JBoss Application Server

4.2

# Getting Started Guide

# Authors

**ISBN:**

**Publication date: Nov, 2007**

# JBoss Application Server: Getting Started Guide: Authors

by JBoss Community Documentation Project

### Introduction

JBoss Application Server is easy to install and you can have it running in a few easy steps. Refer to the *JBoss Application Server: Installation Guide* for information on pre-requisites for installation and the detailed installation steps.

Once you have JBoss Application Server installed, use this guide to familiarize yourself with its layout and the example applications that demonstrate application development and deployment.

# 1. Software Versions

| Software | Description |
| --- | --- |
| JBoss Application Server | 4.2.2 |

**Table 1. Software Versions**

# 2. Help Contribute

If you find a typographical error in the *Getting Started Guide*, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in JIRA: *http://jira.jboss.com* [http://jira.jboss.com/jira/secure/IssueNavigator.jspa?reset=true&&pid=10030&resolution=-1&component=12311310&sorter/field=priority&sorter/order=DESC] against the project *JBoss Application Server* and component *Getting_Started_Guide*.

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

**Note**

Be sure to give us your name so you can receive full credit.

**Note**

This content is taken from svn.jboss.org/repos/jbossas/projects/docs/trunk and has yet to be branched.

To access the content directly and make changes yourself:

```
 svn co https://svn.jboss.org/repos/jbossas/projects/docs/trunk
--username yourname
```

# The JBoss Server - A Quick Tour

## 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.

### 1.1. 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 do not need to do this to be able to run standard J2EE applications though.

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.

The JBoss Application Server ships with four different server configurations. Within the `JBOSS_DIST/jboss-as/server` directory, you will find four subdirectories: `minimal`, `default`, `production` and `all` - one for each server configuration. Each of these configurations provide a different set of services. The `default` configuration is the one used if you don't specify another one when starting up the server.

minimal
: has a minimal configuration—the bare minimum services 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. This is not a J2EE 1.4 compatible configuration.

default
: is a base J2EE 1.4 server profile containing a default set of services. It has the most frequently used services required to deploy a J2EE application. It does not include the JAXR service, the IIOP service, or any of the clustering services.

all
: The all configuration starts all the available services. This includes the RMI/IIOP and clustering services, which aren't loaded in the default configuration.

If you want to know which services are configured in each of these instances, look at the `jboss-service.xml` file in the `JBOSS_DIST/jboss-as/server/<instance-name>/conf/` directory and also the configuration files in the `JBOSS_DIST/jboss-as/server/<instance-name>/deploy` directory.

```
[vsr]$ls server/default/conf
jbossjta-properties.xml  jndi.properties
 standardjbosscmp-jdbc.xml
jboss-log4j.xml          login-config.xml  standardjboss.xml
jboss-minimal.xml        props             xmdesc
jboss-service.xml        standardjaws.xml
```

**Note**

The **default** configuration is the one used if you don't specify another one when starting up the server.

To start the server using an alternate configuration refer to *Section 1.2.2, "Start the Server With Alternate Configuration"*.

### 1.1.1. Server Configuration Directory Structure

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 server configuration. It's where the log output goes, and it's where you deploy your applications. *Table 1.1, "Server Configuration Directory Structure"* shows the directories inside the server configuration directory (`JBOSS_DIST/jboss-as/server/<instance-name>`) and their functions.

| Directory | Description |
| --- | --- |
| `conf` | The `conf` directory contains the `jboss-service.xml` bootstrap descriptor file for a given server configuration. This defines the core services that are fixed for the lifetime of the server. |
| `data` | The `data` directory is available for use by services that want to store content in the file system. It holds persistent data for services intended to survive a server restart. Serveral JBoss services, such as the embedded Hypersonic database instance, store data here. |
| `deploy` | The `deploy` directory contains the hot-deployable services (those which can be added to or removed from the running server). It also contains applications for the current server configuration. You deploy your application code by placing application packages (JAR, WAR and EAR files) in the `deploy` directory. The directory is constantly scanned for updates, and any modified components will be re-deployed automatically. This may be overridden through the URLDeploymentScanner URLs attribute. |
| `lib` | This directory contains JAR files (Java libraries that should not be hot deployed) needed by this server configuration. You can add required library files here for JDBC drivers etc. All JARs in this directory are loaded into the shared classpath at startup. |
| `log` | This is where the log files are written. JBoss uses the Jakarta `log4j` package for logging and you can also use it directly in your own applications from within the server. This may be overridden through the `conf/jboss-log4j.xml` configuration file. |
| `tmp` | The `tmp` directory is used for temporary storage by JBoss services. The deployer, for example, expands application archives in this directory. |
| `work` | This directory is used by Tomcat for compilation of JSPs. |

**Table 1.1. Server Configuration Directory Structure**

### 1.1.2. The "default" Server Configuration File Set

The "`default`" server configuration file set is located in the `JBOSS_DIST/jboss-as/server/default` directory. Let's take a look at the contents of the `default` server configuration file set:

```
jboss-eap-4.2                 // jboss.home_url
 |+ doc
 |+ jboss-as
  |+ bin
  |+ client
  |+ docs
```

```
   |+ icons
   |+ lib                          // jboss.lib.url
   |+ scripts
   |+ server
    |+ all               // jboss.server.name
    |+ default           // jboss.server.home.url
     |+ conf      // jboss.server.config.url
      |+ props
      |+ xmdesc
      -  jbossjta-properties.xml
      -  jboss-minimal.xml
      -  jndi.properties
      -  standardjboss.xml
      -  jboss-log4j.xml
      -  jboss-service.xml
      -  login-config.xml
      -  standardjbosscmp-jdbc.xml
     |+ deploy
      |+ ejb3.deployer
      |+ http-invoker.sar
      |+ jboss-aop-jdk50.deployer
      |+ jboss-bean.deployer
      |+ jboss-web.deployer
      |+ jbossws.sar
      |+ jms
      |+ jmx-console.war
      |+ management
      |+ uuid-key-generator.sar
      -  bsh-deployer.xml
      -  cache-invalidation-service.xml
      -  client-deployer-service.xml
      -  ear-deployer.xml
      -  ejb3-interceptors-aop.xml
      -  ejb-deployer.xml
      -  hsqldb-ds.xml
      -  jboss-ha-local-jdbc.rar
      -  jboss-ha-xa-jdbc.rar
      -  jbossjca-service.xml
      -  jboss-local-jdbc.rar
      -  jboss-xa-jdbc.rar
      -  jmx-invoker-service.xml
      -  jsr88-service.xml
      -  mail-service.xml
      -  monitoring-service.xml
      -  properties-service.xml
      -  quartz-ra.rar
      -  schedule-manager-service.xml
      -  scheduler-service.xml
      -  sqlexception-service.xml
```

```
     |+ lib            // jboss.server.lib.url
    |+ minimal
    |+ production
  |+ seam
  |+ Uninstaller                          // jboss.uninstaller.url
```

#### 1.1.2.1. Contents of "conf" directory

The files in the `conf` directory are explained in the following table.

| File | Description |
| --- | --- |
| `jboss-minimal.xml` | This is a minimalist example of the `jboss-service.xml` configuration file. (This is the `jboss-service.xml` file used in the `minimal` configuration file set) |
| `jboss-service.xml` | `jboss-service.xml` defines the core services and their configurations. |
| `jndi.properties` | The `jndi.properties` file specifies the JNDI `InitialContext` properties that are used within the JBoss server when an `InitialContext` is created using the no-arg constructor. |
| `jboss-log4j.xml` | This file configures the Apache log4j framework category priorities and appenders used by the JBoss server code. |
| `login-config.xml` | This file contains sample server side authentication configurations that are applicable when using JAAS based security. |
| `props/*` | The `props` directory contains the users and roles property files for the `jmx-console`. |
| `standardjaws.xml` | This file provides the default configuration for the legacy EJB 1.1 CMP engine. |
| `standardjboss.xml` | This file provides the default container configurations. |
| `standardjbosscmp-jdbc.xml` | This file provides a default configuration file for the JBoss CMP engine. |
| `xmdesc/*-mbean.xml` | The `xmdesc` directory contains XMBean descriptors for several services configured in the `jboss-service.xml` file. |

**Table 1.2. Contents of "conf" directory**

### 1.1.2.2. Contents of "deploy" directory

The files in the `deploy` directory are explained in the following table.

| File | Description |
| --- | --- |
| `bsh-deployer.xml` | This file configures the bean shell deployer, which deploys bean shell scripts as JBoss services. |
| `cache-invalidation-service.xml` | This is a service that allows for custom invalidation of the EJB caches via JMS notifications. It is disabled by default. |
| `client-deployer-service.xml` | This is a service that provides support for J2EE application clients. It manages the `java:comp/env` enterprise naming context for client applications based on the `application-client.xml` descriptor. |
| `ear-deployer.xml` | The EAR deployer is the service responsible for deploying J2EE EAR files. |
| `ejb-deployer.xml` | The EJB deployer is the service responsible for deploying J2EE EJB JAR files. |
| `hsqldb-ds.xml` | `hsqldb-ds.xml` configures the Hypersonic embedded database service configuration file. It sets up the embedded database and related connection factories. |
| `http-invoker.sar` | `http-invoker.sar` contains the detached invoker that supports RMI over HTTP. It also contains the proxy bindings for accessing JNDI over HTTP. |
| `jboss-aop-jdk50.deployer` | This service configures the `AspectManagerService` and deploys JBoss AOP applications. |
| `jboss-bean.deployer` | `jboss-bean.deployer` provides the JBoss microcontainer, which deploys POJO services wrapped in `.beans` files. |
| `jboss-ha-local-jdbc.rar` | `jboss-ha-local-jdbc.rar` is an experimental version of `jboss-local-jdbc.rar` that supports datasource failover. |
| `jboss-ha-xa-jdbc.rar` | `jboss-ha-xa-jdbc.rar` is an experimental version of `jboss-xa-jdbc.rar` that supports datasource failover. |
| `jboss-local-jdbc.rar` | `jboss-local-jdbc.rar` is a JCA resource adaptor that implements the JCA `ManagedConnectionFactory` interface for JDBC drivers that support the `DataSource` interface but not JCA. |

**Table 1.3. Contents of "deploy" directory**

### 1.1.3. The "all" Server Configuration File Set

The "all" server configuration file set is located in the `JBOSS_DIST/jboss-as/server/all` directory. In addition to the services in the "default" set, the all configuration contains several other services in the `conf/` directory as shown below.

| File | Description |
|---|---|
| `cluster-service.xml` | This service configures clustering communication for most clustered services in JBoss. |
| `deploy-hasingleton-service.xml` | This provides the HA singleton service, allowing JBoss to manage services that must be active on only one node of a cluster. |
| `deploy.last/farm-service.xml` | `farm-service.xml` provides the farm service, which allows for cluster-wide deployment and undeployment of services. |
| `httpha-invoker.sar` | This service provides HTTP tunneling support for clustered environments. |
| `iiop-service.xml` | This provides IIOP invocation support. |
| `juddi-service.sar` | This service provides UDDI lookup services. |
| `snmp-adaptor.sar` | This is a JMX to SNMP adaptor. It allows for the mapping of JMX notifications onto SNMP traps. |
| `tc5-cluster.sar` | Provides AOP support for field-level HTTP session replication. |

**Table 1.4. Additional Services in "conf" directory for "all" configuration**

### 1.1.4. EJB3 Services

The following table explains the files providing ejb3 services.

| File | Description |
| --- | --- |
| `ejb3-interceptors-aop.xml` | This service provides the AOP interceptor stack configurations for EJB3 bean types. |
| `ejb3.deployer` | This service deploys EJB3 applications into JBoss. |
| `jboss-aop-jdk50.deployer` | This is a Java 5 version of the AOP deployer. The AOP deployer configures the `AspectManagerService` and deploys JBoss AOP applications. |
| `jbossws.sar` | This provides Java EE 5 web services support. |

**Table 1.5. EJB3 Services**

Finally, in the EJB3 "all" configuration there are two additional services.

| File | Description |
| --- | --- |
| `ejb3-clustered-sfsbcache-service.xml` | This provides replication and failover for EJB3 stateful session beans. |
| `ejb3-entity-cache-service.xml` | This provides a clustered cache for EJB3 entity beans. |

**Table 1.6. Additional Services in EJB3 "all" Configuration**

### 1.1.5. Adding Your Own 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 `production` directory, renaming it as `myconfig`, remove the `jms` subdirectory and then start JBoss with the new configuration.

```
./run.sh -c myconfig
```

## 1.2. Starting and Stopping the Server

### 1.2.1. Start the Server

Move to `JBOSS_DIST/jboss-as/bin` directory and execute the `run.bat` (for Windows) or `run.sh` (for Linux) script, as appropriate for your operating system. Your output should look like the following (accounting for installation directory differences) and contain no error or exception messages:

```
[user@mypc bin]$ ./run.sh
=========================================================================

  JBoss Bootstrap Environment

  JBOSS_HOME: /home/user/jboss-as-4.2.2/jboss-as

  JAVA: java

  JAVA_OPTS: -Dprogram.name=run.sh -server -Xms1503m -Xmx1503m
 -Dsun.rmi.dgc.client.
gcInterval=3600000 -Dsun.rmi.dgc.server.gcInterval=3600000
 -Djava.net.preferIPv4Stack=true

  CLASSPATH: /home/user/jboss-as-4.2.2/jboss-as/bin/run.jar

=========================================================================
```

**Note**

Note that there is no "Server Started" message shown at the console when the server is started using the `production` profile, which is the default profile used when no other is specified. This message may be observed in the `server.log` file located in the `server/production/log` subdirectory.

### 1.2.2. Start the Server With Alternate Configuration

Using `run.sh` without any arguments starts the server using the `default` server configuration file set. To start with an alternate configuration file set, pass the name of the server configuration file set [same as the name of the server configuration directory under `JBOSS_DIST/jboss-as/server`] that you want to use, as the value to the `-c` command line option. For example, to start with the `minimal` configuration file set you should specify:

```
[bin]$ ./run.sh -c minimal
...
...
...
15:05:40,301 INFO  [Server] JBoss (MX MicroKernel) [4.2.2 (build:
 SVNTag=JBoss_4_2_2 date=200711111042)] Started in 5s:75ms
```

### 1.2.3. Using run.sh

The `run` script supports the following options:

```
usage: run.sh [options]
-h, --help                          Show help message
-V, --version                       Show version information
--                                  Stop processing options
-D<name>[=<value>]      Set a system property
-d, --bootdir=<dir>           Set the boot patch directory; Must be
 absolute or url
-p, --patchdir=<dir>          Set the patch directory; Must be
 absolute or url
-n, --netboot=<url>           Boot from net with the given url as
 base
-c, --configuration=<name>    Set the server configuration name
-B, --bootlib=<filename>      Add an extra library to the front
 bootclasspath
-L, --library=<filename>      Add an extra library to the loaders
 classpath
-C, --classpath=<url>         Add an extra url to the loaders
 classpath
-P, --properties=<url>        Load system properties from the given
 url
-b, --host=<host or ip>       Bind address for all JBoss services
-g, --partition=<name>        HA Partition name
 (default=DefaultDomain)
-u, --udp=<ip>                UDP multicast address
-l, --log=<log4j|jdk>         Specify the logger plugin type
```

### 1.2.4. Stopping the Server

To shutdown the server, you simply issue a Ctrl-C sequence in the console in which JBoss was started. Alternatively, you can use the `shutdown.sh` command.

```
[bin]$ ./shutdown.sh -S
```

The `shutdown` script supports the following options:

```
A JMX client to shutdown (exit or halt) a remote JBoss server.

usage: shutdown [options] <operation>

options:
-h, --help                Show this help message (default)
-D<name>[=<value>]        Set a system property
--                        Stop processing options
-s, --server=<url>        Specify the JNDI URL of the remote server
-n, --serverName=<url>    Specify the JMX name of the ServerImpl
-a, --adapter=<name>      Specify JNDI name of the
 MBeanServerConnection to use
```

```
-u, --user=<name>          Specify the username for authentication
-p, --password=<name>      Specify the password for authentication

operations:
-S, --shutdown             Shutdown the server
-e, --exit=<code>          Force the VM to exit with a status code
-H, --halt=<code>          Force the VM to halt with a status code
```

Using the shutdown command requires a server configuration that contains the `jmx-invoker-service.xml` service. Hence you cannot use the shutdown command with the `minimal` configuration.

### 1.2.5. Running as a Service under Microsoft Windows

You can configure the server to run as a service under Microsoft Windows, and configure it to start automatically if desired.

Download the `JavaService` package from *http://forge.objectweb.org/projects/javaservice/*.

Unzip the package and use the `JBossInstall.bat` file to install the JBoss service. You must set the `JAVA_HOME` and `JBOSS_HOME` environment variables to point to the `jdk` and `jboss-as` directories before running `JBossInstall.bat`. Run `JBossInstall.bat` with the following syntax:

```
JBossInstall.bat <depends> [-auto | -manual]
```

Where <*depends*> is the name of any service that the JBoss AS server depends on, such as the `mysql` database service.

Once the service is installed the server can be started by using the command `net start JBoss`, and stopped with the command `net stop JBoss`.

Please refer to the documentation included in the `JavaService` package for further information.

## 2. The JMX Console

When the JBoss Server is running, you can get a live view of the server by going to the JMX console application at *http://localhost:8080/jmx-console*. You should see something similar to *Figure 1.1, “View of the JMX Management Console Web Application”*.

The JMX Console is the JBoss Management Console which provides a raw view of the JMX MBeans which make up the server. 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` button. 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 know why you can’t resolve a particular EJB name.

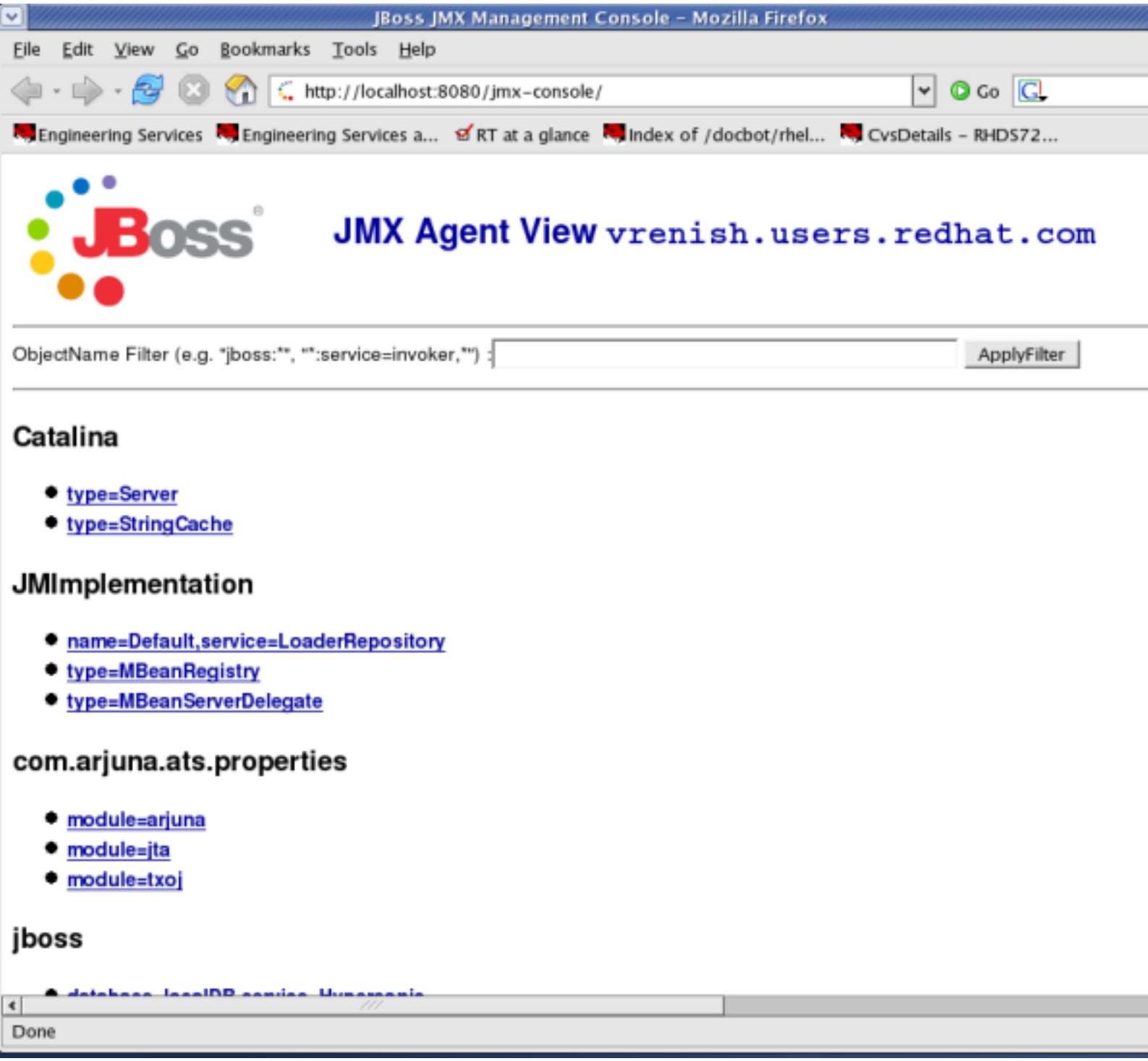


**Figure 1.1. View of the JMX Management Console Web Application**

Look at some of the other MBeans and their listed operations; try changing some of the configuration attributes and see what happens. With a very 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 without doing any permanent damage.

**Note**

If you installed JBoss using the graphical installer, the JMX Console will prompt you for a username and password before you can access it. If you installed using other modes, you can still configure JMX Security manually. We will show you how to secure your console in *Section 4.3, "Security Service"*.

# 3. Hot-deployment of services in JBoss

Hot-deployable services are those which can be added to or removed from the running server. These are placed in the `JBOSS_DIST/jboss-as/server/<instance-name>/deploy` directory. Let's have a look at a practical example of hot-deployment of services in JBoss 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 at the `server/production/deploy` 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 JBoss re-install the service:

```
13:58:54,331 INFO  [MailService] Mail Service bound to java:/Mail
```

This is hot-deployment in action.

# 4. Basic Configuration Issues

Now that we have examined the JBoss server, we will take a look at some of the main configuration files and what they are used for. All paths are relative to the server configuration directory (`server/production`, for example).

## 4.1. 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 will see MBeans for various services including logging, security, JNDI, JNDIView etc. Try commenting out the entry for the `JNDIView` service.

Note that because the mbeans definition had nested comments, we had to comment out the mbean in two sections, leaving the original comment as it was.

```
<!-- Section 1 commented out
<mbean code="org.jboss.naming.JNDIView"
    name="jboss:service=JNDIView"
    xmbean-dd="resource:xmdesc/JNDIView-xmbean.xml">
-->
    <!-- The HANamingService service name -->
<!-- Section two commented out
    <attribute
 name="HANamingService">jboss:service=HAJNDI</attribute></mbean>
-->
```

If you then restart JBoss, you will see that the `JNDIView` service no longer appears in the JMX Management Console (JMX 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.

## 4.2. Logging Service

In JBoss `log4j` is used for logging. If you are 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/jboss-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 (`log/server.log`).

There are 5 basic log levels used: `DEBUG`, `INFO`, `WARN`, `ERROR` and `FATAL`. 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 `server.log` file to see if there are any debug messages which might help you to 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 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 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, `cmp.log`, in order to analyze the generated SQL commands. You would add the following code to the `conf/jboss-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`.

The file appender is set up to produce a new log file every day rather than producing a new one every time you restart the server or writing to a single file indefinitely. The current log file is `cmp.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.

## 4.3. Security Service

The security domain information is stored in the file `conf/login-config.xml` as a list of named security domains, each of which specifies a number of JAAS [1] 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` (used in defining jboss specific configurations for an enterprise application) and/or `jboss-web.xml` (used in defining jboss for a Web application. We'll quickly look at how to do this to secure the JMX Console application that ship with JBoss.

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. [2] This can be done in the `jboss-web.xml` file for the JMX Console, which can be found in `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-constraint` 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>
```

[1] 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 you have more specific requirements.

[2] If you installed JBoss using the Graphical Installer and set the JMX Security up, then you will not have to uncomment the sections, because they are already uncommented. Additionally, the admin password will be set up to whatever you had specified.

```
    <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 have 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">
                props/jmx-console-users.properties
            </module-option>
            <module-option name="rolesProperties">
                props/jmx-console-roles.properties
            </module-option>
        </login-module>
    </authentication>
</application-policy>
```

This configuration uses a simple file based security policy. The configuration files are found in the `conf/props` directory of your server configuration. The usernames and passwords are stored in the `conf/props/jmx-console-users.properties` file and take the form "`username=password`". To assign a user to the `JBossAdmin` group add "`username=JBossAdmin`" to the `jmx-console-roles.properties` file (additional roles on that username can be added comma separated). The existing file creates an `admin` user with the password `admin`. For security, please either remove the user or change the password to a stronger one.

JBoss will re-deploy the JMX Console whenever you update its `web.xml`. You can check the server console to verify that JBoss has seen your changes. If you have

configured everything correctly and re-deployed the application, the next time you try to access the JMX Console, it will ask you for a name and password. [3]

The JMX Console isn't the only web based management interface to JBoss. There is also the Web Console. 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 the file `deploy/management/console-mgr.sar/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.

## 4.4. 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 into a single archive.

Detailed information on all these services can be found in the *JBoss Application Server: Administration and Configuration Guide*, which also provides comprehensive information on server internals and the implementation of services such as JTA and the J2EE Connector Architecture (JCA).

# 5. The Web Container - Tomcat

JBoss Application Server comes with Tomcat as the default web container. The embedded Tomcat service is the expanded `deploy/jboss-web.deployer`. All the necessary jar files needed by Tomcat can be found in there, as well as a `web.xml` (under the ROOT.war/WEB-INF) 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

[3] Since the username and password are session variables in the web browser you may need to shut down your browser and come back in to see the login dialog come back up.

them in the JBoss `deploy` directory and logging output from Tomcat can be found in the JBoss `log` directory.

# EJB3 Caveats in JBoss Application Server 4.2.2

There are a number of implementation features that you should be aware of when developing applications for JBoss Application Server 4.2.2.

## 1. Unimplemented features

The Release Notes for JBoss Application Server 4.2.2 contain information on EJB3 features that are not yet implemented, or partially implemented. The Release Notes include links to issues in JIRA for information on workarounds and further details.

## 2. Referencing EJB3 Session Beans from non-EJB3 Beans

JBoss Application Server 5 will fully support the entire Java 5 Enterprise Edition specification. In the meantime JBoss Application Server 4.2.2 implements EJB3 functionality by way of an EJB MBean container running as a plugin in the JBoss Application Server. This has certain implications for application development.

The EJB3 plugin injects references to an EntityManager and @EJB references from one EJB object to another. However this support is limited to the EJB3 MBean and the JAR files it manages. Any JAR files which are loaded from a WAR (such as Servlets, JSF backing beans, and so forth) do not undergo this processing. The Java 5 Enterprise Edition standard specifies that a Servlet can reference a Session Bean through an @EJB annotated reference, however this is not implemented in JBoss Application Server 4.2.2.

In order to access an EJB3 Session Bean from a Servlet or JSF Backing Bean you will need to do one of two things:

1. **Without Seam - JNDI Lookup .** Without utilizing the Seam framework that is part of JBoss Application Server you will need to use an explicit JNDI lookup to access the EJB3 Session Bean. You can see an example of this being done in the `TodoBean.java` file in the `jsfejb3` example application, described in *Chapter 4, Sample JSF-EJB3 Application*.

```
private TodoDaoInt getDao () {
 try {
 InitialContext ctx = new InitialContext();
 return (TodoDaoInt) ctx.lookup("jsfejb3/TodoDao/local");
 } catch (Exception e) {
 e.printStackTrace();
```

```
  throw new RuntimeException("couldn't lookup Dao", e);
  }
  }
```

`ctx.lookup("jsfejb3/TodoDao/local");` is the method used to reference the EJB3 Session Bean. The form is: *`AppName`*/*`SessionBeanName`*/local.

2. **With Seam - Leave it to the Seam Framework .** When you are using the Seam Framework you don't need to worry about this. Because the Seam framework manages the interaction of Beans anyway, it already automates this type of interaction.

   Refer to *Chapter 5, Using Seam* for a more detailed explanation of achieving this using the Seam framework.

# About the Example Applications

In this guide, we make use of a simple web application to show the use of JSF-EJB3 components. We then illustrate how to use Seam to integrate the JSF and EJB3 components. The example applications (source code) come with this guide and you can find them located in the `JBOSS_DIST/doc/examples` directory. You can also download the sample applications from *http://www.redhat.com/docs/manuals/jboss*. We use two examples in this book:

- A simple "TODO" application to create, view and edit tasks - implemented using JSF and EJB3;
- The same application using the SEAM framework.

If you installed the documentation on your hard drive, then the first example can be found in the `JBOSS_DIST/doc/examples/jsfejb3` directory (if you download the examples the path is: `gettingstarted/jsfejb3`). We will see how to build this example using the `build.xml` file present here and also how to deploy the application. We will also cover in detail the workings of the .java, .xml and .properties files.

The second example used in this guide can be found in the `JBOSS_DIST/doc/examples/seamejb3` directory. Using a simple "TODO" application we will illustrate how Seam ties together the database, the web interface and the EJB3 business logic in a web application. We will use the `build.xml` file present here to compile and build our Seam application.

Within the `JBOSS_DIST/doc/examples/<seamejb3|jsfejb3>` directory, you will find the following sub-directories:

- **src**: contains the Java source code files.
- **view**: contains the web pages.
- **resources**: contains all the configuration files used.

## 1. Install Ant

To compile and package the examples, you must have Apache Ant 1.6+ installed in your machine. You can download it from *http://ant.apache.org* and have it installed in few steps:

- Unzip the downloaded file to the directory of your choice.

- Create an environment variable called `ANT_HOME` pointing to the Ant installation directory. You can do this by adding the following line to your `.bashrc` file (substituting with the actual location of the ant directory on your system):

```
export ANT_HOME=/home/user/apache-ant-1.7.0
```

  On Windows you do this by opening the Control Panel from the Start Menu, switching it to classic view if necessary, then opening System/Advanced/Environment Variables. Create a new variable, call it `ANT_HOME` and set it to be the ant directory.

- Add `$ANT_HOME/bin` to the system path to be able to run `ant` from the command line. You can do this by adding the following line to your `.bashrc` file:

```
export PATH=$PATH:$ANT_HOME/bin
```

  On Windows you do this by opening the Control Panel from the Start Menu, switching it to classic view if necessary, then editing the `PATH` environment variable found in System/Advanced/Environment Variables/System Variables/Path. Add a semicolon and the path to the ant `bin` directory.

- Verify your Ant installation. To do this type `ant -version` at the command prompt. Your output should look something like this:

```
Apache Ant version 1.7.0 compiled on December 13 2006
```

# Sample JSF-EJB3 Application

We use a simple "TODO" application to show how JSF and EJB3 work together in a web application. The "TODO" application works like this: You can create a new 'todo' task item using the "Create" web form. Each 'todo' item has a 'title' and a 'description'. When you submit the form, the application saves your task to a relational database. Using the application, you can view all 'todo' items, edit/delete an existing 'todo' item and update the task in the database.

The sample application comprises the following components:

- Entity objects - These objects represent the data model; the properties in the object are mapped to column values in relational database tables.
- JSF web pages - The web interface used to capture input data and display result data. The data fields on these web pages are mapped to the data model via the JSF Expression Language (EL).
- EJB3 Session Bean - This is where the functionality is implemented. We make use of a Stateless Session Bean.

## 1. Data Model

Let's take a look at the contents of the Data Model represented by the `Todo` class in the `Todo.java` file. Each instance of the `Todo` class corresponds to a row in the relational database table. The 'Todo' class has three properties: id, title and description. Each of these correspond to a column in the database table.

The 'Entity class' to 'Database Table' mapping information is specified using EJB3 Annotations in the 'Todo' class. This eliminates the need for XML configuration and makes it a lot clearer. The `@Entity` annotation defines the `Todo` class as an Entity Bean. The `@Id` and `@GeneratedValue` annotations on the `id` property indicate that the `id` column is the primary key and that the server automatically generates its value for each `Todo` object saved into the database.

```
@Entity
public class Todo implements Serializable {

  private long id;
  private String title;
  private String description;

  public Todo () {
    title ="";
    description ="";
  }
```

```
  @Id @GeneratedValue
  public long getId() { return id;}
  public void setId(long id) { this.id = id; }

  public String getTitle() { return title; }
  public void setTitle(String title) {this.title = title;}

  public String getDescription() { return description; }
  public void setDescription(String description) {
    this.description = description;
  }

}
```

## 2. JSF Web Pages

In this section we will show you how the web interface is defined using JSF pages. We will also see how the data model is mapped to the web form using JSF EL. Using the #{...} notation to reference Java objects is called **JSF EL** (JSF Expression Language). Lets take a look at the pages used in our application:

- **index.xhtml**: This page displays two options: 1. Create New Todo 2. Show all Todos. When you click on the Submit button the corresponding action is invoked.

```
<h:form>
<ul>
  <li><h:commandLink type="submit" value="Create New Todo"
 action="create"/></li>
  <li><h:commandLink type="submit" value="Show All Todos"
 action="todos"/></li>
</ul>
</h:form>
```

- **create.xhtml**: When you try to create a new task, this JSF page captures the input data. We use the `todoBean` to back the form input text fields. The #{todoBean.todo.title} symbol refers to the "title" property of the "todo" object in the "TodoBean" class. The #{todoBean.todo.description} symbol refers to the "description" property of the "todo" object in the "TodoBean" class. The #{todoBean.persist} symbol refers to the "persist" method in the "TodoBean" class. This method creates the "Todo" instance with the input data (title and description) and persists the data.

```
<h:form id="create">
<table>
```

```
   <tr>
     <td>Title:</td>
     <td>
       <h:inputText id="title" value="#{todoBean.todo.title}"
 size="15">
         <f:validateLength minimum="2"/>
       </h:inputText>
     </td>
   </tr>
   <tr>
     <td>Description:</td>
     <td>
       <h:inputTextarea id="description"
 value="#{todoBean.todo.description}">
         <f:validateLength minimum="2" maximum="250"/>
       </h:inputTextarea>
     </td>
   </tr>
</table>
<h:commandButton type="submit" id="create" value="Create"
                 action="#{todoBean.persist}"/>
</h:form>
```

*Figure 4.1, “The "Create Todo" web page ”* shows the "Create Todo" web page with the input fields mapped to the data model.

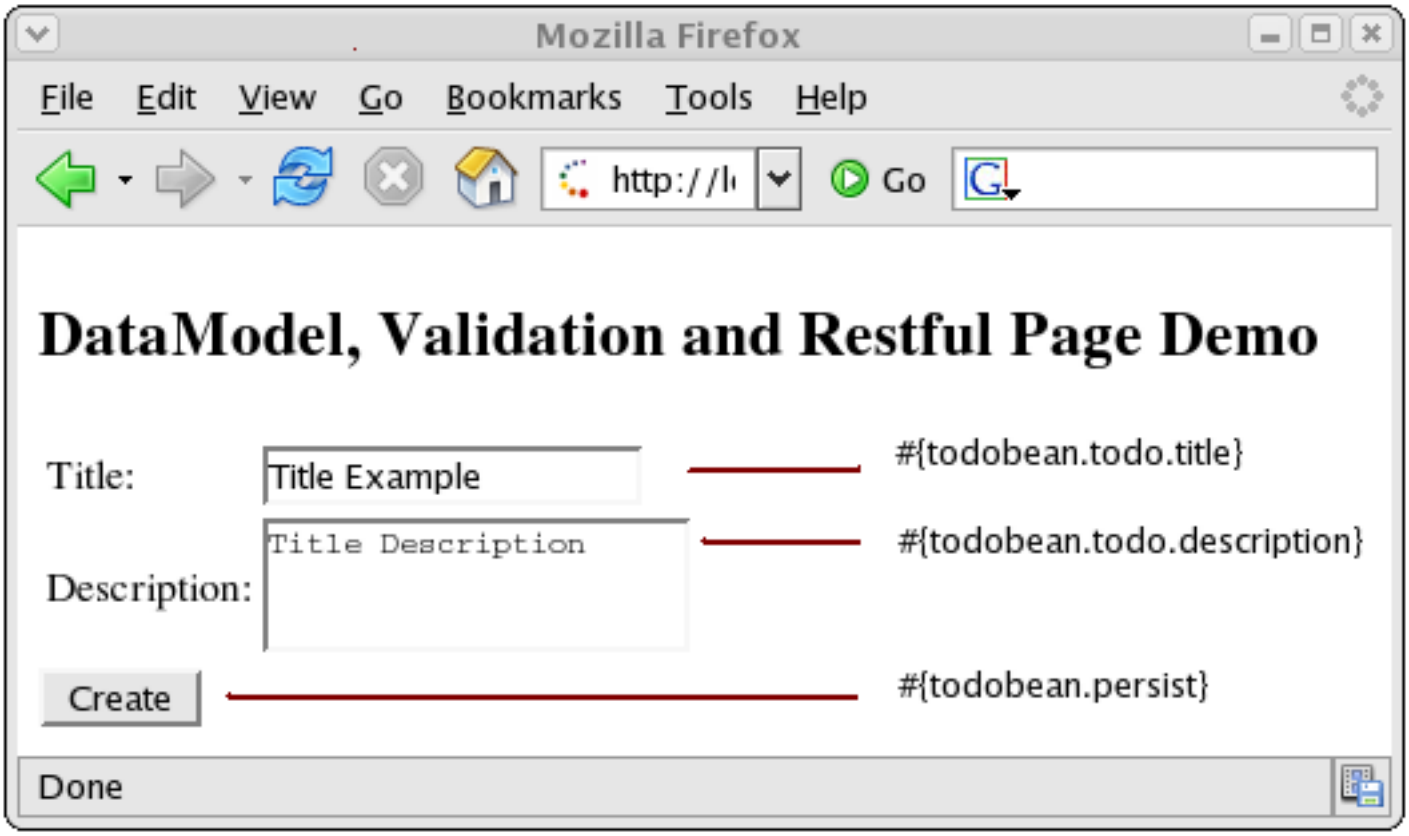


**Figure 4.1. The "Create Todo" web page**

- **todos.xhtml**: This page displays the list of all "todos" created. There is also an option to choose a "todo" item for 'edit' or 'delete'.

  The list of all 'todos' is fetched by #{todoBean.todos} symbol referring to the 'getTodos()' property in the 'TodoBean' class. The JSF `dataTable` iterates through the list and displays each `Todo` object in a row. The 'Edit' option is available across each row. The #{todo.id} symbol represents the "id" property of the "todo" object.

```
<h:form>
<h:dataTable value="#{todoBean.todos}" var="todo">
  <h:column>
    <f:facet name="header">Title</f:facet>
    #{todo.title}
  </h:column>
  <h:column>
    <f:facet name="header">Description</f:facet>
    #{todo.description}
  </h:column>
  <h:column>
    <a href="edit.faces?tid=#{todo.id}">Edit</a>
```

```
  </h:column>
</h:dataTable>
<center>
  <h:commandButton action="create"
            value="Create New Todo" type="submit"/>
</center>
</h:form>
```

*Figure 4.2, "The "Show All Todos" web page "* shows the "Show All Todos" web page with the data fields mapped to the data model.

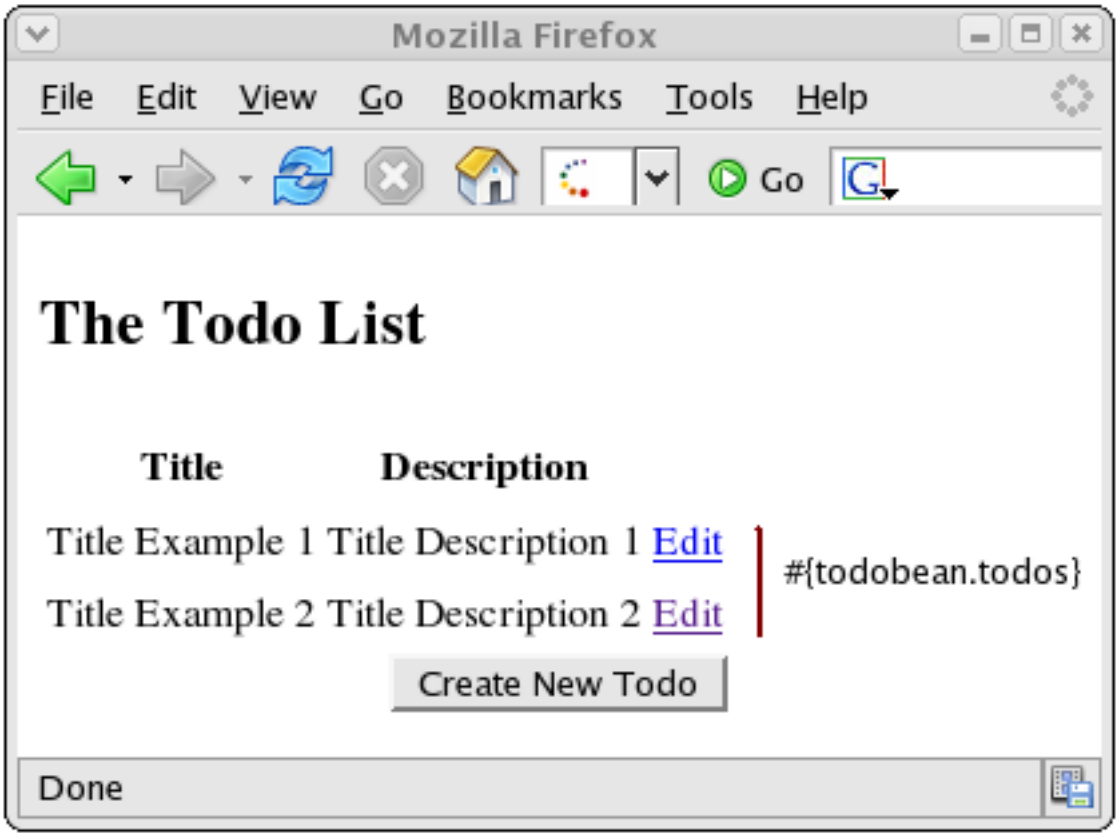


**Figure 4.2. The "Show All Todos" web page**

- **edit.xhtml**: This page allows you to edit the "todo" item's 'title' and 'description' properties. The #{todoBean.update} and #{todoBean.delete} symbols represent the "update" and "delete" methods in the "TodoBean" class.

```
<h2>Edit #{todoBean.todo.title}</h2>
<h:form id="edit">
<input type="hidden" name="tid" value="#{todoBean.todo.id}"/>
<table>
  <tr>
    <td>Title:</td>
```

```
    <td>
      <h:inputText id="title" value="#{todoBean.todo.title}"
 size="15">
        <f:validateLength minimum="2"/>
      </h:inputText>
    </td>
  </tr>
  <tr>
    <td>Description:</td>
    <td>
      <h:inputTextarea id="description"
 value="#{todoBean.todo.description}">
        <f:validateLength minimum="2" maximum="250"/>
      </h:inputTextarea>
    </td>
  </tr>
</table>
<h:commandButton type="submit" id="update" value="Update"
                 action="#{todoBean.update}"/>
<h:commandButton type="submit" id="delete" value="Delete"
                 action="#{todoBean.delete}"/>
</h:form>
```

*Figure 4.3, “The "Edit Todo" web page ”* shows the "Edit Todo" web page with the mapping to the data model.

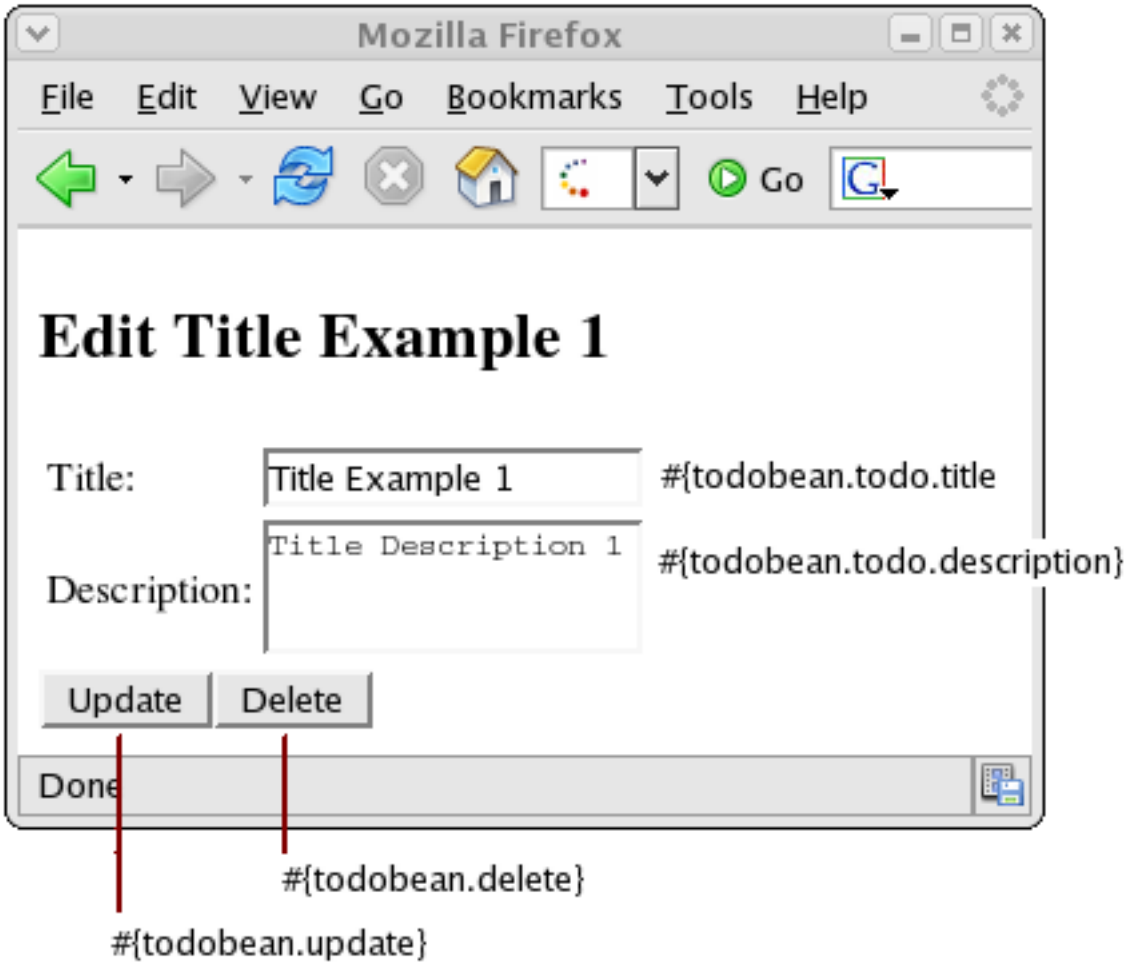


**Figure 4.3. The "Edit Todo" web page**

**Note**

We have used XHTML pages in the sample applications because we recommend using Facelets instead of JSP to render JSF view pages.

## 3. EJB3 Session Beans

EJB 3.0 is one of the major improvements introduced with Java EE 5.0. It aims at reducing the complexity of older versions of EJB and simplifies Enterprise Java development and deployment. You will notice that to declare a class as a 'Session Bean' you simply have to annotate it. Using annotations eliminates the complexity involved with too many deployment descriptors. Also the only interface an EJB3 Session Bean requires is a business interface that declares all the business methods that must be implemented by the bean.

We will explore the two important source files associated with the Bean implementation in our application: `TodoDaoInt.java` and `TodoDao.java`.

- **Business interface**: `TodoDaoInt.java`

  We define here the methods that need to be implemented by the bean implementation class. Basically, the business methods that will be used in our application are defined here.

```
public interface TodoDaoInt {

  public void persist (Todo todo);
  public void delete (Todo todo);
  public void update (Todo todo);

  public List <Todo> findTodos ();
  public Todo findTodo (String id);
}
```

- **Stateless Session Bean**: `TodoDao.java`

  The `@Stateless` annotation marks the bean as a stateless session bean. In this class, we need to access the Entity bean `Todo` defined earlier. For this we need an `EntityManager`. The `@PersistenceContext` annotation tells the JBoss Server to inject an entity manager during deployment.

```
@Stateless
public class TodoDao implements TodoDaoInt {

  @PersistenceContext
  private EntityManager em;

  public void persist (Todo todo) {
    em.persist (todo);
  }

  public void delete (Todo todo) {
    Todo t = em.merge (todo);
    em.remove( t );
  }

  public void update (Todo todo) {
    em.merge (todo);
  }

  public List <Todo> findTodos () {
    return (List <Todo>) em.createQuery("select t from Todo t")
```

```
                                    .getResultList();
    }

    public Todo findTodo (String id) {
      return (Todo) em.find(Todo.class, Long.parseLong(id));
    }

 }
```

# 4. Configuration and Packaging

We will build the sample application using Ant and explore the configuration and packaging details. If you haven't installed Ant yet, do so now.

## 4.1. Building The Application

Let's look at building the example application and then explore the configuration files in detail.

In *Chapter 3, About the Example Applications*, we looked at the directory structure of the `jsfejb3` sample application. At the command line, go to the `jsfejb3` directory. There you will see a `build.xml` file. This is our Ant build script for compiling and packaging the archives. To build the application, you need to first of all edit the `build.xml` file and edit the value of `jboss-dist` to reflect the location where the JBoss Application Server is installed. Once you have done this, just type the command `ant` and your output should look like this:

```
[vrenish@vinux jsfejb3]$ ant
Buildfile: build.xml

compile:
    [mkdir] Created dir:
 /home/vrenish/jboss-eap-4.2/doc/examples/jsfejb3/build/classes
    [javac] Compiling 4 source files to
 /home/vrenish/jboss-eap-4.2/doc/examples/jsfejb3
    /build/classes
    [javac] Note:
 /home/vrenish/jboss-eap-4.2/doc/examples/jsfejb3/src/TodoDao.java
 uses
    unchecked or unsafe operations.
    [javac] Note: Recompile with -Xlint:unchecked for details.

war:
    [mkdir] Created dir:
 /home/vrenish/jboss-eap-4.2/doc/examples/jsfejb3/build/jars
      [war] Building war:
 /home/vrenish/jboss-eap-4.2/doc/examples/jsfejb3/build/jars/
```

```
      app.war

ejb3jar:
      [jar] Building jar:
 /home/vrenish/jboss-eap-4.2/doc/examples/jsfejb3/build/jars/
      app.jar

ear:
      [ear] Building ear:
 /home/vrenish/jboss-eap-4.2/doc/examples/jsfejb3/build/jars/
      jsfejb3.ear

main:

BUILD SUCCESSFUL
Total time: 2 seconds
(vrenish@vinux jsfejb3)$
```

If you get the BUILD SUCCESSFUL message, you will find a newly created `build` directory with 2 sub-directories in it:

- **classes**: containing the compiled class files.
- **jars**: containing three archives - `app.jar`, `app.war` and `jsfejb3.ear`.
  - app.jar : EJB code and descriptors.
  - app.war : web application which provides the front end to allow users to interact with the business components (the EJBs). The web source (HTML, images etc.) contained in the `jsfejb3/view` directory is added unmodified to this archive. The Ant task also adds the `WEB-INF` directory that 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 extra jars required by the web application.
  - jsfejb3.ear : The EAR file is the complete application, containing the EJB modules and the web module. It also contains 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.

## 4.2. Configuration Files

Now that we have built the application, lets take a closer look at some of the important Configuration files. We have built the final archive ready for deployment - `jsfejb3.ear`. The contents of your EAR file should look like this:

```
jsfejb3.ear
|+ app.jar   // contains the EJB code
    |+ import.sql
    |+ Todo.class
    |+ TodoDao.class
    |+ TodoDaoInt.class
    |+ META-INF
        |+ persistence.xml
|+ app.war   // contains web UI
    |+ index.html
    |+ index.xhtml
    |+ create.xhtml
    |+ edit.xhtml
    |+ todos.xhtml
    |+ TodoBean.class
    |+ style.css
    |+ META-INF
    |+ WEB-INF
       |+ faces-config.xml
       |+ navigation.xml
       |+ web.xml
|+ META-INF  // contains the descriptors
    |+ application.xml
    |+ jboss-app.xml
```

- `application.xml`: This file lists the JAR files in the EAR (in our case `app.jar`) and tells the JBoss server what files to look for and where. The root URL for the application is also specified in this file as 'context-root'.

```
<application>
  <display-name>Sample Todo</display-name>
  <module>
    <web>
      <web-uri>app.war</web-uri>
      <context-root>/jsfejb3</context-root>
    </web>
  </module>
  <module>
    <ejb>app.jar</ejb>
  </module>
</application>
```

- `jboss-app.xml`: Every EAR application should specify a unique string name for the class loader. In our case, we use the application name 'jsfejb3' as the class loader name.

```
<jboss-app>
  <loader-repository>
    jsfejb3:archive=jsfejb3.ear
  </loader-repository>
</jboss-app>
```

- `app.jar`: This contains EJB3 Session Bean and Entity Bean classes and the related configuration files. In addition, the `persistence.xml` file configures the back-end data source (in our case the default HSQL database) for the `EntityManager`.

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

- `app.war`: This contains the Web UI files packaged according to the Web Application aRchive (WAR) specification. It contains all the web pages and the required configuration files. The `web.xml` file is an important file for all JAVA EE web applications. It is the web deployment descriptor file. The `faces-config.xml` file is the configuration file for JSF. The `navigation.xml` file contains the rules for JSF page navigation.

```
//faces-config.xml
<faces-config>
  <application>
    <view-handler>
      com.sun.facelets.FaceletViewHandler
    </view-handler>
  </application>
  <managed-bean>
    <description>Dao</description>
    <managed-bean-name>todoBean</managed-bean-name>
    <managed-bean-class>TodoBean</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>
</faces-config>
```

# 5. The Database

## 5.1. Creating the Database Schema

To pre-populate the database, we have supplied SQL Code (`import.sql`) to run with HSQL in the `examples/jsfejb3/resources` directory. When you build the application using Ant, this is packaged in the app.jar file within the jsfejb3.ear file. When the application is deployed, you should be able to view the pre-populated data.

## 5.2. The HSQL Database Manager Tool

Just as a quick aside at this point, start up the JMX console 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 Hypersonic service is enabled in the `hsqldb-ds.xml` file.

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.

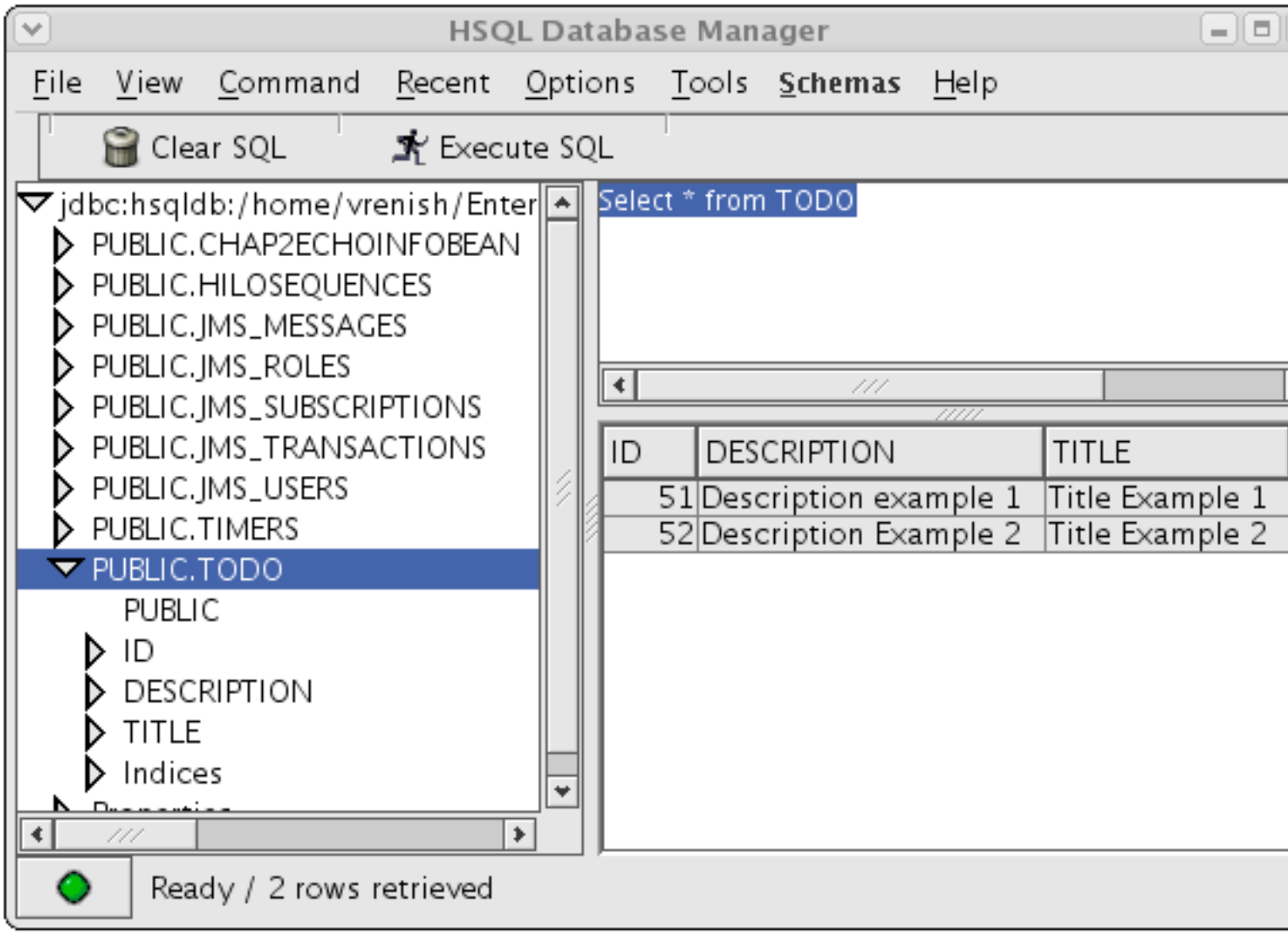


**Figure 4.4. The HSQL Database Manger**

## 6. Deploying the Application

Deploying an application in JBoss is simple and easy. You just have to copy the EAR file to the `deploy` directory in the 'server configuration' directory of your choice. Here, we will deploy it to the 'default' configuration, so we copy the EAR file to the `JBOSS_DIST/jboss-as/server/default/deploy` directory.

You should see something close to the following output from the server:

```
15:32:23,997 INFO  [EARDeployer] Init J2EE application:
 file:/home/vrenish/jboss-eap-4.2
/jboss-as/server/default/deploy/jsfejb3.ear
```

```
15:32:24,212 INFO  [JmxKernelAbstraction] creating wrapper delegate
 for: org.jboss.ejb3.
entity.PersistenceUnitDeployment
15:32:24,213 INFO  [JmxKernelAbstraction] installing MBean:
 persistence.units:ear=
jsfejb3.ear,jar=app.jar,unitName=helloworld with dependencies:
15:32:24,213 INFO  [JmxKernelAbstraction]
 jboss.jca:name=DefaultDS,service=
DataSourceBinding
15:32:24,275 INFO  [PersistenceUnitDeployment] Starting persistence
 unit persistence.
units:ear=jsfejb3.ear,jar=app.jar,unitName=helloworld
15:32:24,392 INFO  [Ejb3Configuration] found EJB3 Entity bean: Todo
15:32:24,450 WARN  [Ejb3Configuration] Persistence provider caller
 does not implements
the EJB3 spec correctly.
 PersistenceUnitInfo.getNewTempClassLoader() is null.
15:32:24,512 INFO  [Configuration] Reading mappings from resource :
 META-INF/orm.xml
15:32:24,512 INFO  [Ejb3Configuration] [PersistenceUnit:
 helloworld] no META-INF/orm.xml
found
15:32:24,585 INFO  [AnnotationBinder] Binding entity from annotated
 class: Todo
15:32:24,586 INFO  [EntityBinder] Bind entity Todo on table Todo
.
.
.
.
15:32:26,311 INFO  [SchemaExport] Running hbm2ddl schema export
15:32:26,312 INFO  [SchemaExport] exporting generated schema to
 database
15:32:26,314 INFO  [SchemaExport] Executing import script:
 /import.sql
15:32:26,418 INFO  [SchemaExport] schema export complete
15:32:26,454 INFO  [NamingHelper] JNDI InitialContext
 properties:{java.naming.factory.
initial=org.jnp.interfaces.NamingContextFactory,
 java.naming.factory.url.pkgs=org.jboss.
naming:org.jnp.interfaces}
15:32:26,484 INFO  [JmxKernelAbstraction] creating wrapper delegate
 for: org.jboss.ejb3.
stateless.StatelessContainer
15:32:26,485 INFO  [JmxKernelAbstraction] installing MBean:
 jboss.j2ee:ear=jsfejb3.ear,
jar=app.jar,name=TodoDao,service=EJB3 with dependencies:
15:32:26,513 INFO  [JmxKernelAbstraction]
 persistence.units:ear=jsfejb3.ear,
jar=app.jar,unitName=helloworld
```

```
15:32:26,557 INFO  [EJBContainer] STARTED EJB: TodoDao ejbName:
 TodoDao
15:32:26,596 INFO  [EJB3Deployer] Deployed:
 file:/home/vrenish/jboss-eap-4.2/jboss-as/
server/default/tmp/deploy/
tmp33761jsfejb3.ear-contents/app.jar
15:32:26,625 INFO  [TomcatDeployer] deploy, ctxPath=/jsfejb3,
 warUrl=.../tmp/deploy/
tmp33761jsfejb3.ear-contents/app-exp.war/
15:32:26,914 INFO  [EARDeployer] Started J2EE application:
 file:/home/vrenish/jboss-eap-
4.2/jboss-as/server/default/deploy/jsfejb3.ear
```

If there are any errors or exceptions, make a note of the error message. Check that the EAR is complete and inspect the WAR file and 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/jsfejb3*

You will be forwarded to the application main page. *Figure 4.5, "Sample TODO"* shows the sample application in action.


**Figure 4.5. Sample TODO**

# Using Seam

JBoss Seam is a framework that provides the glue between the new EJB3 and JSF frameworks that are part of the Java EE 5.0 standard. In fact, the name Seam refers to the seamless manner in which it enables developers to use these two frameworks in an integrated manner. Seam automates many of the common tasks, and makes extensive use of annotations to reduce the amount of xml code that needs to be written. The overall effect is to significantly reduce the total amount of coding that needs to be done.

If you are new to Seam, you can find more introductory information from the following url and book:

- *The Seam Reference Guide* [http://docs.jboss.com/seam/2.0.0.GA/reference/en/html_single/].
- *Beginning JBoss Seam* by Joseph Faisal Nusairat, Apress 2007.

We have included two versions of the example application, one coded using EJB3 / JSF without using Seam, and one using Seam, to demonstrate clearly the difference in application development using the Seam framework.

## 1. Data Model

Let's start off our examination of the Seam implementation in the same way, by examining how the Data Model is implemented. This is done in the `Todo.java` file.

```
@Entity
@Name("todo")
public class Todo implements Serializable {

  private long id;
  private String title;
  private String description;

  public Todo () {
    title ="";
    description ="";
  }

  @Id @GeneratedValue
  public long getId() { return id;}
  public void setId(long id) { this.id = id; }

  @NotNull
  public String getTitle() { return title; }
  public void setTitle(String title) {this.title = title;}
```

```
  @NotNull
  @Length(max=250)
  public String getDescription() { return description; }
  public void setDescription(String description) {
    this.description = description;
  }

}
```

The `@Entity` annotation defines the class as an EJB3 entity bean, and tells the container to map the `Todo` class to a relational database table. Each property of the class will become a column in the table. Each instance of the class will become a row in this table. Since we have not used the `@Table` annotation, Seam's "configuration by exception" default will name the table after the class.

`@Entity` and `@Table` are both EJB3 annotations, and are not specific to Seam. It is possible to use Seam completely with POJOs (Plain Old Java Objects) without any EJB3-specific annotations. However, EJB3 brings a lot of advantages to the table, including container managed security, message-driven components, transaction and component level persistence context, and `@PersistenceContext` injection, which we will encounter a little further on.

The `@Name` annotation is specific to Seam, and defines the string name for Seam to use to register the Entity Bean. This will be the default name for the relational database table. Each component in a Seam application must have a unique name. In the other components in the Seam framework, such as JSF web pages and session beans, you can reference the managed `Todo` bean using this name. If no instance of this class exists when it is referenced from another component, then Seam will instantiate one.

The `@Id` annotation defines a primary key `id` field for the component. `@GeneratedValue` specifies that the server will automatically generate this value for the component when it is saved to the database.

Seam provides support for model-based constraints defined using Hibernate Validator, although Hibernate does not have to be the object persister used. The `@NotNull` annotation is a validation constraint that requires this property to have a value before the component can be persisted into the database. Using this annotation allows the validation to be enforced by the JSF code at the view level, without having to specify the exact validation constraint in the JSF code.

At this point the only apparent difference between the Seam version and the EJB3/JSF version of the app is the inclusion of the validator annotation `@NotNull`, and the `@Name` annotation. However, while the EJB3/JSF version of this application requires a further `TodoBean` class to be manually coded and managed in order to handle the interaction between the `Todo` class and the web interface, when using

Seam the Seam framework takes care of this work for us. We'll see how this is done in practice as we examine the implementation of the user interface.

## 2. JSF Web Pages - index.xhtml and create.xhtml

The **index.xhtml** file used is the same as in the EJB3/JSF example.

**create.xhtml** begins to reveal the difference that coding using the Seam framework makes.

```
<h:form id="create">

<f:facet name="beforeInvalidField">
  <h:graphicImage styleClass="errorImg" value="error.png"/>
</f:facet>
<f:facet name="afterInvalidField">
  <s:message styleClass="errorMsg" />
</f:facet>
<f:facet name="aroundInvalidField">
  <s:div styleClass="error"/>
</f:facet>

<s:validateAll>

<table>

  <tr>
    <td>Title:</td>
    <td>
      <s:decorate>
        <h:inputText id="title" value="#{todo.title}" size="15"/>
      </s:decorate>
    </td>
  </tr>

  <tr>
    <td>Description:</td>
    <td>
      <s:decorate>
        <h:inputTextarea id="description"
 value="#{todo.description}"/>
      </s:decorate>
    </td>
  </tr>

</table>

</s:validateAll>
```

```
<h:commandButton type="submit" id="create" value="Create"
                 action="#{todoDao.persist}"/>
</h:form>
```

The first thing that is different here is the Java Server Facelet code at the beginning, which works with the `@NotNull` validation constraint of our `todo` class to enforce and indicate invalid input to the user.

Also notice here that rather than requiring the use of a `TodoBean` class as we did in the EJB3/JSF example we back the form directly with a `Todo` entity bean. When this page is called, JSF asks Seam to resolve the variable `todo` due to JSF EL references such as `#{todo.title}`. Since there is no value already bound to that variable name, Seam will instantiate an entity bean of the `todo` class and return it to JSF, after storing it in the Seam context. The Seam context replaces the need for an intermediary bean.

The form input values are validated against the Hibernate Validator constraints specified in the `todo` class. JSF will redisplay the page if the constraints are violated, or it will bind the form input values to the `Todo` entity bean.

Entity beans shouldn't do database access or transaction management, so we can't use the `Todo` entity bean as a JSF action listener. Instead, creation of a new todo item in the database is accomplished by calling the `persist` method of a `TodoDao` session bean. When JSF requests Seam to resolve the variable `todoDao` through the JSF EL expression `#{todoDao.persist}`, Seam will either instantiate an object if one does not already exist, or else pass the existing stateful `todoDao` object from the Seam context. Seam will intercept the `persist` method call and inject the `todo` entity from the session context.

Let's have a look at the `TodoDao` class (defined in `TodoDao.java`) to see how this injection capability is implemented.

## 3. Data Access using a Session Bean

Let's go through a listing of the code for the `TodoDao` class.

```
@Stateful
@Name("todoDao")
public class TodoDao implements TodoDaoInt {

  @In (required=false) @Out (required=false)
  private Todo todo;

  @PersistenceContext (type=EXTENDED)
  private EntityManager em;

  // Injected from pages.xml
```

```
   Long id;

   public String persist () {
     em.persist (todo);
     return "persisted";
   }

   @DataModel
   private List <Todo> todos;

   @Factory("todos")
   public void findTodos () {
     todos = em.createQuery("select t from Todo t")
                                  .getResultList();
   }

   public void setId (Long id) {
     this.id = id;

     if (id != null) {
       todo = (Todo) em.find(Todo.class, id);
     } else {
       todo = new Todo ();
     }
   }

   public Long getId () {
     return id;
   }

   public String delete () {
     em.remove( todo );
     return "removed";
   }

   public String update () {
     return "updated";
   }

   @Remove @Destroy
   public void destroy() {}

 }
```

First of all notice that this is a stateful session bean. Seam can use both stateful and stateless session beans, the two most common types of EJB3 beans.

The `@In` and `@Out` annotations define an attribute that is injected by Seam. The attribute is injected to this object or from this object to another via a Seam context

variable named `todo`, a reference to the Seam registered name of our `Todo` class defined in `Todo.java`.

The `@PersistenceContext` annotation injects the EJB3 Entity manager, allowing this object to persist objects to the database. Because this is a stateful session bean and the `PersistenceContext` type is set to `EXTENDED`, the same Entity Manager instance is used until the Remove method of the session bean is called. The database to be used (a `persistence-unit`) is defined in the file `resources/META-INF/persistence.xml`

Note that this session bean has simultaneous access to context associated with web request (the form values of the `todo` object), and state held in transactional resources (the `EntityManager`). This is a break from traditional J2EE architectures, but Seam does not force you to work this way. You can use more traditional forms of application layering if you wish.

The `@DataModel` annotation initializes the `todos` property, which will be outjected or "exposed" to the view. The `@Factory` annotated method performs the work of generating the `todos` list, and is called by Seam if it attempts to access the exposed `DataModel` property and finds it to be null. Notice the absence of property access methods for the `todos` property. Seam takes care of this for you automatically.

Let's take a look at the JSF code that we use for displaying and editing the list of todos, to get an idea of how to use these interfaces in practice.

## 4. JSF Web Pages - todos.xhtml and edit.xhtml

Using the `DataModel` exposed property of the Session Bean it becomes trivial to produce a list of todos:

```
<h:form>

<h:dataTable value="#{todos}" var="todo">
  <h:column>
    <f:facet name="header">Title</f:facet>
    #{todo.title}
  </h:column>
  <h:column>
    <f:facet name="header">Description</f:facet>
    #{todo.description}
  </h:column>
  <h:column>
    <a href="edit.seam?tid=#{todo.id}">Edit</a>
  </h:column>
</h:dataTable>

<center>
```

```
  <h:commandButton action="create"
            value="Create New Todo" type="submit"/>
</center>

</h:form>
```

When the JSF variable resolver encounters `{#todos}` and requests `todos`, Seam finds that there is no "todos" component in the current scope, so it calls the @Factory("todos") method to make one. The todos object is then outjected once the factory method is done since it is annotated with the @DataModel annotation.

Constructing the view for the edit page is similarly straight forward:

```
<h:form id="edit">

<f:facet name="beforeInvalidField">
  <h:graphicImage styleClass="errorImg" value="error.png"/>
</f:facet>
<f:facet name="afterInvalidField">
  <s:message styleClass="errorMsg" />
</f:facet>
<f:facet name="aroundInvalidField">
  <s:div styleClass="error"/>
</f:facet>

<s:validateAll>

<table>

  <tr>
    <td>Title:</td>
    <td>
      <s:decorate>
        <h:inputText id="title" value="#{todo.title}" size="15"/>
      </s:decorate>
    </td>
  </tr>

  <tr>
    <td>Description:</td>
    <td>
      <s:decorate>
        <h:inputTextarea id="description"
 value="#{todo.description}"/>
      </s:decorate>
    </td>
  </tr>

</table>
```

```
</s:validateAll>

<h:commandButton type="submit" id="update" value="Update"
                 action="#{todoDao.update}"/>

<h:commandButton type="submit" id="delete" value="Delete"
                 action="#{todoDao.delete}"/>
</h:form>
```

Here we see the same factors in play. JSF validation code taking advantage of the validation constraints defined in our Entity Bean, and the use of the `todoDao` Session Bean's `update` and `delete` methods to update the database.

The call from `todos.xhtml`: `edit.seam?tid=#{todo.id}` causes Seam to create a `todoDao` and set it's `id` property to `tid`. Setting its `id` property causes the `todoDao` to retrieve the appropriate record from the database.

The functionality that allows the edit page to be called with a parameter in this way is implemented through `pages.xml`. Let's have a look at the `pages.xml` file and how it is used by Seam applications.

## 5. Xml Files

Seam drastically reduces the amount of xml coding that needs to be done. One file that is of interest is the `pages.xml`, packaged in the `app.war` file's `WEB-INF` directory. This file is available in the `resources/WEB-INF` directory in the source code bundle. The `pages.xml` file is used to define page descriptions including Seam page parameters (HTTP `GET` parameters), page actions, page navigation rules, error pages etc. Among other things it can be used in a Seam application to define exception handlers and redirections.

In the case of our sample application we are using it to define a Seam page parameter. The `pages.xml` in this example contains the following code:

```
<page view-id="/edit.xhtml">
    <param name="tid" value="#{todoDao.id}"
           converterId="javax.faces.Long"/>
</page>
```

This defines a parameter named `tid` for the `edit.xhtml` page. When the `edit.xhtml` page is loaded, the HTTP `GET` request parameter `tid` is converted to a `Long` value and assigned to the `id` property of the `todoDao` object. You can have as many page parameters as required to bind HTTP `GET` request parameters to the back-end components in your application.

# 6. Further Information

This completes our walkthrough of the sample Seam application.
For further, detailed information on developing applications using the Seam framework, please refer to the *The Seam Reference Guide* [http://docs.jboss.com/seam/2.0.0.GA/reference/en/html_single/].

# Using other Databases

In the previous chapters, we've just been using the JBossAS default datasource in our applications. This datasource is configured to use the embedded Hypersonic database instance shipped by default with the distribution. This datasource is bound to the JNDI name `java:/DefaultDS` and its descriptor is named `hsqldb-ds.xml` under the deploy directory

Having a database included with JBossAS is very convenient for running the server and examples out-of-the-box . However, this database is not a production quality database and as such should not be used with enterprise-class deployments. As a consequence of this JBoss Support does not provide any official support for Hypersonic.

In this chapter we will explain in details how to configure and deploy a datasource to connect JBossAS to the most popular database servers available on the market today.

## 1. 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_5.dtd`. Additional documentation on the files and the JBoss JCA implementation can also be found in the JBoss Application Server Guide available at *http://labs.jboss.com/projects/docs/*.

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

If you look at the example files `firebird-ds.xml`, `facets-ds.xml` and `sap3-ds.xml`, you'll notice that they have a completely different format, with the root element being `connection-factories` rather than `datasources`. These use an alternative, more generic JCA configuration syntax used with a pre-packaged JCA resource adapter. 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.

We would also highly recommend consulting the JCA wiki pages at http://wiki.jboss.org/wiki/Wiki.jsp?page=JBossJCA

Next, we'll work through some step-by-step examples to illustrate what's involved setting up a datasource for a specific database.

# 2. Using MySQL as the Default DataSource

The MySQL® database has become the world's most popular open source database thanks to its consistent fast performance, high reliability and ease of use. This database server is used in millions of installations ranging from large corporations to specialized embedded applications across every continent of the world. . In this section, we'll be using the community version of their database server (GA 5.0.45) and the latest JDBC driver (GA 5.1.5) both available at *http://www.mysql.com*.

## 2.1. Installing the JDBC Driver and Deploying the datasource

To make the JDBC driver classes available to the JBoss Application Server, copy the archive `mysql-mysql-connector-java-5.1.5-bin.jar` from the Connector/J distribution to the `lib` directory in the `default` server configuration (assuming that is the server configuration you're running).

Then create a text file in the deploy directory called mysql-ds.xml with the following datasource descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
 <local-tx-datasource>
   <jndi-name>DefaultDS</jndi-name>

 <connection-url>jdbc:mysql://localhost:3306/test</connection-url>
   <driver-class>com.mysql.jdbc.Driver</driver-class>
   <user-name>root</user-name>
   <password>jboss</password>

 <valid-connection-checker-class-
name>org.jboss.resource.adapter.jdbc.vendor.MySQLValidConnectionChecker</
valid-connection-checker-class-name>
   <metadata>
 <type-mapping>mySQL</type-mapping>
 </metadata>
 </local-tx-datasource>
</datasources>
```

The datasource is pointing at the database called test provided by default with MySQL 5.x. Remember to update the connection url attributes as well as the combo username/password to match your environment setup.

## 2.2. Configuring JBoss MQ Persistence Manager

The persistence manager of JBoss MQ uses the default datasource to create tables to store JMS messages, transaction data and other indexes. The DDLs used by this

service are vendor-specific and should be updated accordingly as follows: From docs/examples/jms, copy mysql-jdbc2-service.xml over to deploy/jms. Then delete the existing hsqldb-jdbc2-service.xml file located in the same folder. Last, make sure to point the service at the right datasource jndi name i.e. “DefaultDS”:

```
<mbean code="org.jboss.mq.pm.jdbc2.PersistenceManager"
 name="jboss.mq:service=PersistenceManager">
 <depends
 optional-attribute-
name="ConnectionManager">jboss.jca:service=DataSourceBinding,name=DefaultDS</
depends>
....
```

**Note**

The StateManager and other services such as Timer or HiLo make use of standard DDLs thus don't require any changes.

## 2.3. Testing the MySQL DataSource

Using the test client described in *Section 5, “Creating a JDBC client”*, you may now verify the proper installation of your datasource.

# 3. Configuring a datasource for Oracle DB

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/technology/products/database/xe/index.html*

In this section, we'll connect the server to Oracle Database 10g Express Edition using the latest JDBC driver (11g) available at *http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html*

## 3.1. Installing the JDBC Driver and Deploying the DataSource

To make the JDBC driver classes available to JBoss Application Server, copy the archive ojdbc5.jar to the lib directory in the default server configuration (assuming that is the server configuration you’re running).

Then create a text file in the `deploy` directory called `oracle-ds.xml` with the following datasource descriptor :

```
    <?xml version="1.0" encoding="UTF-8"?>
    <datasources>
     <local-tx-datasource>
      <jndi-name>DefaultDS</jndi-name>

 <connection-url>jdbc:oracle:thin:@localhost:1521:xe</connection-
url>
      <driver-class>oracle.jdbc.driver.OracleDriver</driver-class>
      <user-name>SYSTEM</user-name>
      <password>jboss</password>

 <valid-connection-checker-class-
name>org.jboss.resource.adapter.jdbc.vendor.OracleValidConnectionChecker</
valid-connection-checker-class-name>
      <metadata>
       <type-mapping>Oracle9i</type-mapping>
      </metadata>
     </local-tx-datasource>
    </datasources>
```

The datasource is pointing at the database/SID called “xe” provided by default with Oracle XE.

Of course, you need to update the connection url attributes as well as the username/password combination to match your environment setup.

## 3.2. Configuring JBoss MQ Persistence Manager

The persistence manager of JBoss MQ uses the default datasource to create tables to store JMS messages, transaction data and other indexes. The DDLs used by this service are vendor-specific and should be updated accordingly as follows:

From `docs/examples/jms`, copy `oracle-jdbc2-service.xml` over to `deploy/jms`. Then delete the existing `hsqldb-jdbc2-service.xml` file located in the same folder. Last, make sure to point the service at the right datasource jndi name i.e. “DefaultDS”:

```
<mbean code="org.jboss.mq.pm.jdbc2.PersistenceManager"
     name="jboss.mq:service=PersistenceManager">
<depends
 optional-attribute-
name="ConnectionManager">jboss.jca:service=DataSourceBinding,name=DefaultDS</
depends>
....
```

**Note**

The StateManager and other services such as Timer or HiLo make use of standard DDLs thus don't require any changes.

## 3.3. Testing the Oracle DataSource

Before you can verify the datasource configuration, Oracle XE should be reconfigured to avoid port conflict with JBossAS as by default they both start a web server on port 8080.

Open up an Oracle SQLcommand line and execute the following commands:

```
SQL> connect;
Enter user-name: SYSTEM
Enter password:
Connected.
SQL> begin
2  dbms_xdb.sethttpport('8090');
3  end;
4  /
PL/SQL procedure successfully completed.
SQL> select dbms_xdb.gethttpport from dual;
GETHTTPPORT
-----------
8090
```

The web server started by Oracle XE to provide http-based administration tools is now running on port 8090. Start the JBossAS server instance as you would normally do. You are now ready to use the test client described in Chapter 6.5 to verify the proper installation of your datasource.

# 4. Configuring a datasource for Microsoft SQL Server 200x

In this section, we'll connect the server to MS SQL Server 2000 using the latest JDBC driver (v1.2) available at *http://msdn2.microsoft.com/en-us/data/aa937724.aspx*.

## 4.1. Installing the JDBC Driver and Deploying the DataSource

To make the JDBC driver classes available to JBoss Application Server, copy the archive sqljdbc.jar from the sqljdbc_1.2 distribution to the lib directory in the default server configuration (assuming that is the server configuration you're running).

Then create a text file in the deploy directory called mssql-ds.xml with the following datasource descriptor :

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
  <local-tx-datasource>
  <jndi-name>DefaultDS</jndi-name>

 <connection-url>jdbc:sqlserver://
localhost:1433;DatabaseName=pubs</connection-url>

 <driver-class>com.microsoft.sqlserver.jdbc.SQLServerDriver</
driver-class>
  <user-name>sa</user-name>
  <password>jboss</password>
  <check-valid-connection-sql>SELECT 1 FROM
 sysobjects</check-valid-connection-sql>
  <metadata>
   <type-mapping>MS SQLSERVER2000</type-mapping>
  </metadata>
  </local-tx-datasource>
</datasources>
```

The datasource is pointing at a database “pubs” provided by default with MS SQL Server 2000.

Remember to update the connection url attributes as well as the username/password combination to match your environment setup.

**Note**

The class MSSQLValidConnectionChecker is provided with the AS distribution. Unfortunately it throws an exception when the latest 2005 version of the MS driver is installed *http://jira.jboss.com/jira/browse/JBAS-4911*.

### 4.1.1. Configuring JBoss MQ Persistence Manager

The persistence manager of JBoss MQ uses the default datasource to create tables to store JMS messages, transaction data and other indexes. The DDLs used by this service are vendor-specific and should be updated accordingly as follows: From `docs/examples/jms`, copy `mssql-jdbc2-service.xml` over to deploy/jms. Then delete the existing `hsqldb-jdbc2-service.xml` file located in the same folder. Last, make sure to point the service at the right datasource jndi name i.e. “DefaultDS”:

```
<mbean code="org.jboss.mq.pm.jdbc2.PersistenceManager"
 name="jboss.mq:service=PersistenceManager">
 <depends
 optional-attribute-
name="ConnectionManager">jboss.jca:service=DataSourceBinding,name=DefaultDS</
depends>
....
```

**Note**

The StateManager and other services such as Timer or HiLo make use of standard DDLs thus don't require any changes.

### 4.1.2. Testing the datasource

Using the test client described in *Section 5, "Creating a JDBC client"*, you may now verify the proper installation of your datasource.

## 5. Creating a JDBC client

When testing a newly configured datasource we suggest using some very basic JDBC client code embedded in a JSP page. First of all, you should create an exploded WAR archive under the deploy directory which is simply a folder named "`jdbcclient.war`". In this folder, create a text document named client.jsp and paste the code below:

```
<%@page contentType="text/html"
 import="java.util.*,javax.naming.*,javax.sql.DataSource,java.sql.*"
 %>
 <%

  DataSource ds = null;
  Connection con = null;
  PreparedStatement pr = null;
  InitialContext ic;
  try {
  ic = new InitialContext();
  ds = (DataSource)ic.lookup( "java:/DefaultDS" );
  con = ds.getConnection();
  pr = con.prepareStatement("SELECT USERID, PASSWD FROM
 JMS_USERS");
  ResultSet rs = pr.executeQuery();
  while (rs.next()) {
  out.println("<br> " +rs.getString("USERID") + " | "
 +rs.getString("PASSWD"));
```

```
  }
  rs.close();
  pr.close();
  }catch(Exception e){
  out.println("Exception thrown " +e);
  }finally{
  if(con != null){
  con.close();
 }
}
```

Open up a web browser and hit the url: *http://localhost:8080/jdbcclient/client.jsp*. A list of users and password should show up as a result of the jdbc query:

```
dynsub | dynsub
guest | guest
j2ee | j2ee
john | needle
nobody | nobody
```

# Appendix A. Further Information Sources

Revision History

| | | |
|---|---|---|
| Revision 4.2.2-1 | Oct 15 2007 | Michael Hideo |
| Migrated Content from 4.0.4 | | |
| Revision 4.2.2 | Nov 28 2007 | Samson Kittoli |
| Updated Content from tech review. | | |

For a longer introduction to JBoss, see *JBoss: A Developer's Notebook*. (O'Reilly, 2005. Norman Richards, Sam Griffith).

For more comprehensive JBoss documentation covering advanced JBoss topics, refer to the manuals available online at *http://www.redhat.com/docs/manuals/jboss*.

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

To learn more about Hibernate, see *Java Persistence With Hibernate*. (Manning, 2007. Christian Bauer, Gavin King)

For complete coverage of the JBoss Seam framework, we recommend *JBoss Seam: Simplicity And Power Beyond Java EE*. (Prentice Hall, 2007. Michael Yuan, Thomas Heute).