Embedding Cloudscape in BEA WebLogic Server

A Cloudscape White Paper

Phil Lopez

July 19, 2000

Synopsis

This paper describes how the Cloudscape DBMS, a 100% Pure Java™ object-relational database management system, may be embedded inside BEA WebLogic Server. The method of embedding is similar to that of the Informix Cloudconnector product. The source code provided is based on Cloudscape 3.5 and WebLogic 5.1, but the principles can be applied to other versions.

This document discusses the motivation behind embedding Cloudscape within WebLogic Server, and the process of integration. The general steps to embed Cloudscape within WebLogic Server are:

1. Implement the WebLogic server startup and shutdown interfaces to boot and shutdown the Cloudscape DBMS respectively.
2. Modify the weblogic.properties file to register the startup/shutdown class.
3. Verify the configuration by connecting to a Cloudscape database via WebLogic using Cloudscape's simplification of the WebLogic multi-tier JDBC driver.
4. Configure and use a WebLogic data source and connection pool.

Requirements

This paper assumes that the reader has installed following software:

- BEA WebLogic Server (An evaluation version is available from www.bea.com)
- Informix Cloudscape (An older evaluation version comes with BEA WebLogic Server; the latest free developer version can be downloaded from www.cloudscape.com)

It is expected that the reader is somewhat familiar with both products, has successfully started both products and, preferably, worked through some tutorials. The following documents are particularly relevant to this paper:

- Cloudscape Developer's Guide
- Cloudscape Server and Administration Guide
- WebLogic Administrator Guides
- WebLogic Developer Guides
- WebLogic API Reference

The reader is also expected to have some knowledge of Java, JDBC, and J2EE technologies such as servlets and JNDI. Tutorials for these technologies are available at www.java.sun.com.

Introduction

To use Cloudscape in a multi-user environment, it must be embedded within a server framework. The server framework manages the connectivity between the DBMS and its clients, possibly through a non-JDBC style of communication such as a servlet or EJB. Cloudscape must always be embedded within either an application or a server framework.

When embedded within an application, multiple internal client threads (those in the same JVM as Cloudscape) access a Cloudscape database through the local Cloudscape JDBC driver. However, when embedded within a server framework, multiple external clients access the same Cloudscape database by connecting to the server framework.

Informix currently provides two server frameworks for Cloudscape:

- **RmiJdbc** is a simple and free server framework included in the basic Cloudscape product, and is suitable for small applications and prototyping.
- **Cloudconnector** is an add-on product that provides a complete server framework based on BEA WebLogic Server, and includes features such as connection pooling, support for servlets, and secure connectivity via SSL.

This paper describes how to customize BEA WebLogic Server to provide a powerful server framework for Cloudscape in a way similar to the Cloudconnector product. Sites that already have BEA WebLogic server installed, or that desire to use a WebLogic version more recent than that provided by Cloudconnector, can follow the instructions herein to embed Cloudscape within WebLogic.

Motivation

There are a number of advantages to embedding a DBMS within an application server such as WebLogic Server (and its JVM) as opposed to maintaining data in an enterprise-scale database server outside the application server (and usually on a different host). These advantages include:

- *Performance*: For some applications and deployments, the cost of communicating between processes and hosts, as between an application server and a database server, offsets any performance advantages gained by using a highly scalable enterprise DBMS.

- *Cost of Ownership*: An enterprise DBMS is generally more expensive to purchase and administer than Cloudscape, which is practically a zero administration system. For smaller deployments with limited resources (e.g. a workgroup Intranet), this is an important consideration.

- *Scalability*: At certain stages of the lifecycle for a multi-tier application (especially development and QA), it may make sense to use Cloudscape embedded in the application server rather than purchase and manage an enterprise DBMS. Such an approach permits scalability downwards rather than just upwards.

Note on Application Server Clusters

It is important to note that, to maintain integrity, Cloudscape should not be embedded into a cluster of application servers unless the database is static and it is embedded in every server. This is because a single Cloudscape database cannot be managed by more than one Cloudscape DBMS (i.e. single JVM) instance at any time.

The Cloudscape DBMS instance is started when the local Cloudscape JDBC driver (`COM.cloudscape.core.JDBCDriver`) is loaded (e.g., via a `Class.forName()` invocation). Consequently, if a cluster of application servers (i.e. multiple JVMs) is to make use of a Cloudscape database, they will need to each have their own copy of that database. If the databases need to remain identical at all times, they should be static (i.e. read-only). If the database cannot be static, consider the following alternatives:

- Maintain a single Cloudscape database and instance by having the application servers connect to a Cloudscape server framework (e.g. RmiJdbc, Cloudconnector, or WebLogic when prepared in the manner described in this paper). Depending on the level of interaction with the database, however, this may mitigate the benefits of clustering the application servers.
- Consider using an enterprise-level DBMS (e.g. Informix Internet Foundation).
- Consider redesigning the application so that an asynchronous database synchronization technology (such as the Informix Cloudsync product) may be used to synchronize multiple databases.

Configuration

Background

Cloudscape is an embeddable database written entirely in Java. In fact, it must always be embedded in order to run. For single user applications, Cloudscape is usually embedded in the application itself. However, for multi-user applications, Cloudscape must be embedded into another application that manages the multi-user connectivity to the database; this is known as a "server framework." This is a very modular and flexible architecture. RmiJdbc and Cloudconnector are both examples of server frameworks that embed Cloudscape.

In this paper, we describe the process of embedding Cloudscape inside the BEA WebLogic application server (see Figure 1) in a similar manner to the Cloudconnector product.

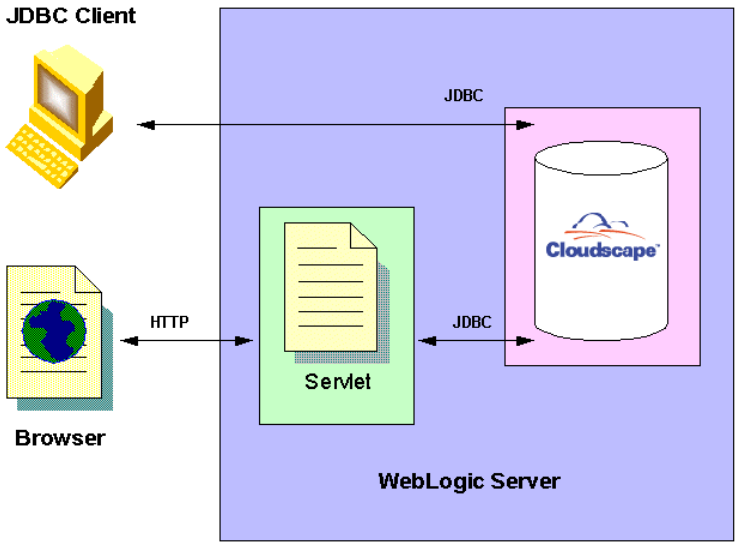

Figure 1: Cloudscape Embedded in WebLogic Server

In a client/server DBMS such as the Informix Dynamic Server, the DBMS itself provides the database engine (i.e. data storage and access) as well as multi-user connectivity to that database engine (e.g. listening on TCP ports). When an application server such as BEA WebLogic connects via JDBC to a client/server DBMS, it is communicating with a DBMS that is already running.

Starting and Stopping Cloudscape

With an embeddable database such as Cloudscape, however, the DBMS is *not* always running. Like any other application, it needs to be started (or *booted*) before it can be used. Cloudscape boots the first time that the local JDBC driver is loaded in a JVM.

The following statement loads the JDBC driver:

```
Class.forName("COM.cloudscape.core.JDBCDriver");
```

Similarly, Cloudscape is shutdown in the following manner:

```
DriverManager.getConnection("jdbc:cloudscape:;shutdown=true");
```

The issue, then, is when to start and stop Cloudscape.

One option is to let the application server load the local Cloudscape JDBC driver when necessary and to pool its connections. This is the typical behavior of an application server for client/server databases. For example, in BEA WebLogic, one can define a connection pool by specifying a "weblogic.jdbc.connectionPool" property. WebLogic will then automatically load the JDBC driver and manage a pool of connections to the given DBMS and database.

The problem with this approach is that, while the Cloudscape system will boot when the JDBC driver is loaded for the first time, it will never be explicitly shut down (i.e. through a call to `getConnection("jdbc:cloudscape:;shutdown=true")`. Consequently, the next time it is booted, Cloudscape must perform the standard recovery procedures after any abnormal termination.

The preferable alternative is to start and stop Cloudscape as part of the startup and graceful shutdown sequence of the application server. Connection pooling can still be set up through the usual mechanism; nothing untoward occurs if the application server attempts to re-load the JDBC driver class.

The next section describes how to configure WebLogic to start and stop Cloudscape from the application server startup and shutdown sequences.

WebLogic Properties for Startup and Shutdown Actions

BEA WebLogic can be directed to invoke user-provided code at startup and shutdown (see http://www.weblogic.com/docs51/classdocs/API_startup.html). This is achieved through the provision of classes that implement the following two interfaces:

- weblogic.common.T3StartupDef
- weblogic.common.T3ShutdownDef

Classes that implement these interfaces can be specified in the weblogic.properties file (typically located in the directory into which WebLogic was installed). In the weblogic.properties file, set the properties *weblogic.system.startupClass* and *weblogic.system.shutdownClass* to point to classes implementing T3StartupDef and T3ShutdownDef, respectively. For example, you could add the following lines to weblogic.properties if using the sample class provided in this document (WebLogicCloduscapeServierTie):

```
weblogic.system.startupClass.CloudscapeStartup=WebLogicCloudscapeServerTie
weblogic.system.shutdownClass.CloudscapeShutdown=WebLogicCloudscapeServerTie
```

The token following the WebLogic property name (i.e. “CloudscapeStartup” and “CloudscapeShutdown” in the example above) is the “virtual name” that WebLogic uses to identify the action and to relate other directives (e.g. “initialArgs”). For example, to require that Cloudscape start successfully before WebLogic can start, add the following property:

```
weblogic.system.startupFailureIsFatal.CloudscapeStartup=true
```

The weblogic.properties file can also be used to specify the Cloudscape system home directory (i.e. where the log and databases are stored). This is done by the following property:

```
java.system.property.cloudscape.system.home=c:\\cloudscape_data
```

Make sure that the path (i.e. directory) exists before starting WebLogic. If the server is running on Windows, double backslashes should be used in the path since “\” is the usual escape character. A single forward slash (e.g. “c:/cloudscape_data”) can be used instead.

The WebLogicCloudscapeServerTie Class

For convenience, we have implemented both T3StartupDef and T3ShutdownDef interfaces in the same class, “WebLogicCloudscapeServerTie.” The full source code for this class is provided below.

The **T3StartupDef** interface defines two methods:

```
void setServices(T3ServicesDef services)
String startup(String name, Hashtable ht)
```

As can be expected, the `startup()` method is invoked on the startup class as part of the WebLogic server startup process. The `setServices()`method is invoked prior to `startup()` to provide the startup class with access to WebLogic services such as logging.

The **T3ShutdownDef** interface is identical to T3StartupDef except that it defines a `shutdown()` method instead of `startup()`.

A class that implements either or both of the interfaces must have a public no-arguments constructor.

```
// File: WebLogicCloudscapeServerTie.java

import java.util.Hashtable;
import java.sql.SQLException;
import java.sql.DriverManager;
import weblogic.common.LogServicesDef;
import weblogic.common.T3ServicesDef;
import weblogic.common.T3StartupDef;
import weblogic.common.T3ShutdownDef;

public class WebLogicCloudscapeServerTie
    implements T3StartupDef,
               T3ShutdownDef
{
    private T3ServicesDef services;
    private LogServicesDef log;

    public WebLogicCloudscapeServerTie() {} // explicit default constructor

    public void setServices(T3ServicesDef services)
    {
        this.services = services;
        this.log      = services.log();
    } // Method: void setServices(T3ServicesDef)

    public String startup(String name, Hashtable args)
        throws Exception
    {
        try
        {
            // Log that we are starting Cloudscape
            log.log("Starting Cloudscape...");

            // Load the local JDBC driver; this will boot Cloudscape
            Class.forName("COM.cloudscape.core.JDBCDriver").newInstance();

            // Return a string to be added to the WebLogic log
            return "Cloudscape started successfully.";
        }
        catch (Throwable t)
        {
            // Something went wrong, so write it to the log.
            log.log(t.getMessage());
            return "** Error ** Cloudscape did not start!";
        }
    } // Method: String startup(String, Hashtable)

    public String shutdown(String name, Hashtable args)
        throws Exception
    {
                // Log that we are terminating Cloudscape
        log.log("Terminating Cloudscape...");

        // First see if the Cloudscape JDBC driver is actually loaded
        if (DriverManager.getDriver("jdbc:cloudscape:") != null)
        {
            try
```

```
            {
                // Issue a getConnection("jdbc:cloudscape:;shutdown=true")
                // NB: Cloudscape will raise a SQLException at shutdown
                try
                {
                    DriverManager.getConnection(
                        "jdbc:cloudscape:;shutdown=true");
                }
                catch (SQLException sqlEx)
                {
                    String sqlState = sqlEx.getSQLState();

                    // We're expecting an exception to indicate that the
                    // database has been shutdown. This corresponds to an
                    // exception with SQLState of "XJ015"
                    if ((sqlState == null) || (!sqlState.equals("XJ015")))
                    {
                        sqlEx.printStackTrace();
                        log.log("SQL Exception: " + sqlEx.toString());
                        log.log("SQL State: " + sqlEx.getSQLState());
                        log.log("SQL Error code: " + sqlEx.getErrorCode());
                        throw sqlEx;
                    }
                }

                // Return a string to be added to the WebLogic log
                return "Cloudscape shut down successfully.";
            }
            catch (Throwable t)
            {
                // Something went wrong, so write it to the log.
                log.log(t.toString());
                log.log(t.getMessage());
                return "** Error ** Cloudscape did not shutdown cleanly!";
            }
        }
        else
        {
            // Cloudscape driver isn't loaded
            return "** Warning ** Cloudscape driver was not loaded";
        }
    } // Method: String shutdown(String, Hashtable)

} // Class: WebLogicCloudscapeServerTie
```

For simplicity, this example does not support Cloudscape user authentication. If the "cloudscape.connection.requireAuthentication" property has been set to "true" in the cloudscape.properties file then the username and password must be provided to the getConnection() request to shutdown the server. The username and password can be specified in the weblogic.properties file using the "weblogic.system.startupArgs" property and then extracted from the Hashtable argument to `shutdown()`.

After compiling the above Java code (which requires having the imported libraries in the classpath), the resulting WebLogicCloudscapeServerTie.class file should be placed in the classpath for the WebLogic server, along with the cloudscape.jar file (unless using the evaluation version of Cloudscape that already comes with WebLogic Server). If necessary, modify the classpath settings in the startup scripts for WebLogic (e.g. startWebLogic.cmd or startWebLogic.sh) . One location for WebLogicCloudscapeServerTie.class is the "serverclasses" directory (e.g. C:\weblogic\myserver\serverclasses). Refer to the BEA WebLogic documentation for more information about WebLogic classpath settings.

Verifying the Configuration

As configured above, WebLogic provides a server framework for Cloudscape. There are several architectural possibilities with an application server such as WebLogic, including:

- JDBC client to WebLogic, which then passes through the JDBC calls
- Thin client (e.g. web browser) connected to a servlet that calls JDBC
- Thin client connected to a servlet connected to EJBs that call JDBC

After first testing that Cloudscape is properly embedded in WebLogic, we will describe the process of acquiring a database connection for JDBC clients as well as internal clients (e.g. servlets, EJBs). The following sections assume that both WebLogic and Cloudscape are running.

Starting and Stopping WebLogic

Start WebLogic Server in the usual way. Examine the logs to determine whether Cloudscape was booted successfully. Shut down WebLogic (e.g. through the console application, or through the weblogic.Admin class) rather than just terminating the server JVM. Again, examine the logs to ensure that Cloudscape was shutdown cleanly.[1]

Verifying Connectivity

WebLogic provides a utility to test database connectivity called "t3dbping." To verify that Cloudscape is running properly within WebLogic, use the following command after setting the classpath for WebLogic (and started the WebLogic server):

```
java utils.t3dbping t3://localhost:7001 "" ""   \
    COM.cloudscape.core.JDBCDriver              \
    jdbc:cloudscape:testme;create=true
```

If successful, the result should begin as follows:

```
Connecting to WebLogic with the WebLogic JDBC Driver

Success!!!
```

[1] Note: In the evaluation version of Cloudscape provided with WebLogic Server 5.1, Cloudscape appears to not shut down cleanly, although it most probably did. This is because getSQLState() returns NULL rather than "XJ015." This problem disappears when using a more recent version of Cloudscape.

```
You can connect to the database in your WebLogic JDBC program using:

import java.sql.*;
import java.util.Properties;
import weblogic.common.*;

T3Client   t3   = null;
Connection conn = null;

try {
  t3 = new T3Client("t3://localhost:7001");
  t3.connect();
…
```

WebLogic JDBC Driver

WebLogic provides a server framework for Cloudscape. One of the features of WebLogic is its own JDBC driver that provides multi-tier JDBC access. What this means is that client applications connect via JDBC to the WebLogic server, which then passes the JDBC calls on to the real DBMS. This facility can be used to provide multi-user JDBC access (i.e. multiple external clients) to a Cloudscape database.

The JDBC driver provided with BEA WebLogic is the “weblogic.jdbc.t3.Driver” class. The connection URL to be used with this driver should actually specify the JDBC driver to be used by WebLogic in the middle tier, along with any parameters for that driver. For example, to access a Cloudscape database “testDB”, the following URL is required:

```
jdbc:weblogic:t3?weblogic.t3.serverURL=t3://localhost:7001
&weblogic.t3.driverClassName=COM.cloudscape.core.JDBCDriver
&weblogic.t3.driverURL=jdbc.cloudscape:testDB
```

This is a fairly complex URL already, and it does not even include optional arguments to Cloudscape. For this reason, Informix provides a class that translates a relatively simple Cloudscape URL into the WebLogic JDBC driver equivalent. This class is COM.cloudscape.core.WebLogicDriver, and is provided in the Cloudscape “client.jar” archive.

Using the WebLogicDriver class, the code to connect to a Cloudscape database via WebLogic is:

```
Class.forName("COM.cloudscape.core.WebLogicDriver");
DriverManager.getConnection("jdbc:cloudscape:weblogic:testDB;create=true");

Class.forName("COM.cloudscape.core.WebLogicDriver");
Connection conn = DriverManager.getConnection(
                     "jdbc:cloudscape:weblogic:testDB;create=true");

// Use the connection and then close it
conn.close();
```

Setting up a WebLogic Data Source and Connection Pool

To access a Cloudscape database from within a WebLogic application component (e.g. a servlet or Enterprise JavaBean), a data source should be created for that database by defining a WebLogic connection pool in the "weblogic.properties" file. A DataSource object is a factory for Connection objects and, as such, supports the notion of a connection pool. Typically, a DataSource is registered with a JNDI service in the application server so as to decouple the connection pool parameters from the data source name used by applications.

The traditional client/server approach is to connect to the database at the start of the application (which is an expensive operation), and then release that connection at application termination. However, when a connection pool is available, the tactic is to acquire a connection from the connection pool just before database activity, and then release that connection back into the pool just after the database activity. Since the connection is not held while the application is performing non-database activity, it can be shared with other tasks.

In WebLogic Server, one must separately define both the DataSource object and the connection pool resource in the "weblogic.properties" file, as in the following example:

```
weblogic.jdbc.TXDataSource.TestCloudscapeSource=TestCloudscapePool

weblogic.jdbc.connectionPool.TestCloudscapePool=\
        url=jdbc:cloudscape:demo;upgrade=true,\
        driver=COM.cloudscape.core.JDBCDriver,\
        loginDelaySecs=1,\
        initialCapacity=2,\
        maxCapacity=10,\
        capacityIncrement=1,\
        allowShrinking=true,\
        shrinkPeriodMins=15,\
        props=user=none;password=none;server=none

weblogic.allow.reserve.weblogic.jdbc.connectionPool.TestCloudscapePool=\
     everyone
```

It is necessary to specify who may access the connection pool via the "weblogic.allow.reserve" property otherwise WebLogic reports a run-time error when trying to acquire a connection from the pool.

If the "initialCapacity" attribute is defined to be a value greater than zero, then the connections to the specified database (e.g. "demo") will be initialized during server startup. This may occur prior to the `startup()` method of the WebLogicCloudscapeServerTie class being invoked. However, this is not a problem, as Cloudscape is booted the first time the COM.cloudscape.core.JDBCDriver, and subsequent attempts to load the driver do not cause problems. Importantly, the `shutdown()` method of the class is still invoked during WebLogic shutdown, so the databases can be closed properly.

When WebLogic is started, you should see the following output:

```
Tue May 09 17:03:33 PDT 2000:<I> <JDBC Pool> Creating connection pool
 TestCloudscapePool with:
```

```
{poolName=TestCloudscapePool, loginDelaySecs=1, maxCapacity=10,
 props=user=none;password=none;server=none, allowShrinking=true,
 driver=COM.cloudscape.core.JDBCDriver,
 aclName=weblogic.jdbc.connectionPool.TestCloudscapePool,
 capacityIncrement=1, initialCapacity=2,
 url=jdbc:cloudscape:demo;upgrade=true, shrinkPeriodMins=15}
Delaying 1 seconds before making a TestCloudscapePool pool connection.
Tue May 09 17:03:43 PDT 2000:<I> <JDBC Pool> Connection for pool
 "TestCloudscapePool" created.
Delaying 1 seconds before making a TestCloudscapePool pool connection.
Tue May 09 17:03:45 PDT 2000:<I> <JDBC Pool> Connection for pool
 "TestCloudscapePool" created.
Tue May 09 17:03:45 PDT 2000:<I> <JDBC Init> Creating Tx DataSource named
 TestCloudscapeSource for TestCloudscapePool pool
```

If a Cloudscape error occurs when WebLogic starts, you may need to examine the "cloudscape.LOG" file for more information. This log file is in the directory specified by the "cloudscape.system.home" property.

The following provides the salient Java statements required to make use of the connection pool from within WebLogic:

```
// Required classes
import java.sql.Connection;
import javax.sql.DataSource;
import javax.naming.InitialContext;
import javax.naming.Context;

// Setup an initial JNDI context
Context ctx = new InitialContext();

// Get the "TestCloudscapeSource" data source from the JNDI directory
DataSource ds = (javax.sql.DataSource) ctx.lookup("TestCloudscapeSource");

// Acquire a connection from the pool
Connection conn = ds.getConnection();

// Use the connection

// Close the connection to release it back to the poool
conn.close();
```

Summary

This white paper provided the source code and instruction necessary to embed the Informix Cloudscape DBMS within BEA WebLogic Server in a manner similar to the Informix Cloudconnector product. It explained the motivations behind adopting this architecture, as well as the features of Cloudscape that support such an application.

With its combination of robust SQL features, support for Java, and embeddable, pure Java architecture, Cloudscape is the data management product of choice for data-driven Java applications.