SmartPool 1.5 User Documentation

There is always one more bug

Introduction

SmartPool is a connection-pooling component modeled on connection pooling features provided by an Application Server. SmartPool makes an attempt to solve critical issues like connection leaks, connection blocking, open JDBC objects like Statements, PreparedStatements etc. Features of SmartPool include support for multiple pools, automatic closing of associated JDBC objects, detect connection leaks based on configurable time-outs, track connection usage, wrap SmartPool to an existing pool, monitor run time status of the pools apart from conventional pooling support.

The focus of this release is to support distributed transactions in a clustered database environment like Oracle RAC. Each pool is now a container for one or more subpools that inherit all the characteristics from the container pool and are exactly identical to each other. Each subpool should be configured to point to a different database instance using the connect-string parameter or the connection-loader-class parameter. SmartPool then associates a thread with a particular subpool

Loading SmartPool

Using the *PoolManager*:
***PoolManager* is an interface defining the behavior of the class that manages each Pool. You can load SmartPool component by using the following:**

PoolManager pm = new PoolManagerImpl("/home/ssshetty/pool-config.xml");

Once the PoolManger is instantiated you can call the methods on the PoolManager interface with the following:

String poolName = "MyPool";
Connection conn = pm.getConnection(poolName);

For more details refer to Java Documentation of *PoolManager* Interface.
Note: You are creating an Instance of a *PoolManager*, which is not Singleton by itself, and thus multiple *PoolManager* instances would result in multiple pools.

Using *SmartPoolFactory*:
***SmartPoolFactory* provides a Singleton access to SmartPool. This is a Singleton Wrapper to the *PoolManagerImpl*. This is the recommended way to load the Pool.**

SmartPoolFactory smp = new SmartPoolFactory("/home/pool-config.xml");
This would create a Singleton object smp in the memory. Static methods on this object can be invoked from any class loaded within the same JVM.

Connection con = SmartPoolFactory.getConnection();
Note: Static methods on *SmartPoolFactory* can only be used after it is initialized with a configuration file as shown above. It is recommended to load this *SmartPoolFactory* on your application start-up (like the start-up servlet in a web application).
For more details refer to Java Documentation of *SmartPoolFactory*.

You can now specify the config file using the system property "config.file". SmartPoolFactory will pick up the file pointed by this system property. (new**)**

- **Accessing Connections from the pool**

Connection can be accessed from the *PoolManager*/*SmartPoolFactory* depending on the method you have used to load the pool. The following examples are applicable when you use *SmartPoolFactory*.
You can draw connections from the pool is the following way:

Connection conn = SmartPoolFactory.getConnection();
This fetches a connection from the default pool, with owner set to N/A i.e. Not Applicable

Default Pool: Default pool is one where is set to default-pool="true" in the configuration file. In a simple application where only one pool is required and no ownership is to be tracked, developers need not provide the pool name each time they take a connection from the pool. At most one pool can be marked as a default pool.
Owner: Owner is the identity of the user/class drawing the connection from the pool. Ownership is used for debugging while detecting connection leaks to exactly identify the owners of connections. Preferably this should be a combination of the class and method that is drawing the connection since this would directly help in attacking and solving the problem of Connection leaks.
Anonymous Connection: A connection drawn from the pool without specifying the owner name is an anonymous connection. It is not possible to keep track of ownerships if anonymous connections are allowed.
To disallow anonymous connections set allow-anonymous-connections="false" in the configuration file.
Other methods to get Connections are:

Connection conn = SmartPoolFactory.getConnection("poolName");

Here poolName is the pool name.

Connection conn = SmartPoolFactory.getConnection("poolName", "TestSmartPool:getMyConnection");
Here TestSmartPool:getMyConnection is the owner name.

Releasing Connection to the pool:
To release connection to the Pool just call close on the method.

```
Connection conn = SmartPoolFactory.getConnection("poolName")
....
....
conn.close();
```

This will return the connections to the pool.

Following points are worth noting when a connection is released.
1. If auto-commit is set to false for the connection, SmartPool will call rollback on that method and set auto-commit to true. This is to destroy the existing state of the connection before it is made available to others.

2. If auto-close="true" in the configuration file, all the associated Statements, PreparedStatements, and CallableStatements will be closed when the connection is released to the pool.

- **Detecting Leaks**

Leak detection is time-out based.

leak-time-out="100"
To set a time-interval of 100 seconds for which a class can hold the connection without being considered as leak set leak-time-out="100".
A *Connection Leak* is said to have occurred when a class is holding on to a connection for more than leak-time-out.

poll-thread-time="100"
A leak detector thread will poll all used connections every 100 seconds. As soon as the leak detector Poll thread finds out a Connection leak, it will notify all the registered connection leak listeners.
Connection leak listener is a class implementing the *ConnectionLeakListener* interface.
Connection leak listener class receives a notification of a leak through the following call back method:

public void connectionTimeOut(ConnectionLeakEvent cle)

Where *ConnectionLeakEvent* is the class encapsulating the information about the leak.
See Java documentation of *ConnectionLeakEvent*, *ConnectionLeakListener* for more info.

You can register a *ConnectionLeakLister* by calling the following method on *PoolManager/SmartPoolFactory*

PoolManager.addConnectionLeakListener(ConnectionLeakListener cle);

You can also provide a default listener in the configuration file. For e.g.:

default-listener="testconnectionpool.LeakDetector"

***PoolManager* will load this listener on load and it will be able to listen to leaks right from the instance *PoolManager* object is created.**
NOTE: The connection leak event will also provide you with access to the actual

Connection object which you can be returned back to the pool by calling close() in it, however be sure of what you are doing, because this may result in functionality bugs which would be very difficult to debug.

- **Monitoring Pool Status**
  Monitoring Interfaces allows you to keep track how the connections are being used, who is blocking up the connections, detect leaks if any, thus helping in identifying and attacking problems when the application/system crashes because of excessive connections or tables get locked etc.
  To monitor the Pools you can use the Following method:

  PoolMonitor pm = SmartPoolFactory.getPoolMonitor("Sachin");

  ***PoolMonitor* is the interface defining methods to monitor the current status of the pool.**
  To get the Configuration of the pool, use the following method:

  ConfigMonitor cm = pm.getConfigMonitor();

  ***ConfigMonitor* provides access to pool configuration.**
  To get information on the Connections being used:

  Vector v = pm.getConnectionsInUse();

  This will return a vector of *ConnectionMonitor* providing all the information about the connection in use.
  For more information see Java Documentation of *PoolMonitor*, *ConfigMonitor*, *ConnectionMonitor*.

- **Automatic closing of associated Statements, PreparedStatements, and CallableStatements**
  If auto-close="true" in the configuration file then all the Statements, PreparedStatements, and CallableStatements associated with the connection will be closed when the connection is released back to the pool.

- **Wrapping SmartPool to an existing pool**
  Wrapping SmartPool to an existing pool implies that SmartPool will not draw raw connections to the database; it would rather draw connections from another connection pool, for e.g. DataSource of an Application Server.
  In this case SmartPool does not maintain any pool, it simply delegates the connection requests to the parent pool. SmartPool however keeps a track of connections in use, and thus can be used for leak detecting, automatic closing of associated Statements, PreparedStatements, CallableStatements objects.
  SmartPool can be wrapped to an existing pool with the help of the interface ConnectionProvider. ConnectionProvider provides two methods:

  public Connection getConnection() throws Exception

  This method should return a connection from the pool to which SmartPool is wrapped.

  public void returnConnection(Connection conn) throws Exception

  This method should return the connection conn back to the parent pool.

  For more information see Java Documentation of ConnectionProvider.

  To wrap smart pool to an existing pool, you need to write an implementation of ConnectionProvider and provide the fully qualified class name in the configuration file. For e.g. If the class implementing ConnectionProvider is com.your_company.your_project.SampleConnectionProviderImpl, then in the

configuration file, you will have to put the following entry:

connection-loader-class= "com.your_company.your_project.SampleConnectionProviderImpl"

- **Forced recovery of recently used connections**

max-connection-idle-time="100"

The above specifies the maximum amount of time for which a consumer can hold on to a connection, without using it. hence if the poll thread finds any connections that have not been used for more than 100 seconds, it will take the connection back, and any attempt to use the connection afterwards will result in *StaleConnectionException*.

- **Configuring MultiPools**

With release 1.5, each pool in SmartPool as defined in configuration file with the pool element is a MultiPool. Each MultiPool encapsulates one or more subpools that are exactly identical to each other except for their connect-strings or connection-loader-class. This support for MultiPool has been specifically designed to support clustered database environments like Oracle RAC where multiple database instances provide simultaneous access to a single database. In such scenarios, a multi pool with subpools as many as database instances should be configured, with each subpool pointing to a database instance using the connect-string parameter. This allows the same multi pool to load balance connections across different database instances transparently. Instance stickiness for distributed transactions is explained in the next section.

To configure a MultiPool with two subpools use the following configuration:

```
<connect-strings>
    <thread-stickiness>true</thread-stickiness>
    <connect-string
name="instance1">jdbc:oracle:thin:@db01.dev:1521:dev1</connect-string>
    <connect-string
name="instance2">jdbc:oracle:thin:@db02.dev:1521:dev2</connect-string>
</connect-strings>
```

To configure a Multipool with two subpools that loads a connection from an external source (wrapping SmartPool to other pools)

```
<external-pooling>
    <thread-stickiness>true</thread-stickiness>
    <connection-loader-class
name="loader1">com.mycom.test.ConnectionProvider</connection-loader-class>
    <connection-loader-class
name="loader2">com.mycom.test.ConnectionProvider</connection-loader-class>
</external-pooling>
```

Note: A MultiPool with a single connection-string or connection-loader-class is exactly same as convention pools with not subpools.

- **Using SmartPool to achieve sticky distributed transactions with Oracle RAC**

Oracle RAC and Clustered database environments provide a whole new set of challenges and SmartPool tries to address one of them, i.e. support sticky distributed transactions without sacrificing failover. In a clustered database environments, all connections participating in a single distributed transactions are required to be connected to the same database instance thus eliminating the option of random load balancing. This

forces the user to leverage RAC only for failover and rule out load balancing. SmartPool addresses this problem by associating a thread to an instance, and thus all connections request from the same thread are directed to the same instance. Distributed transactions maintain transaction context association in the same way and thus every new thread gets attached to a different instance achieving load balancing but still maintaining the stickiness to the instance for the lifespan of the thread, thus supporting distributed transactions.

To configure a MultiPool that provides thread sticky loadbalancing for Oracle RAC:

1. Use appropriate jdbc connect strings that disable server side load balancing by setting the *instance_name* parameter in connect string
2. set the *thread-stickiness* parameter without which SmartPool will pick a random connections rather then examine the thread association with an instance.

For example:

```
<connect-strings>
    <thread-stickiness>true</thread-stickiness>
    <connect-string name="instance1">
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=tcp)
(HOST=MYDBHOST1)(PORT=1522)))(CONNECT_DATA=(SERVICE_NAME=MYDB)
(INSTANCE_NAME=MYDB1)))</connect-string>
    <connect-string name="instance2">
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=tcp)
(HOST=MYDBHOST2)(PORT=1522)))(CONNECT_DATA=(SERVICE_NAME=MYDB)
(INSTANCE_NAME=MYDB2)))</connect-string>
</connect-strings>
```

For more information refer to: Using global transactions with Oracle RAC

- **Features to expect in future releases**

  1. Detect instance failover and adjust the subpools accordingly
  2. Using Oracle 10g fan events to listen to server side callbacks

- **XSD for configuration file**
  $DOWNLOAD_EXTRACT_DIR/samples/pool-config.xsd

- **Sample configuration file**
  $DOWNLOAD_EXTRACT_DIR/samples/pool-config-sample.xml

- **Sample ConnectionProvider Implementation**
  $DOWNLOAD_EXTRACT_DIR/samples/ConnectionProviderImpl1.java

- **Sample Connection Leak Listener**
  $DOWNLOAD_EXTRACT_DIR/samples/LeakDetectorImpl.java

- **Java/API Documentation**
  $DOWNLOAD_EXTRACT_DIR/doc/javadoc

- **Glossary**
  Consumer: Consumer is a class/component that draws connections from the SmartPool.

  Default Pool: Default pool is the pool where you have said default-pool="true" in the configuration file. Thus in a simple application where only one pool is required and no ownership is to be tracked, developers need not provide the pool name each time they take a connection from the pool. Only one pool can be marked as a default pool at the

most.

Owner: Owner is an identity of the user/class drawing the connection from the pool. This is used for debugging while detecting connection leaks so as to exactly identify who is blocking the connections. Preferably this should be a combination of the class and method that is drawing the connection as this would directly help in attacking and solving the problem of Connection leaks.

Anonymous Connection: An anonymous connection is a connection drawn from the pool without specifying the owner name. Hence not possible to keep track of where the connection is being blocked.
When allow-anonymous-connections="false" in the configuration file, anonymous connections are not allowed.

Connection Leak: When a Consumer holds on to a connection for more than the time specified in leak-time-out in configuration file, a connection leak is said to have occurred and the Consumer is the owner of the connection and is responsible for the connection leak.