JdbcConfiguration13

JDBC pools configuration

BTM XA datasources can be created via some java code or via a BTM-specific tool called the Resource Loader. You are free to choose the method you prefer, there is absolutely no difference between them.

Contents

- Using the BTM API
  - Minimal settings
  - Eager initialization
- Using the Resource Loader

Using the BTM API

BTM comes bundled with a JDBC XA connection pool which is very easy to configure. You basically have to create an instance of bitronix.tm.resource.jdbc.PoolingDataSource set some properties and you're done.

Here is an example of datasource creation that connects to an Oracle database:

```
PoolingDataSource myDataSource = new
PoolingDataSource();
(1)
myDataSource.setClassName("oracle.jdbc.xa.cl
ient.OracleXADataSource");
(2)
myDataSource.setUniqueName("oracle");
(3)
myDataSource.setMinPoolSize(0);
(4)
myDataSource.setMaxPoolSize(5);
(5)
myDataSource.setAcquireIncrement(1);
(6)
myDataSource.setAllowLocalTransactions(true)
;
(7)
```

```
myDataSource.setTestQuery("SELECT 1 FROM
DUAL");
(8)
myDataSource.setUseTmJoin(true);
(9)
myDataSource.setDeferConnectionRelease(true)
;
(10)
myDataSource.setAutomaticEnlistingEnabled(tr
ue);
(11)
myDataSource.setAcquisitionTimeout(30);
(12)
myDataSource.setAcquisitionInterval(1);
(13)
myDataSource.setPreparedStatementCacheSize(5
);
(14)
myDataSource.setTwoPcOrderingPosition(0);
(15)
myDataSource.setApplyTransactionTimeout(true
);
(16)
myDataSource.getDriverProperties().setProper
ty("user", "users1");
(17)
myDataSource.getDriverProperties().setProper
ty("password", "users1");
(18)
myDataSource.getDriverProperties().setProper
ty("URL",
```

```
"jdbc:oracle:thin:@localhost:1521:XE");
(19)


Connection c = myDataSource.getConnection();
(20)
 // do some SQL
c.close();
(21)
```

```
myDataSource.close();
(22)
```

1. The Bitronix `PoolingDataSource` is a javabean that implements `java.sql.DataSource`.
2. You have to specify the driver's `XADataSource` implementation here.
3. Each datasource must be assigned a unique name. This is required for distributed crash recovery.
4. This datasource can contain at least 0 connection. 0 is the default value when unspecified.
5. This datasource can contain at most 5 connections.
6. If there aren't enough connections in the pool to fulfill a request, new connections will be created, by increments of 1 at a time.
7. You have to set `allowLocalTransactions` to true if you want to be able to run SQL statements outside of XA transactions scope. Defaults to false.
8. When specified, this query will be executed to check that the connection is still valid before handing it to the application code.
9. Set `useTmJoin` to false if the vendor's `XADataSource` implementation does not implement `XAResource.isSameRM()` properly. Refer to the JdbcXaSupportEvaluation page to see if your database needs it. Defaults to true.
10. Set `deferConnectionRelease` to false if the vendor's `XADataSource` implementation supports transactions interleaving. Refer to the JdbcXaSupportEvaluation page to see if your database supports it. Defaults to true.
11. Set `automaticEnlistingEnabled` to false if you do not want the `PoolingDataSource` to automatically enlist/delist the connections into the XA transactions. You then have to enlist `XAResource` objects manually into the `Transaction` objects for them to participate in XA transactions. Defaults to true.
12. The amount of seconds the pool will block when a connection is requested but the pool is empty and cannot grow anymore. Defaults to 30.
13. The amount of seconds the pool will wait when a connection has been tested invalid before trying to acquire a new one. Defaults to 1.
14. The amount of prepared statements cached per pooled connection. Defaults to 0, meaning statement caching is disabled.
15. The position of this resource during the 2PC protocol execution. This is required if you want to guarantee that a resource commits before another one. Defaults to 0.
16. Should the transaction timeout be passed to the resource via XAResource.setTransactionTimeout() ? Defaults to false.
17,18,19. The driverProperties is a `java.util.Properties` object. You have to add into it a set of property name / property value of the `OracleXADataSource` class. You have to refer to the driver's documentation to know what can / has to be set. The OracleXADataSource javadoc contains this list for the Oracle case. BTM will perform conversion from `String` to `boolean` or to `int` when necessary.
20,21. You can now use the `PoolingDataSource` like any other `java.sql.DataSource`.
22. Remember to close the `PoolingDataSource` after you're done with it to release the connections.

> ✅ **No XADataSource implementation ?**
>
> If your database vendor does not provide a `XADataSource` implementation, you should have a look at the Last Resource Commit optimization.

Minimal settings

You do not have to set properties that have a default value. Here is a simplified version of the previous code creating a `PoolingDataSource` with minimal settings:

```
PoolingDataSource myDataSource = new
PoolingDataSource();
(1)
myDataSource.setClassName("oracle.jdbc.xa.cl
ient.OracleXADataSource");
(2)
myDataSource.setUniqueName("oracle");
(3)
myDataSource.setMaxPoolSize(5);
(4)
myDataSource.setAllowLocalTransactions(true)
;
(5)
myDataSource.setTestQuery("SELECT 1 FROM
DUAL");
(6)
myDataSource.getDriverProperties().setProper
ty("user", "users1");
(7)
myDataSource.getDriverProperties().setProper
ty("password", "users1");
(8)
myDataSource.getDriverProperties().setProper
ty("URL",
"jdbc:oracle:thin:@localhost:1521:XE");
(9)


Connection c = myDataSource.getConnection();
(10)
```

```
 // do some SQL
c.close();
```

(11)

```
myDataSource.close();
(12)
```

This will create a `PoolingDataSource` that will work exactly the same as the previous one. The only difference is that unspecified properties have been left untouched with their default value.

Eager initialization

The connection pool will be initialized during the first call to `getConnection()`. It might be desirable to initialize the pool eagerly, like during application startup rather than having to wait for the first requests. This can be done by calling `init()`:

```
PoolingDataSource myDataSource = new
PoolingDataSource();
(1)
myDataSource.setClassName("oracle.jdbc.xa.cl
ient.OracleXADataSource");
(2)
myDataSource.setUniqueName("oracle");
(3)
myDataSource.setMaxPoolSize(5);
(4)
myDataSource.setAllowLocalTransactions(true)
;
(5)
myDataSource.setTestQuery("SELECT 1 FROM
DUAL");
(6)
myDataSource.getDriverProperties().setProper
ty("user", "users1");
(7)
myDataSource.getDriverProperties().setProper
```

```
ty("password", "users1");
(8)
myDataSource.getDriverProperties().setProper
ty("URL",
"jdbc:oracle:thin:@localhost:1521:XE");
(9)
myDataSource.init();
(10)

Connection c = myDataSource.getConnection();
(11)
 // do some SQL
c.close();
(12)
```

```
myDataSource.close();
(13)
```

Now line 10 will initialize the pool instead of line 11.

Using the Resource Loader

A datasource configuration utility is also bundled with BTM. It is convenient to use it rather than create your datasources in code. Refer to the Resource Loader page for more details.

Here is the equivalent Resource Loader configuration of the previous code example:

```
resource.ds.className=oracle.jdbc.xa.client.
OracleXADataSource
resource.ds.uniqueName=oracle
resource.ds.maxPoolSize=5
resource.ds.allowLocalTransactions=true
resource.ds.testQuery=SELECT 1 FROM DUAL
resource.ds.driverProperties.user=users1
resource.ds.driverProperties.password=users1
resource.ds.driverProperties.URL=jdbc:oracle
:thin:@localhost:1521:XE
```

✅ **Datasource initialization / shutdown**

The Resource Loader will always eager initialize the created datasources and close them when the transaction manager shuts down.

You just have to write those properties in a simple text file and tell BTM where to load it by setting the `resourceConfigurationFilename` property of the Configuration object.

Now you also have to know how to get the datasource created by the Resource Loader. There are multiple ways:

- Look up resources using the embedded JNDI provider.
- Another way is to bind a bitronix.tm.resource.ResourceObjectFactory object, passing it a javax.naming.Reference containing a javax.naming.StringRefAddr containing the datasource's `uniqueName` as `addrType` some

where in your JNDI tree. The `bitronix.tm.resource.ResourceObjectFactory` class will just return the datasource with the specified `uniqueName`. This is explained more in-depth in the Tomcat and Jetty integ ration page.

- The last way is to call bitronix.tm.resource.ResourceRegistrar.get(String uniqueName). This is the least preferred method as this ties your code to BTM which you probably want to avoid.