DBCP

DBCP Overview

{{
 	
http://jakarta.apache.org/commons/dbcp/images/dbcp-logo-white.png
 	
}}
	Commons-DBCP provides database connection pooling services.
	A lot of information is available on the DBCP website. If you don't find the information you need you can always contact us using one of the mailing lists.

Release Plans

External Resources

Here's a little diagram showing what datastructure DBCP sets up and how that datastructure is used in case the DriverManager is employed to obtain Drivers at runtime. Please send any corrections (for now) to d.tonhofer@m-plify.com.

- Apache_DBCP_Structure.pdf
- Apache_DBCP_Structure.vsd
- Apache_DBCP_Structure.png

Diagrams hosted by http://public.m-plify.net

The Database Connection Pool (DBCP) component can be used in applications where JDBC resources need to be pooled. Apart from JDBC connections, this provides support for pooling Statement and PreparedStatement instances as well. Complete article can be found here http://www.devx.com/Java/Article/29795/0/page/2.

Security Manager settings

If you're running tomcat with the Security Manager, you will need to add to your catalina.policy:

```
grant {
        permission java.lang.RuntimePermission "accessClassInPackage.org.apache.tomcat.dbcp.*";
};
```

(This is in addition to any java.net.SocketPermission and java.sql.SQLPermission needed by the database itself)

Tomcat 5.0 Configuration examples

Some Tomcat JNDI Datasource examples (in addition to the Tomcat 5.5 JNDI datasource howto).

BasicDataSource

```
<Resource name="jdbc/abandoned" auth="Container" type="javax.sql.DataSource"/>
<ResourceParams name="jdbc/abandoned">
   <parameter><name>factory</name><value>org.apache.commons.dbcp.BasicDataSourceFactory</value></parameter>
   <parameter><name>username</name><value>sa</value></parameter>
   <parameter><name>password</name><value></value></parameter>
   <parameter><name>driverClassName</name><value>org.hsqldb.jdbcDriver</value></parameter>
   <parameter><name>url</name><value>jdbc:hsqldb:database</value></parameter>
   <parameter><name>removeAbandoned</name><value>false</value></parameter>
   <parameter><name>removeAbandonedTimeout</name><value>300</value></parameter>
   <parameter><name>logAbandoned</name><value>true</value></parameter>
   <parameter><name>maxActive</name><value>10</value></parameter>
</ResourceParams>
```

PerUserPoolDataSource

```
<Resource name="jdbc/TestDBCPDS" auth="Container" type="org.apache.commons.dbcp.cpdsadapter.DriverAdapterCPDS"/>
<ResourceParams name="jdbc/TestDBCPDS">
   <parameter><name>factory</name><value>org.apache.commons.dbcp.cpdsadapter.DriverAdapterCPDS</value><
/parameter>
   <parameter><name>user</name><value>sa</value></parameter>
   <parameter><name>password</name><value></value></parameter>
   <parameter><name>driver</name><value>com.sybase.jdbc2.jdbc.SybDrivers</value></parameter>
   <parameter><name>url</name><value>jdbc:sybase:Tds:<myServerName>:<myPort>?charset=iso_1</value></parameter>
</ResourceParams>

<Resource auth="Container" name="jdbc/TestDB" type="org.apache.commons.dbcp.datasources.PerUserPoolDataSource"/>
<ResourceParams name="jdbc/TestDB">
   <parameter><name>factory</name><value>org.apache.commons.dbcp.datasources.PerUserPoolDataSourceFactory<
/value></parameter>
   <parameter><name>defaultMaxActive</name><value>10</value></parameter>
   <parameter><name>defaultMaxIdle</name><value>2</value></parameter>
   <parameter><name>defaultMaxWait</name><value>-1</value></parameter>
   <parameter><name>dataSourceName</name><value>java:comp/env/jdbc/TestDBCPDS</value></parameter>
</ResourceParams>
```

Tomcat 5.5 Configuration examples

Less DBCP, more Tomcat (and Tomcat 5.5.7 in particular):

A more specialized example, in which we want to set up a Tomcat Authentication Realm based on a database. The database shall be accessed through connections from a DBCP pool. We edit server.xml directly. This configuration can be tricky to get right, so here is a complete example:

First, define the 'javax.sql.DataSource' available to web applications in the JNDI default context under 'jdbc/m3p_5_0' by:

1: Saying what class or interface a JNDI context lookup will return: 'javax.sql.DataSource'.

2: Saying what class will actually create instances of the above, i.e. give the factory. This is actually not necessary if you want 'org.apache.commons.dbcp.BasicDataSourceFactory' as that is the default factory used by Tomcat whenever 'javax.sql.DataSource' objects should be created. By setting 'factory', you can override that default value. 'o.a.c.d.BasicDataSourceFactory' creates 'org.apache.commons.dbcp.BasicDataSource' instances. These use a resource pool of type 'org.apache.commons.pool.impl.GenericObjectPool'. And for this type of pool we can demand that objects be verified at borrowing time - which is what we want as it will prevent Tomcat getting its paws on stale database connections.

3: Configuring the attributes of the 'BasicDataSourceFactory'. The allowed attributes can be found by looking for JavaBean-compliant set() methods and members in the source or the DBCP API doc. In particular: what driver shall be used by the factory to actually get database connections: 'com.mysql.jdbc.Driver'.

Additionally (and one level down, if you will) the 'Driver' named in '!driverClassName' is itself configured through the URL used when a new connection is created

I have put the 'Resource' element into the 'GlobalNamingResource' element instead of the 'Context' element. If you want to put it in the 'Context' element, the 'Realm' must be in the same 'Context' element and must additionally have the attribute 'localDataSource' set to 'true'.

- Tomcat 5.5 'server' element configuration.
- Tomcat 5.5 JNDI-DataSource examples
- The definition of 'resource'
- Tomcat 4.1 JDNI resources howto

```
<Server ...>

    <GlobalNamingResources>

        <Resource name="jdbc/mydatabase"
                  auth="Container"
                  type="javax.sql.DataSource"
                  factory="org.apache.commons.dbcp.BasicDataSourceFactory"
                  driverClassName="com.mysql.jdbc.Driver"
                  validationQuery="SELECT 1"
                  loginTimeout="10"
                  maxWait="5000"
                  username="i_am_tomcat"
                  password="my_password_is_foo"
                  testOnBorrow="true"
                  url="jdbc:mysql://127.0.0.1/mydatabase?connectTimeout=5000&amp;socketTimeout=8000&amp;
useUsageAdvisor=true"
        />

    </GlobalNamingResources>

    ...something something...

</Server>
```

Now define the Tomcat Authentication Realm. I have put the 'Realm' element inside the 'Host' element, but as said above you can put it into the 'Context' element, too.

The Realm implementation is 'org.apache.catalina.realm.DataSourceRealm'; it will use a 'javax.sql.DataSource' interface found in the JNDI initial context. The location of that interface inside the JNDI namespace is given by 'dataSourceName': 'java:com/env/jdbc/mydatabase'.

- Tomcat 5.5 Realm configuration
- Tomcat 5.5 DataSourceRealm

```
<Host ....>

  <Context ...> ... </Context>

  <Context ...> ... </Context>

  <Realm className="org.apache.catalina.realm.DataSourceRealm"
                    dataSourceName="jdbc/mydatabase"
                    digest="MD5"
                    roleNameCol="web_user_role_name"
                    userCredCol="account_md5_password"
                    userNameCol="account_login"
                    userRoleTable="web_user_role_t"
                    userTable="account_t" />

</Host>
```

Finally, beware bug 33357 in 5.5.7 which should be fixed soon though :-P

Hibernate

Hibernate is a powerful, ultra-high performance object/relational persistence and query service for Java. Hibernate lets you develop persistent classes following common Java idiom - including association, inheritance, polymorphism, composition and the Java collections framework.

- DBCP/Hibernate configuration

FAQ

Q: Is this project still active or have they just not released anything in an excessive period of time?

A: DBCP is still active. We don't do a lot of releases but

Q: What is the best way, ASIDE from running a query against the DB, to be sure that the connection is still valid when it is retrieved from the pool?

A: How would you do this without a pool? I.e., what is the best way aside from running a query to be sure that a connection is still valid when returned from DriverManager DataSource.getConnection?

When upgrading from DBCP 1.0 to DBCP 1.1 you can encounter the following issue (reported on commons-user):

We are accessing mainframe data using a JDBC driver, but it is not truly a database. Specifically, the setAutoCommit and setReadOnly Connection methods fail. Version 1.0 of DBCP silently discarded these errors and allowed the connections to be created and used anyway. Version 1.1 of activateObject surfaces these exceptions and fails to create new connections in the pool. (solution: use custom activateObject method)

Q: I see in the javadocs that AbandonedConnectionPool was deprecated (DBCP 1.1). What replaced it?

A: The original reason for deprecation was the danger in reusing a abandoned connection without knowing if it is safe to do so. There was a discussion about it in april. Some people wanted to remove it completely.

I took a different approach. In 1.1 an abandoned connection will not be reused but closed (and a new one created). The classes remained deprecated because I think the AbandonedPool should move to the pool package (and made more generic/safe).

The abandoned connection feature on BasicDataSource will remain supported in one form or another. I'm not using it on my tomcat production configurations but there are junit tests to make sure everything works.

So if you are using the Abandoned* classes directly then it is possible you are affected by a future refactoring (we will try to remain compatible if possible of course). If you use BasicDataSource then you can be sure the feature will remain.

Q: When using DBCP what parameter needs to be set on the ObjectPool so that a minimum number of connections are created when the pool instance is created ? I thought the setMinIdle() does this, but doesn't look so.

A: You can setMinIdle() to always have a minimum amount of idle connection in the pool. (not only at the start of the pool but the whole time) The minIdle check is done in the evictor thread so timeBetweenEvictionRunsMillis has to be set to a non negative-value. (you can optionally turn off the eviction by setting numTestsPerEvictionRun to zero or just set minEvictableIdleTimeMillis very high).

A more simple way to create a number of connections at startup is to use the pool.addObject() method. (add the following after creating the PoolableConnectionFactory)

```
for (int i=0 ; i<initialsize ; i++) {
   connectionPool.addObject()
}
```

Q: <nowiki>Does the current 1.1 release support the poolable/caching of PreparedStatements. I noted that the PoolableConnectionFactory can take a KeyedObjectPoolFactory as a statement pool factory. But there is not concrete implementation for the KeyedPoolableObjectFactory which is required when creating a GenericKeyedObjectPoolFactory. If I pass in a null as shown in the examples, does it cache prepared statements or should I do that in local objects?</nowiki>

A:
Yes, prepared statements are being cached. Here is a an example how to use the statement pool:

```
ConnectionFactory connectionFactory = new DriverManagerConnectionFactory(
    url, username, password);

GenericObjectPool connectionPool = new GenericObjectPool();

// null can be used as parameter because this parameter is set in
// PoolableConnectionFactory when creating a new PoolableConnection
KeyedObjectPoolFactory statementPool = new GenericKeyedObjectPoolFactory(null);

final boolean defaultReadOnly = false;
final boolean defaultAutoCommit = false;
final String validationQuery = null;
new PoolableConnectionFactory(connectionFactory, connectionPool, statementPool,
      validationQuery, defaultReadOnly, defaultAutoCommit);
```

Q: Where do I get a concrete example of PerUserPoolDataSource? How do we use it in a situation where multiple pools are required for different modules of a project and their properties differ on the database transactional load?

Q: Can DBCP be compiled with JDK1.3?

A: The ant build has facilities to comment out the JDBC3 method making DBCP source compatible with JDK1.3. DBCP1.2 had some JDK1.4 method but those were removed, see issue 29454.

Q: Uh... what about PoolableConnectionFactory line 51 & others which use Boolean.valueOf(boolean) which was introduced in 1.4?

A: They were replaced: http://cvs.apache.org/viewcvs.cgi/jakarta-commons/dbcp/src/java/org/apache/commons/dbcp/PoolableConnectionFactory.java?r1=1.22&r2=1.23&diff_format=h

---

Q: Without using validation of connections (testOnBorrow = false, testOnReturn = false, timeBetweenEvictionRunsMillis = -1) and after shutdown and restarting the database again, it looks like the pool is cleaning its old connections by itself. So it turns out that we always have valid connections. How can you explain this and when is explicit validation necessary?

A: During the connection activation (when borrowing a connection) the setAutoCommit and other connection init methods are called. If one of these methods throws a SQLException then the connection is also considered broken and removed from the pool.

So if you are using one of the "default*" properties and the JDBC driver correctly reports the SQLExceptions on the "set*" methods then you don't need an extra validationQuery.

Notes on DBCP Connection Validation

The following links are of use for the discussion:

- org.apache.commons.pool.PoolableObjectFactory.validateObject(java.lang.Object)
- org.apache.commons.dbcp.PoolableConnectionFactory
- org.apache.commons.dbcp.PoolableConnectionFactory.validateConnection(java.sql.Connection)
- org.apache.commons.pool.StackObjectPool
- org.apache.commons.pool.GenericObjectPool

Q: What is "Validation of Connections"?

Before a new Connection is obtained from a pool or before an existing one is returned to a pool it may be *validated*: a check is run to see whether the Connection has become stale - for example because the database unilaterally decided to close the underlying socket.

To validate the Connection, a *validationQuery* (e.g. 'SELECT 1') is passed to the constructor of *org.apache.dbcp.PoolableConnectionFactory*.

It is used in the following way:

According to the pooling implementation that underlies DBCP, objects in a pool are validated through a call to the interface method *org.apache.commons.pool.PoolableObjectFactory.validateObject()*. Its description is:

```
Ensures that the instance is safe to be returned by the pool. Returns false if this object should be destroyed.
```

org.apache.dbcp.PoolableConnectionFactory implements that interface method:

```
    public boolean validateObject(Object obj) {
        if(obj instanceof Connection) {
            try {
                validateConnection((Connection) obj);
                return true;
            } catch(Exception e) {
                return false;
            }
        } else {
            return false;
        }
    }
```

Note the call to *PoolableConnectionFactory.validateConnection()*.

PoolableConnectionFactory.validateConnection() tries to run the *validationQuery* passed in the constructor. If this fails, it throws an *SQLException* and *validateObject()* will return *false*.

Q: When are Connections validated? In other words, when you have a PoolableConnectionFactory, when is its validateObject() method called on a Connection?

A: This depends on the pool implementation that you are using

Case 1: Your Pool is a org.apache.commons.pool.StackObjectPool

Connections are **not** validated when you borrow them from the pool. They are only validated - unconditionally - in *returnObject()*, which is called by *addObject()*. In other words, they are only validated if they are put into the pool, either immediately after creation or when return them to the pool by a call to *Connection.close()*.

The consequence of this is that you will get Exceptions if you borrow a stale Connection because immediately after the Object has been pop()-ed from the stack, the factory's *activateObject()* is called on the Connection and the first thing it does is set the autocommit default value. This will throw on a stale Connection.

Case 2: Your Pool is a org.apache.commons.pool.GenericObjectPool

Connections are validated in the following three *GenericObjectPool* methods:
evict(), *borrowObject()* and *addObjectToPool()*. Thus, *GenericObjectPool* gives you the possibility to also validate the Connections at 'borrow' time.

GenericObjectPool.borrowObject()

Only calls *validateObject()* if 'testOnBorrow' has been set. This parameter
is passed in the pool's constructor.
If validation fails and the Connection has not been newly created, the Connection is
discarded and a new one is created in its place. However, if the Connections has
just been newly created, the Exception is left up the stack.
Note that if the database goes away, you will see Exceptions coming out of
borrowObject() as newly created Connections will fail their validation.

GenericObjectPool.evict()

This method is called by the *Evictor* which clears out idle Objects in the
pool. It unconditionally runs *validateObject()* on a pooled Object. If that method
returns false, the pooled Object is removed.

GenericObjectPool.addObjectToPool()

Only calls *validateObject()* if 'testOnReturn' has been set. This parameter
is passed in the pool's constructor. If the (returned) Object fails validation,
it is immediately destroyed.

Q: What if the pooled Connections are com.mysql.jdbc.Connection instances

This works perfectly well if you are using a *GenericObjectPool* that has 'testOnBorrow' set. Connections will be validated at borrow time. Any stale ones will be discarded, and new ones created in their place. The MySQL Connector/J Connection allows to set the parameter 'autoReconnect', but it is useless in this case: 'autoReconnect' just means that you will get an Exception when you run a query on a stale Connection, but that you can run the query *again* and the Connection will try to reconnect on this second try. As DBCP does not do second tries on the same Connection instance, the 'autoReconnect' case will not arise. Retrying twice before giving up is for example what a Tomcat 4 JDBCRealm does. (The above is not clear from the MySQL documentation and the behaviour one would intuitively give to 'autoReconnect' is misleading and so this is a potential headache.)

There is a 'autoReconnectForPools' starting from MySQL-Connector-J which I haven't looked into yet.

See Connector-J Connection Properties for additional details, and also the source for *com.mysql.jdbc.Connection.execSQL()*.