts to do the job. This is also a convenient way of pre-populating the database with data.

4.1.7.1 Enabling the HSQL MBean and TCP/IP Connections

The HSQL database can be run in one of two modes: in-process or client-server. Since we are going to be running the SQL scripts using a tool which connects to the database we want to make sure the database is running in client-server mode and will accept TCP/IP connections (the HSQL documentation refers to this as "server" mode). In later versions of JBoss, the client-server mode is disabled to prevent direct database access which could be a security risk if the default login and password had not been modified. Open the *hsqldb-ds.xml* file which you'll find in the deploy directory and which sets up the default datasource. Near the top of the file, you'll find the `<connection-url>` element. Make sure the value is set to

```
jdbc:hsqldb:hsql://localhost:1701
```

and that any other examples are commented out. So you should have something like:

```
      <!-- The jndi name of the DataSource, it is prefixed with java:/ -->
      <!-- Datasources are not available outside the virtual machine -->
         <jndi-name>DefaultDS</jndi-name>

         <!-- for tcp connection, allowing other processes to use the hsqldb
         database. This requires the org.jboss.jdbc.HypersonicDatabase mbean. -->
         <connection-url>jdbc:hsqldb:hsql://localhost:1701</connection-url>

         <!-- for totally in-memory db, not saved when jboss stops.
         The org.jboss.jdbc.HypersonicDatabase mbean is unnecessary
         <connection-url>jdbc:hsqldb:.</connection-url>
   -->
         <!-- for in-process db with file store, saved when jboss stops. The
         org.jboss.jdbc.HypersonicDatabase is unnecessary

         <connection-url>jdbc:hsqldb:${jboss.server.data.dir}/hypersonic/localDB
         </connection-url>
   -->
```

Now scroll down to the bottom of the file and you should find the MBean declaration for the Hypersonic service:

```
   <mbean code="org.jboss.jdbc.HypersonicDatabase" name="jboss:service=Hypersonic">
     <attribute name="Port">1701</attribute>
     <attribute name="Silent">true</attribute>
     <attribute name="Database">default</attribute>
     <attribute name="Trace">false</attribute>
```

```
    <attribute name="No_system_exit">true</attribute>
  </mbean>
```

Make sure this is also uncommented. This is also needed if you want to be able to run the HSQL Database Manager tool which we'll be looking at shortly.

4.1.7.2 Creating the Database Schema

Where necessary, we have supplied modified scripts to run with HSQL and you'll find them in the *sql* directory[3]. The main differences are in the SQL syntax for applying constraints in the table creation script *hsql-create-table.sql*. Apart from that the changes are trivial.

We've modified the corresponding tasks in the build file to call the appropriate HSQL tool for running the script. If JBoss isn't already running, you should start it now, so that the HSQL database is available. First we need to create the necessary tables by running the command

```
ant -f jboss-build.xml db-create-table
```

Then run the following command to populate them with the required data

```
ant -f jboss-build.xml db-insert
```

and finally, if everything has gone according to plan, you should be able to view some of the data using

```
ant -f jboss-build.xml db-list
```

which lists the transactions for a specific account.

3. Those prefixed with *"hsql-"* have been altered. The others are identical to the originals.

4.1.7.3 The HSQL Database Manager Tool

Just as a quick aside at this point, start up the JMX console application web application and click on the `service=Hypersonic` link which you'll find under the section "jboss".

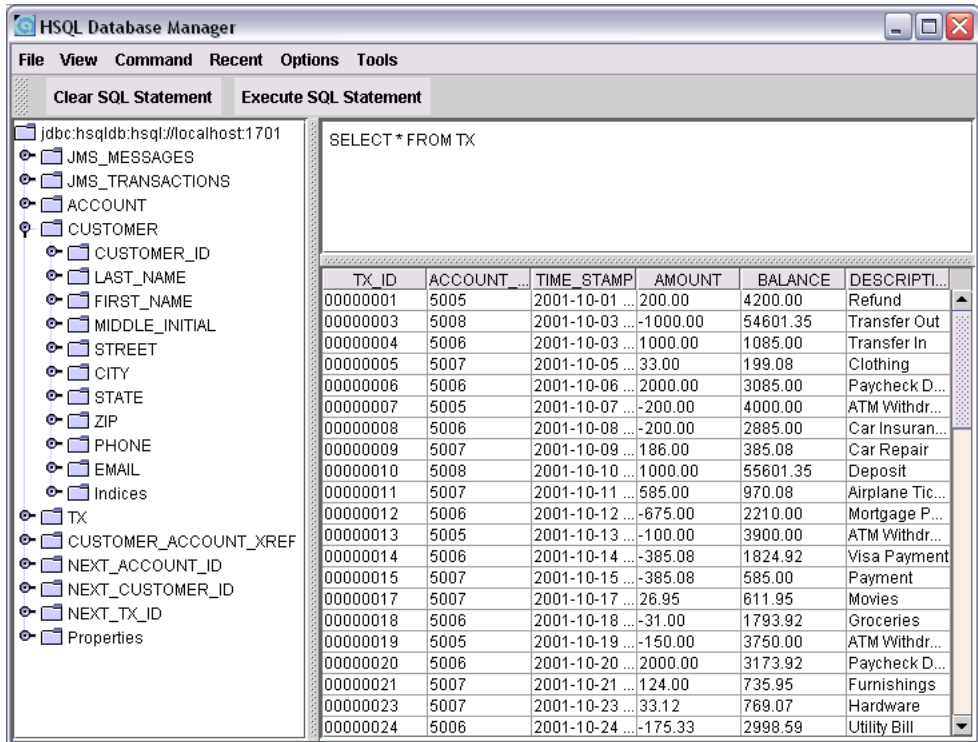

FIGURE 4.1. View of the HSQL Database Manger

This will take you to the information for the Hypersonic service MBean[4]. Scroll down to the bottom of the page and click the "invoke" button for the `startDatabaseManager()` operation. This starts up the HSQL Manager – a Java GUI application which you can use to manipulate the database directly.

4. If you can't find this, make sure the service is enabled as described in "Enabling the HSQL MBean and TCP/IP Connections" on page 21.

4.1.8. Deploying the Application

Deployment in JBoss is easy – you just have to copy the EAR file to the deploy directory. There's also a target in the build file for this so you can type

```
ant -f jboss-build.xml deploy
```

and this will assemble the EAR file and deploy it. You should see something close to the following output from the server (reduced for brevity):

```
19:30:32,966 INFO  [MainDeployer] Starting deployment of package: file:/F:/servers/
jboss-3.2.3/server/default/deploy/JBossDukesBank.ear
19:30:32,997 INFO  [EARDeployer] Init J2EE application: file:/F:/servers/jboss-3.2.3/
server/default/deploy/JBossDukesBank.ear
19:30:34,513 INFO  [EjbModule] Deploying AccountEJB
...
19:30:45,356 INFO  [Engine] StandardManager[/bank]: Seeding random number generator
class java.security.SecureRandom
19:30:45,356 INFO  [Engine] StandardManager[/bank]: Seeding of random number generator
has been completed
19:30:45,356 INFO  [Engine] StandardWrapper[/bank:default]: Loading container servlet
default
19:30:45,356 INFO  [Engine] StandardWrapper[/bank:invoker]: Loading container servlet
invoker
19:30:45,685 INFO  [EARDeployer] Started J2EE application: file:/F:/servers/jboss-
3.2.3/server/default/deploy/JBossDukesBank.ear
19:30:45,685 INFO  [MainDeployer] Deployed package: file:/F:/servers/jboss-3.2.3/
server/default/deploy/JBossDukesBank.ear
```

If there are any errors or exceptions, make a note of the error message and at what point it occurs (e.g. during the deployment of a particular EJB, the web application or whatever). Check that the EAR is complete and inspect the WAR file and each of the EJB jar files produced by the build to make sure they contain all the necessary components (classes, descriptors etc.).

You can safely redeploy the application if it is already deployed. To undeploy it you just have to remove the archive from the deploy directory. There's no need to restart the server in either case. If everything seems to have gone OK, then point your browser at the application URL

http://localhost:8080/bank/main

You should be forwarded to the application login page. As explained in the tutorial, you can login with a customer Id of 200 and the password "j2ee"[5].

5. If you get an error at this point, check again that you have set up the database correctly as described in "Enabling the HSQL MBean and TCP/IP Connections" on page 21. In particular, check that the `connection-url` is right. Then make sure that you have populated the database with data.

You should also be able to run the standalone client application using the command

```
ant -f jboss-build.xml run-client
```

This is a Swing GUI client which allows you to administer the customers and accounts.

4.2. JNDI and Java Clients

It’s worth taking a brief look at the use of JNDI with standalone clients. The example makes use of the J2EE “Application Client” framework which has been introduced in JBoss 3.2.3[6]. This introduces the concept of a client-side local environment naming context (within which JNDI names are resolved with the prefix “`java:/comp/env`”). This is identical to the usage on the server side; the additional level of indirection means you can avoid using hard-coded names in the client. The name mapping is effected by the use of the proprietary *jboss-client.xml* which resolves the references defined in the standard *application-client.xml.* See “Container-Specific Deployment Descriptors” on page 15 for more information on how this works.

4.2.1. The *jndi.properties* File

One issue with a Java client is how it bootstraps itself into the system – how it manages to connect to the correct JNDI server to lookup the references it needs. The information is supplied by using standard Java properties. You can find details of these and how they work in the JDK API documentation for the `javax.naming.Context` class. They can either be coded, or supplied in a file named *jndi.properties.* The file we’ve used looks like this:

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming.client
j2ee.clientName=bank-client
```

The first three are standad properties, which are set up to use the JBoss JNDI implementation. The fourth “j2ee.clientName” is a custom property which identifies the client deployment information on the server side. The name must match the `jndi-name` specified in the *jboss-client.xml* descriptor:

```
<jboss-client>
   <jndi-name>bank-client</jndi-name>

   <ejb-ref>
      <ejb-ref-name>ejb/customerController</ejb-ref-name>
      <jndi-name>MyCustomerController</jndi-name>
   </ejb-ref>
```

6. See the changenote at **http://sourceforge.net/tracker/index.php?func=detail&aid=840598&group_id=22866&atid=381174** for full details.

```
    <ejb-ref>
        <ejb-ref-name>ejb/accountController</ejb-ref-name>
        <jndi-name>MyAccountController</jndi-name>
    </ejb-ref>
</jboss-client>
```

You don't need to worry about any of this if you're building web applications.

4.3. Security

You may have noticed that we haven't done anything so far to set up any security configuration for the application. In fact there isn't any security to speak of and you can login with any password and gain access to the account – not much use for an on-line bank. Logging in with an invalid Id will cause the application to crash when the first JSP tries to access the (non-existent) user's accounts – not exactly ideal either.

If a web application doesn't have a "security domain" specified[7], JBoss assigns it a "NullSecurityManager" instance by default. This will allow any login to succeed, explaining the above behaviour.

4.3.1. Configuring a Security Domain

Enabling security for your application is done through the JBoss-specific deployment descriptors. To protect the web application, you have to include a `security-domain` element in the *jboss-web.xml*:

```
<jboss-web>
    <security-domain>java:/jaas/dukesbank</security-domain>
    ...
</jboss-web>
```

If you also want access controls to be applied at the EJB layer, you should add an identical element to the *jboss.xml* file too:

```
<jboss>
    <security-domain>java:/jaas/dukesbank</security-domain>

    <enterprise-beans>
    ...

</jboss>
```

7. The term "security domain" is widely used in security parlance, not always with the same meaning. It generally refers to a set of users (or components) operating under a common set of authentication and access-control mechanisms. In JBoss this is seen in the mapping of a security domain name to a particular set of login modules in the *login-config.xml* file. The term is often used interchangeably with "realm".

What this means is that JBoss will bind a security manager instance for our application under the JNDI name `java:/jaas/dukesbank`. The security domain for our application is named "dukesbank" and you can configure it in the *conf/login-config.xml* file which we first saw in "Security Service" on page 10. If you take a look at that file, you'll see how each security domain has an `application-policy` element. The `name` attribute is the security domain name, so to add a login configuration for our application, we would insert an extra entry of the form

```
<application-policy name = "dukesbank">
  <authentication>
     <login-module code="org.jboss.security.auth.spi.UsersRolesLoginModule"
         flag = "required" />
   </authentication>
</application-policy>
```

where the `authentication` element contains a sequence of `login-module` child elements, each of which specifies a JAAS login module implementation which will be used to authenticate users. The "required" flag means that login under this module must succeed for the user to be authenticated. The `UsersRoles-LoginModule` which we've specified here is a simple login module which stores valid user names, passwords and roles in properties files. Any security domains which don't have a login configuration entry will default to the policy named "other" which you will find at the bottom of the *login-config.xml* file. By default it uses this same login module, so we don't really need to add a specific entry for our application. However it's a good idea for completeness sake and you may want to experiment with adding different login modules later.

To recap, here are the steps you need to follow to secure Duke's Bank:

1. Add the `security-domain` element to each of the *jboss.xml* and *jboss-web.xml* descriptors in the *dd* directory. It should already be there, commented out.
2. Add an entry to the *conf/login.xml file* for the "dukesbank" security domain as above (optional).
3. Create the *users.properties* and *roles.properties* files which contain the security information for the information for the application and include these in the EAR file (this has already been done for you).
4. Follow through the build steps to re-package the EJBs and the web application (to make sure the modified descriptors are included).
5. Assemble the EAR file and re-deploy it to JBoss.

Again make sure that the application deploys OK without any errors and exceptions and try accessing it with your browser as before. This time you should not be able to login without the correct username and password combination.

4.3.2. UsersRolesLoginModule Files

Have a quick look at the format of the files so that you can experiment with adding users of your own. You'll find them in the *src* directory. The *users.properties* file contains name-value pairs of the form

`username=password`. The *roles.properties* entries are the user name and a comma-separated list of roles for that user.

```
username=role1,role2...
```

In Duke's Bank, the user "200" must be given the role "BankCustomer" to be able to access the web application and the EJB methods which it calls.

In a real project you will want to use a more sophisticated approach. You can find out more about using JAAS login modules in the JBoss "JAAS Howto" document which you can download from **http://sourceforge.net/docman/?group_id=22866**. We'll also look at security in more detail in "Security Configuration" on page 56.

4.3.3. The J2EE Security Model

We've only covered the proprietary aspects of securing a J2EE application in JBoss and we won't go into the details of standard J2EE security as this is covered elsewhere. However a brief overview in the context of the Duke's Bank application is worthwhile. For more details you should see the relevant sections in the tutorial, the EJB and servlet specifications, or any textbook on J2EE applications.

4.3.3.1 Authentication

The servlet spec. defines a standard means of configuring the login process for web applications. You will find an example in the element `login-config` in the *web.xml* file for Duke's Bank:

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>Default</realm-name>
  <form-login-config>
    <form-login-page>/logon</form-login-page>
    <form-error-page>/logonError</form-error-page>
  </form-login-config>
</login-config>
```

This is specifying that a form-based login should be used to obtain a username and password (as opposed to HTTP basic authentication for example, where the browser pops up a login dialog). It also specifies the URL that should be used for the login (`/logon`) and the URL which the user is forwarded to on a login error, such as a bad password. The format of the login form – the URL to submit to and the field names for username and password are defined in the spec. You can see an example in the file *logon.jsp* which is used in the application[8].

You should keep in mind that the authentication *logic* which decides whether a login succeeds or fails is outside the scope of the spec. The actual authentication mechanism is contained in the login modules that a security domain uses. So by adding the security-domain tag to your application, and thus linking

it to an entry in *login-config.xml*, you are effectively specifying what authentication logic will be used, be it a database, LDAP or whatever.

4.3.3.2 Access Control (Authorization)

J2EE uses a role-based access-control model, with the emphasis placed on configuration rather than code; you can restrict access to EJBs or individual EJB methods in the *ejb-jar.xml* file or to specific URLs in the *web.xml* file by defining which user roles are allowed to access them. A set of roles, again decided by the underlying security mechanism, will be assigned to a user as part of the logon process and each subsequent attempt to access a protected resource will be checked to see if it is allowed.

If you have a look at in *web.xml* you will find the access controls under the `security-constraint` element. You can see the list of restricted URLs there under `web-resource-collection` and the role which is allowed to access them (BankCustomer) under the `auth-constraint` element. In the *ejb-jar.xml* file, method access is controlled using a series of `method-permission` elements which contain lists of method definitions and the roles that can call them (or `<unchecked/>` for any role).

4.3.4. Application JNDI Information in the JMX Console

Lets take a quick look at the JBoss JMX console again and see what information it shows about our application. This time click on the `service=JNDIView` link and then invoke the `list()` operation at the bottom of the page which displays the JNDI tree for the server. You should see the EJB modules

8. Note that the URL for form logins "`j_security_check`" is implemented by the web container (Tomcat) and your code doesn't play any part in the login process. In practice it isn't too hard to break the example application login (especially when it's running in Tomcat 4) and you will get exceptions if you do things like browsing directly to the login page or attempting to login twice in the same session. These are issues with the servlet specification and Tomcat and you can find a lot of discussion of them online, e.g. in the Tomcat bugs database:

   http://nagoya.apache.org/bugzilla/show_bug.cgi?id=6279.

   The example code just assumes that the standard model will be followed and doesn't provide any workarounds. So don't push it too hard.

from Duke's Bank listed near the top and the contents of their private environment naming contexts as well as the names the entries are linked to in the server.

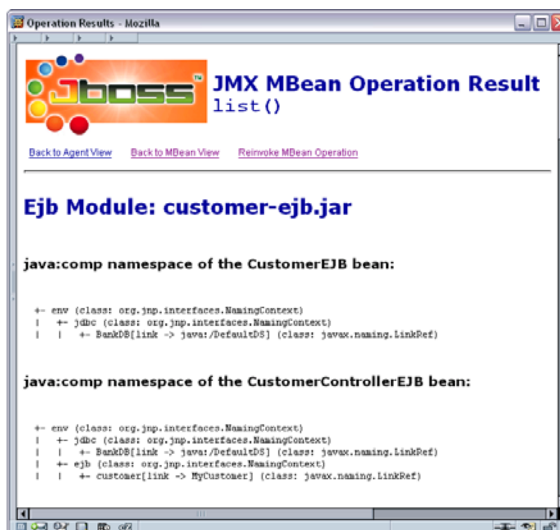

FIGURE 4.2. JMX Console JNDI View

Further down, under the *java:* namespace[9], you should see

```
+- jaas (class: javax.naming.Context)
|   +- dukesbank (class: org.jboss.security.plugins.SecurityDomainContext)
|   +- JmsXARealm (class: org.jboss.security.plugins.SecurityDomainContext)
|   +- jbossmq (class: org.jboss.security.plugins.SecurityDomainContext)
|   +- HsqlDbRealm (class: org.jboss.security.plugins.SecurityDomainContext)
```

which is a list of the active security managers, bound under their security-domain names. Note that these objects are created on demand, so the `dukesbank` entry will only appear if you have tried to log in to the application.

9. The *java:* namespace is for names which can only be resolved within the VM. Remote clients can't resolve them, unlike those in the global namespace.

CHAPTER 5

JMS and Message-Driven Beans

One thing that's missing from the Duke's Bank application is any use of JMS messaging, so we'll work through the tutorial example on Message Driven Beans (MDB) to see how to use messaging in JBoss. We'll assume you're already familiar with general JMS and MDB concepts. The J2EE tutorial code for the MDB is in *j2eetutorial/examples/src/ejb/simplemessage.* We've supplied a build file in the *examples* directory which will allow you to build the example from scratch and run it in JBoss. We've also added the *ejb-jar.xml* file and the *jboss.xml* file.

The example code is *very* simple. There are only two classes, one for the client and one for the bean (unlike normal EJBs, MDBs don't need any interfaces). The client publishes messages to a JMS Queue and the MDB handles them via its standard `onMessage` method. The messages are all of type `javax.jms.TextMessage` and the bean simply prints out the text contained in each message.

The only container-specific tasks required are setting up the Queue in JBoss, and configuring the MDB to accept messages from it.

5.1. Building the Example

5.1.1. Compiling and Packaging the MDB and Client

As before, the build file is called *jboss-build.xml* so, from the *examples* directory, first compile the files with the command

```
ant -f jboss-build.xml compile-mdb
```

Then run the following commands:

```
ant -f jboss-build.xml package-mdb
ant -f jboss-build.xml package-mdb-client
ant -f jboss-build.xml assemble-mdb
```

Which will produce archives for the bean and the client and a combined EAR file in the *jars* directory. We've retained the same layout as the Duke's Bank build – with a *dd* directory containing the deployment descriptors and the *jars* directory containing the archives produced by the build. The examples come with a set of pre-built EAR files in the *ears* directory but we won't use these.

5.1.1.1 Specifying the Source Queue for the MDB

As with other container-specific information, the queue name for the MDB is specified in the *jboss.xml* file:

```
<jboss>
  <enterprise-beans>
    <message-driven>
      <ejb-name>SimpleMessageBean</ejb-name>
      <destination-jndi-name>queue/MyQueue</destination-jndi-name>
    </message-driven>
  </enterprise-beans>
</jboss>
```

So the MDB will receive messages from the queue with JNDI name `queue/MyQueue`.

5.2. Deploying and Running the Example

To deploy the MDB, just copy the *simplemessage.jar* file to the JBoss deploy directory. A successful deployment should look something like this:

```
19:51:07,872 INFO  [MainDeployer] Starting deployment of package: file:/F:/servers/
jboss-3.2.3/server/default/deploy/SimpleMessage.ear
19:51:07,888 INFO  [EARDeployer] Init J2EE application: file:/F:/servers/jboss-3.2.3/
server/default/deploy/SimpleMessage.ear
```

```
19:51:08,653 INFO  [EjbModule] Deploying SimpleMessageEJB
19:51:11,060 WARN  [JMSContainerInvoker] Could not find the queue destination-jndi-
name=queue/MyQueue
19:51:11,075 WARN  [JMSContainerInvoker] destination not found: queue/MyQueue reason:
javax.naming.NameNotFoundException: MyQueue not bound
19:51:11,075 WARN  [JMSContainerInvoker] creating a new temporary destination: queue/
MyQueue
19:51:11,091 INFO  [MyQueue] Bound to JNDI name: queue/MyQueue
19:51:11,091 INFO  [MyQueue] Started jboss.mq.destination:service=Queue,name=MyQueue
19:51:11,263 INFO  [DLQHandler] Started null
19:51:11,263 INFO  [JMSContainerInvoker] Started jboss.j2ee:binding=message-driven-
bean,jndiName=local/SimpleMessageEJB,plugin=invoker,service=EJB
19:51:11,263 INFO  [MessageDrivenInstancePool] Started jboss.j2ee:jndiName=local/Sim-
pleMessageEJB,plugin=pool,service=EJB
19:51:11,263 INFO  [MessageDrivenContainer] Started jboss.j2ee:jndiName=local/Sim-
pleMessageEJB,service=EJB
19:51:11,263 INFO  [EjbModule] Started jboss.j2ee:module=simplemessage.jar,serv-
ice=EjbModule
19:51:11,263 INFO  [EJBDeployer] Deployed: file:/F:/servers/jboss-3.2.3/server/
default/tmp/deploy/tmp17621SimpleMessage.ear-contents/simplemessage.jar
19:51:11,294 INFO  [EARDeployer] Started J2EE application: file:/F:/servers/jboss-
3.2.3/server/default/deploy/SimpleMessage.ear
19:51:11,294 INFO  [MainDeployer] Deployed package: file:/F:/servers/jboss-3.2.3/
server/default/deploy/SimpleMessage.ear
```

If you look more closely at this, you will see warnings that the message queue specified in the deployment doesn't exist. In this case JBoss will create a temporary one for the application and bind it under the supplied name. You can check it exists using the JNDIView MBean again – look under the "global" JNDI namespace. We'll look at how to explicitly create JMS destinations below.

5.2.1. Running the Client

Run the client with the command

```
ant -f jboss-build.xml run-mdb
```

and you should see output in both the client and server windows as they send and receive the messages respectively.

5.3. Managing JMS Destinations

As with most things in JBoss, JMS Topics and Queues are implemented using MBeans. There are two ways you can create them: you can add MBean declarations to the appropriate configuration file, or you

can create them dynamically using the jmx-console application. If you use the latter method, they won't survive a server restart.

5.3.1. The *jbossmq-destinations-service.xml* File

You'll find this file in the *jms* directory inside the deploy directory. It contains a list of JMS destinations and sets up a list of test topics and queues which illustrate the syntax used. To add an extra queue for our example, you simply add the following MBean declaration to the file:

```
<mbean code="org.jboss.mq.server.jmx.Queue"
                name="jboss.mq.destination:service=Queue,name=MyQueue">
</mbean>
```

5.3.2. Using the DestinationManager from the JMX Console

With JBoss running, bring up the JMX Console in your browser and look for the section labelled "jboss.mq" in the main agent view. Click on the link which says `service=DestinationManager`. The DestinationManager MBean is the main JMS service in JBoss and you can use it to create and destroy queues and topics at runtime. Look for the operation called `createQueue`. This takes two parameters – the first is a name for the Queue MBean (so will not usually be relevant to your application code) and the second is the JNDI name. So enter "MyQueue" and "queue/MyQueue"[1] for these respectively. Note that this will fail if either of these names is already in use (for example if you have deployed the application as above or added a Queue MBean using the XML configuration file. If this is the case you can either remove the existing queue or just try another name.

5.3.3. Administering Destinations

Once you've created a Queue or Topic, you can also access the attributes and operations which it exposes via JMX. Look at the main JMX Console view again and you'll find a separate "jboss.mq.destination" section which should contain an entry for our Queue (no matter how it was created). Click on this and you'll see the attributes for the queue. Amongst these is the "QueueDepth" which is the number of messages which are currently on the queue.

As an excercise, you can try temporarily stopping the delivery of messages to the MDB. Locate the section called "jboss.j2ee" in the JMX console and you should find an MBean listed there which is responsible for invoking your MDB. The name should be something like

```
binding=message-driven-bean,jndiName=local/SimpleMessageEJB,plugin=invoker,service=EJB
```

1. We've adopted the standard JBoss convention of binding queues under the JNDI name "queue" and topics under "topic" but this isn't necessary. You can use any name.

and you can start and stop the delivery of messages using the corresponding MBean operations which it supports.

Then run the client a few times and monitor the queue depth as the messages accumulate. If you re-start message delivery you should see all the messages arriving at once.

CHAPTER 6

Container-Managed Persistence

The Duke's Bank application which we saw in Chapter 4 uses bean-managed persistence (BMP). However, the improvements to container-managed persistence (CMP) introduced in the EJB 2.0 specification make it unlikely that you would use BMP in practice. In this chapter we'll look at the "RosterApp" example application from the J2EE tutorial which covers the use of container-managed persistence and relationships. You should read through the CMP tutorial notes before proceeding so that you have a good overview of the beans and their relationships.

You'll find the code in *j2eetutorial/examples/src/ejb/cmproster.* The application implements a player roster for sports' teams playing in leagues. There are three entity beans PlayerEJB, TeamEJB and LeagueEJB and a single session bean, RosterEJB, which manipulates them and provides the business methods accessed by the client application. Only the session bean has a remote interface.

6.1. Building the Example

The EJBs are packaged in two separate jar files, one for the entity beans and one for the session bean. As before, we've provided a *ejb-jar.xml* files for each one. You don't need a *jboss.xml* file for this example – all the CMP information

needed to build the database schema is included in the standard descriptor. We'll look at JBoss-specific customization later.

6.1.1. Compiling the Code

Make sure you're in the *examples* directory. Running the following command should compile all the code in one go:

```
ant -f jboss-build.xml compile-cmp
```

6.1.2. Packaging the Jars

Run the following command to build the "team" jar file which contains the entity beans:

```
ant -f jboss-build.xml package-team
```

Then build the "roster" jar with:

```
ant -f jboss-build.xml package-roster
```

Both jar files will be created in the *jar* directory. Build the client jar using

```
ant -f jboss-build.xml package-roster-client
```

Finally assemble the "RosterApp" EAR using the command:

```
ant -f jboss-build.xml assemble-roster
```

6.2. Deploying and Running the Application

Copy the *RosterApp.ear* file from the jar directory to the JBoss deploy directory (or run ant with the "deploy-cmp" target) and check the output from the server:

```
19:55:49,138 INFO  [MainDeployer] Starting deployment of package: file:/F:/servers/
jboss-3.2.3/server/default/deploy/RosterApp.ear
19:55:49,153 INFO  [EARDeployer] Init J2EE application: file:/F:/servers/jboss-3.2.3/
server/default/deploy/RosterApp.ear
19:55:49,731 INFO  [EjbModule] Deploying RosterEJB
19:55:50,153 INFO  [EjbModule] Deploying PlayerEJB
19:55:50,216 INFO  [EjbModule] Deploying TeamEJB
19:55:50,216 INFO  [EjbModule] Deploying LeagueEJB
19:55:52,919 INFO  [StatefulSessionInstancePool] Started jboss.j2ee:jndiName=Ros-
terEJB,plugin=pool,service=EJB
19:55:52,919 INFO  [StatefulSessionFilePersistenceManager] Started null
19:55:52,919 INFO  [StatefulSessionContainer] Started jboss.j2ee:jndiName=Ros-
terEJB,service=EJB
```

```
19:55:52,935 INFO  [EjbModule] Started jboss.j2ee:module=roster-ejb.jar,service=Ejb-
Module
19:55:52,935 INFO  [EJBDeployer] Deployed: file:/F:/servers/jboss-3.2.3/server/
default/tmp/deploy/tmp17622RosterApp.ear-contents/roster-ejb.jar
19:55:53,653 INFO  [EntityInstancePool] Started jboss.j2ee:jndiName=local/Play-
erEJB,plugin=pool,service=EJB
19:55:53,653 INFO  [EntityContainer] Started jboss.j2ee:jndiName=local/PlayerEJB,serv-
ice=EJB
19:55:53,685 INFO  [EntityInstancePool] Started jboss.j2ee:jndiName=local/
TeamEJB,plugin=pool,service=EJB
19:55:53,685 INFO  [EntityContainer] Started jboss.j2ee:jndiName=local/TeamEJB,serv-
ice=EJB
19:55:54,622 INFO  [TeamEJB] Created table 'TEAMEJB' successfully.
19:55:54,669 INFO  [PlayerEJB] Created table 'PLAYEREJB' successfully.
19:55:54,669 INFO  [PlayerEJB] Created table 'TEAMEJB_PLAYERS_PLAYEREJB_TEAMS' suc-
cessfully.
19:55:54,966 INFO  [LeagueEJB] Created table 'LEAGUEEJB' successfully.
19:55:54,966 INFO  [EntityInstancePool] Started jboss.j2ee:jndiName=local/
LeagueEJB,plugin=pool,service=EJB
19:55:54,966 INFO  [EntityContainer] Started jboss.j2ee:jndiName=local/LeagueEJB,serv-
ice=EJB
19:55:54,966 INFO  [EjbModule] Started jboss.j2ee:module=team-ejb.jar,service=EjbMod-
ule
19:55:54,966 INFO  [EJBDeployer] Deployed: file:/F:/servers/jboss-3.2.3/server/
default/tmp/deploy/tmp17622RosterApp.ear-contents/team-ejb.jar
19:55:54,997 INFO  [EARDeployer] Started J2EE application: file:/F:/servers/jboss-
3.2.3/server/default/deploy/RosterApp.ear
19:55:54,997 INFO  [MainDeployer] Deployed package: file:/F:/servers/jboss-3.2.3/
server/default/deploy/RosterApp.ear
```

There are a few things worth noting here. In the Duke's Bank application, we specified the JNDI name we wanted a particular EJBHome reference to be bound under in the *jboss.xml* file. Without that information JBoss will default to using the EJB name. So the session bean is bound under “RosterEJB” and so on. You can check these in the jmx-console as before. You will also see that the database tables have been automatically created – there is one for each entity bean and an additional join table to handle the many-to-many relationship between players and teams. There is no standard naming convention for either table names or columns but if you take a look at the database schema as we did before (See “The HSQL Database Manager Tool” on page 23.), you can see that the columns are named after the corresponding fields. This behaviour can be customized (to match an existing schema, for example) by supplying a *jbosscmp-jdbc.xml* file.

Note that if you increase the logging level for the `org.jboss.ejb.plugins.cmp` package (See “Logging Service” on page 9.) to DEBUG, the engine will log the SQL commands which it is executing. This can be useful in understanding how the engine works and how the various tuning parameters affect the loading of data (see below).

6.2.1. Running the Client

The client performs some data creation and retrieval operations via the session bean interface. It creates leagues, teams and players which will be inserted into the database (check with the HSQL manager tool). The session bean methods it calls to retrieve data are mainly wrappers for EJB finder methods. The command to run the client and the expected output are shown below:

```
$ ant -f jboss-build.xml run-cmp
Buildfile: jboss-build.xml

run-cmp:
     [java] P10 Terry Smithson midfielder 100.0
     [java] P6 Ian Carlyle goalkeeper 555.0
     [java] P7 Rebecca Struthers midfielder 777.0
     [java] P8 Anne Anderson forward 65.0
     [java] P9 Jan Wesley defender 100.0

     [java] T1 Honey Bees Visalia
     [java] T2 Gophers Manteca
     [java] T5 Crows Orland

     [java] P2 Alice Smith defender 505.0
     [java] P22 Janice Walker defender 857.0
     [java] P25 Frank Fletcher defender 399.0
     [java] P5 Barney Bold defender 100.0
     [java] P9 Jan Wesley defender 100.0

     [java] L1 Mountain Soccer
     [java] L2 Valley Basketball
```

Note that the client doesn't remove the data, so if you run it twice it will fail because it tries to create entities which already exist! If you want to run it again you'll have to remove the data. The easiest way to do this (if you're using HSQL) is to delete the contents of the *data/hypersonic* directory in the server configuration you are using (assuming you don't have any other important data in there!) and restart the server. We've also provided a simple delete SQL script which you can run with the command

```
ant -f jboss-build.xml db-delete
```

You could also use SQL commands directly through the HSQL Manager tool to delete the data.

6.3. CMP Customization

There are many ways you can customize the CMP engines's behaviour by using the *jbosscmp-jdbc.xml* file. It is used for basic information such as the datasource name and type-mapping (Hypersonic, Oracle

etc.) and whether the database tables should be automatically created on deployment and deleted on shutdown. You can customize the names of database tables and columns which the EJBs are mapped to and you can also tune the way in which the engine loads the data depending on how you expect it to be used. For example, by using the “read-ahead” element you can get it to read and cache blocks of data for multiple EJBs with a single SQL call, anticipating further access. “Eager-loading” groups can be specified, meaning that only some fields are loaded initially with the entity; the others are “lazy-loaded” if and when they are required. The accessing of relationships between EJBs can be tuned using similar mechanisms. This flexibility is impossible to achieve if you are using BMP where each bean must be loaded with a single SQL call. If on top of that you include having to write all your SQL and relationship management code by hand then the choice should be obvious. You should rarely, if ever, have to use BMP in your applications.

The details of tuning the CMP engine are beyond the scope of this document but you can get an idea of what’s available by examining the DTD (*docs/dtd/jbosscmp-jdbc_3_2.dtd*) which is well commented. There is also a standard setup in the *conf* directory called *standardjbosscmp-jdbc.xml* which contains values for the “default” settings and a list of type-mappings for common databases. The beginning of the file is shown below:

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

      <create-table>true</create-table>
      <remove-table>false</remove-table>
      <read-only>false</read-only>
      <read-time-out>300000</read-time-out>
      <row-locking>false</row-locking>
      <pk-constraint>true</pk-constraint>
      <fk-constraint>false</fk-constraint>
      <preferred-relation-mapping>foreign-key</preferred-relation-mapping>
      <read-ahead>
         <strategy>on-load</strategy>
         <page-size>1000</page-size>
         <eager-load-group>*</eager-load-group>
      </read-ahead>
      <list-cache-max>1000</list-cache-max>
      ...
```

You can see that, among other things, this sets the datasource and mapping for use with the embedded Hypersonic database and sets table-creation to “true” and removal to “false” (so the schema will be created if it doesn’t already exist). The “read-only” and “read-time-out” elements specify whether data should be read-only and the time in milliseconds it is valid for. Note that many of these elements can be used at different granularities such as per-entity or even on a field-by-field basis (consult the DTD for details). The read-ahead element uses an “on-load” strategy which means that the EJB data will be loaded when it is accessed (rather than when the finder method is called) and the “page-size” setting means that the data for up to 1000 entities will be loaded with one SQL command. You can override this either in your own *jbosscmp-jdbc.xml* file’s list of default settings or by adding the information to a specific query configuration in the file.

A comprehensive explanation of the CMP engine and its various loading strategies can be found in the full JBoss Admin. and Development Guide (See Table 1 on page 17).

6.3.1. XDoclet

Writing and maintaining deployment descriptors is a labour-intensive and error-prone job at the best of times and detailed customization of the CMP engine can lead to some large and complex files. If you are using CMP (or indeed EJBs) in anger then it is worth getting to grips with the XDoclet code generation engine (**http://xdoclet.sourceforge.net**). Using Javadoc-style attribute tags in your code it will generate the deployment descriptors for you as well as the EJB interfaces and other artifacts if required. It fully supports JBoss CMP and though the learning curve is quite steep and a bit much when you're trying to get to grips with the basics, its use is thoroughly recommended (almost essential in fact) for real projects.

CHAPTER 7

Web Services with JBoss.Net

Web services are all the rage these days. By transmitting XML data using platform and language-independent protocols (e.g. SOAP over HTTP), the aim is to achieve genuine interoperability, based on clearly-defined standards. Web services are a required part of the J2EE 1.4 specification. There is a lot to learn (starting with a whole pile of new acronyms) so if you're not already familiar with the subject, we would recommend you do some reading in advance. A good place to start would be the **JBoss.Net documentation** on the JBoss web site, which provides an excellent overview and links to other sources of information. Another good source of reference material is the Apache Web Services Project web site at

http://ws.apache.org.

7.1. JBoss.net

JBoss.Net is the JBoss module responsible for providing web services. It is built around the Apache Axis SOAP implementation (**http://ws.apache.org/axis**) and is intended to provide integration with J2EE and JMX. It introduces a new archive type – the web service archive (WSR) – which allows you to package and deploy your web services in a similar fashion to standard J2EE modules, taking advantage of the JBoss hot-deployment mechanism.

The JBoss.Net service is included in the “all” server configuration, not the “default” one which we’ve been using up until now. It’s implemented by the expanded JBoss service archive, *jboss-net.sar*, in the deploy directory. To make it available, you have to start JBoss with the command

```
run -c all
```

Alternatively you can move the whole SAR into the default configuration or create your own custom configuration (See “Server Configurations” on page 7.).

With JBoss.Net it is easy to can expose an EJB as a web service, so we’ll do this, using one of the session beans from Duke’s Bank as an example. Make sure you have the latest version of Ant, as some versions of the version of the Xerces parser which come with it can cause problems. We used Ant 1.5.4 (which contains Xerces 2.5) without any problems.

7.2. Duke’s Bank as a Web Service

It’s really very straightforward to make your EJB available as a web service. If you haven’t already worked through the Duke’s Bank example, then you should do that first. We’ll use the `AccountController` session bean and call the `getDetails` method which takes a `String` argument for the accountId and returns an `AccountDetails` value object containing the data for that account. So it’s more complicated than the average “Hello World” example as we have to deserialize the returned object on the client side. There are two things we have to do, assuming you already have the application deployed. We have to write and deploy a WSR file containing the descriptor for the web service, and we have to write a client.

7.2.1. The Web Service Archive (WSR) File

The WSR file is just a standard Jar archive with a *.wsr* extension and a *META-INF/web-service.xml* file. The latter is a standard Axis descriptor (WSDD file). You can read more on this in the Axis documentation. If you’ve unpacked the supplementary JBoss files for the J2EE tutorial, you’ll find the file in the *bank/dd* directory.

```
<deployment
    name="Bank"
    xmlns="http://xml.apache.org/axis/wsdd/"
    targetNamespace="http://net.jboss.org/bank"
    xmlns:bank="http://net.jboss.org/bank"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

<!-- AccountController Session bean exposed as a web service -->
  <service name="AccountController" provider="Handler">
    <parameter name="handlerClass" value="org.jboss.net.axis.server.EJBProvider"/>
    <parameter name="beanJndiName" value="MyAccountController"/>
    <parameter name="allowedMethods" value="getAccountsOfCustomer getDetails"/>
  </service>
```

```
<!-- Type-mapping for the AccountDetails value object. -->
  <beanMapping qname="bank:AccountDetails"
             languageSpecificType="java:com.sun.ebank.util.AccountDetails"/>

</deployment>
```

From the listing, you can see how the `<service>` tag is used to expose the bean is exposed as the web service “AccountController”. This obviously doesn’t have to be the same as the bean name – the bean is specified by the `beanJndiName` parameter. The `org.jboss.net.axis.server.EJBProvider` class, which is an extension of the corresponding Axis EJBProvider, is responsible for handling the details. The `allowed-Methods` parameter defines which EJB methods should be exposed by the service.

The final `beanMapping` element specifies that the `AccountDetails` object should be treated as a Java bean which means that the (de)serialization to and from SOAP messages will be handled by the Axis bean serialization classes. An alternative is to use the `typeMapping` element to set up custom serialization and deserialization.

To make `com.sun.ebank.util.AccountDetails` into a valid Javabean class, we have to add a default constructor:

```
    public AccountDetails() {  }
```

or it won’t be possible for the client to create the return object from the SOAP message. So you should do this and then recompile and deploy Duke’s Bank before proceeding. Make sure you deploy it into the correct server configuration! For example, if you’ve been running the “default” configuration until now, but have switched to “all” to enable Web Services, then you must obviously place the EAR file in the *JBOSS_DIST/server/all/deploy* directory. You can do a sanity check by browsing to the web application.

7.2.2. Building and Deploying the WSR File

If you run the command

```
ant -f jboss-build.xml wsr
```

from the *bank* directory, this should produce the WSR file in the Jar directory. You can then copy the file to the deploy directory (make sure you copy it to the server configuration you are running). You should see a short message in the server console to say it has deployed the archive. Duke’s Bank *must* be deployed prior to this or you’ll get a ClassNotFoundException for `AccountDetails`. An alternative approach, which you would probably adopt in practice, would be to add the WSR file to the EAR and deploy everything as a single unit.

Once the service is deployed you can view the WSDL (Web Service Description Language) for it by browsing to to the URL **http://localhost:8080/jboss-net/services/AccountController?wsdl**

This description of the service interface is the web service equivalent of IDL in CORBA. In this example it is generated for us but it is also possible to write the WSDL for the service and then compile code for it using a tool such as *wsdl2java* (which comes with Axis).

7.2.3. Running the Client

We've also supplied a Java client and an ant task to run it.

```
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import com.sun.ebank.util.AccountDetails;
import javax.xml.namespace.QName;

public class WSClient
{
   public static void main(String [] args) throws Exception
   {
      String endpoint = "http://localhost:8080/jboss-net/services/AccountController";
      Service service = new Service();
      Call    call    = (Call) service.createCall();
      call.setTargetEndpointAddress( new java.net.URL(endpoint) );
      call.setOperationName("getDetails");
      call.setReturnClass(AccountDetails.class);
      QName qn = new QName("http://net.jboss.org/bank","AccountDetails");
      call.registerTypeMapping(AccountDetails.class, qn,
        new org.apache.axis.encoding.ser.BeanSerializerFactory(AccountDetails.class, qn),
         new org.apache.axis.encoding.ser.BeanDeserializerFactory(AccountDetails.class, qn));

      AccountDetails ret = (AccountDetails) call.invoke( new Object[] { "5005" } );
      System.out.println(ret.getDescription() + ", " + ret.getType());
   }
}
```

The client uses the JAXRPC Call interface to invoke the service dynamically, rather than using stub code compiled from the WSDL. It specifies the method to be invoked using `call.setOperationName` and then uses `call.registerTypeMapping` to define how the returned object should be handled (the latter is an Axis-specific method and we again use the Axis bean-(de)serialization facilities).

7.2.4. Net Traffic Analysis

Axis comes with some useful utilities for monitoring your web service traffic. The “TCPMonitor” tool act as a TCP tunnel for connections between the client and server: it listens on one port for client connections, forwarding client requests to the server and returning responses on the client. From this “man-in-the-middle” position it will print out all the traffic in both directions, so you can use it to view HTTP headers, SOAP messages or anything else you want to pass over a TCP connection. There's nothing specific to web services involved. There's an additional target in the build file to run the tool:

```
ant -f jboss-build.xml tcpmon
```

which will pop up the initial configuration window. You can check the Axis user guide for details on using this but it just involves specifying a local port to listen on (we chose 7070) and the information for the host and port to forward to (the defaults are “localhost” and “8080” respectively, so you shouldn’t need to change them). You then have to modify the client to connect to the new port and recompile. You can then run the client and view the output:

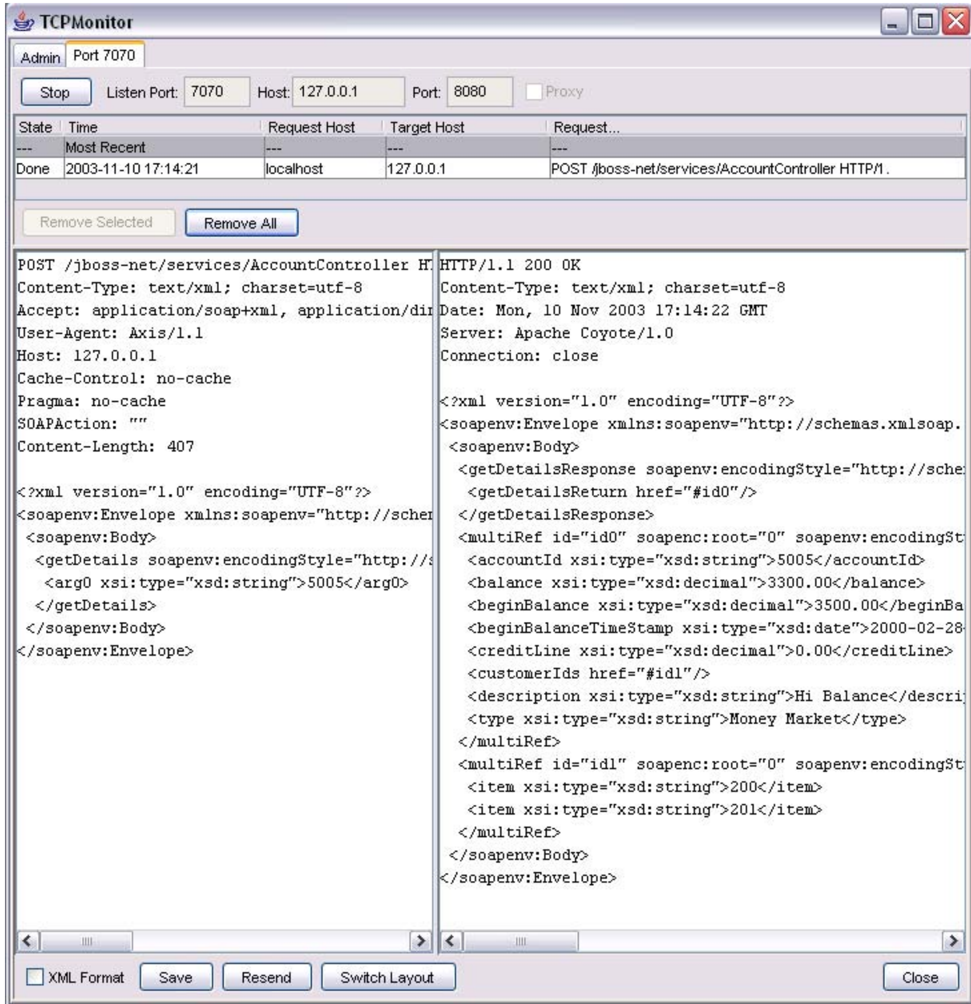

FIGURE 7.1. TCPMon output of Web Services Call

You can also make changes to the request message and resend it, making TCPMon a useful debugging tool as well.

CHAPTER 8

Using other Databases

In the previous chapters, we've just been using the JBoss default datasource in our applications. This is provided by the embedded HSQL database instance and is bound to the JNDI name "java:/DefaultDS". Having a database included with JBoss is very convenient for running examples and HSQL is adequate for many purposes. However, at some stage you will want to use another database, either to replace the default datasource or to access multiple databases from within the server.

8.1. DataSource Configuration

Database connection management in JBoss is entirely handled by the JCA implementation. So all databases are accessed via JCA resource adapters which handle connection pooling, security and transactions.

8.1.1. JDBC-Wrapper Resource Adapters

If there is no proprietary adapter for the database in question then you can configure it to use one of the two JDBC-wrapper resource adapters which we mentioned when we were looking at the various services deployed in JBoss (See "Additional Services" on page 11). Obviously you need a JDBC driver for this to

work and the classes have to be made available (by copying the driver jar or zip file to the *lib* directory of the server configuration you are working with). The main distinction between different datasource configurations is whether they are set up to use the local or XA-transaction JDBC adapters. The latter option is only available if the JDBC driver in question provides an implementation of `javax.sql.XADataSource`[1] but you can still choose the local option even if an XADataSource implementation is available (see the two oracle configuration files for example). There is also a “no-transaction” configuration but you would rarely use this with a database.

8.1.2. DataSource Configuration Files

DataSource configuration file names end with the suffix “*-ds.xml*” so that they will be recognised correctly by the JCA deployer. The *docs/example/jca* directory contains sample files for a wide selection of databases and it is a good idea to use one of these as a starting point. For a full description of the configuration format the best place to look is the DTD file *docs/dtd/jboss-ds_1_0.dtd*. Additional documentation on the files and the JBoss JCA implementation can also be found in the JBoss Admin. and Development Guide.

Local-transaction datasources are configured using the `<local-tx-datasource>` element and XA-compliant ones using `<xa-tx-datasource>`. The example file *generic-ds.xml* shows how to use both types and also some of the other elements that are available for things like connection-pool configuration. Examples of both local and XA configurations are available for Oracle, DB2 and Informix.

If you look at the example files *firebird-ds.xml*, *facets-ds.xml* and *sap3-ds.xml,* you’ll notice that they have a completely different format, with the root element being `<connection-factories>` rather than `<datasources>`. These use an alternative, more generic JCA configuration syntax used with a pre-packaged JCA resource adapter. As we mentioned in “Additional Services” on page 11, the syntax is not specific to datasource configuration and is used, for example, in the *jms-ds.xml* file to configure the JMS resource adapter.

8.2. Examples

We’ll work through some step-by-step examples to illustrate what’s involved.

8.2.1. Using MySQL as the Default DataSource

MySQL is a one of the most popular open source databases around and is used by many prominent organizations from Yahoo to NASA. The official JDBC driver for it is called “Connector/J”. For this

1. The local and XA transaction contracts are discussed in chapter 7 of the JCA 1.5 Specification.

example we've used MySQL 4.0.13 and Connector/J 3.0.9 on Windows XP. You can download them both from **http://www.mysql.com**.

8.2.1.1 Creating a Database and User

We'll assume that you've already installed MySQL and that you have it running and are familiar with the basics. Run the *mysql* client program from the command line so we can execute some administration commands. You should make sure that you are connected as a user with sufficient privileges (e.g. by specifying the "-u root" option to run as the MySQL "root" user).

First create a database called "jboss" within MySQL for use by JBoss

```
mysql> CREATE DATABASE jboss;
Query OK, 1 row affected (0.05 sec)
```

and check that it has been created using

```
mysql> SHOW DATABASES;
+----------+
| Database |
+----------+
| jboss    |
| mysql    |
| test     |
+----------+
3 rows in set (0.00 sec)
```

Then create a user called "jboss" with password "password" to access the database

```
mysql> GRANT ALL PRIVILEGES ON jboss.* TO jboss@localhost IDENTIFIED BY 'password';
Query OK, 0 rows affected (0.06 sec)
```

and again check that everything has gone smoothly

```
mysql> select User,Host,Password from mysql.User;
+-------+-----------+------------------+
| User  | Host      | Password         |
+-------+-----------+------------------+
| root  | localhost |                  |
| root  | %         |                  |
|       | localhost |                  |
|       | %         |                  |
| jboss | localhost | 5d2e19393cc5ef67 |
+-------+-----------+------------------+
5 rows in set (0.02 sec)
```

8.2.1.2 Installing the JDBC Driver and Deploying the DataSource

To make the JDBC driver classes available to JBoss, copy the file *mysql-connector-java-3.0.9-stable-bin.jar* from the Connector/J distribution to the *lib* directory in the default server configuration (assuming you're running this one, of course). Then create a file called *mysql-ds.xml* with the following datasource configuration:

```
<datasources>
  <local-tx-datasource>
    <jndi-name>DefaultDS</jndi-name>
    <connection-url>jdbc:mysql://localhost:3306/jbossdb</connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <user-name>jboss</user-name>
    <password>password</password>
  </local-tx-datasource>
</datasources>
```

which mirrors the database and user information we set up in the previous section. Our aim here is to replace the default datasource in JBoss with a MySQL version, so you have to remove the existing *hsqldb-ds.xml* from the deploy directory or there will be a conflict between the JNDI names of the two datasources. Copy the new file in its place and start JBoss.

You may notice some exceptions during JMS startup and error messages about SQL syntax. This is because the message persistence manager uses SQL subqueries (nested select statements) which have been introduced in MySQL 4.1 (which is still in alpha release). There are alternative service files for use with MySQL and other databases in the *examples/jms* directory[2].

8.2.1.3 Testing the MySQL DataSource

We'll use the CMP "roster" application. The only change that has to be made is to change the type-mapping from "Hypersonic" to "MySQL". You can either do this by adding a *jbosscmp-jdbc.xml* to the EJB deployment or modify the global default settings in *conf/standardjbosscmp-jdbc.xml.* The latter approach is simpler, as you don't have to re-package the application. The disadvantage is that you have to restart JBoss for the changes to take place. Edit the file and change the `datasource-mapping` element to "mySQL":

```
<jbosscmp-jdbc>
   <defaults>
      <datasource>java:/DefaultDS</datasource>
      <datasource-mapping>mySQL</datasource-mapping>
```

2. The file for MySQL is *mysql-jdbc2-service.xml.* Make sure you don't use the "mssql" one by mistake. Replace the occurrence of "MySqlDS" with "DefaultDS" and replace the file *jms/hsql-jdbc2-service.xml* in the deploy directory with this one.

After restarting JBoss, you should be able to deploy the application and see the tables being created as we did in "Deploying and Running the Application" on page 37. The tables should be visible from the MySQL client:

```
mysql> show tables;
+--------------------------------+
| Tables_in_jboss                |
+--------------------------------+
| jms_messages                   |
| jms_transactions               |
| leagueejb                      |
| playerejb                      |
| teamejb                        |
| teamejb_players_playerejb_teams |
+--------------------------------+
6 rows in set (0.00 sec)
```

You can see the JMS persistence tables in there too, since we're using MySQL as the default data-source.

8.2.2. Setting up an XADataSource with Oracle 9i

Oracle is one of the main players in the commercial database field and most readers will probably have come across it at some point. You can download it freely for non-commercial purposes from

http://www.oracle.com.

Installing and configuring Oracle is not for the faint of heart – it isn't really just a simple database but is heavy on extra features and technologies which you may not actually want (another Apache web server, multiple JDKs, Orbs etc.) but which are usually installed anyway. So we'll assume you already have an Oracle installation available – for this example, we've used Oracle 9.2.0.1 for Linux[3].

8.2.2.1 Padding Xid Values for Oracle Compatibility

If you look in the *jboss-service.xml* file in the *default/conf* directory, you'll find the following service MBean.

```
<!-- The configurable Xid factory. For use with Oracle, set pad to true -->
<mbean code="org.jboss.tm.XidFactory"
   name="jboss:service=XidFactory">
   <!--attribute name="Pad">true</attribute-->
</mbean>
```

3. If you are installing on Linux and are using Redhat, you have to tweak the installation a bit as it won't work out of the box. Read the article linked to from Oracle's web site and make sure you have plenty of spare time.

The transaction service uses this to create XA transactions identifiers. The comment explains the situation: for use with Oracle you have to include the line which sets the attribute "Pad" to "true". This activates padding the identifiers out to their maximum length of 64 bytes. Remember that you'll have to restart JBoss for this change to be put into effect, but wait until you've installed the JDBC driver classes which we'll talk about next.

8.2.2.2 Installing the JDBC Driver and Deploying the DataSource

The Oracle JDBC drivers can be found in the directory *$ORACLE_HOME/jdbc/lib.* Older versions, which may be more familiar to some users, had rather uninformative names like "*classes12.zip*" but at the time of writing the latest driver version can be found in the file *ojdbc14.jar.* There is also a debug version of the classes with "*_g*" appended to the name which may be useful if you run into problems. Again, you should copy one of these to the *lib* directory of the JBoss default configuration. The basic driver class you would use for the non-XA setup is called `oracle.jdbc.driver.OracleDriver`. The XADataSource class, which we'll use here, is called `oracle.jdbc.xa.client.OracleXADataSource`.

For the configuration file, make a copy of the *oracle-xa-ds.xml* example file and edit it to set the correct URL, username and password:

```
<datasources>
  <xa-datasource>
   <jndi-name>XAOracleDS</jndi-name>
   <track-connection-by-tx>true</track-connection-by-tx>
   <isSameRM-override-value>false</isSameRM-override-value>
   <xa-datasource-class>oracle.jdbc.xa.client.OracleXADataSource</xa-datasource-class>
   <xa-datasource-property name="URL">jdbc:oracle:thin:@monkeymachine:1521:jboss
   </xa-datasource-property>
   <xa-datasource-property name="User">jboss</xa-datasource-property>
   <xa-datasource-property name="Password">password</xa-datasource-property>
   <exception-sorter-class-name>org.jboss.resource.adapter.jdbc.vendor.OracleExceptionSorter
   </exception-sorter-class-name>
   <no-tx-separate-pools/>
  </xa-datasource>

  <mbean code="org.jboss.resource.adapter.jdbc.xa.oracle.OracleXAExceptionFormatter"
         name="jboss.jca:service=OracleXAExceptionFormatter">
   <depends
     optional-attribute-name="TransactionManagerService">jboss:service=TransactionManager
   </depends>
  </mbean>
</datasources>
```

We've used the oracle thin (pure java) driver here and assumed the database is running on the host "monkeymachine" and that the database name (or SID in Oracle terminology) is "jboss". We've also assumed that you've created a user "jboss" with all the sufficient privileges. You can just use "dba" privileges for this example:

```
[oracle@monkeymachine oradata]$ sqlplus /nolog
SQL*Plus: Release 9.2.0.1.0 - Production on Sun Nov 9 23:11:25 2003
Copyright (c) 1982, 2002, Oracle Corporation.  All rights reserved.
```

```
SQL> connect / as sysdba
Connected.
SQL> create user jboss identified by password;
User created.
SQL> grant dba to jboss;
Grant succeeded.
```

Now copy the file to the deploy directory. You should get the following output:

```
21:11:00,046 INFO  [MainDeployer] Starting deployment of package: file:/F:/servers/
jboss-3.2.2/server/default/deploy/oracle-xa-ds.xml
21:11:00,171 INFO  [RARDeployment] Started jboss.jca:service=ManagedConnectionFac-
tory,name=XAOracleDS
21:11:00,171 INFO  [JBossManagedConnectionPool] Started jboss.jca:service=ManagedCon-
nectionPool,name=XAOracleDS
21:11:00,187 INFO  [XAOracleDS] Bound connection factory for resource adapter for Con-
nectionManager
'jboss.jca:service=XATxCM,name=XAOracleDS to JNDI name 'java:/XAOracleDS'
21:11:00,187 INFO  [TxConnectionManager] Started jboss.jca:service=XATxCM,name=XAOra-
cleDS
21:11:00,234 INFO  [OracleXAExceptionFormatter] Started jboss.jca:service=OracleXAEx-
ceptionFormatter
21:11:00,234 INFO  [MainDeployer] Deployed package: file:/F:/servers/jboss-3.2.2/
server/default/deploy/oracle-xa-ds.xml
```

and if you use the JNDIView service from the JMX console as before, you should see the name “java:/XAOracleDS” listed.

8.2.2.3 Testing the Oracle DataSource

Again we’ll use the CMP example to try out the new DataSource. The *jbosscmp-jdbc.xml* file should contain the following:

```
<jbosscmp-jdbc>
   <defaults>
      <datasource>java:/XAOracleDS</datasource>
      <datasource-mapping>Oracle9i</datasource-mapping>
   </defaults>
</jbosscmp-jdbc>
```

There are other oracle type-mappings available too – if you’re using an earlier version, have a look in the *conf/standardjbosscmp-jdbc.xml* file to find the names. As above, you can also modify the default values directly in this file which will set them for all CMP deployments and also save you having to re-package the EAR file.

Deploy the application as before, check the output for errors and then check that the tables have been created using Oracle SQLPlus again from the command line:

```
SQL> select table_name from user_tables;

TABLE_NAME
------------------------------
LEAGUEEJB
PLAYEREJB
TEAMEJB
TEAMEJB_PLAYERS_PLAYE_1TKRO4S
```

CHAPTER 9 Security Configuration

We've already seen how to set up simple security when we looked at the Duke's Bank application (See "Security" on page 26.). We looked at how to enable security by adding a security domain element to the jboss-specific deployment descriptors and thus linking your application to a configuration in the *login-config.xml* file. However we only used simple file based security in that chapter.

In this chapter, we'll examine some more advanced configuration options and find out how to use some of the other login modules that are available.

9.1. Security Using a Database

One of the most likely scenarios is that your user and role information is stored and maintained in a database. JBoss comes with a login module called `DatabaseServerLoginModule` which just needs some simple configuration options to set it up. You need to supply

- the SQL query to retrieve the password for a specified user.
- the query to retrieve a user's roles.
- the JNDI name of the DataSource to be used.

This gives you the flexibility to use an existing database schema. Let's suppose that the security database tables were created using the following SQL

```
CREATE TABLE Users(username VARCHAR(64) PRIMARY KEY, passwd VARCHAR(64))
CREATE TABLE UserRoles(username VARCHAR(64), userRoles VARCHAR(32))
```

then to use this as the security database for Duke's Bank, you would modify the "dukesbank" entry in the JBoss *login-config.xml* file as follows:

```
<application-policy name="dukesbank">
  <authentication>
    <login-module code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
           flag="required">
      <module-option name="dsJndiName">java:/DefaultDS</module-option>
      <module-option name="principalsQuery">
         select passwd from Users where username=?
      </module-option>
      <module-option name="rolesQuery">
         select userRoles,'Roles' from UserRoles where username=?
      </module-option>
    </login-module>
  </authentication>
</application-policy>
```

The query to retrieve the password is straightforward. In the case of the roles query you will notice that there is an additional field with value "Roles" which is the "role group". This allows you to store additional roles (for whatever purpose) classified by the role group. The ones which will affect JBoss permissions are expected to have the value "Roles". In this simple example we only have a single set of roles in the database and no role group information[1].

We've used the default DataSource here. If you're using Hypersonic, then you can easily create the tables and insert some data using the Database Manager tool which we also used in the Duke's Bank chapter. Just execute the two commands above and then the following ones to insert the information for the user with customer Id "200" :

```
INSERT INTO Users VALUES('200','j2ee')
INSERT INTO UserRoles VALUES('200','BankCustomer')
```

and you should be able to login as before.

1. You can also use the default schema which is to have a table called "Principals" with columns "PrincipalID" and "Password" and a table called "Roles" with columns "PrincipalID", "Role" and "RoleGroup". In this case you don't have to specify the SQL queries for the login module. The RoleGroup entries for JBoss permissions should be set to the value "Roles" as before.

9.2. Using Password Hashing

The login modules we've used so far all have support for password hashing; rather than storing passwords in plain text, a one-way hash of the password is stored (using an algorithm such as MD5), in a similar fashion to the passwd file on a Unix system. This has the advantage that anyone reading the hash won't be able to use it to log in. However there is no way of recovering the password should the user forget it and it also makes administration slightly more complicated because you also have to calculate the password hash yourself to put it in your security database. This isn't a major problem though. To enable password hashing in the database example above, you would add the following module options to the configuration

```
<module-option name="hashAlgorithm">MD5</module-option>
<module-option name="hashEncoding">base64</module-option>
```

This indicates that we want to use MD5 hashes and use Base64 encoding to covert the binary hash value to a string. JBoss will now calculate the hash of the supplied password using these options before authenticating the user, so it's important that we store the correctly hashed information in the database. If you're on a Unix system or have Cygwin installed on Windows you can use the following command:

```
$ echo -n "j2ee" | openssl dgst -md5 -binary | openssl base64
glcikLhvxq1BwPBZN0EGMQ==
```

and insert the string "`glcikLhvxq1BwPBZN0EGMQ==`" into the database in place of the password "j2ee". If you don't have this option, you can use the class `org.jboss.security.Base64Encoder` which you'll find in the *jbosssx.jar* file.

```
$ java -classpath ./jbosssx.jar org.jboss.security.Base64Encoder j2ee MD5
[glcikLhvxq1BwPBZN0EGMQ==]
```

With a single argument it will just encode the given string but if you supply the name of a digest algorithm as a second argument it will calculate the hash of the string first.

APPENDIX A The Web Console

Throughout this book we have been referring to the JMX Console web application which you can view by browsing to **http://localhost:8080/jmx-console**. However there is also a new management console application which extends the functionality to include statistics on deployed J2EE components such as EJBs and servlets.

The URL for the web console is **http://localhost:8080/web-console**. You will get more out of it if you have some applications deployed and been running them to accumulate some statistics. For example, with the Duke's Bank application deployed you'll see something like Figure A.1, which shows the statistics for the AccountController stateful session bean. The invocation statistics are self-explanatory; you have a list of methods and the max, min, average time per invocation as well as the total time spent in the method and the number of invocations. The number of concurrent invocations is shown underneath the table of methods.

The information in the "Bean Statistics" section shows information on the bean instance numbers. The details vary depending on the type of bean and the possible values are shown in Table 1 on page 60. For a complete description of the bean states ("method-ready", "pooled", "ready" etc.) see the EJB specification.

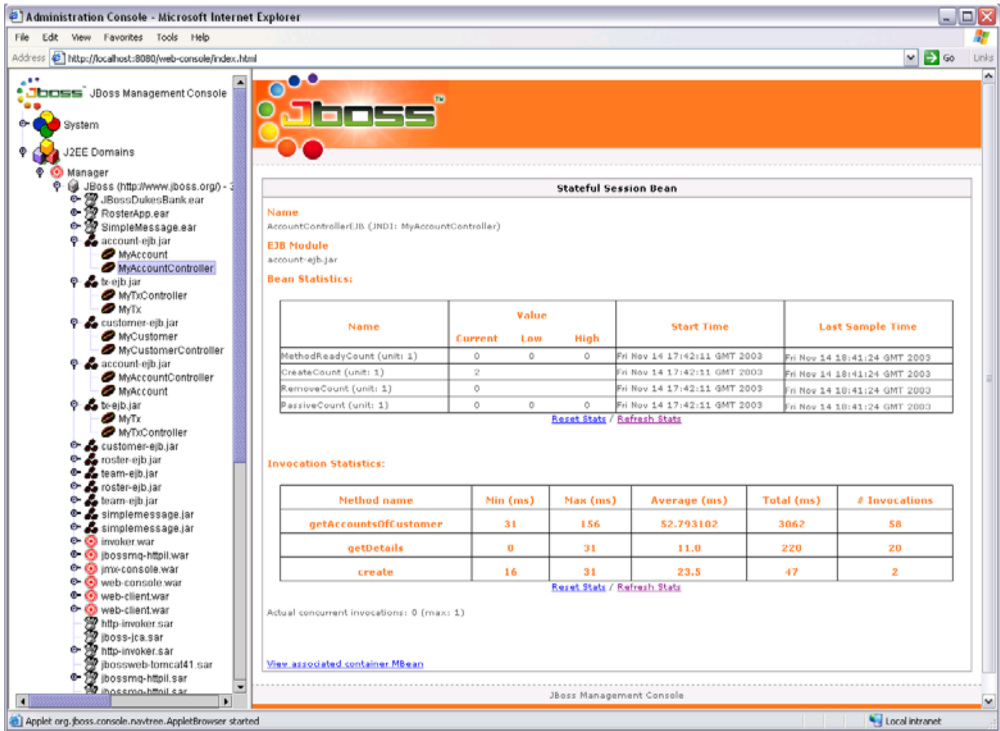

FIGURE A.1. Web Admin. Console Showing Stateful Session Bean Statistics.

TABLE 1. Bean Statistics Data

Stateless Session Bean	Description
MethodReadyCount	Number of beans in the “method-ready” state.
CreateCount	Number of times create method has been called.
RemoveCount	Number of times remove method has been called.
Stateful Session Bean	
MethodReadyCount	The number of beans in the “method-ready” state.
CreateCount	The number of beans that have been created

TABLE 1. **Bean Statistics Data**

RemoveCount	The number of beans that have been explicitly removed[a].
PassiveCount	The number of beans that have been passivated by the container.
Entity Bean	
CreateCount	Number of entities that have been created by calls to create method.
RemoveCount	Number of entities that have been removed (deleted) by calling remove method.
ReadyCount	Number of beans that are in the "ready" state - assigned an entity object and ready to handle invocations.
PooledCount	Number of beans in the "pooled" state. JBoss doesn't use entity instance pooling so this will be zero.

a. Note that RemoveCount may not equal CreateCount over time as the beans may be passivated and then time-out without the remove method being called.

The web-console isn't a pure web application but uses a Java applet for the tree view on the left-hand side. So you'll need to have the Java plugin installed and have Java enabled to make it work.