Paper 2037-2014

Using Java to Harness the Power of SAS®

Jeremy J. Palbicki, Mayo Clinic, Rochester, MN

ABSTRACT

Are you a Java programmer who has been asked to work with SAS® or a SAS programmer who has been asked to provide an interface to your IT colleagues? Let's face it, not a lot of Java programmers are heavy SAS users. If this is the case in your company, then you are in luck because SAS provides a couple of really slick features to allow Java programmers to access both SAS data and SAS programming from within a Java program.

This paper walks beginner Java or SAS programmers through the simple task of accessing SAS data and SAS programs from a Java program. All that you need is a Java environment and access to a running SAS process, such as a SAS server. This SAS server can either be a SAS/SHARE® server or an IOM server. However, if you do not have either of these two servers that is okay; with the tools that are provided by SAS, you can start up a remote SAS session within Java and harness the power of SAS.

INTRODUCTION

Let's say you have a project in Java or are working with a Java programmer at your site and you or the Java developer needs to be able to deliver data that is prepared by the stats team. The stats team previously produced a large collection of data sets with the desired data but they only use SAS. One way to solve this problem would be to ask the stats team to convert the SAS data sets to MS Excel, CSV, tab delimited, etc. However, converting SAS data sets to a flat file may cause conversion issues or type mismatches if you don't parse the flat file correctly. The best solution would be to have Java work with the SAS data sets directly as if they were in a database management system. Using the SAS tools you can. These tools include a SAS Driver for JDBC and the SAS/CONNECT® Driver for Java.

SETUP

In order to take advantage of the power of SAS within Java, there are a couple of things required. First you need a Java environment (obviously), and within that Java environment you need to add some class files supplied by SAS. Here are the links to the setup programs that allow you to download the jars. Both jars for the JDBC and SAS/CONNECT are downloaded and installed from the same package.

www.sas.com/apps/demosdownloads/setupintro.jsp

```
sas.core.jar
sas.svc.connection.platform.jar
sas.internet.javatools.jar
sas.svc.connection.jar
```

The second thing you will need is a SAS environment for the Java connections. This can be a SAS/SHARE server, SAS/CONNECT server or a SAS/IOM Server. In this presentation I will show examples using SAS/SHARE and SAS/CONNECT. If you are using an IOM server the connection parameters will be slightly different but the general concept will be the same.

For this example, we have a SAS/SHARE server called levsh01 on port number 8551. We also have a SAS/CONNECT server using the spawner called levcn01 at port 7551. If your site does not have a spawner running, as long as you have telnet capability into the machine with SAS running and the ability to launch SAS via the command line, you can use the script concept for connections.

USING JDBC

SAS provides a type 2 JDBC driver for connecting to SAS/SHARE or IOM. Using this driver treats libraries as if they were databases and the data sets as if they were tables. Assuming that the jars listed above are in your classpath and the SAS/SHARE server is running as described above, here is code used to establish the JDBC connection.

Here is the list of imports used for the examples provided below:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;
import com.sas.net.sharenet.ShareNetDriver;
import com.sas.net.connect.ConnectException;
import com.sas.net.connect.ConnectLogException;
import com.sas.net.connect.TelnetConnectClient;
```

Here is an example of code used within the Java program to create the connection:

```
…

Connection conn;

Class.forName("com.sas.net.sharenet.ShareNetDriver");


String url = "jdbc:sharenet://levsh01:8551;
Properties prop = new Properties();

prop.setProperty("shareUser","MYUSERNAME");
prop.setProperty("sharePassword", "MYPASSWORD");
prop.setProperty("appname", "MYAPPNAME");
prop.setProperty("librefs", "MYLIB '/home/stats/sasdata/';" );

try {

      conn = DriverManager.getConnection(url,prop);
      return true;

} catch (SQLException e1) {

      e1.printStackTrace();
      return false;
}
```

Once this connection is made you can then proceed with using the connection object to build queries to run on the SAS server. The example below assumes there is a data set called statsdata1 in the folder "/home/stats/sasdata/":

```
Statement stmt;
ResultSet rslt;

stmt = conn.createStatement();
rslt = stmt.executeQuery("Select * from mylib.statsdata1");
```

At this point you now have a Java ResultSet that contains all of the contents of the data set statsdata1.

USING SAS/CONNECT

Another wonderful product that one can use to integrate SAS and JAVA together is the SAS/CONNECT Driver for Java. With this product you can create a connection from Java to a SAS server to actually submit SAS code and interact with a SAS session. There are a couple of options available to connecting to the SAS server. Just like in SAS, SAS/CONNECT has two methods of connections, scripted and spawner. In using the

SAS/CONNECT Driver for Java we do not need to actually use a script file to connect, but the concept is the same. Below is code from Java to create both types of connections to a MVS server. In this example, we assume the name of the MVS server is “levcnsrv”.

If you are not using a spawner, use the code below. This example uses a standard telnet port and for the case of MVS requires that the user has TSO access:

```
Properties prop = new Properties();

prop.setProperty("prompt1", "USERID");
prop.setProperty("response1", username);
prop.setProperty("prompt2", "PASSWORD");
prop.setProperty("response2", password);
prop.setProperty("prompt3", "===>");
prop.setProperty("response3", "5");
prop.setProperty("prompt4", "READY");
prop.setProperty("response4","sas
options('dmr,comamid=TCP,noterminal,no$syntaxcheck')ddsasusr(work)");
prop.setProperty("sasPortTagTimeout", "45");
prop.setProperty("epilogPrompt1", "READY");
prop.setProperty("epilogResponse1", "logoff");
prop.setProperty("epilogPrompt2", "LOGGED OFF TSO");
try {
      conn = new TelnetConnectClient(prop);
      conn.connect(“levcnsrv”);

} catch (ConnectException e) {
      e.printStackTrace();
      return false;
}
```

Using a spawner creates more flexibility because you do not need TSO access to log on. Here is the code to connect to the spawner process:

```
Properties prop = new Properties();


prop.setProperty("prompt1", "Username:");
prop.setProperty("response1", username);
prop.setProperty("prompt2", "Password:");
prop.setProperty("response2", password);
prop.setProperty("prompt3", "Hello>");
prop.setProperty("response3","/u/SAS93/spncshel.sh nosasuser opt(''dmr
comamid=tcp'')");

try {
      conn = new TelnetConnectClient(prop);
      conn.connect(“levcnsrv”, 7551); //7551=spawner process port number

} catch (ConnectException e) {
      e.printStackTrace();
      return false;
}
```

You should contact your SAS system administrator if you have questions about the ports and SAS startup commands.

Now that you have a connection called conn, what are some of the functions you can do with it? Here are just a couple of examples of things you can do with the connection:

```
String libcode = "libname a 'userx.sasdata.sas'; ";
String datacode = "data a.test; x=1; y=2; output; x=3; y=4; output; run; ";
conn.rsubmit(libcode);
String log = conn.getLogLines();
conn.rsubmit(datacode);
String log2 = conn.getLogLines();
```

At this point you have now executed a couple of SAS steps, one to create a libref to a MVS library and another to create a data set called test in that library. You also have two variables called log and log2 that contain the system log from the execution. The links at the end of this paper will provide you a complete list of methods you can call from the SAS/CONNECT Driver for Java.

Another great feature of the SAS/CONNECT Driver for Java is the ability to return a JDBC connection object. This is where the power of SAS really takes hold. Once you have created the above connection and created a data set, what if you wanted to read that data set back in to a Java ResultSet? Here is how:

```
try {

      Connection jdbc_con = conn.getSharenet();// get the JDBC connection
      PreparedStatement ps = jdbc_con.prepareStatement("select * from
a.test");
      ResultSet rs = ps.executeQuery();
      while (rs.next()){
            // perform operations on ResultSet.
      }
      rs.close();
      jdbc_con.close();
      conn.disconnect();

} catch (ConnectException e) {
      e.printStackTrace();
} catch (SQLException e) {
      e.printStackTrace();
}
```

CONCLUSION

Harnessing the power of SAS using Java can be a very useful tool. It can eliminate the need for additional translational steps between Java applications and SAS data sets and it can also allow for easier collaboration between IT staff and analysis staff. These tools can provide greater flexibility when developing Java applications that need to display advanced statistical analysis in real time.

RECOMMENDED READING

- *SAS/CONNECT® Driver for Java*

  http://support.sas.com/documentation/cdl/en/inetjava/64897/HTML/default/viewer.htm#titlepage.htm

- *SAS® Drivers for JDBC Cookbook*

  http://support.sas.com/documentation/cdl/en/jdbcref/65037/HTML/default/viewer.htm#titlepage.htm

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Name: Jeremy J. Palbicki
Organization: Mayo Clinic

Address: 200 First St. S.W.
City, State ZIP: Rochester, MN 55905
Email: palbicki.jeremy@mayo.edu