Getting Started with JBoss

J2EE Applications on JBoss 4.0

Table of Contents

Preface

1. 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.

2. Target Audience

The goal of this book is to get you up and running J2EE 1.4 applications on JBoss 4.0 as quickly as possible. At the time of writing, the latest release is version 4.0.0. You should use this version or later with the examples. We will use Sun's J2EE 1.4 tutorial examples (http://java.sun.com/j2ee/1.4/docs/tutorial/doc/) 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.

3. What this Book Covers

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.

Chapter 5 covers web services. We work through how to expose EJB methods from the Duke's Bank application through web services and then call them with a Java client.

Configuration of databases is an important issue and this is covered in Chapter 8. We also work through some step-by-step examples.

In Chapter 9 we look at some more advanced security configuration options.

Suggestions for additional topics are always welcome.

Chapter 1. Getting Started

1.1. Downloading and Installing JBoss

The JBoss application server is available as a free download from the JBoss website. (http://www.jboss.org/downloads/index) We provide both a binary and source distribution, but if you are just getting started with JBoss, stick to the binary distribution, which can be run straight out of the box.

The binary versions are available as either `.zip`, `.tar.gz`, `.bz2` files. The contents are the same so grab whichever flavor 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-4.0.0`. Of course the version number suffix will be different if you are running a later release. Make sure you don't use a directory which has any spaces in the path (such as the `Program Files` directory on Windows) as this may cause problems.

The only additional requirement to run JBoss is to have an up-to-date version of Java on your machine. JBoss 4.0 requires at least a 1.4 JDK to run. Make sure to get the JDK and not the JRE. Although JBoss will startup with the JRE only, you'll experience problems compiling JSPs with it. You should also make sure the `JAVA_HOME` environment variable is set to point to your JDK installation.

1.2. Starting the Server

Our first step is to 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, OS X, 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 look like the following.

```
11:29:39,944 INFO  [Server] JBoss (MX MicroKernel) [4.0.0 (build: CVSTag=JBoss_4_0_0 date=200409200418
```

You can verify that the server is running by going the JBoss web server, which is running on port 8080. (Make sure you don't have anything else already on your machine using that port) The default page has links to a few useful JBoss resources.

1.3. The JMX Console

You can get a live view of the server by going to the JMX console application at http://localhost:8080/jmx-console [1]. 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. 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 it. 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 returns a view of the current names bound into the JNDI tree, which is very useful when you start deploying your own applications and want to

[1] Note that on some machines, the name `localhost` won't resolve properly and you should use the local loopback address 127.0.0.1 instead.

know why you can't resolve a particular EJB name.

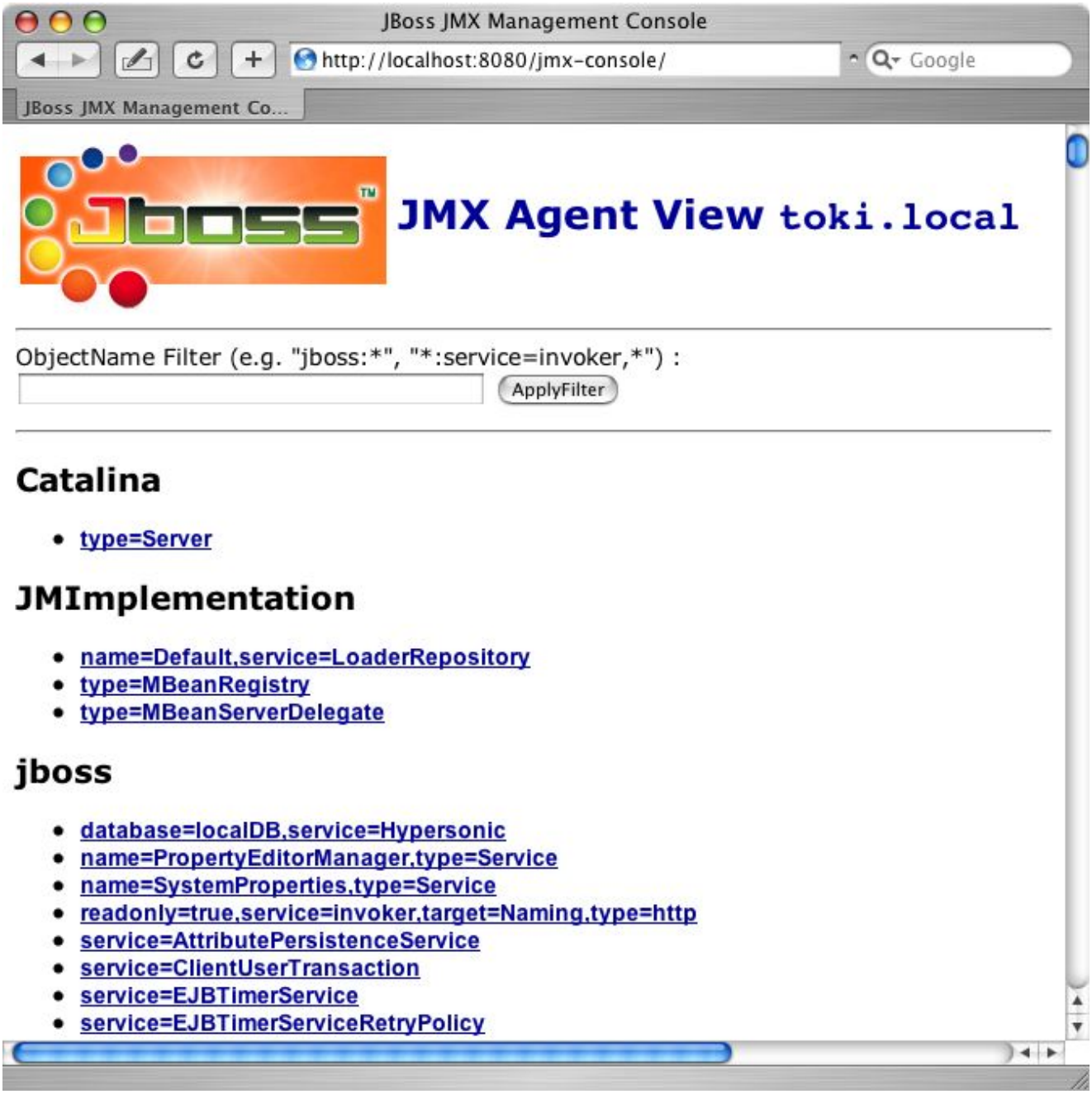

Figure 1.1. View of the JMX Management Console Web Application

Look at some of the other MBeans and their listed operations; try changing some of the configuration attributes and see what happens. With a few exceptions, 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.

1.4. Stopping the Server

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).

1.5. 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 are examples in the JBoss `bin` directory called `jboss_init_redhat.sh` and `jboss_init_suse.sh`which you can modify and use. On a Windows system, you can use a utility like Javaservice[2] to manage JBoss as a system service.

[2]Javaservice 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 JBoss and have run the server for the first time, the next thing you will want to know is how the installation is 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 ignore for the time being. To remedy that, we'll explore the server directory structure, locations of the key configuration files, log files, deployment and so on. It's worth familiarizing yourself with the layout at this stage as it will help you understand the JBoss service architecture so that you'll be able to find 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-4.0.0` directory. Throughout the book, we will refer to this as the `JBOSS_DIST` directory. There are four sub-directories immediately below this:

- **bin**: contains startup and shutdown and other system-specific scripts. 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[3] configuration files for setting up datasources for different databases (such as MySQL, Oracle, Postgres)[4].
- **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 `-c <config name>` to the run script. We'll look at the standard server configurations next.

[3] J2EE Connector Architecture - provides a standard for providing connectivity between application servers and existing Enterprise Information Systems (EIS).

[4] 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.

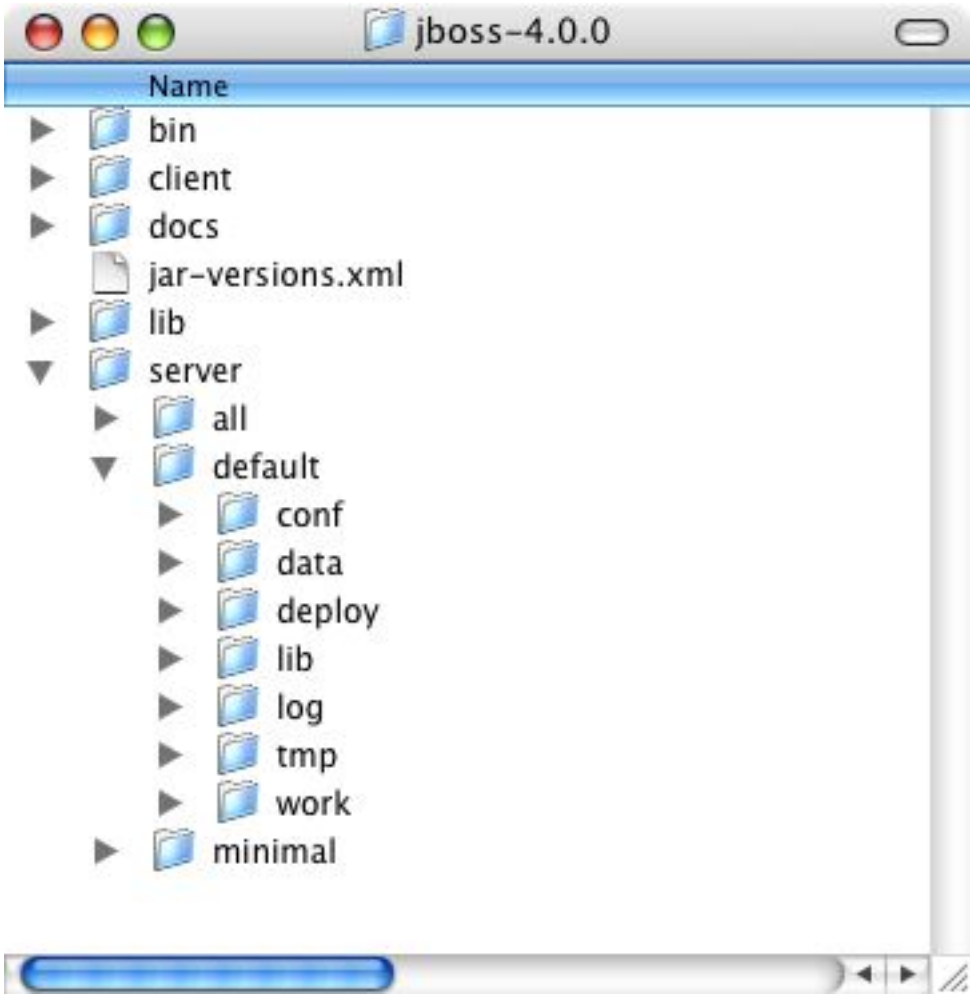

Figure 2.1. JBoss Directory Structure

2.1.2. Server Configurations

Fundamentally, the JBoss architecture consists of the JMX MBean server, the microkernel, and a set of pluggable component services, the 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 to use JBoss, 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 four example server configurations: `all`, `default`, `standard` and `minimal`, each of which provides a different set of services. Not surprisingly, the **default** configuration is the one used if you don't specify another one when starting up the server, so that's the one we were running in the previous chapter. It is the J2EE 1.4 certified configuration, and contains everything you need to run a stand-alone J2EE server. The other configurations are explained below.

- **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 any other J2EE technologies. This is just the bare server. There is no web container, no EJB or JMS support.

- **standard**: this is the base J2EE 1.4 compliant configuration and is similar to the `default` configuration in JBoss 3.2. It does not include the JAXR service, the IIOP service, or any of the clustering services.
- **all**: starts all the available services. This includes the RMI/IIOP and clustering services, 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 new configuration.

```
run -c myconfig
```

The directory server configuration you're using, is effectively the server root while JBoss is running. It contains all the code and configuration information for the services provided by the particular configuration. 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` server configuration directory. If you haven't tried running the server yet, then do so now, as a few of the sub-directories are only created when JBoss starts for the first time.

- **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[5]. 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) that 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 and 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:

```
13:10:05,235 INFO  [MailService] Mail service 'java:/Mail' removed from JNDI
```

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

[5] The J2EE Connector Architecture defines the Resource Adapter Archive (RAR) file, which is 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. All Path are relative to the `default` configuration directory.

2.2.1. Core Services

The core services specified in the `conf/jboss-service.xml` file are started first when the server starts up. 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 that we saw earlier). Try commenting out the entry for the JNDIView service.

```
<!--
   <mbean code="org.jboss.naming.JNDIView"
               name="jboss:service=JNDIView"
               xmbean-dd="resource:xmdesc/JNDIView-xmbean.xml">
   </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, which allows your service to be hot deployable.

2.2.2. Logging Service

We mentioned already that log4j is used for logging. If you're not familiar with the log4j package and would like to use it in your applications, you can read more about it at the Jakarta web site. (http://jakarta.apache.org/log4j/) Logging is controlled from a central `conf/log4j.xml`file. This file defines a set of appenders, specifying the log files, what categories of messages should go there, the message format and the level of filtering. By default, JBoss produces output to both the console and a log file (`server.log` in the log directory).

There are 4 basic log levels used: `DEBUG`, `INFO`, `WARN` and `ERROR`. The logging threshold on the console is `INFO`, which means that you will see informational messages, warning messages and error messages on the console but not general debug messages. In contrast, there is no threshold set for the `server.log` file, so all generated logging messages will be logged there. 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. However, be aware that just because the logging threshold allows debug messages to be displayed, that doesn't mean that all of JBoss will produce detailed debug information for the log file. You will also have to boost the logging limits set for individual categories. Take the following category in `server.xml` for example.

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

This limits the level of logging to `INFO` for all JBoss classes, apart from those which have more specific overrides provided. If you were to change this to `DEBUG`, it would produce a much more detailed logging output.

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>
```

This 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 (Chapter 7).

The file appender is set up to produce a new log file every day, so it doesn't produce a one every time you re-start the server and it won't write to a single file indefinitely. The current log file is `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.

2.2.3. 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[6] 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`. We'll quickly look at how to do this to secure the JMX Console and the Web Console applications that ship with JBoss.

We saw the JMX Console briefly in Section 1.3. Almost every aspect of the JBoss server can be controlled through the JMX Console, so it is important to make sure that, at the very least, the application is password protected. Otherwise, any remote user could completely control your server. To protect it, we will add a security domain to cover the application. This can be done in the `jboss-web.xml` file for the JMX Console, which can be found in `server/default/deploy/jmx-console.war/WEB-INF/` directory. Uncomment the `security-domain` in that file, as shown below.

```
<jboss-web>
    <security-domain>java:/jaas/jmx-console</security-domain>
</jboss-web>
```

This links the security domain to the web application, but it doesn't tell the web application what security policy to enforce. What URLs are we trying to protect, and who is allowed to access them? To configure this, go to the `web.xml` file in the same directory and uncomment the `security-contraint` that is already there. This security constraint will require a valid user name and password for a user in the `JBossAdmin` group.

```
<!--
   A security constraint that restricts access to the HTML JMX console
   to users with the role JBossAdmin. Edit the roles to what you want and
   uncomment the WEB-INF/jboss-web.xml/security-domain element to enable
   secured access to the HTML JMX console.
-->
<security-constraint>
```

[6] 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.

```
    <web-resource-collection>
        <web-resource-name>HtmlAdaptor</web-resource-name>
        <description>
            An example security config that only allows users with the
            role JBossAdmin to access the HTML JMX console web application
        </description>
        <url-pattern>/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>JBossAdmin</role-name>
    </auth-constraint>
</security-constraint>
```

That's great, but where do the user names and passwords come from? They come from the `jmx-console` security domain we linked the application to. We've provided the configuration for this in the `conf/login-config.xml`.

```
<application-policy name="jmx-console">
    <authentication>
        <login-module code="org.jboss.security.auth.spi.UsersRolesLoginModule"
                      flag="required">
            <module-option name="usersProperties">
                jmx-console-users.properties
            </module-option>
            <module-option name="rolesProperties">
                jmx-console-roles.properties
            </module-option>
        </login-module>
    </authentication>
</application-policy>
```

This configuration uses a simple file based security policy. The usernames and passwords are stored in `jmx-console-users.properties` and take the form "`username=password`". To assign a user to the `JBossAdmin` group add "`username=JBossAdmin`" to the `jmx-console-roles.properties` file. The existing file creates an `admin` user with the password `admin`. You'll want to remove that user or change the password to something stronger.

JBoss will re-deploy the JMX Console whenever you update it's `web.xml`. You can check the server console to verify that JBoss has seen your changes. If you've configured everything correctly and re-deployed the application, the next time you try to access the JMX Console, JBoss will ask you for a name and password.

The JMX Console isn't the only web based management interface to JBoss. There is also the Web Console. (See Appendix A) Although it's a Java applet, the corresponding web application can be secured in the same way as the JMX Console. The Web Console is in `default/deploy/management/web-console.war`. The only difference is that the Web Console is provided as a simple WAR file instead of using the exploded directory structure that the JMX Console did. The only real difference between the two is that editing the files inside the WAR file is a bit more cumbersome.

2.2.4. Additional Services

The non-core, hot-deployable services are added to the `deploy` directory. They can be either XML descriptor files, `*-service.xml`, or JBoss Service Archive (SAR) files. SARs contain both the XML descriptor and additional resources 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:

- **bsh-deployer**: deploys bean shell scripts as JBoss services.
- **client-deployer-service.xml**: deploys J2EE application clients.
- **ear-deployer.xml**: deploys J2EE EJB JAR files.
- **ebxmlrr-service.xml**: the JAXR registry service implementation.
- **ejb-deployer.xml**: deploys J2EE EAR files.
- **hibernate-deployer-service.xml**: deploys Hibernate archives (HAR files).
- **hsqldb-ds.xml**: sets up the embedded Hypersonic database service and the default data source.
- **iiop-service.xml**: enables CORBA and IIOP support.
- **jboss-jdbc-metadata.sar**: a service that allows the datasource-mapping for a CMP2 deployment to be externalized from the jbosscmp-jdbc.xml descriptor so that deployments can be independent of the type of datasource.
- **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.
- **jboss-ws4ee.sar**: provides J2EE web services support.
- **jbossjca-service.xml**: the JBoss JCA implementation. Allows the deployment of JCA resource adaptors within JBoss.
- **jbossweb-tomcat50-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 that we used in the previous chapter.
- **jmx-invoker-server.xml**: provides remote access to the JMX MBean server.
- **mail-service.xml**: allows applications and services to use JavaMail from within JBoss. Must be configured with relevant mail server information.
- **management**: sub-directory containing alternative management services, including an improved web console.
- **monitoring-service.xml**: Alert monitors like the console listener and email listener are configured here.
- **properties-service.xml**: amongst other things, allows the setting of global system properties (as returned by `System.getProperties`).
- **sqlexception-service.xml**: provides a means of identifying non-fatal SQL exceptions for a given JDBC driver.

- **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 that define the transport protocols over which message transfer can take 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[7].

Looking beyond the `default` configuration into the `standard` configuration, a few additional services are provided.

- **cache-invalidation-service.xml**: allows customized control of the EJB cache via JMS.
- **http-invoker.sar**: provides RMI/HTTP access for MBeans and EJBs.
- **jboss-aop.deployer**: provides the `AspectManagerService` and deploys JBoss AOP applications.
- **schedule-manager-service.xml** and **scheduler-service.xml**: task scheduling service.

The `all` configuration contains all the additional services provided by JBoss that you might want to incorporate into your configuration.

- **cluster-service.xml**: the cluster services, including the JGroups integration service, HA-JNDI, stateful session bean replication, and the CMP2 cache invalidation service.
- **deploy-hasingleton-service.xml**: the `HASingletonSDeployer` which insures that only a single node in a cluster deploys the services in the `deploy-hasingleton` directory.
- **deploy.last/farm-service.xml**: the farm cluster deployment service. It is in the deploy.last directory to ensure that it is deployed after all other services.
- **jbossha-httpsession.sar**: the legacy HTTP session replication service.
- **remoting-service.xml**: the experimental next generation detached invoker framework.
- **snmp-adaptor.sar**: translates JMX notifications into SNMP traps.
- **tc5-cluster-service.xml**: the TreeCache configuration for the new HTTP replication service.

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).

[7] 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.

2.3. The Web Container - Tomcat

JBoss now comes with Tomcat 5.0 as the default web container. The embedded Tomcat service is the expanded SAR `jbossweb-tomcat50.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 `server.xml`, 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).

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/1.4/docs/tutorial/doc/. You should read the getting started information there and download the example code from http://java.sun.com/j2ee/1.4/download.html#tutorial.

Duke's Bank also makes use of the JavaServer Pages Standard Tag Library (JSTL) framework. We'll use the Apache implementation, which you can get from the Jakarta web site at http://jakarta.apache.org/taglibs/doc/standard-doc/intro.html. You will need to copy the JSTL JAR files over in a moment, so make sure to download the libraries and keep them available.

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, which is available along side this document on the JBoss documentation page, http://www.jboss.org/docs/index.

The tutorial uses the Apache Ant build tool, which you should download and install[8]. 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.

3.2. 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 standard and also do not need to 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.

3.2.1. 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. 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.

[8] 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.

3.2.2. 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 Chapter 8.

3.2.3. 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.

3.3. J2EE in the Real World

The examples here are only intended to get you up and running with JBoss and to help you familiarize 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.

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.

Finally, we hope you'll realize 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 4.0 series.

Chapter 4. The Duke’s Bank Application

Now that you have the server running, we use the Duke’s Bank example from the J2EE tutorial to illustrate how to get an application up and running in JBoss. Duke’s Bank demonstrates a selection of J2EE technologies working together to implement a simple on-line banking application. It uses EJBs and 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 1.4 tutorial, which includes Duke’s Bank, as well as the JSTL libraries described in Section 3.1. 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 is packaged as a ZIP archive called `jbossj2ee-src.zip`. Download this and unpack it into the `j2eetutorial14` directory, adding to the existing tutorial files. All the Duke’s Bank code is in a the `examples/bank` subdirectory. If you've unpacked the JBoss extensions correctly, you will see a `jboss-build.xml` there. This is our Ant build script for the JBoss version of the application. Rather than just overwriting the existing `build.xm` file, we’ve used a different name from the default. This means that ant must now be run as `ant -f jboss-build.xml`.

Before we can build, we'll need to perform two setup steps. First, we'll need to install the Jakarta JSTL libraries we downloaded earlier. Copy the `jstl.jar` and `standard.jar` files from the Jakarta JSTL distribution to `bank/jar`. Next, you'll need to edit the `jboss-build.properties` file in the `j2eetutorial14` to point to your JBoss install directory. Set the `jboss.home` property to the full path to your JBoss 4.0 installation. If you’ve unpacked JBoss 4.0 in the `C:` drive on a windows machine, you would set it as follows.

```
# 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-4.0.0
```

4.1.2. Compiling the Java Source

At the command line, go to the `bank` directory. All the build commands will be run from here. Compilation is pretty straightforward; just type the following command to invoke the `compile` Ant target.

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

If there aren’t any errors, you will find a newly created `build` directory with the class files in it.

4.1.3. Package the EJBs

The application has one EJB jar, `bank-ejb.jar`, which contains the code and descriptors (`ejb-jar.xml` and

`jboss.xml`) for the entity beans and associated controller session beans which the clients interact with. The `pacakge-ejb` Ant target will create them in the `jar` directory.

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

4.1.4. Package the WAR File

The 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 the extra JSP tag-library descriptors required by the web application. The `package-web` Ant target builds the web client WAR file.

```
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 `package-client` Ant target.

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

The generated WAR file 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, `application.xml`. It is also possible to deploy EJBs and web application modules individually but the EAR provides a convenient single unit. The `assemble-app` Ant target will produce the final file `JBossDukesBank.ear`.

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

4.1.7. The Database

Before we can deploy the application, we need to populate the database it will run against. If you are writing an application that uses container-managed persistence, you can configure the CMP engine to create the tables for you at deployment, but otherwise you will need have to have a set of scripts to do the job. This is also a convenient place 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 (the HSQL documentation

refers to this as server mode). Since we are going to be running the SQL scripts using a tool that connects to the database, we want to make sure the database is running in client-server mode and will accept TCP/IP connections. In later versions of JBoss, client-server mode is disabled to prevent direct database access, which could be a security risk if the default login and password have 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.

```
<!-- 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 so JBoss will start the database in the correct mode.

4.1.7.2. Creating the Database Schema

We have supplied scripts to run with HSQL in the `sql` directory. The database tasks in the build file will try to contact HSQL database. If JBoss isn't already running, you should start it now, so that the database is available.

First we need to create the necessary tables with the `db-create-table` target.

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

Then run `db-insert` target to populate them with the required data.

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

Finally, if everything has gone according to plan, you should be able to view some of the data using the `db-list` target, which lists the transactions for a specific account.

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

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`. If you can’t find this, make sure the service is enabled as described in Section 4.1.7.1.

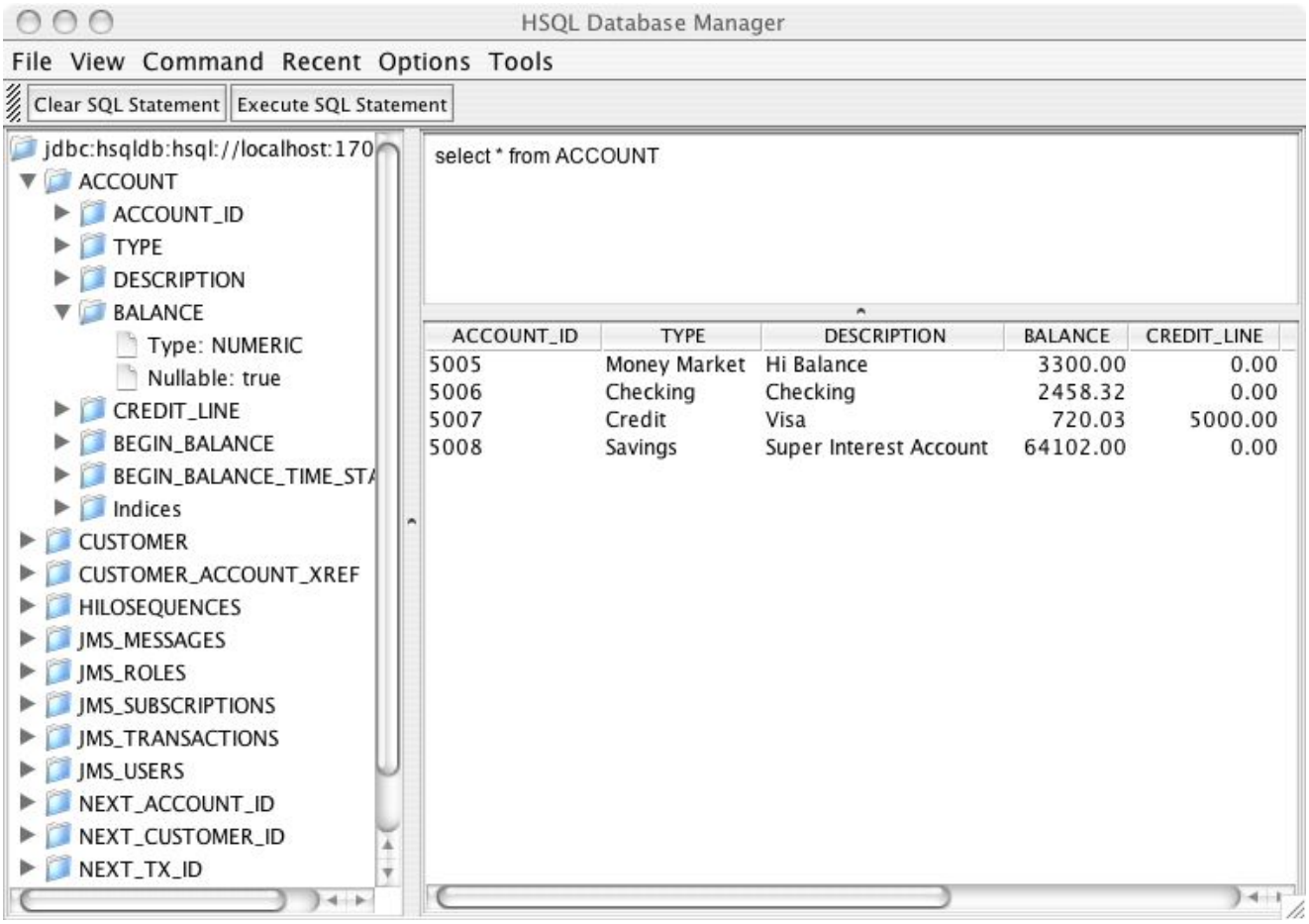

Figure 4.1. The HSQL Database Manger

This will take you to the information for the Hypersonic service MBean. 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.

4.1.8. Deploying the Application

Deploying an application in JBoss is easy. You just have to copy the EAR file to the deploy directory. The `deploy` target in the build file does this for our application.

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

You should see something close to the following output from the server (reduced for brevity).

```
14:14:53,763 INFO  [EARDeployer] Init J2EE application: file:/Users/orb/java/jboss-4.0.0/
server/default/deploy/JBossDukesBank.ear
14:14:55,588 INFO  [EjbModule] Deploying CustomerEJB
14:14:56,049 INFO  [EjbModule] Deploying CustomerControllerEJB
...
```

```
14:14:59,292 INFO  [EJBDeployer] Deployed: file:/Users/orb/java/jboss-4.0.0/server/
default/tmp/deploy/tmp43515JBossDukesBank.ear-contents/customer-ejb.jar
14:14:59,727 INFO  [EJBDeployer] Deployed: file:/Users/orb/java/jboss-4.0.0/server/
default/tmp/deploy/tmp43515JBossDukesBank.ear-contents/tx-ejb.jar
14:15:00,321 INFO  [EJBDeployer] Deployed: file:/Users/orb/java/jboss-4.0.0/server/
default/tmp/deploy/tmp43515JBossDukesBank.ear-contents/account-ejb.jar
14:15:00,608 INFO  [TomcatDeployer] deploy, ctxPath=/bank, warUrl=file:/Users/orb/java/
jboss-4.0.0/server/default/tmp/deploy/tmp43515JBossDukesBank.ear-contents/web-client-
exp.war/
14:15:01,594 INFO  [EARDeployer] Started J2EE application: file:/Users/orb/java/
jboss-4.0.0/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 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 will 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` If you get an error at this point, check again that you have set up the database correctly as described in Section 4.1.7.1. In particular, check that the `connection-url` is right. Then make sure that you have populated the database with data.

You can also run the standalone client application using the `run-client` target.

```
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 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 Section 3.2.1 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. The properties can either be hard-coded, or supplied in a file named `jndi.properties` on the classpath. The file we've used is shown below.

```
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 standard properties, which are set up in order to use the JBoss JNDI implementation. The `j2ee.clientName` property 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>
    <ejb-ref>
        <ejb-ref-name>ejb/accountController</ejb-ref-name>
        <jndi-name>MyAccountController</jndi-name>
    </ejb-ref>
</jboss-client>
```

Of course if you were only building a simple web application, you wouldn't need to worry about remote clients

4.2.2. Application JNDI Information in the JMX Console

While we’re on the subject of JNDI, let’s 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, which displays the JNDI tree for the server. You should see the EJB modules 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.

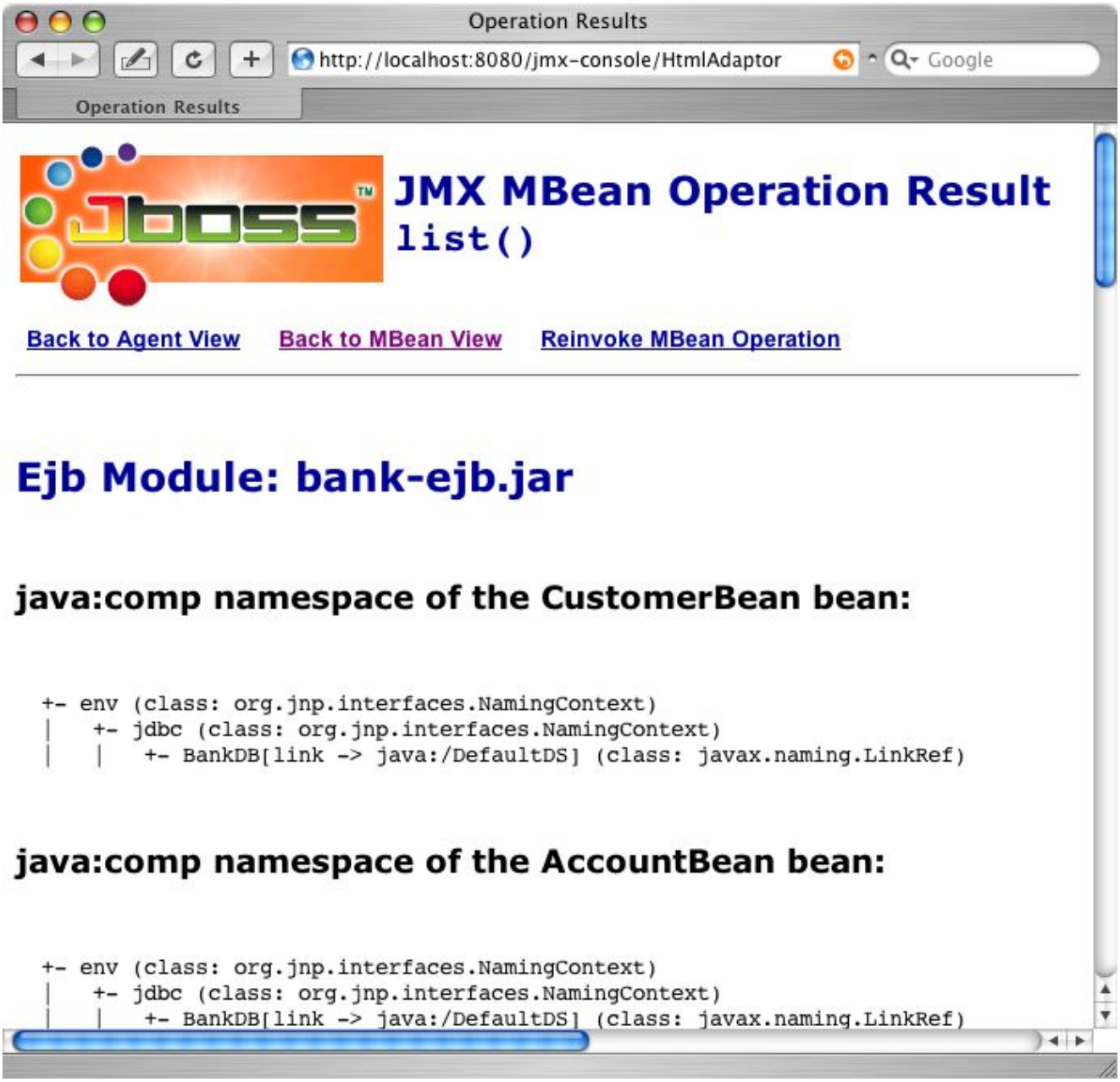

Figure 4.2. JMX Console JNDI View

Further down, under the `java:` namespace[9] is a list of the active security managers, bound under their security-domain names

```
+- 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)
```

Note that these objects are created on demand, so the `dukesbank` entry will only appear if you have configured the application to use the dukesbank domain and 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.

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. That is not very good 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. That's not exactly ideal either.

If a web application doesn't have a security domain [10] specified, 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 also add an identical `security-domain` element to the `jboss.xml` file too

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

    <enterprise-beans>
        ...
    </enterprise-beans>
</jboss>
```

What this means is that JBoss will bind a security manager instance for our application under the JNDI name `java:/jaas/dukesbank`. You can configure it in the `conf/login-config.xml` file, which we first saw in Section 2.2.3. 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 like the following.

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

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 `UsersRolesLoginModule` 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.

[10] 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 the term realm.

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.
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 of the form `username=role1,role2,...`.

In Duke's Bank, the user id 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 to authentication. 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 Chapter 9.

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 adequately 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 specifies that a HTML form login should be used to obtain a username and password. (The alternative would be HTTP BASIC authentication, where the browser presents 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, namely the URL to submit to and the field names for username and password, are defined in the servlets specification. You can see how it works in the `logon.jsp` file in this application.

You should keep in mind that the authentication logic which decides whether a login succeeds or fails is outside the scope of the J2EE specification. 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 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).

Chapter 5. J2EE Web Services

From the start, web services have promised genuine interoperability by transmitting XML data using platform and language-independent protocols such as SOAP over HTTP. While the early days of multiple competing standards and general developer confusion may have made this more of a dream than a reality, web services have matured and standardized enough to have been incorporated into the J2EE 1.4 specification.

Keeping with the spirit of this guide, we'll assume you have some experience with web services already. If you don't, we would recommend you do some reading in advance. A good place to start would be http://www.jboss.org/wiki/Wiki.jsp?JBossWS on the JBoss wiki, which covers web services on JBoss in more depth. We also recommend *J2EE Web Services* by Richard Monson-Haefel for more general coverage of J2EE web services.

5.1. Web services in JBoss

JBossWS is the JBoss module responsible for providing web services in JBoss 4.0, replacing the previous JBoss.NET package. Like it's predecessor, it is also based on Apache Axis (http://ws.apache.org/axis). However, JBossWS provides the complete set of J2EE 1.4 web services technologies, including SOAP, SAAJ, JAX-RPC and JAXR.

J2EE web services provides for two types of endpoints. If you think of a web service as a platform-independent invocation layer, then the endpoint is the object you are exposing the operations of and invoking operations on. Naturally, J2EE web services support exposing EJBs as web services, but only stateless session beans can be used. That makes sense given the stateless nature of web services requests. Additionally, J2EE web services provide for JAX-RPC service endpoints, (JSEs) which are nothing more than simple Java classes. We'll only be working with EJB endpoints in this example.

5.2. Duke’s Bank as a Web Service

We'll continue working with the Duke's Bank application from Chapter 4 and create a simple web service for querying accounts and balances. The `AccountController` session bean provides this functionality to the Duke's Bank web application. Unfortunately the application uses stateful session beans as it's external interface, so we can't expose the `AccountController` session bean directly. Instead, we'll create a new stateless session bean, the `TellerBean`, which will provide a more suitable web service endpoint.

Before we start, make sure that you have built and deployed Duke's Bank according to the instructions in Chapter 4. As with that example, we'll be working from the `examples/bank` directory. Although `TellerBean` will have already been compiled when you deployed Duke's Bank, you'll need to remember to invoke the `compile` target to compile any changes you might make.

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

The magic of J2EE is in the deployment descriptors. We've seen how to deploy session beans already. Deploying a session bean as a web service is as simple as adding a `service-endpoint` element to the session bean definition in `ejb-jar.xml`. The service-endpoint specifies the class that provides the interface corresponding to the methods on the session bean being exposed as a web service.

```
<session>
    <ejb-name>TellerBean</ejb-name>
    <service-endpoint>com.jboss.ebank.TellerEndpoint</service-endpoint>
    <ejb-class>com.jboss.ebank.TellerBean</ejb-class>
```

```
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
    <ejb-ref>
        <ejb-ref-name>ejb/account</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <home>com.sun.ebank.ejb.customer.CustomerHome</home>
        <remote>com.sun.ebank.ejb.customer.Customer</remote>
    </ejb-ref>
</session>
```

You might have noticed that we didn't declare a home or remote interface for `TellerBean`. If your session bean is only accessed by the web services interface, you don't need one, so we've left them out here. Instead, we've declared the `TellerEndpoint` class as our endpoint interface. Our web service interface exposes two operations, both of which map onto the equivalent methods on `TellerBean`.

```
public interface TellerEndpoint
    extends Remote
{
    public String[] getAccountsOfCustomer(String customerId)
        throws RemoteException;

    public BigDecimal getAccountBalance(String accountID)
        throws RemoteException;
}
```

We'll generate our WSDL, the interoperable web services definition, from this interface using `java2wsdl`, an Axis tool which comes with the JBossWS. The `wsdl` target does this.

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

This generates the `dd/ws/wsdl/teller.wsdl` file representing our service. WSDL can be very verbose, so we won't duplicate the file here. But, we will point out two important things. First, the `wsdlsoap:address` in the `wsdl:service` is deliberately bogus.

```
<wsdl:service name="TellerService">
    <wsdl:port name="TellerEndpoint" binding="impl:TellerEndpointSoapBinding">
        <wsdlsoap:address location="http://this.value.is.replaced.by.jboss"/>
    </wsdl:port>
</wsdl:service>
```

JBoss will replace the `wsdlsoap:address` with the actual value when it deploys the web service, so there is no need to worry about it at this point.

The other detail to note from the generated WSDL file is that the namespace for our webservice is `http://ebank.jboss.com`. We'll need to make sure we map the `namespaceURI` in our JAX-RPC mapping file.

```
<java-wsdl-mapping xmlns="http://java.sun.com/xml/ns/j2ee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
          http://www.ibm.com/webservices/xsd/j2ee_jaxrpc_mapping_1_1.xsd"
      version="1.1">

    <package-mapping>
        <package-type>com.jboss.ebank</package-type>
        <namespaceURI>http://ebank.jboss.com</namespaceURI>
    </package-mapping>
</java-wsdl-mapping>
```

The last piece of the deployment descriptor puzzle is the `webservices.xml` file, which associates our webservice with the WSDL and mapping files we've created.

```
<webservices xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                        http://www.ibm.com/webservices/xsd/j2ee_web_services_1_1.xsd"
    version="1.1">
    <webservice-description>
        <webservice-description-name>TellerService</webservice-description-name>
        <wsdl-file>META-INF/wsdl/teller.wsdl</wsdl-file>
        <jaxrpc-mapping-file>META-INF/mapping.xml</jaxrpc-mapping-file>
        <port-component>
            <port-component-name>Teller</port-component-name>
            <wsdl-port>TellerEndpoint</wsdl-port>
            <service-endpoint-interface>
                com.jboss.ebank.TellerEndpoint
            </service-endpoint-interface>
            <service-impl-bean>
                <ejb-link>TellerBean</ejb-link>
            </service-impl-bean>
        </port-component>
    </webservice-description>
</webservices>
```

Our web service is a simple session bean, so deploying it only requires us to package up the bean and the associated deployment descriptors into an EJB JAR file. The `package-ws` task accomplishes this, and the `deploy-ws` target deploys the EJB JAR to JBoss.

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

Once the service is deployed you can view the WSDL (Web Service Description Language) for it by browsing to the URL http://localhost:8080/bankws-ejb/TellerService?wsdl. In this example we generate the WSDL, but it would also have been possible to write the WSDL for the service by hand and then generate a Java endpoint interface for it using `wsdl2java`, which is also provided with JBossWS.

5.3. Running the Web Service Client

We've also supplied a Java client which accesses the web service from a non-J2EE environment.

```
public class WSClient {
    public static void main(String[] args)
        throws Exception
    {
        URL url =
            new URL("http://localhost:8080/ws4ee/services/TellerService?wsdl");

        QName qname = new QName("http://ebank.jboss.com",
                                "TellerService");

        ServiceFactory factory = ServiceFactory.newInstance();
        Service        service = factory.createService(url, qname);

        TellerEndpoint endpoint = (TellerEndpoint)
            service.getPort(TellerEndpoint.class);

        String   customer = "200";
        String[] ids      = endpoint.getAccountsOfCustomer(customer);

        System.out.println("Customer: " + customer);
        for (int i=0; i<ids.length; i++) {
            System.out.println("account[" + ids[i] +  "]  " +
                               endpoint.getAccountBalance(ids[i]));
        }
    }
}
```

The client can be run using the `run-ws` target.

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

The client will display the balance for each account belonging to the given customer.

```
     [java] Customer: 200
     [java] account[5005]  3300.00
     [java] account[5006]  2458.32
     [java] account[5007]  220.03
     [java] account[5008]  59601.35
```

5.4. Network Traffic Analysis

JBoss comes with a very useful network monitoring utilities to help monitoring your web service traffic. The TCP monitor 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. The `tcpmon` target will launch the tool.

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

When `tcpmon` starts, it will present an initial configuration window. You just need to specify 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.

To route your web service traffic through the TCP monitor, you'll need to change the port number used to look-up the WSDL file in the client. You'll also need to change the port number in the WSDL file that gets returned. If you recall, JBoss overwrites the `wsdlsoap:address` using the correct service name, host and port. To get JBoss to use port 7070 instead, you'll need to change the `WebServicePort` in `jboss-ws4ee.sar/META-INF/jboss-service.xml` to 7070 as shown below.

```
<mbean code="org.jboss.webservice.AxisService"
       name="jboss.ws4ee:service=AxisService">
    <depends>jboss:service=WebService</depends>
    <attribute name="WebServiceHost">localhost</attribute>
    <attribute name="WebServiceSecurePort">8443</attribute>
    <attribute name="WebServicePort">7070</attribute>
    <attribute name="ValidateWsdlRequest">false</attribute>
</mbean>
```

If everything is configured correctly, you can then run the client and view the output.

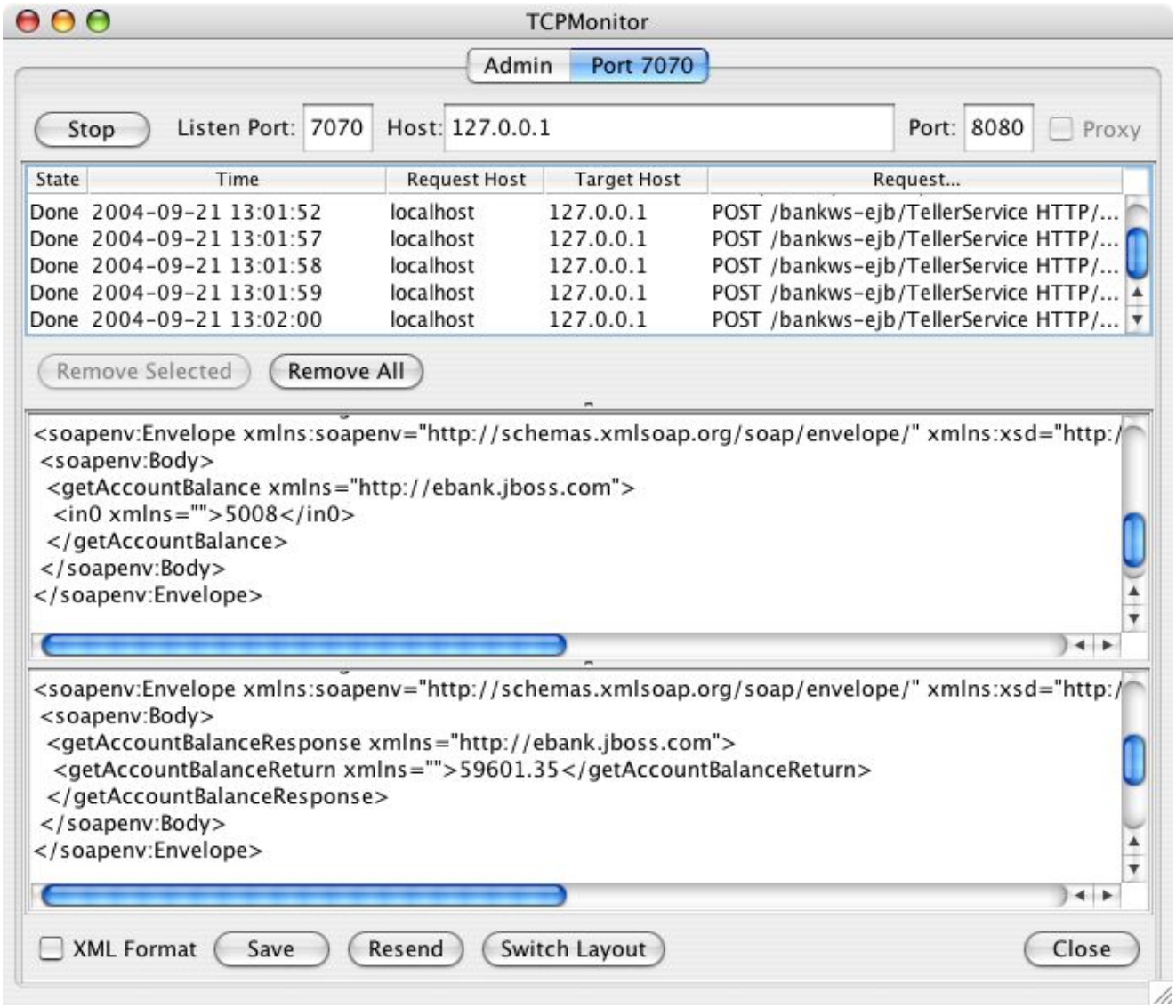

Figure 5.1. TCPMon output of Web Services Call

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

Chapter 6. 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 (MDBs) 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 `j2eetutorial14/examples/ejb/simplemessage`. We've supplied a `jboss-build.xml` file in the `simplemessage` directory which will allow you to build the example from scratch and run it in JBoss.

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.

6.1. Building the Example

6.1.1. Compiling and Packaging the MDB and Client

To compile the files, invoke the `compile-mdb` target from the `simplemessage` directory.

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

Then run the following targets to produce archives for the bean and the client and a combined EAR file in the `jar` directory.

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

We've retained the same layout we used in the Duke's Bank build, with a `dd` directory containing the deployment descriptors and the `jar` directory containing the archives produced by the build.

6.1.1.1. 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`.

6.2. Deploying and Running the Example

To deploy the MDB, just copy the `SimpleMessage.ear` file to the JBoss deploy directory. The `deploy-mdb` tar-

get does this.

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

A successful deployment should look something like this:

```
15:42:02,437 INFO  [EARDeployer] Init J2EE application: file:/Users/orb/java/jboss-
4.0.0/server/default/deploy/SimpleMessage.ear
15:42:03,274 INFO  [EjbModule] Deploying SimpleMessageEJB
15:42:04,857 WARN  [JMSContainerInvoker] Could not find the queue destination-jndi-
name=queue/MyQueue
15:42:04,951 WARN  [JMSContainerInvoker] destination not found: queue/MyQueue reaso
n: javax.naming.NameNotFoundException: MyQueue not bound
15:42:04,955 WARN  [JMSContainerInvoker] creating a new temporary destination: queu
e/MyQueue
15:42:05,204 INFO  [MyQueue] Bound to JNDI name: queue/MyQueue
15:42:06,216 INFO  [EJBDeployer] Deployed: file:/Users/orb/java/jboss-4.0.0/server/
default/tmp/deploy/tmp18923SimpleMessage.ear-contents/simplemessage.jar
15:42:06,909 INFO  [EARDeployer] Started J2EE application: file:/Users/orb/java/jbo
ss-4.0.0/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.

6.2.1. Running the Client

Run the client with the `run-mdb` Ant target.

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

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

6.3. 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. However, if you use the latter method, they won't survive a server restart.

6.3.1. 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 the queue for our example, you would 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>
```

6.3.2. 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` for these respectively. 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. 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.

6.3.3. Administering Destinations

You can access the attributes and operations that the MBeans representing a queue or topic 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 exercise, 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 will be `binding=message-driven-bean, jndiName=local/SimpleMessageEJB, plugin=invoker,service=EJB`

You can start and stop the delivery of messages using the corresponding MBean operations which it supports. Invoke the `stopDelivery()` method, and then run the client a few times. You will see the `QueueDepth` increase as the messages accumulate. If you re-start message delivery, with the `startDelivery()` method, you should see all the messages arriving at once.

Chapter 7. Container-Managed Persistence

The Duke's Bank application 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 `j2eetutorial14/examples/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.

7.1. 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.

To compile the code, first make sure you're in the `examples` directory. Running the `compile-cmp` target will compile all the code in one go.

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

Run the following `package-team` to build the team JAR file which contains the entity beans.

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

The `package-roster` target builds the `roster` JAR.

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

Both JAR files will be created in the `jar` directory. Build the client jar using the `package-roster-client` target.

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

Finally assemble the RosterApp EAR using the `assemble-roster` target.

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

7.2. Deploying and Running the Application

Deploying the application is done with the `deploy-cmp` Ant target.

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

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.

```
17:19:13,603 INFO  [EARDeployer] Init J2EE application: file:/Users/orb/java/jboss-4.0.0/s
erver/default/deploy/RosterApp.ear
17:19:14,068 INFO  [EjbModule] Deploying RosterBean
17:19:14,320 INFO  [EjbModule] Deploying TeamBean
17:19:14,357 INFO  [EjbModule] Deploying LeagueBean
17:19:14,383 INFO  [EjbModule] Deploying PlayerBean
17:19:14,614 INFO  [EJBDeployer] Deployed: file:/Users/orb/java/jboss-4.0.0/server/default
/tmp/deploy/tmp35981RosterApp.ear-contents/roster-ejb.jar
17:19:15,334 INFO  [EJBDeployer] Deployed: file:/Users/orb/java/jboss-4.0.0/server/default
/tmp/deploy/tmp35981RosterApp.ear-contents/team-ejb.jar
17:19:15,419 INFO  [EARDeployer] Started J2EE application: file:/Users/orb/java/jboss-4.0.
0/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 RosterBean and so on. You can check these in the jmx-console as before.

The first time you deploy the application, JBoss will automatically create the required database tables. If you take a look at the database schema (as we did in Section 4.1.7.3), you will see that JBoss has created one table for each entity bean and an addition join table needed to handle the many-to-many relationship between players and teams. The table and column names correspond the names of the entity beans and their attributes. If these names are suitable, you won't need to further refine the schema. In this case we've had to manually map the `position` field on `PlayerBean` to a column named `pos` because the default column name, `position`, is a reserved token in HSQL. The schema is in the `jbosscmp-jdbc.xml` file.

Note that if you increase the logging level for the `org.jboss.ejb.plugins.cmp` package (Section 2.2.2) 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.

7.2.1. 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. 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 `db-delete` target.

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

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

7.3. CMP Customization

There are many ways you can further customize the CMP engine'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.

7.3.1. 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 learning to use the XDoclet code generation engine (http://xdoclet.sourceforge.net/). Using Javadoc-style attribute tags you place in your code, XDoclet 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, its use is thoroughly recommended (almost essential in fact) for real projects.

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 that 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 Section 2.2.4). Obviously you need a JDBC driver for this to work, and the classes have to be made available (just 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`, 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 recognized 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 `data-sources`. These use an alternative, more generic JCA configuration syntax used with a pre-packaged JCA resource adapter. As we mentioned in Section 2.2.4, 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.

Next, we'll work through some step-by-step examples to illustrate what's involved.

8.2. 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 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. 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)
```

Then check that it has been created.

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

Next, 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)
```

Again, you can 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)
```

8.2.2. 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>
```

This 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[11].

8.2.3. 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>
```

After restarting JBoss, you should be able to deploy the application and see the tables being created as we did in Section 7.2. 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 datasource.

8.3. 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

[11]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.

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[12].

8.3.1. 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>
```

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.

8.3.2. 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>
```

[12] 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.

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.

```
11:33:45,174 INFO  [WrapperDataSourceService] Bound connection factory for resource adapter
for ConnectionManager 'jboss.jca:name=XAOracleDS,service=DataSourceBinding to JNDI name
'java:XAOracleDS'
```

If you use the JNDIView service from the JMX console as before, you should see the name `java:/XAOracleDS` listed.

8.3.3. 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 (Section 4.3). 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 following.

- 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 SQ.L

```
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[13].

[13] 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.

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 and you should be able to login as before.

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

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 `/etc/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 `openssl` to hash the value.

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

You would then insert the resulting string, `glcikLhvxq1BwPBZN0EGMQ==`, into the database instead of the plaintext 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 guide, we have been using to the JMX Console web application to inspect and manage the server. However there is another management console application which offers extended functionality such as live graphs, alerts, and 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 A.1. For a complete description of the bean states (method-ready, pooled, ready etc.) see the EJB specification.

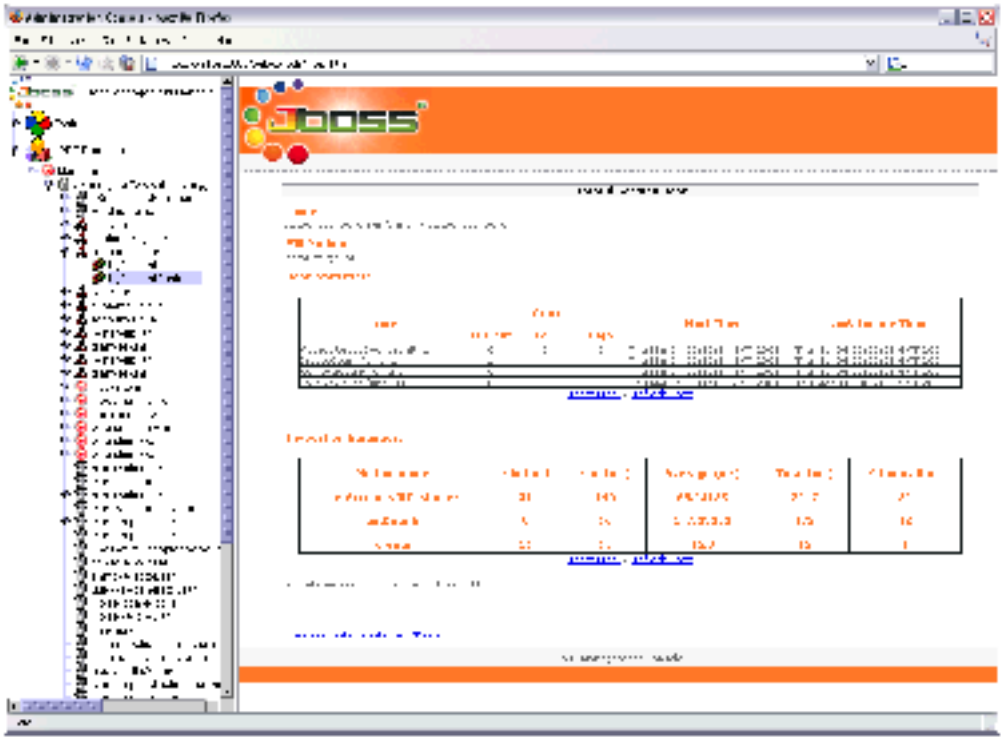

Figure A.1. Web Admin. Console Showing Stateful Session Bean Statistics.

Table A.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
RemoveCount[a]	The number of beans that have been explicitly removed.
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.

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.

Appendix B. Further Information Sources

For comprehensive JBoss documentation covering advanced JBoss topics, see *JBoss Admin. and Development Guide*, available at http://www.jboss.org/docs/index.

For information on how to run clustered JBoss servers for performance and high availability, see *JBoss Clustering* (Sacha Labourey and Bill Burke) also available at http://www.jboss.org/docs/index

For general EJB instruction, with thorough JBoss coverage, see *Enterprise JavaBeans, 4th Edition*. (O'Reilly, 2004. Richard Monson-Haeful, Bill Burke, Sacha Labourey)

For additional, but dated, EJB instruction, we also recommend the classic *Mastering Enterprise JavaBeans, Second Edition*. (Wiley, 2001. Ed. Roman et al.) A free PDF version of the first edition is available online at http://www.theserverside.com/books/masteringEJB/index.jsp.

For more information about using XDoclet to simplify J2EE development, see *XDoclet in Action*. (Manning, 2003. Craig Walls, Norman Richards)