JBoss to Geronimo - Hibernate Migration

This article will help you migrate applications using Hibernate 3.3.1 as the ORM tool from JBoss Application Server 4.2.1 to Apache Geronimo 2.1.4.

Hibernate is a powerful, high performance object/relational persistence and query service. It helps in the development of persistent (POJO) classes which have the getter and setter methods for attributes that are then mapped to fields in the database. You may follow object-oriented idiom - including association, inheritance, polymorphism, composition, and collections. Hibernate allows you to express queries in its own portable SQL extension (HQL), as well as in native SQL, or with an object-oriented Criteria and Example API.

Basically, Hibernate maps the Java classes to the database tables. It also provides the data query and retrieval facilities that significantly reduce the development time. It fits in naturally to the object-oriented mode of code development in the JAVA middle-tier. A unique feature of Hibernate is that it facilitates transparent persistence that enables the applications to switch to any database be it MySQL, Oracle or DB2. Hibernate can be used in Java Swing applications, Java Servlet-based applications, or J2EE applications using EJB session beans (Java Servlet based application in our case).

In order to illustrate the steps involved in the migration a sample application is provided which we will first deploy in JBoss and then migrate to Geronimo. The sample application will be the Online Brokerage Application which we already used in the JDBC migration example. This application has been changed to use Hibernate for persistence.

This article is organized in the following sections:

Back to Top

Hibernate implementation analysis

Hibernate can provide the following services:

- Connection Management
- Transaction Management
- Object Relational Mapping

It is also very flexible and we can use any combinations of these services. In this article we will be configuring hibernate to provide only the O/R Mapping feature while JBoss/Geronimo will provide both Transaction Management and Connection Management. This is the most commonly used configuration of Hibernate with an application server. Hibernate requires a configuration file `hibernate.cfg.xml` which creates the connection pool and sets up the required environment. It contains parameters like the database driver, connection url, dialect (this specifies which RDBMS is being used), username, password, pool size among others. It also contains the location of the Mapping File `*.hbm.xml`. A Mapping File is used to map the fields in the database table to the persistent class attributes.

These properties are **common** to any application server including **Apache Geronimo v2.1**.

However, JBoss (more specifically the Hibernate MBean) provides **two** additional deployment mechanisms.

The first is the Hibernate Archive (HAR file). Here, all the Hibernate classes and mapping files are packaged in a special archive, the HAR file. JBoss deploys this archive in the same way as it does with an EAR or a WAR file.

The alternative to the HAR file is to simply put all the Hibernate classes and mapping files with the other application classes, like in an EAR for example. The Hibernate MBean would be separately configured and told to search all application JARs for mapping files. Both deployment mechanisms allow you to provide Hibernate objects to your application code without performing any manual configuration or writing of the setup code normally required.

Structurally, a HAR file resembles a JBoss service archive (SAR file). The HAR file contains the Hibernate class files and the mapping files (*.hbm.xml) along with the standard `jboss-service.xml` file containing a definition for the Hibernate MBean configured for the needs of the Hibernate application being created. In the latest JBoss distribution, it has been renamed as `hibernate-service.xml`, but it retains the same structure and purpose.

Hibernate archives can be deployed as top-level packages or as a component of an EAR file. Since Hibernate archives are not a standard J2EE deployment type, we need to declare them in the `jboss-app.xml` file of an EAR file to use them in that context.

The following table provides a feature-to-feature comparison between these two applicaiton servers.

Feature	Apache Geronimo v2.1	JBoss v4.2.1
Container-managed datasource	Supported. Hibernate is able to use a datasource given its JNDI name. This is because it is running in the same thread as the application.	Supported. Hibernate can lookup a datasource from JNDI given its JNDI name.
Automatic JNDI binding	Not Supported.	Supported. Once the property is set the session factory is bound to the JNDI context.
JTA Session binding	This feature is not supported out of the box. We need to write a lookup for the Geronimo Transaction Manager to enable this.	Supported out of the Box. Hibernate provides a lookup class for the JBoss Transaction Manager.

JMX deployment	Not Supported out of the box. **Can be implemented by writing a GBean and a Hibernate Connection Provider class**.	Supported. Hibernate is distributed with `org.hibernate.jmx.HibernateService` which can be deployed on JBoss.
Hibernate Archive (HAR)	Not Supported. Hibernate classes are deployed as a part of the J2EE archives.	Supported. A HAR packages the configuration and mapping files enabling extra server support to deployment.
Caching	You can use caching mechanisms provided by hibernate.	You can use caching mechanisms provided by hibernate. Integration with JBoss Cache is also supported.
Session Management	Not Supported. It is required to manually open sessions. Only the transaction needs to be closed.	The Hibernate Session's lifecycle can be automatically bound to the scope of a JTA transaction. This means you no longer have to manually open and close the Session, this becomes the job of a JBoss EJB interceptor.
Hibernate Mapping Files	We need to specify the locations of the Hibernate mapping files.	If we use HAR deployment JBoss will automatically lookup the Hibernate mapping files.

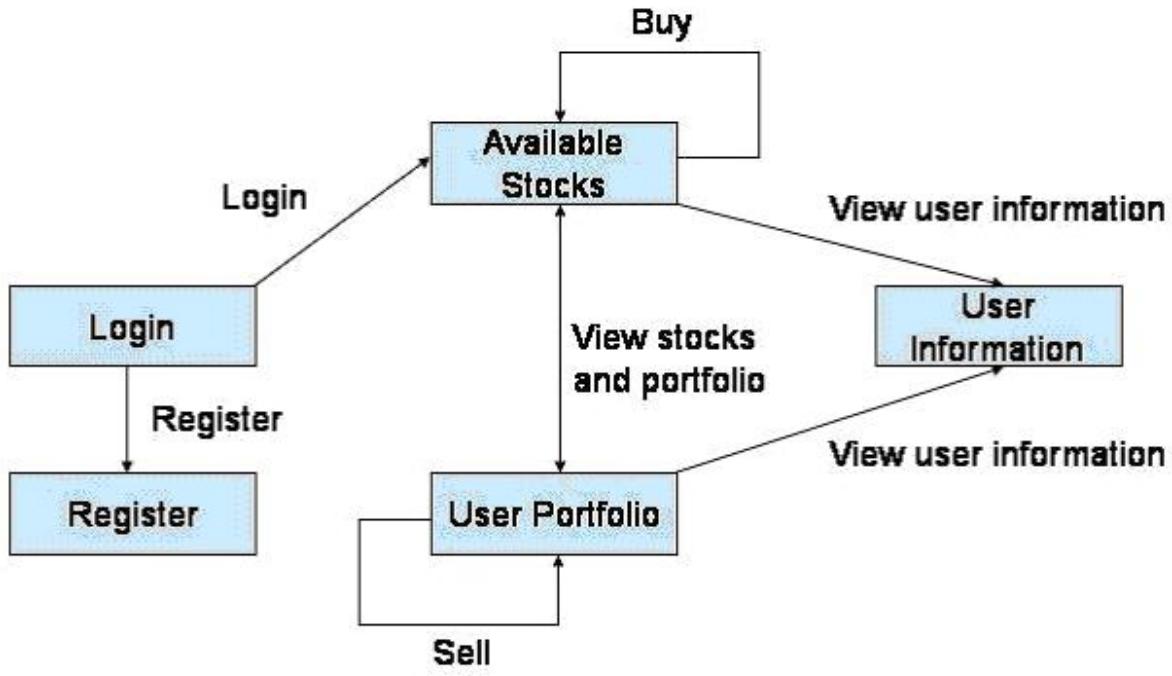
Back to Top

Sample application

This article contains a sample application to demonstrate migrating an application from JBoss to Geronimo, called Online Brokerage . It represents an online trading scenario in which users can buy and sell stocks. The application has the following five pages:

- Login Page
- Registration Page
- User Information Page
- Available Stocks Page
- User Portfolio Page

The following figure illustrates the application flow:



First, the user accesses the Login page. From the Login page the user enters the user name and password. If the user name or password is not valid the application throws an error message and rejects the user's login attempt. If the user name and password are correct, the user is taken to the Available Stocks page where he/she can view all the stocks that are present for sale at that time.
The user can choose to buy as many stocks as wanted, depending on the available money in the account, by clicking the Buy button. After the transaction completes successfully the user is brought back to the Available Stocks page where he/she can buy more stocks if required. If the user has insufficient funds to buy stocks the application will throw an error and will not process the transaction. The error message is shown at the top of the Available Stocks page. There is a User Info button on this page. By clicking this button the user is taken to the User Info page and shown the user details.
From the Available Stocks page there is a View your Portfolio link that shows all the stocks that the user owns. In that page, the user can select the stocks and quantity to sell. This page also shows the user's available cash in the User Cash field. If the user tries to sell more stocks than he/she has, the application will throw an error. The error message will be displayed on the same page. For each successful sale, the sale amount is added to the user's cash balance. The quantity text box shows the quantity of stocks of a particular company that the user has. The Quantity to Sell field allows the user to enter the quantity of stocks to sell for a specific company. For selling and buying, the radio button should be checked. This should be done after entering the values. If either the quantity to sell textbox is not filled or the selection box is not checked and you press on sell a JavaScript alert will be triggered saying that the required field is empty. On entering non numeric characters for quantity another alert will be triggered. This behavior is similar for the Available Stocks page as well.
New users can register by clicking the Register button in the login page. In the Registration page the user will enter a user id, user name, password, address and available cash.
Back to Top

Application classes and JSP pages

The Online Brokerage sample application consists of the following packages:

- com.dev.trade.bo
  - Stock - Represents the stock of a company.
  - User - Represents the user.
  - UserStock - Represents the stock owned by a user.
- com.dev.trade.dao
  - TradeDAO - Contains all the database access methods.
- com.dev.trade.exception
  - DBException - Custom exception class that is thrown for all database exceptions.
- com.dev.trade.servlet
  - TradeDispatcherServlet - Dispatches all the requests to the JSPs after performing required database functions.

The Online Brokerage also includes the following JSP pages:

- login.jsp - The login page of the application.
- error.jsp - The default error page of the application.
- register.jsp - The user registration page.
- stocks.jsp - The Available Stocks page from where the user can buy stocks.
- userstocks.jsp - The user portfolio page which shows the user's stocks. The user can sell stocks from this page.

Back to Top

Tools used

The tools used for developing and building the sample application are:

Apache Ant

Ant is a pure Java build tool. It is used for building the war files and populating the database for the Online Brokerage application. Ant can be downloaded from the following URL:

http://ant.apache.org

Hibernate

At the time of writing this article, Hibernate 3.2 is the latest version available and can be downloaded at the following URL:
http://www.hibernate.org

Additional documentation on Hibernate may be found at the following URL:
http://hibernate.org/5.html
http://www.hibernate.org/hib_docs/v3/reference/en/html/tutorial.html

Download and install Hibernate, the installation directory will be later referred as **<hibernate_home>**.

Back to Top

Sample database

The database used for demonstrating this application is MySQL. The sample database name is **adi** and it consists of the following three tables, STOCKS, USERS and USERSTOCKS. The fields for each of these tables are described below.

Table Name	Fields
STOCKS	ID (PRIMARY KEY) NAME PRICE
USERS	USERID (PRIMARY KEY) NAME PASSWORD ADDRESS CASH
USERSTOCKS	ID (PRIMARY KEY) USERID (PRIMARY KEY) NAME PRICE QUANTITY

The USERSTOCKS table is used to store the stocks that are owned by each user. The USERS and STOCKS tables are used to store the user and stock details. Because this is just a sample application the amount of money that a user has is entered during user registration by the user itself.

The DDL file used for populating this database is **db.sql** and it is located in the <brokerage_home>\sql directory.

Back to Top

The JBoss environment

This section shows you how and where the sample JBoss reference environment was installed so you can map this scenario to your own implementation.

Detailed instructions for installing, configuring, and managing JBoss are provided in the product documentation. Check the product Web site for the most updated documents.

The following list highlights the general tasks you will need to complete to install and configure the initial environment as the starting point for deploying the sample application.

1. Download and install JBoss v4.2.1 as explained in the product documentation guides. From now on the installation directory will be referred as **<jboss_home>**
2. Create a copy of the default JBoss v4.2.1 application server. Copy recursively **<jboss_home>\server\default** to **<jboss_home>\server\<your_server_name>**
3. Start the new server by running the **`run.sh -c <your_server_name>`** command from the **<jboss_home>\bin** directory.

4. Once the server is started, you can verify that it is running by opening a Web browser and pointing it to this URL: http://localhost:8080. You should see the JBoss Welcome window and be able to access the JBoss console.
5. Once the application server is up and running, the next step is to install and configure all the remaining prerequisite software required by the sample application. This step is described in the following section.

Install and configure prerequisite software

In order to build and run the Library application included in this article, you need to install and configure the build tool and the database that is used by the application.

Install the database

As mentioned earlier in the article, this application is using the MySQL database that can be downloaded from the following URL:

http://www.mysql.com

Note: The appropriate version of the MySQL Connector/J should also be downloaded (3.1, 5.0, or 5.1), depending on your MySQL version.

This MySQL connector must also be placed in <JBOSS_HOME>/server/default/lib.

The Installation and configuration of MySQL is fairly intuitive and it is well documented in the MySQL Reference Manual available at the following URL:

http://dev.mysql.com/doc/mysql/en

Note: During the instance configuration I modified the security settings and specified "password" as the password for the root user. This user ID and password will later be used for accessing the database from the sample application.

Create sample database

Once the MySQL instance is configured you need to create the sample database that will be used by the Library application. From a command line, type the following command to start the MySQL monitor:

```
mysql -u root -ppassword
```

Note that there is no blank between the flag -p and the password.

This will bring up the MySQL command interface as shown in the following example:

MySQL monitor interface

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 7 to server version: 4.1.14-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

From the MySQL command interface create the **adi** sample database by typing the following command:

```
mysql> create database adi;
```

Configure Ant

As mentioned before, Apache Ant is used to build the binaries for the Online Brokerage application. If you do not have Ant installed this is a good time for doing it and make sure that **<ant_home>\bin** directory is added to the system's path variable.

Apache Ant can be downloaded from the following URL:

http://ant.apache.org

Configure Hibernate

JBoss comes bundled with Hibernate so there is no need to download separate jars for hibernate.

Back to Top

Build the sample application

The Online Brokerage application included with this article provides an Ant script that you will use in order to build the application and populate the database. Download the sample application by cliking on Online Brokerage.

After extracting the zip file, a brokerage directory is created, throughout the rest of the article we will refer to this directory as **<brokerage_home>**. In that directory open the **build.properties** file and edit the properties to match your environment as shown in the following example:

build.properties

```
#Replace java.home with your jdk directory
java.home=<java_home>

#Replace jboss.home with the parent directory of lib/j2ee.jar
jboss.home=<jboss_home>/server/<your_server_name>

#Replace geronimo.home with the geronimo home directory.
geronimo.home=<geronimo_home>

#Fully qualified name of the JDBC driver class
db.driver=com.mysql.jdbc.Driver

#database url
db.url=jdbc:mysql://localhost:3306/adi

#database userId
db.userid=root

#database password
db.password=password

#script file for creating the tables
sql.file=sql/db.sql

#location of the jdbc driver jar.
driver.classpath=<mysql-connector_home>/mysql-connector-java-3.1.14-bin.jar

#location of the hibernate jars.
dependency.dir=<hibernate_home>/lib
```

Note that the path to the Hibernate jars defined by the **`dependency.dir`** tag are necessary by Geronimo, JBoss comes with it's own copy of Hibernate. Still you will need to copy to that directory the **hibernate3.jar** file located in the **<hibernate_home>** directory.

Important: When specifying the paths in this file, make sure you use forward slash "**/**" otherwise you will get compilation errors.

From a command prompt or shell change directory to <brokerage_home> and run the **ant** command. This will build the war, har and ear files and place them in the **<brokerage_home>\jboss-artefact** directory. The war created by the **ant** build contains a JBoss specific deployment descriptor **jboss-web.xml** file in the WEB-INF directory of the WAR. The HAR contains a JBoss specific **hibernate-service.xml** file in the META-INF directory and the EAR contains a JBoss specific deployment descriptor **jboss-app.xml**. These files are shown below.

jboss-web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
  <context-root>/brokerage</context-root>
  <resource-ref>
      <res-ref-name>jdbc/HibernateDB</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <jndi-name>jdbc/HibernateDS</jndi-name>
  </resource-ref>
</jboss-web>
```

The resource-ref element is used to map the resource referred to by the name **jdbc/HibernateDB** in the web.xml file to the resource with the JNDI name **java:jdbc/ HibernateDS**, for example MySQL datasource.

hibernate-service.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<server>
    <mbean code="org.jboss.hibernate.jmx.Hibernate"
           name="jboss.har:service=Hibernate">
        <attribute name="DatasourceName">java:jdbc/HibernateDS</attribute>

        <attribute name="Dialect">
            org.hibernate.dialect.MySQLDialect
        </attribute>
        <attribute name="SessionFactoryName">
            java:/hibernate/BrokerageSessionFactory
        </attribute>
        <attribute name="CacheProviderClass">
            org.hibernate.cache.HashtableCacheProvider
        </attribute>
         <!-- <attribute name="ScanForMappingsEnabled">true</attribute> -->
        <attribute name="ShowSqlEnabled">true</attribute>
    </mbean>
</server>
```

This file contains the hibernate properties that need to be set. The property names and their functions are:

- **DatasourceName** - The JNDI name of the datasource that Hibernate should use.
- **Dialect** - The SQL Dialect that should be used.
- **SessionFactoryName** - JNDI name to which the session factory should be bound.
- **CacheProviderClass** - The Class of the Cache Provider
- **ShowSqlEnabled** - Show the executed SQL statements in the output.

The **hibernate-service.xml** file should be present in the META-INF directory of the EAR.

jboss-app.xml

```
<!DOCTYPE jboss-app PUBLIC "-//JBoss//DTD J2EE Application 1.4//EN" "http://www.jboss.org/j2ee/dtd/jboss-
app_4_0.dtd">
<jboss-app>
    <module>
        <har>brokerage.har</har>
    </module>
</jboss-app>
```

The **jboss-app.xml** file located in the META-INF directory of the EAR specifies the name of the har file present in the EAR.

The following example shows the deployment descriptor **web.xml** used in this application. The **web.xml** file located in the WEB-INF directory of **`brokerage.war`** is the deployment descriptor where you specify, among other things, the servlets name and default JSP for the application.

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee" version="2.4" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <display-name>brokerage</display-name>
    <servlet>
        <display-name>Trade-Dispatcher</display-name>
        <servlet-name>TradeDispatcher</servlet-name>
        <servlet-class>com.dev.trade.servlet.TradeDispatcherServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>TradeDispatcher</servlet-name>
        <url-pattern>/login</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>TradeDispatcher</servlet-name>
        <url-pattern>/stocks</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>TradeDispatcher</servlet-name>
        <url-pattern>/userstocks</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>TradeDispatcher</servlet-name>
        <url-pattern>/buy</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>TradeDispatcher</servlet-name>
        <url-pattern>/sell</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>TradeDispatcher</servlet-name>
        <url-pattern>/register</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
  <welcome-file>/login.jsp</welcome-file>
    </welcome-file-list>
    <error-page>
     <exception-type>javax.servlet.ServletException</exception-type>
     <location>/error.jsp</location>
    </error-page>

    <resource-ref>
        <res-ref-name>jdbc/HibernateDB</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
        <res-sharing-scope>Shareable</res-sharing-scope>
    </resource-ref>
</web-app>
```

Back to Top

Populate sample data

As we mentioned before, the **db.sql** script is provided to populate the sample data. The location of this file is already defined in the **build.properties** by the tag **sql.file**. To populate the sample data simply run the following command from the **<brokerage_home>** directory.

```
ant populateDB
```

Back to Top

Deploy the sample application

Before you deploy the sample application you need to configure the datasource required by this application. To deploy the datasource configuration in JBoss copy the **mysql-ds.xml** file located in the **<brokerage_home>\plan** directory to the following directory:

<jboss_home>\server\<your_server_name>\deploy

Similarly to the database deployment, to deploy the Online Brokerage application in JBoss, copy the **brokerage.ear** file you just built with Ant to the following directory:

<jboss_home>\server\<your_server_name>\deploy

If JBoss is already started, it will automatically deploy and start the application; otherwise, the application will be deployed and started at the next startup.

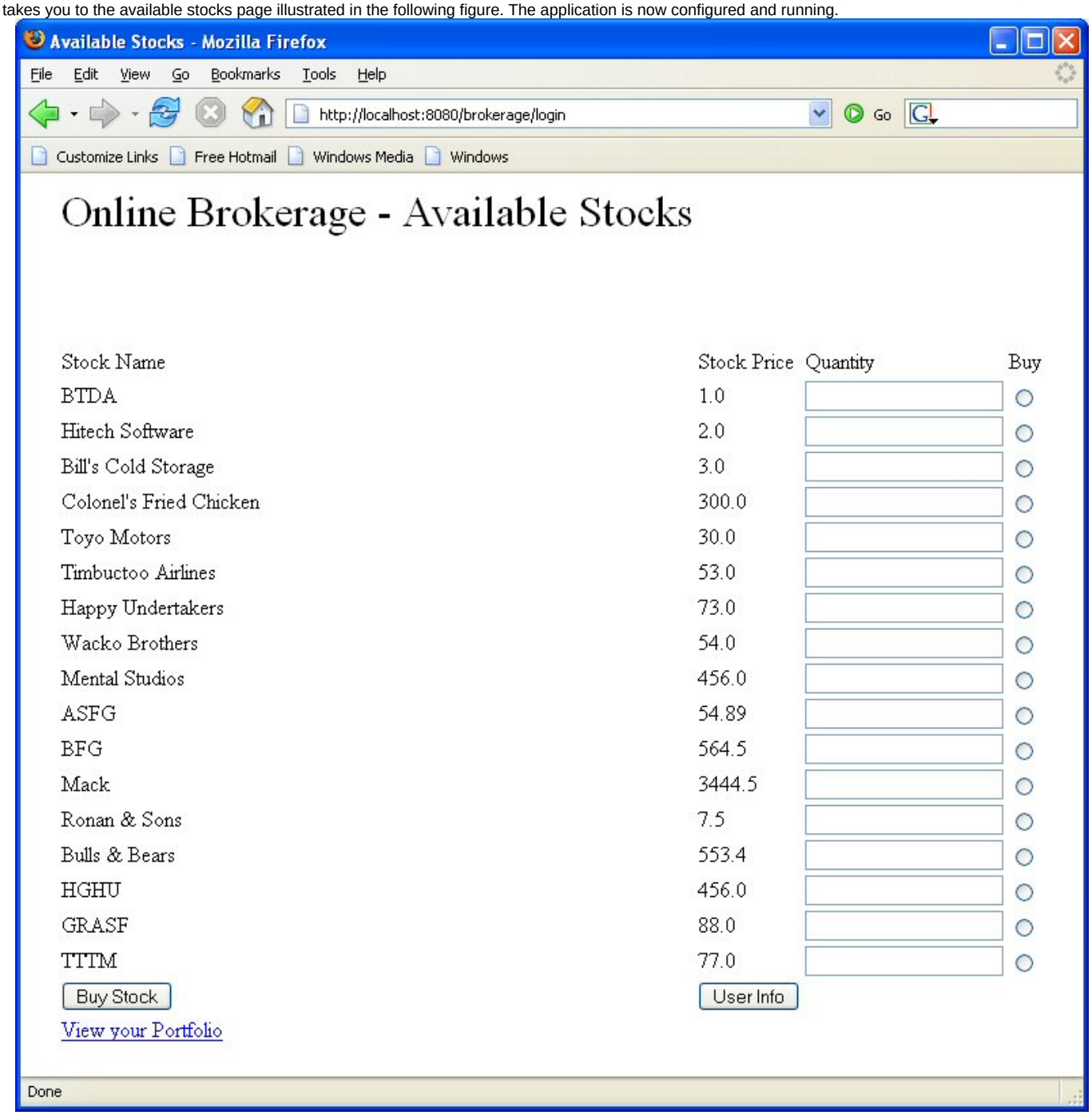
Back to Top

Test the sample application

To test the application, open a Web browser and access the following URL:

http://localhost:8080/brokerage

This brings up the login screen of the Online Brokerage application. Enter j2ee as the user name and password as the password and click on login. This takes you to the available stocks page illustrated in the following figure. The application is now configured and running.



Back to Top

The Geronimo environment

Download and install Geronimo from the following URL:

http://geronimo.apache.org/downloads.html

The release notes available there provide clear instructions on system requirements and how to install and start Geronimo. Throughout the rest of this article we will refer to the Geronimo installation directory as **<geronimo_home>**.

Back to Top

Configure Resources

For running the Online Brokerage application in Geronimo, you will be using the same MySQL database that was used with JBoss. So the first task you need to do in order to prepare the Geronimo environment is to configure the data source.

Configure the datasource

You need to copy the MySQL database driver into the Geronimo repository so that you can refer to it in the data source deployment plan. The Geronimo repository is located in the **<geronimo_home>/repository** directory. Inside this directory, create a new directory called **mysql/jars** and copy the **mysql-connector-java-3.1.14-bin.jar** file into it.

Now, you need to define a data source deployment plan. For your convenience, the sample application already provides a deployment plan called **mysql-geronimo-plan.xml** located in the **<brokerage_home>\plan** directory. The following example shows the content of this deployment plan.

mysql-geronimo-plan.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<connector xmlns="http://geronimo.apache.org/xml/ns/j2ee/connector-1.2">
  <dep:environment xmlns:dep="http://geronimo.apache.org/xml/ns/deployment-1.1">
    <dep:moduleId>
          <dep:groupId>user</dep:groupId>
          <dep:artifactId>database-pool-HibernateDB</dep:artifactId>
          <dep:version>2.0</dep:version>
          <dep:type>car</dep:type>
        </dep:moduleId>
        <dep:dependencies>
          <dep:dependency
            <dep:groupId>mysql</dep:groupId>
                <dep:artifactId>mysql-connector-java</dep:artifactId>
                <dep:version>3.1.14-bin</dep:version>
                <dep:type>jar</dep:type>
          </dep:dependency>
        </dep:dependencies>
  </dep:environment>
  <resourceadapter>
    <outbound-resourceadapter>
      <connection-definition>
        <connectionfactory-interface>javax.sql.DataSource</connectionfactory-interface>
        <connectiondefinition-instance>
          <name>HibernateDS</name>
          <config-property-setting name="Password">password</config-property-setting>
          <config-property-setting name="CommitBeforeAutocommit">false</config-property-setting>
          <config-property-setting name="Driver">com.mysql.jdbc.Driver</config-property-setting>
          <config-property-setting name="ExceptionSorterClass">org.tranql.connector.AllExceptionsAreFatalSorter<
/config-property-setting>
          <config-property-setting name="UserName">root</config-property-setting>
          <config-property-setting name="ConnectionURL">jdbc:mysql://localhost:3306/adi</config-property-
setting>
          <connectionmanager>
            <local-transaction/>
            <single-pool>
              <max-size>10</max-size>
              <min-size>0</min-size>
              <blocking-timeout-milliseconds>5000</blocking-timeout-milliseconds>
              <idle-timeout-minutes>30</idle-timeout-minutes>
              <match-one/>
            </single-pool>
          </connectionmanager>
        </connectiondefinition-instance>
      </connection-definition>
    </outbound-resourceadapter>
  </resourceadapter>
</connector>
```

Now that you have a deployment plan and have copied the drivers the next step is to acutally deploy the datasource connection pool. If you have not yet started Geronimo please do so by running the following command:

<geronimo_home>\bin\geronimo start

To deploy the datasource connection pool run the following command:

<geronimo_home>\bin\deploy --user system --password manager deploy <brokerage_home>\plan\mysql-geronimo-plan.xml ..\repository\org\tranql\tranql-connector-ra\1.3\tranql-connector-ra-1.3.rar

Depending on your environment you should see a confirmation message similar to this one:

```
C:\geronimo-2.0\bin>deploy --user system --password manager deploy \brokerage\plan\mysql-geronimo-plan.xml
..\repository\org\tranql\tranql-connector-ra\1.3\tranql-connector-ra-1.3.rar
    Deployed user/database-pool-HibernateDS/2.0/car
```

Back to Top

Step-by-step migration

Apache Geronimo does not support HAR archives so we will be having all the classes inside a WAR archive. We need to write two classes for making the application work on Geronimo. One is a TransactionManagerLookup class for Geronimo and the other is a utility class HibernateUtil to get the SessionFactory. In addition to this we will need to make some changes to the TradeDispatcherServlet and TradeDAO classes.

As a first step we will see the changes that need to be made to the application as illustrated in the following list.

#TradeDAO
#HibernateUtil
#TradeDispatcherServlet

TradeDAO

In JBoss the HibernateMBean will create and bind the SessionFactory to the Global JNDI Context. So we can obtain the SessionFactory through a simple lookup. We can then get the session from the SessionFactory.

`TradeDAO.java` is located in the `<brokerage_home>/src/com/dev/trade/dao` directory.

Excerpt from TradeDAO.java for JBoss

```
package com.dev.trade.dao;

import java.sql.Connection;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Set;

import javax.naming.InitialContext;
import javax.naming.NamingException;

import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;

import com.dev.trade.bo.Stock;
import com.dev.trade.bo.User;
import com.dev.trade.bo.UserStock;
import com.dev.trade.exception.DBException;
import com.dev.trade.util.HibernateUtil;

public class TradeDAO {

        SessionFactory factory  = null;
        Session session = null;

        public TradeDAO() throws Exception {

                try {
                        InitialContext ctx = new InitialContext();
                    factory  = (SessionFactory)ctx.lookup("java:hibernate/BrokerageSessionFactory");

                } catch (NamingException e1) {
                        // TODO Auto-generated catch block
                        e1.printStackTrace();
                }
        }

        public User getUserByUserId(String userId) throws DBException {

                session = factory.getCurrentSession();
                Query q = session.createQuery("from User u where u.userId=:userId");
                q.setString("userId", userId);
                return (User) q.uniqueResult();

        }
        ...
```

In Geronimo we will write a new utility class to create the SessionFactory, this class will be the HibernateUtil class. It has a method getCurrentSession() that will return the session.

Modify the first part of `TradeDAO.java` to match the changes in the following excerpt.

Excerpt from TradeDAO.java for Geronimo

```
package com.dev.trade.dao;

import java.sql.Connection;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Set;

import org.hibernate.Query;
import org.hibernate.Session;

import com.dev.trade.bo.Stock;
import com.dev.trade.bo.User;
import com.dev.trade.bo.UserStock;
import com.dev.trade.exception.DBException;
import com.dev.trade.util.HibernateUtil;

public class TradeDAO {


      Session session = null;

       public TradeDAO() throws Exception {

        }

       public User getUserByUserId(String userId) throws DBException {

            session = HibernateUtil.getCurrentSession();
                Query q = session.createQuery("from User u where u.userId=:userId");
                q.setString("userId", userId);
              return (User) q.uniqueResult();

       }
       ...
```

Do a search and replace, look for the following string:

`factory.getCurrentSession()`

and replace it with

`HibernateUtil.getCurrentSession()`

You should count 9 replacements.

This difference in obtaining the session will be the main change in the codebase for the application to run in Apache Geronimo.

HibernateUtil

As mentioned above we will use this class to create the SessionFactory and provide the application with Hibernate sessions. The source code for the class is given below

HibernateUtil.java for Geronimo

```
package com.dev.trade.util;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.cfg.Configuration;

/**
 * Configures and provides access to Hibernate sessions, tied to the
```

```
 * current thread of execution.  Follows the Thread Local Session
 * pattern, see {@link http://hibernate.org/42.html}.
 */
public class HibernateUtil {

    /** location of the Hibernate Configuration File */
    private static String CONFIG_FILE_LOCATION = "hibernate.cfg.xml";

    /** Holds a single instance of Session */
    private static final ThreadLocal threadLocal = new ThreadLocal();

    /** The single instance of hibernate configuration */
    private static final Configuration cfg = new Configuration();

    /** The single instance of hibernate SessionFactory */
    private static org.hibernate.SessionFactory sessionFactory;

    /**
     * Returns the ThreadLocal Session instance.  Lazy initialize
     * the <code>SessionFactory</code> if needed.
     *
     *  @return Session
     *  @throws HibernateException
     */
    public static Session getCurrentSession() throws HibernateException {
        Session session = (Session) threadLocal.get();

        if (session == null || ! session.isConnected()) {
            if (sessionFactory == null) {
                try {
                    cfg.configure(CONFIG_FILE_LOCATION);
                    sessionFactory = cfg.buildSessionFactory();
                }
                catch (Exception e) {
                    System.err.println("%%%% Error Creating SessionFactory %%%%");
                    e.printStackTrace();
                }
            }
            session = sessionFactory.openSession();
            threadLocal.set(session);
        }

        return session;
    }

    /**
     *  Close the single hibernate session instance.
     *
     *  @throws HibernateException
     */
    public static void closeSession() throws HibernateException {
        Session session = (Session) threadLocal.get();


        if (session != null) {
            session.close();
        }
    }



}
```

`HibernateUtil.java` is located in the `<brokerage_home>/src/com/dev/trade/util` directory. For your convenience, a copy of this file is already provided with the sample application.

TradeDispatcherServlet

In this class there will also be a difference in getting the SessionFactory. In JBoss we get it from the JNDI Context but in Geronimo we will get it through the utility class.

doGet method for JBoss

```
protected void doGet(HttpServletRequest request,
   HttpServletResponse response) throws ServletException, IOException {

   Session hsession = null;
   try {
         InitialContext ctx = new InitialContext();
         SessionFactory factory  =  (SessionFactory)ctx.lookup("java:hibernate/BrokerageSessionFactory");
         hsession = factory.openSession();
       } catch (NamingException e1) {
           e1.printStackTrace();
       }

    Transaction tr = hsession.beginTransaction();
```

`TradeDispatcherServlet.java` is located in the `<brokerage_home>/src/com/dev/trade/servlet` directory. Replace the doGet method for the one shown in the following example.

doGet method for Geronimo

```
protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {

    Transaction tr = HibernateUtil.getCurrentSession().getTransaction();
    tr.begin();
```

Hibernate comes with transaction manager lookup classes for many application servers. Unfortunately Hibernate 3.2 does not have a lookup class specific for Apache Geronimo, so we need to write our own. The code for the Geronimo specific transaction manager lookup is shown in the following example.

GeronimoTransactionManagerLookup

```
package org.hibernate.transaction;

import java.util.Iterator;
import java.util.Properties;
import java.util.Set;

import javax.transaction.TransactionManager;
import org.hibernate.HibernateException;
import org.hibernate.transaction.TransactionManagerLookup;

import org.apache.geronimo.gbean.AbstractName;
import org.apache.geronimo.gbean.AbstractNameQuery;
import org.apache.geronimo.kernel.Kernel;
import org.apache.geronimo.kernel.KernelRegistry;
import org.apache.geronimo.kernel.proxy.ProxyManager;

public class GeronimoTransactionManagerLookup implements
        TransactionManagerLookup {

    public Object getTransactionIdentifier(Transaction arg0) {
                return null;
    }

    public static final String UserTransactionName = "java:comp/UserTransaction";

    public TransactionManager getTransactionManager(Properties props) throws HibernateException {
       /*
         * try { Kernel kernel = KernelRegistry.getSingleKernel(); ProxyManager
         * proxyManager = kernel.getProxyManager(); AbstractNameQuery query =
         * new AbstractNameQuery(TransactionManager.class.getName()); Set names =
         * kernel.listGBeans(query); AbstractName name = null; for (Iterator it =
         * names.iterator(); it.hasNext();) name = (AbstractName) it.next();
         * Object transMg = (Object) proxyManager.createProxy(name,
         * TransactionManager.class); return (TransactionManager)transMg; }catch
         * (Exception e) { e.printStackTrace(); System.out.println(); throw new
         * HibernateException("Geronimo Transaction Manager Lookup Failed", e); }
         */
       try {
       Kernel kernel = KernelRegistry.getSingleKernel();
       AbstractNameQuery query = new AbstractNameQuery(TransactionManager.class.getName ());
       Set<AbstractName> names = kernel.listGBeans(query);
       if (names.size() != 1) {
           throw new IllegalStateException("Expected one transaction manager, not " + names.size());
       }
       AbstractName name = names.iterator().next();
       TransactionManager transMg = (TransactionManager)
       kernel.getGBean(name);
       return (TransactionManager)transMg;
       } catch (Exception e) {
           e.printStackTrace();
           System.out.println();
           throw new HibernateException("Geronimo Transaction Manager Lookup Failed", e);
       }
   }

    public String getUserTransactionName() {
        return UserTransactionName;
    }
}
```

For your convenience this class is already provided in the `<brokerage_home>/TransactionManager` directory. Create the following directory structure and copy the `GeronimoTransactionManagerLookup.java` there.

`*brokerage_home>/src/org/hibernate/transaction`

Now you need to create the hibernate configuration file hibernate-cfg.xml. Inside this file you will specify the required hibernate configuration attributes. Note that the `hibernate-service.xml` in JBoss is for doing the same thing. For your convenience this file is also provided with the sample application in the `<brokerage_home>/hibrenate` directory.

hibernate-cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
 <session-factory>
 <!-- properties -->
 <property name="connection.datasource">java:comp/env/jdbc/HibernateDB</property>
 <property name="hibernate.transaction.manager_lookup_class">
    org.hibernate.transaction.GeronimoTransactionManagerLookup
 </property>

  <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

 <!-- Disable the second-level cache  -->
<property name="hibernate.cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
<property name="hibernate.current_session_context_class">org.hibernate.context.JTASessionContext</property>
 <!-- Echo all executed SQL to stdout -->
 <property name="hibernate.show_sql">true</property>

 <!-- Drop and re-create the database schema on startup -->
<!--<property name="hibernate.hbm2ddl.auto">create</property> -->

 <!-- mapping files -->
 <mapping resource="Stock.hbm.xml"/>
 <mapping resource="UserStock.hbm.xml"/>
 <mapping resource="User.hbm.xml"/>

 </session-factory>
</hibernate-configuration>
```

Additional details for the specific functions of each of these properties can be found in the hibernate manual.

One last step before building is to create a **geronimo-web.xml** file which is the Geronimo specific deployment descriptor as illustrated in the following example. Once again, for your convenience this file is also provided with the sample application in the `<brokerage_home>/web/descriptors/geronimo` directory.

geronimo-web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://geronimo.apache.org/xml/ns/j2ee/web-1.1" xmlns:naming="http://geronimo.apache.org/xml/ns
/naming-1.1">
    <dep:environment xmlns:dep="http://geronimo.apache.org/xml/ns/deployment-1.1">
            <dep:moduleId>
                    <dep:groupId>BrokerageApp</dep:groupId>
                        <dep:artifactId>MySqlDS</dep:artifactId>
                        <dep:version>2.0</dep:version>
                        <dep:type>car</dep:type>
                </dep:moduleId>

                <dep:dependencies>
                    <dep:dependency>
                        <dep:groupId>user</dep:groupId>
                            <dep:artifactId>database-pool-HibernateDB</dep:artifactId>
                            <dep:version>2.0</dep:version>
                            <dep:type>car</dep:type>
                        </dep:dependency>
                </dep:dependencies>

                <dep:hidden-classes>
                    <dep:filter>org.springframework</dep:filter>
                        <dep:filter>META-INF/spring</dep:filter>
                        <!--dep:filter>antlr</dep:filter-->
                </dep:hidden-classes>
        </dep:environment>

        <context-root>/brokerage</context-root>

    <resource-ref>
        <ref-name>jdbc/HibernateDB</ref-name>
        <resource-link>HibernateDS</resource-link>
    </resource-ref>
</web-app>
```

The `<hidden-classes>` element is for preventing certain classes that come with Apache Geronimo from being visible to the war classloader which may result in version problems.

Back to Top

Build the migrated sample application

To build the application run the following commands

- Add the hibernate jar to the classpath. You will now need to build hibernate with the `GeronimoTransactionManagerLookup` class. If you have not downloaded the hibernate source you can compile the class after putting the hibernate jar in the classpath and then manually add that class to the hibernate jar file.
  `set CLASSPATH=%CLASSPATH%;<hibernate_home>/lib/hibernate3.jar`
- Add the Geronimo kernel to your classpath
  `set CLASSPATH=%CLASSPATH%;<geronimo_home>/lib/geronimo-kernel-2.1.4.jar`

- To add the class manually, although not needed for this particular sample, you can use the tool of your preference and add the GeronimoTransactionManagerLookup.class to the `org\hibernate\transaction` directory in the `hibernate_home>/lib/hibernate3.jar` file.

- Now build the migrated application by running the following command:

`<brokerage_home>/ant war`

Note that we are just building the **war** target because the Online Brokerage sample application is just a web application. In the JBoss section we used an **ear** to package the **war** and the **har**, this last one is not supported by Apache Geronimo.

This will create a **brokerage.war** file in the **<brokerage_home>/geronimo-artefact** directory.

The **<brokerage_home>/solutions** directory has the source files already migrated to compile for and run in Apache Geronimo.

Back to Top

Deploy the migrated sample application

To deploy the migrated Online Brokerage application, make sure the Geronimo server is up and running and run the following command:

```
deploy --user system --password manager deploy <brokerage_home>/geronimo-artefact/brokerage.war
```

Once the application is deployed, open a Web browser and access the following URL:

http://localhost:8080/brokerage

Login with the same user name and password you used when testing the application from JBoss.

Back to Top

Summary

This article has shown you how to migrate a sample application that uses Hibernate as its O/R mapping layer, from JBoss to the Apache Geronimo application server. Some of the features/functionalities already provided by JBoss may not yet be implemented in Geronimo, as a consequence some minor additional coding may be required but the overall migration complexity is low.

Back to Top