WEBADE
Connection Pooling Guide

Date:	July 20, 2012
Revision:	1.5

Vivid Solutions Inc.
Suite #1A, 2328 Government St.
Victoria, BC V8T 5G5
Phone: (250) 385-6040
Fax: (250) 385-6046
Website: www.vividsolutions.com

Document Change Control

REVISION NUMBER	DATE OF ISSUE	AUTHOR(S)	DESCRIPTION
1.0	Dec 4, 2003	Jason Ross	Initial Document
1.1	Dec 7, 2003	Jason Ross	Updated with some new configuration settings
1.2	June 14, 2005	Jason Ross	Updated to include connection wrapping documentation
1.3	August 11, 2005	Jason Ross	Updated the documentation on disabling connection wrapping and connection wrapper logging.
1.4	November 4, 2005	Jason Ross	Updated with minor layout changes.
1.5	January 31, 2012	Andrew Wilkinson	Corrected a code sample.

Table of Contents

1. INTRODUCTION 5

2. NEW FEATURES 6

2.1 WRAPPED CONNECTION CLASSES 6

2.1.1 RETRIEVING THE WRAPPED OBJECT 7

2.1.2 DISABLING WEBADE CONNECTION WRAPPING 7

2.2 DATABASE CONNECTION IDLE TIMEOUTS 8

2.3 DATABASE PINGING 8

2.4 SIMPLIFIED CONNECTION POOL CONFIGURATION 8

2.5 CONFIGURABLE BLOCKING WAIT TIMES 8

2.6 IMPROVED LOGGING AND DEBUGGING 8

2.7 OPTIONAL NON-POOLING CONNECTIONS 9

3. CONFIGURATION 10

3.1 CONNECTION POOL SETTINGS 10

3.1.1 DATABASE URL 10

3.1.2 DATABASE USER 10

3.1.3 DATABASE PASSWORD 10

3.1.4 MIN CONNECTIONS 10

3.1.5 MAX CONNECTIONS 10

3.1.6 MAX CONNECTION IDLE TIME 11

3.1.7 MAX CONNECTION WAIT TIME 11

3.1.8 MONITOR SLEEP TIME 11

3.1.9 POOL CONNECTIONS FLAG 11

3.1.10 PING CONNECTIONS FLAG 11

4. FUTURE CHANGES 12

4.1 CHANGING THE WAIT QUEUE TO A PRIORITY QUEUE 12

4.2 FORCE QUIT AND MONITORING OF LOST CONNECTIONS 12

4.3 DYNAMIC POOL SETTINGS 12

4.4 JMX SUPPORT 12

1. INTRODUCTION

This document details the changes to the WebADE core API, regarding the new implementation of connection pooling, replacing the Oracle reference implementation (found in classes12.zip).

2. NEW FEATURES

Below is a description of the new additions to the WebADE connection pooling, and changes to the core API.

2.1 WRAPPED CONNECTION CLASSES

The WebADE connection pooling API, by default, now wraps many of the classes in the java.sql package of the underlying JDBC implementation with internal WebADE classes implementing the same standard JDBC interfaces.

By wrapping these classes, WebADE can step in and handle situations where bugs in a deployed application cause database resources to be left open, such as where a connection is not closed due to an exception being thrown in the application code. In a situation such as this, WebADE can clean up the open resource on garbage collection.

Instances of the following classes are wrapped by WebADE classes, when returned by calls to the WebADE API:

- java.sql.CallableStatement
- java.sql.Connection
- java.sql.PreparedStatement
- java.sql.ResultSet
- java.sql.Statement

Because WebADE wraps these objects with an internal class implementing the appropriate java.sql interface, you cannot directly cast these objects to a database-specific implementation class, like OracleConnection, OracleStatement, and OracleResultSet.

If you require access to these wrapped database-specific implementation objects, you can either cast the returned object to the WebADE-wrapper implementation class and call a special getWrapped*XXX*() method on this object or disable the connection wrapping by setting the appropriate system property.

2.1.1 Retrieving the Wrapped Object

Below are code sample for retrieving the wrapped database-specific implementation objects.

CONNECTION CLASS

```
import ca.bc.gov.webade.dbpool.WrapperConnection;

...

Application app = ...
Connection conn = app.getConnectionByAction(...);
WrapperConnection wconn = (WrapperConnection)conn;
Connection wrappedConn = wconn.getWrappedConnection();
```

STATEMENT CLASS

```
import ca.bc.gov.webade.dbpool.WrapperStatement;

...

Application app = ...
Connection conn = app.getConnectionByAction(...);
Statement stmt = conn.createStatement();
WrapperStatement wstmt = (WrapperStatement)conn;
Statement wrappedStmt = wstmt.getWrappedStatement();
```

NOTE: This code sample also works for WrapperPreparedStatement and WrapperCallableStatement classes, as these classes extend WrapperStatement.

RESULTSET CLASS

```
import ca.bc.gov.webade.dbpool.WrapperResultSet;

...

Application app = ...
Connection conn = app.getConnectionByAction(...);
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(...)
WrapperResultSet wrs = (WrapperResultSet)rs;
ResultSet wrappedRs = wrs.getWrappedResultSet();
```

2.1.2 Disabling WebADE Connection Wrapping

In extreme circumstances, it may be necessary to disable the WebADE connection wrapping altogether. This is not recommended, but, if needed, it can be done by setting the "webade.use.wrapper.connections" system property to “false”.

NOTE: This should be set as a system property at the java command line, as follows:

```
-Dwebade.use.wrapper.connections=false
```

This property should only be set as a short term fix for deployment situations, where the deployed application is not behaving properly.

2.2 DATABASE CONNECTION IDLE TIMEOUTS

Database connections are now closed, if the connection has been sitting available in the pool past a given idle time, in minutes.

2.3 DATABASE PINGING

Database connections can be configured to be “ping”-ed before being handed out to the application, to ensure that the database connection has not been closed, due to databases going down.

2.4 SIMPLIFIED CONNECTION POOL CONFIGURATION

WebADE connection pool configuration is now much more simplified, by default not requiring an application to write any code to get the connection pools initialized. Previously, the application developer was required to create each connection pool and register these pools with the WebADE.

2.5 CONFIGURABLE BLOCKING WAIT TIMES

Previously, applications could only either block indefinitely while waiting for a connection, or not block at all. Application threads requesting connection pools can now have a third option to obtaining connection from the pool. Block for a specific amount of time (In milliseconds).

2.6 IMPROVED LOGGING AND DEBUGGING

Logs are now output when a connection is retrieved from the queue, returned to the queue, and when database errors occur. These logs will indicate the target pooled connection (by hash code) and which pool it is associated with (by role).

When a connection, statement, or result set is not closed properly, by calling the close() method on the wrapper object, at garbage collection time, the wrapper object will print a full stack trace at the time the wrapper object was created, to help the developer to locate the section of offending code in their application. (See below for an example)

```
WARN  ca.bc.gov.webade.dbpool.WrapperConnection - Connection: 23047631 closed
by garbage collector.  Connection checked out
	at
ca.bc.gov.webade.dbpool.WebADEConnectionCache.getConnection(WebADEConnectionC
ache.java:187)
	at
ca.bc.gov.webade.dbpool.ConnectionCacheTester.openConnection(ConnectionCacheT
ester.java:76)
	at
ca.bc.gov.webade.dbpool.ConnectionCacheTester.testUnclosedConnection(Connecti
onCacheTester.java:62)
	at
sun.reflect.NativeMethodAccessorImpl.invoke0(NativeMethodAccessorImpl.java:Na
tive Method)
	at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
	at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.
java:25)
	at java.lang.reflect.Method.invoke(Method.java:324)
	at junit.framework.TestCase.runTest(TestCase.java:154)
	at junit.framework.TestCase.runBare(TestCase.java:127)
	at junit.framework.TestResult$1.protect(TestResult.java:106)
	at junit.framework.TestResult.runProtected(TestResult.java:124)
	at junit.framework.TestResult.run(TestResult.java:109)
	at junit.framework.TestCase.run(TestCase.java:118)
	at junit.framework.TestSuite.runTest(TestSuite.java:208)
	at junit.framework.TestSuite.run(TestSuite.java:203)
	at
org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRun
ner.java:478)
	at
org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.run(RemoteTestRunner.j
ava:344)
	at
org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.main(RemoteTestRunner.
java:196)
```

2.7 OPTIONAL NON-POOLING CONNECTIONS

It is possible to prevent connections from being pooled, by setting the appropriate flag in the configuration settings.

3. CONFIGURATION

3.1 CONNECTION POOL SETTINGS

Pools, by default, now require no initialization from the application developer. The default settings for each connection pool parameter are described below. If you wish to override the settings for your application roles' connection pools, modify the appropriate columns in the WebADE PROXY_CONTROL table for the role's connection poll table entry.

3.1.1 DATABASE URL

The Database URL is a properly formatted JDBC URL for the target database. This field is mandatory, and will be retrieved from the WebADE database by the Application object.

3.1.2 DATABASE USER

The Database User is a user id with access to the target database. This field is mandatory, and will be retrieved from the WebADE database by the Application object.

3.1.3 DATABASE PASSWORD

The Database Password is the password for the above user id. This field is mandatory, and will be retrieved from the WebADE database by the Application object.

3.1.4 MIN CONNECTIONS

The minimum number of connections that will be open at any given time. This field is optional, with a default value of 0.

3.1.5 MAX CONNECTIONS

The maximum number of connections that will be open at any given time. This field is optional, with a default value of 5.

3.1.6 Max Connection Idle Time

The maximum time, in minutes, a connection should remain open while available in the pool. If the idle time is set to 0, the connection will be left open indefinitely. This field is optional, with a default value of 10 minutes.

3.1.7 Max Connection Wait Time

The maximum time, in milliseconds, a thread should block while waiting for an available connection. If the wait time is set to 0, the thread will block indefinitely. If the wait time is set to –1, the thread will not block at all. If the thread waits past its wait time (or is set not to block), the connection request will return null. This field is optional, with a default value of 0 (Indefinite blocking).

3.1.8 Monitor Sleep Time

The time the cache monitor thread will wait, in minutes, between connection pool checks. During each check, the monitor will close any connections in the pool that have been idle for longer than the max connection idle time. This field is optional, with a default value of 1 minute.

3.1.9 Pool Connections Flag

A flag indicating whether or not to pool database connections. Valid values are "true" and "false". This field is optional, with a default value of true.

3.1.10 Ping Connections Flag

A flag indicating whether or not to ping database connections before handing them out. This setting is intended to allow a testing of the connection, before the connection is handed out, to prevent closed connections from being used by an application. If the connection is closed, the error is trapped, and a good connection is created, and handed to the requesting thread. Valid values are "true" and "false". This field is optional, with a default value of false.

4. FUTURE CHANGES

4.1 CHANGING THE WAIT QUEUE TO A PRIORITY QUEUE

Allow requests for connections to have a priority, to allow behind-the-scenes processes (batch processes) to be superseded by user requests for connection pools, for more responsive user requests.

4.2 FORCE QUIT AND MONITORING OF LOST CONNECTIONS

Allow the connection pool monitor to forcefully terminate connections that have been checked out for too long. This is particularly useful for connections that were not closed before the code using them loses scope.

4.3 DYNAMIC POOL SETTINGS

Allow connection pool settings to be modified on the fly, instead of only at initialization time, as it is now.

4.4 JMX SUPPORT

Add support for the Java Management Extensions API. This will allow the remote administration/monitoring of the pools at runtime. It could also allow for actually growing and shrinking of the pools without stopping the application. The JMX support could also be expanded to monitor and manage all aspects of the WebADE. Once the JMX infrastructure is in place, it is simply a matter of creating a MBean (Managed Bean) to instrument the WebADE objects. Little effort with great gains.