J2EE and Connection Pools

Right now GeoTools makes use of JDBC Connections in the following scenarios:

- EPSG Authority implementations (for Postgres, HSQL, Access and soon Oracle)
- DataStore implementations (for PostGIS, Oracle, HSQL, H2, MySQL, etc...)

We need to ensure that GeoTools can be used safely in a J2EE environment; specifically we need to allow GeoTools based applications to configure the library with an externally defined DataSource (usually manged by a J2EE container provided connection pool).

Background on the Use of DataSource

Because GeoTools predates almost all J2EE technology we missed out on the introduction of the DataSource interface in Java 1.4.

DataSource Javadocs
A factory for connections to the physical data source that this DataSource object represents. An alternative to the DriverManager facility, a DataSource object is the preferred means of getting a connection. An object that implements the DataSource interface will typically be registered with a naming service based on the JavaTM Naming and Directory (JNDI) API.

The DataSource interface is implemented by a driver vendor. There are three types of implementations:

1. Basic implementation – produces a standard Connection object
2. Connection pooling implementation – produces a Connection object that will automatically participate in connection pooling. This implementation works with a middle-tier connection pooling manager.
3. Distributed transaction implementation – produces a Connection object that may be used for distributed transactions and almost always participates in connection pooling. This implementation works with a middle-tier transaction manager and almost always with a connection pooling manager.

Read more information in DataSource javadocs.

This is similar in intent to our own ConnectionPool implementation (that makes use of DriverManager to look up a JDBC Driver etc...).

Implementations of DataSource

As indicated in the javadocs there are many implementations of DataSource around, here is a list of known implementations.

- DBCP - This is the connection pool implementation used by Tomcat
- C3PO - A connection pool often used with hibernate
- WebSphere
- OC4L
- etc...

If we strict to the strict DataSource API we should be okay .... too about Oracle.

Why we are not Okay with Oracle

For most cases we are fine with the DataSource API as provided out of the box. There is **one** case we are not happy:

Oracle ☹

The Oracle datastore wants to have direct access to an OracleConnection (so it can play directly with the Oracle STRUCT interface etc...). To do that we need to "unwrap" the Connection provided by the DataSource.

How to "unwrap" changes based on the implementation returned by DataSource.getConnection():

- Direct: cast returned Connection to (OracleConnection)
- DBCP: cast to specific DBCPPooledConnection and ask for the wrapped connection
- C3PO: use reflection to access a private field
- etc...

Because this is an open ended set .. and we do not have access to all implementations (say OC4L) we are going to have to make an extension point to handle this case.

Sun has recognized this problem and defined the Wrapper interface in Java 6:

Java 6 Wrapper Interface

```
interface Wrapper {
   boolean isWrapperFor( Class iface );
 <T> T unwrap( Class<T> iface );
}
```

We cannot use this directly (as we need to let other programmers "Extend" our system with new wrappers). But we can get pretty close:

```
interface UnWrapper<T> {
     boolean isWrapperOn( Object obj );
     T unwrap( Object obj );
}
```

We can account for the "direct" case pretty easiliy (ie OracleConnection unwrapped to OracleConnection):

```
class OracleConnectionUnwrapper extends UnWrapper<OracleConnection> {
     boolean isWrapperOn( Object obj ){
         return obj instanceof OracleConnection;
     }
     OracleConnection unwrap( Object obj ){
          return (OracleConnection)  obj;
     }
}
```

Because reflection will be a common theme we may as well just break down and provide a couple nice superclasses for people to extend. The other advantage is that we do not introduce additional dependencies into our code.

```
class C3POUnwrap extends ReflectionUnwrap<OracleConnection> {
      public C3POUnwrap(){
            super( OracleConnection.class );
      }
      String getOnClassName(){
           return "com.mchange.v2.c3p0.ComboPooledDataSource";
      }
      String getTargetField(){
           return "conn";
      }
}
class DBPCUnwrap extends ReflectionUnwrap<OracleConnection>{
      public DBPCUnwrap(){
            super( OracleConnection.class );
      }
      String getSourceConnectionType(){
           return "org.apache.commons.dbcp.DelegatingConnection";
      }
      String getTargetProperty(){
           return "delegate";
      }
}
```

While it is a pain to produce a bunch of code just to make Oracle happy; this is life we may as well get on with it.

Existing Example

This problem has been reported (and fixed) for the case of Oracle in a JBoss environment:

- http://jira.codehaus.org/browse/GEOS-778

The solution was not general (a dependency on JBoss was introduced in the attached Java code). Good to know we are on the right track.

EPSG Authority Implementations

The base class (EPSGDefaultFactory) is already set up to use DataSource, we had some initial trouble where it would try to register a DataSource if one was not already provided (this has since been repressed).

- http://jira.codehaus.org/browse/GEOT-909 - EPSG DefaultFactory does not work in an EJB environment

The code currently uses JNDI to look for a DataSource registered at "java:EPSG". There are a couple of problems here ...

- Each subclass targets a DataSource in the same location
  - Actual location of DataSource to a EPSG database may change from application to application (or site to site)
- Database independence
  - It looks like we need to synchronize the DataSource and EPSGDefaultFactory used? It would be nice to have JNDI be the only thing an application needs to get right (so they can switch between Postgres and Oracle without having to redeploy all GeoTools based applications with different jars)
  - Write the EPSGDefaultFactory support in such a way that it is database independent. That is use strategy objects for implementation specific SQL needs, rather then subclassing.

Rough Plan:

1. Add DBCP connection pool implementation to our build (and use it as the "default" rather then DirectConnection)
2. Test that EPSGDefaultFactory works with DataSource
3. Produce a second Factory that accepts an arbitraty JNDI datasource, it will be up to client applications to configure this Factory and add it into the GeoTools mix (using Hints?)

Alternatives to JDBC & DataSource

We can rewrite the implementations to use Hibernate; and rely on Hibernate handling cross database issues (and DataSource) correctly.

DataStore Implementations

Connection pool configuration requires different parameter sets depending on the library and method chosen. For example, a JNDI based one just needs a JNDI key, a DBCP one can go up to 27 parameters, a C3P0 one may have even more.
Database vendors sometime do provide their own implementations, so, long story short, we need the set of usable connection pools to be open ended just like connection unwrappers.

Given the number of parameters, we suggest to have a DataSourceFactorySpi, much like the DataStoreSpi, that is, something that can be looked up with SPI, provides a list of parameters, and given a map with the required values, builds a DataSource.
JDBC data store factories will be cloned and modified to accept a DataSource, and all connections will be gathered from it. The old data store factories will be left there for compatibility, but deprecated, and new ones will be created that deal with the new connection pool setup. The old ones will use a default pool, which replaces the ConnectionPool class and fixes many of its illnesses.

In order to replace ConnectionPool with DataSource the following classes/methods will have to be modified:

- JDBC2DataStore.createConnection()
- JDBCTransactionState(ConnectionPool)
- Various bits in VersioningPostgisDataStore, which now do call the connection pool directly;
- JDBCAuthorityFactory and PostgisAuthorityFactory
- Various bits in the JDBC datastore factories
- Finally, we have to get rid of all the connection factories making up the PoolDataSource instances the ConnectionPool uses, and the ConnectionPoolManager too.
  The total number of references to ConnectionPool in the code is 53, the total number of ConnectionPoolManager references is 9.

So, to sum up, the idea is:

1. Update JDBCDataStore to use DataSource (rather than ConnectionPool),
   a. Add the default pool manager as a maven dependency (probably DBCP?)
   b. Review QueryData error handling code (we went to great trouble to clear the error status of Connections because our "pool" was so simple it would not do it)
   c. Fix all places that are using ConnectionPool and use DataSource instead
2. Change existing DataStoreFactorySPI (like PostGISDataStoreFactory) to use the default pool based on traditional parameters, and pass the generated DataSource to stores.
3. Create an additional set of factories (like PostGISDataStoreFactory2) that will accept a preconfigured DataSource
4. Fix up Oracle
   a. Implement the UnWrap idea outlined above for DBCP
   b. Add additional UnWrap implementations as GeoServer is tested in new environments (JBoss, OC4L and so on)

Issues left open:

- shall we allow for an open ended set of parameters? Some connection pools have way too many parameters, shall we expose them all? An alternative would be to allow user to add his own keys and values, and expose just the typical ones by default.
- decide which connection pool to make the default. Choice is not simple, see this blog for example.
- shall we consider doing a JNDI lokup when the "datasource" key is a String? (I'd say, not, because we would duplicate the job of data source factories in each data store, but this is still open for debate).