ConnectionPooling

Installing a JDBC driver

Consult the database documentation to find out where an appropriate JDBC driver can be obtained for the database server that'll be used with Cocoon. Or see list of drivers]. Sun also mantains a [http://industry.java.sun.com/products/jdbc/drivers.

Drivers are usually packaged as zip or jar files. Installation is simply a matter of ensuring that the archive containing the classes is correctly configured in the CLASSPATH.

There are several ways to achieve this:

- Edit the startup script for the application server, and ensure that it's CLASSPATH contains the archive
- For Tomcat, the jar files can be placed in $TOMCAT_HOME/server/lib. Tomcat will automatically add all jar files in this directory to it's CLASSPATH. However it has problems reading zip files, so you may need to rename the file.
- Copy the archive to the $COCOON_HOME/WEB-INF/lib directory
- Instruct Cocoon to load the additional zip/jar files from anywhere in the file-system using the `extra-classpath` configuration parameter in the $COCOON_HOME/WEB-INF/web.xml file.

The `extra-classpath` parameter is normally commented out. Simply remove the comments, leaving the following:

```
<init-param>
  <param-name>extra-classpath</param-name>
   <param-value>WEB-INF/extra-classes1:/[YOU-ABSOLUTE-PATH-TO]/own.jar</param-value>
</init-param>
```

Note that the absolute path to the database driver should be specified.

The recommended method is to simply add the drivers to the $TOMCAT_HOME/server/lib directory. It's the simplest option and avoids the need to maintain a configuration file. However if you have several web applications running under Tomcat that use slightly different versions of the drivers, e.g. to access different versions of the database server, place the files in the $COCOON_HOME/WEB-INF/lib directory instead.

Loading the Driver

The classes for the selected JDBC driver must be loaded prior to connections being created. This allows the driver to automatically register itself with the DriverManager class so that it can be properly used to create connections.

To ensure that the appropriate classes get loaded when Cocoon starts, the class name for the JDBC driver should be added to the `load-class` parameter in the $COCOON_HOME/WEB-INF/web.xml configuuration file.

```
<init-param>
 <param-name>load-class</param-name>
  <param-value>
   <!-- For IBM WebSphere: -->
   com.ibm.servlet.classloader.Handler

   <!-- For JDBC-ODBC Bridge: -->
   sun.jdbc.odbc.JdbcOdbcDriver

   <!-- For Interbase DBMS: -->
   interbase.interclient.Driver
  </param-value>
</init-param>
```

Classes should be referenced using their fully-specified names. Consult the driver documentation for the correct class name.

Creating a Connection Pool

Connection pools, like many other aspects of Cocoon are configured using the `cocoon.xconf` XML configuration file. (See ConfiguringCocoon for more information on performing other tasks).

Datasources such as database connections are configured within the `datasource` element as follows:

```
<datasources>
...
   <jdbc name="pool-name">
     <pool-controller min="1" max="5"/>
     <auto-commit>true|false</auto-commit>
     <dburl>JDBC-connection-string</dburl>
     <user>database-username</user>
     <password>database-password</password>
   </jdbc>
...
</datasources>
```

Note that because connection pools are configured in the `cocoon.xconf` config file, the application server must be restarted to alter these configuration parameters. Unlike the Sitemap which can be re-loaded if it changes, the main configuration file is only read when Cocoon initialises.

The elements in the above examine have the following responsibilities:

- `jdbc` – indicates a JDBC datasource. Must be named. This name is used to refer to the connection pool in XSP pages, etc.
- `dburl` – the JDBC connection string, just as would be provided to

Driver.createConnection(...);

- `user` – the username of the database account to which the connections will be made.
- `password` – the password for the account
- `auto-commit` – whether connections are set to automatically commit or not. The default is `true`

There is one additional set of properties which instructs Cocoon on how many database should be created when the application is started, etc. These are attributes of the `pool-controller` element:

- `min` – the minimum number of connections to maintain in the pool. This will be the number created when Cocoon initialises
- `max` – the maximum number of connections that the pool should ever contain. A pool will grow to this maximum limit under heavy usage, but will shrink (as far as `min`) if the connections become unused
- `oradb` – this attribute, which can be set to `true` or `false` only, is only required when creating connections to Oracle databases. It will ensure that the test to see if the connection is still valid.

Checking that the Pool is initialised

It's possible to check that the pool is intialised correctly in several places.

Firstly the Cocoon `root.log` file will contain a log entry similar to the following:

```
(Unknown-URI) Unknown-thread/CocoonServlet: Trying to load class: org.hsqldb.jdbcDriver
```

If there is a stack trace following this entry, for example a `ClassNotFoundException`, check that the CLASSPATH has been configured correctly.

- Check the logs for SQLExceptions. These may indicate that the classes have not been correctly loaded, or that the connection parameters (the JDBC connect string, username, password, etc) are incorrect.

Getting a Connection

Once a connection pool has been configured, it's possible to obtain connections from it, e.g. in a custom Action, using the following code snippet:

The class must implement the ((Composable)) interface, and is passed the `ComponentManager` automatically. The DataSource can then be cached for later use

```
...
ComponentSelector selector =
        (ComponentSelector) manager.lookup(Roles.DB_CONNECTION);
DataSourceComponent datasource = (DataSourceComponent)
selector.select("my_pool");
...
```

The above code, or its expanded form in the Apache Cocoon docs, did not work for me. Specifically, importing `org.apache.cocoon.Roles` failed. It does not seem to exist anywhere in Cocoon. I have Cocoon 2.1.5.1. --JasonStitt, 2004-07-17.

I found this alternative, presented here in skeleton form:

```
import java.sql.Connection;
import java.sql.SQLException;

// These are from the jar avalon-framework-api-VERSION.jar, in Cocoon's WEB-INF/lib
import org.apache.avalon.framework.component.ComponentException;
import org.apache.avalon.framework.component.ComponentManager;
import org.apache.avalon.framework.component.ComponentSelector;
import org.apache.avalon.framework.component.Composable;

// This is from the jar excalibur-datasource-VERSION.jar in the same directory
import org.apache.avalon.excalibur.datasource.DataSourceComponent;

public class CLASSNAME implements Composable {
    DataSourceComponent datasource = null;

    public void compose(ComponentManager manager) throws ComponentException {
            ComponentSelector dbselector =
                (ComponentSelector) manager.lookup(DataSourceComponent.ROLE + "Selector");
            datasource = (DataSourceComponent) dbselector.select("NameOfPool"); //name as defined in cocoon.
xconf
    }

    // You can then use datasource.getConnection() to get a java.sql.Connection object.
}
```

Of course, it can also be a good idea to separate your data-access object from the Avalon framework. For example, create a go-between class that gets a connection from Cocoon's pool and exposes it as a generic Connection object. If your DAO does not depend on Cocoon, then you can use it easily in other contexts, whether those are other apps or unit tests.

Using the Pool in flowscript

From Flowscript, you can instantiate a Java object that uses the connection pool, use the pool directly, or even query a database without using the pool. The petstore example, placed in `samples/blocks/petstore` in recent Cocoon versions, accesses the database in Flowscript. It demonstrates both using the pool and connecting to the database directly.

The bulk of the code handling database connections is in PetStoreImpl.js. To see the definition of the Database object used, you need the source code to Cocoon. You can find the imported Database.js (and the files it calls) at:

`cocoon-2.1.4/src/blocks/petstore/java/org/apache/cocoon/components/flow/javascript`

Using Java data-access objects in Flowscript

This assumes that you have created a Java object that implements Composable, as described in the previous section.

In Flowscript, you can use the cocoon.createObject() method to instantiate a class that implements an Avalon interface, such as Composable, but is not registered as a component. For example:

`var dataObject = cocoon.createObject(Packages.package.DataObject);`

Note that this doesn't seem to work if your constructor needs any parameters. But you also cannot just instantiate the class using `new`, because then `compose()` and any other functions implementing the Avalon component interfaces will not be called.

Connecting to the database pool in Flowscript

It's fairly simple:

```
var selector = cocoon.getComponent(Packages.org.apache.avalon.excalibur.datasource.DataSourceComponent.ROLE +
"Selector");
var datasource = selector.select("NameOfPool");
var connection = datasource.getConnection();
```

It is also wise to check out the mailing lists; you just may find your solution waiting for you 🙂