Paper 017-2010

The Best of Both Worlds: Integrating a Java Web Application with SAS® Using the SAS/SHARE® Driver for JDBC

Adam Rauch, LabKey Corporation, Seattle, WA

ABSTRACT

Life would be simple if all data were stored and accessed via SAS® software. In reality, most organizations store data in multiple repositories and access this data using a variety of analytical tools that don't communicate with each other. Technology solutions that bridge between analytical systems and data repositories are critical to sharing information throughout the organization. The SAS/SHARE® driver for JDBC is a powerful example of bridging technology that increases the value of SAS data to an organization.

We demonstrate that an integration solution using the SAS/SHARE driver is both useful and suitable for high-throughput production use. We extended an existing Java-based life sciences web application to integrate secure, dynamic access to SAS data sets. We document the implementation process, detailing best practices, pitfalls, and workarounds discovered along the way.

BACKGROUND AND PROBLEM STATEMENT

LabKey Server is an open-source, web-based system for collaborative biomedical research. It provides a secure data repository for managing and sharing laboratory data, including proteomics, microarray, flow cytometry and plate-based assay data. Life sciences organizations have deployed the system to solve a variety of data management problems, including managing large HIV/AIDS vaccine trials, analyzing high-throughput proteomics data for cancer biomarker research, and providing repositories of infectious disease research using flow cytometry. More information about the product is available at the LabKey Software Foundation website, http://www.labkey.org.

LabKey Server is written in Java and uses a relational database (PostgreSQL or Microsoft SQL Server) to store its data. Data is added to the system via a combination of automated systems (e.g., analytic pipelines that pull assay data directly from laboratory instruments, nightly uploads of specimen information from external data dumps) and manual processes (e.g., direct entry of subject responses into web forms, manual uploads of assay data by remote laboratories). The system organizes all data within a security framework that ensures sensitive information can be viewed and altered only by those who are explicitly granted authorization.

Once in the system, general-purpose data management tools allow querying, joining, reporting, charting, and exporting of data via a web-based user interface. Also provided is an extensive set of domain-specific data analysis and management tools customized to understand the intricacies of specific assays and the requirements of scientists using those assays. Every attempt to access data is subject to authorization checks performed at the application level.

The Statistical Center for HIV/AIDS Research and Prevention (SCHARP) provides data management and statistical analysis expertise to several large consortia working toward an HIV/AIDS vaccine. SCHARP is part of the Vaccine and Infectious Disease Institute of the Fred Hutchinson Cancer Research Center based in Seattle, WA. SCHARP operates one of the most broadly used installations of LabKey Server to help coordinate these large-scale collaborations. As a statistical center, SCHARP also makes extensive use of SAS as a repository and analysis system.

The SCHARP installation of LabKey Server houses hundreds of observational studies; it provides secure analysis and publishing of study and assay data to thousands of individual collaborators. External users are not permitted direct access to SCHARP's SAS installation. Instead, select data sets are published on LabKey Server where specific groups are granted access. In the past, publishing a SAS data set meant exporting it to text format, typically a tab-separated value (TSV) file, and instructing LabKey Server to load the text file; this is accomplished via automated scripts in some cases and via manual processes in others.

This type of export/import process has many drawbacks. It is complex, fragile, and prone to errors that are difficult to detect. It is also time consuming, awkward, and unnecessarily resource intensive. Once transferred, the resulting data is less than ideal. It may be out-of-date immediately, since the data set could change in SAS. The transfer

process via TSV files is also inherently “lossy”: we lose important information about SAS formats, special missing values, data set labels, variable descriptions, and other critical meta data.

In an effort to resolve these issues, SCHARP sponsored a project to create more seamless integration between SAS data sets and LabKey Server. The key requirements for this solution included:

- Relatively low cost (this precluded solutions that entailed extensive re-architecting of the Java application)
- Support for the entire suite of analytical tools on SAS data sets. SCHARP had built expertise and code around the standard LabKey features that operate on tables stored in the primary database; they wanted SAS data sets to behave identically. This meant that SAS data set integration had to support:
    - Sortable, filterable HTML grids with paging.
    - User-defined views saved with custom filters, sorts, and column lists.
    - Custom queries and reports.
    - Export to Excel, web query, and TSV formats.
    - Access from JavaScript, Java, R, and SAS client libraries.
- Strict adherence to the existing LabKey Server security model. SAS data sets must follow the same authorization steps as any other data set. SCHARP wanted to continue using the groups and permissions they had configured for their 2,000+ users on the system and did not want to open up their SAS installation beyond their internal use.
- Transfer of SAS meta data along with the data sets.

SOLUTION OVERVIEW

After considering several possible solutions, an approach employing SAS/SHARE and the SAS/SHARE driver for JDBC quickly became the leading contender for meeting the above requirements. JDBC is the Java Database Connectivity API, an industry-standard interface for connecting Java applications with a wide range of databases. LabKey Server accesses its primary database server using JDBC, so its existing data management tools are already tuned to this API.

A SAS/SHARE-to-Java solution involves three key components:

- SAS/SHARE service. SAS/SHARE provides concurrent access to SAS data sets over a network. This service is configured to run as part of the SAS installation. Configuration involves setting up a TCP/IP port, defining libraries to publish, and starting the service.
- SAS/SHARE driver for JDBC. The driver JAR files must be on the classpath of the Java application.
- Java code that loads the driver, connects to the SAS/SHARE service via its URL and port, and executes queries against the service via the JDBC API.

Once configured, the Java application sees the SAS repository as a relational database similar to PostgreSQL or Microsoft SQL Server. Using JDBC means that several familiar SAS constructs are accessed and referred to by their database counterparts:

- Each SAS library defined by SAS/SHARE appears as a database “schema.”
- Each data set appears as a “table” within its schema.
- Each SAS variable appears as a “column” within its table.

The Java application uses JDBC to retrieve data stored in SAS data sets. It can also retrieve “meta data” associated with schemas, tables, and columns; meta data is information about types and properties, for example: name, data type, and description of each column or the primary keys and descriptions of each table.

TEST IMPLEMENTATION

We decided to test the approach by writing a simple command line Java application that connects to a local instance of SAS/SHARE. During the development and testing phase, we ran SAS, our Java development environment, our web application, and our test database server on our development machines. This is highly recommend since it allowed us to change the SAS/SHARE service configuration, change the published libraries, experiment with authentication, execute performance tests, modify data sets, etc. without disturbing a production SAS installation.

The *SAS® 9.2 Drivers for JDBC Cookbook* from SAS Institute provides detailed reference information for configuring and connecting with the SAS/SHARE service. Below are the basic steps for setting up SAS/SHARE on a development machine configured with Windows 7 and SAS 9.2 (changing file paths should make these steps compatible with other configurations):

- In your "services" file, configure a TCP/IP port for SAS/SHARE to use. We chose to assign port 5010 to the SAS/SHARE service by adding this line to our services file:

  ```
  sasshare    5010/tcp    #SAS/SHARE server
  ```

  The location of the services file changes based on your operating system, so consult your operating system documentation for the correct path. It's likely located in c:\windows\system32\drivers\etc\services on Windows; on Linux, Unix, and OS X, you may find it in /etc/services.

- Define one or more libraries and start the SAS/SHARE service. Run SAS and execute the following statements:

  ```
  libname sample 'C:\Program Files\SAS\SASFoundation\9.2\core\sample';
  proc server authenticate=optional id=sasshare;
  run;
  ```

  The SAS/SHARE service should now be running. **Important:** The "authenticate=optional" setting runs SAS/SHARE in a completely open, unsecured manner. It should be used for development purposes only.

- Place the SAS/SHARE driver for JDBC JAR files (sas.core.jar and sas.intrnet.javatools.jar) on your java classpath.

- Write Java code that connects to the SAS/SHARE service defined above. The basic approach looks like this:

  ```
  Class.forName("com.sas.net.sharenet.ShareNetDriver");
  String url = "jdbc:sharenet://localhost:5010?appname=JdbcTest";
  conn = DriverManager.getConnection(url);

  Statement stmt = conn.createStatement();
  String sql = "SELECT * FROM sample.empinfo";
  ResultSet rs = stmt.executeQuery(sql);

  while (rs.next())
     System.out.println(rs.getString("name") + " " + rs.getInt("idNum"));

  rs.close();
  stmt.close();
  conn.close();
  ```

  A full test application would include exception handling and would probably log more columns and meta data. It would also exercise functionality beyond a simple SELECT statement. See Appendix A for Java source code of a complete SAS/SHARE JDBC test application that can be compiled and run from the command line.

The test application we wrote retrieved data and meta data from many sample data sets. It also confirmed that basic SQL operations such as WHERE, ORDER BY, GROUP BY, aggregates, aliases, sub-selects, and standard JOINs were all supported. After successful tests, the next step was to integrate the SAS/SHARE connection capability into LabKey Server.

INTEGRATION

Writing an application to test Java and SAS/SHARE integration was quite straightforward; the driver worked as advertised and seemed to follow the JDBC specification. Integrating SAS/SHARE into a production system was significantly more difficult; it required some changes to core code and revealed problems with the driver that had to be dealt with.

SAS/SHARE CONFIGURATION

The test application includes a hard-coded connection string ("jdbc:sharenet://localhost:5010?appname=JdbcTest") and no authentication credentials. This is unacceptable in a production environment, where the code must be independent of the SAS/SHARE configuration details. Also, establishing and closing a database connection for each query is unnecessarily expensive.

In Java web applications, configuration of external resources is often done using the Java Naming and Directory Interface (JNDI). Configuration details are specified in a separate XML file, keeping code and settings independent. JDBC database connection configurations are specified by adding a DataSource element to the web application's context file, for example:

```
<Resource name="jdbc/sasDataSource"
        auth="Container"
        type="javax.sql.DataSource"
        driverClassName="com.sas.net.sharenet.ShareNetDriver"
        url="jdbc:sharenet://localhost:5010?appname=LabKey"
        maxActive="8"
        maxIdle="4"/>
```

This element defines a DataSource named "sasDataSource" with settings very similar to the test application above. The URL is the same, other than changing the "appname" parameter to the production server's name. (The "appname" parameter is optional and SAS-specific, but it can be helpful because it appears whenever SAS/SHARE logs activity from this connection. You should change the value of this parameter to your own application's name.) This particular configuration is suitable only for development use since it assumes authentication is optional on the SAS/SHARE service. However, it's a trivial process to add credentials once you configure your SAS/SHARE service to require authentication: simply add "username" and "password" properties and they will be sent to SAS/SHARE when making a connection.

The SAS/SHARE driver for JDBC JAR files must be on the classpath on the server. The exact location will vary by application server. With Apache Tomcat, placing the JAR files in <tomcat>/common/lib is sufficient.

A JDBC connection to a JNDI-defined DataSource is obtained via code that looks like this:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/sasDataSource");
Connection conn = ds.getConnection();
```

This code loads a specific DataSource by name, but many applications will take a more general approach. LabKey Server, for example, enumerates all the DataSources specified in a context file and displays their names in a drop-down list available to administrators. Administrators then choose which set of DataSources, schemas, and tables to publish in each area of the web site, which lets the administrator fine-tune permissions for each data set. The system can connect to multiple databases of different types; for example, a server could connect to three SAS/SHARE servers, a PostgreSQL database server, and two Microsoft SQL Server databases.

In addition to hiding the configuration details (and in particular, the credentials) from the code, our DataSource definition provides another very important feature: a connection pool. The "maxActive" and "maxIdle" properties define a pool of connections that are maintained by the connection manager. The connection manager doesn't make a database connection every time your code asks for a connection; instead, it loans out previously established connections from its pool. A properly configured database connection pool is critical for most high-throughput web applications. The pool defined here is relatively small; consult your connection pool documentation to determine the proper settings for your application.

HANDLING SAS/SHARE DIFFERENCES

JDBC hides many of the differences between databases from the Java code that operates on them, but it can't hide all differences. Despite SQL standardization efforts, every database server requires different SQL syntax for common features and supports a unique set of capabilities. An application built to communicate with different database servers needs a way to address these differences.

Because it supported multiple primary databases (PostgreSQL and Microsoft SQL Server), LabKey Server had in place an abstraction layer that hides database-specific implementation details from the application layer. Adding a third provider to this layer to support SAS/SHARE was relatively straightforward.

We call this the "SQL dialect" layer. It's implemented as an abstract class with an implementation subclass for every database we support. The application layer calls SQL dialect methods to determine capabilities and SQL syntax supported by each database. For example, the SAS/SHARE driver doesn't allow in-line comments in SQL queries, so

the SAS dialect's supportsComments() method returns false. Likewise, the SAS dialect's getConcatenationOperator() method returns "||".

ISSUES, WORKAROUNDS, AND RECOMMENDATIONS

Developing and testing the SAS/SHARE integration uncovered a significant number of issues that had to be addressed. Some issues were very minor, requiring a simple code change or a new setting in the SAS SQL dialect. Others were more involved, requiring major reworking of the code. Issues we uncovered included:

- Obtaining the JDBC driver. Most database server vendors post their JDBC drivers on the Internet, available for free download. As of the time of this writing, the most recent JDBC driver is not available for download (only version 9.1.3). We strongly recommend using the most recent driver (see below), so you'll need to **acquire the 9.2 driver from SAS Institute**. Note that the standard installation places the JDBC driver JAR files in an obscure directory such as:

  ```
  C:\Program Files\SAS\SASFoundation\9.2\eclipse\plugins
  ```

  Copy the JDBC driver JAR files (sas.core.jar and sas.intrnet.javatools.jar) from there to your application server or Java project.

- JDBC driver differences. The 9.2 JDBC driver changed significantly from the 9.1.3 JDBC driver. For example, the names used to retrieve both column and primary key meta data changed. (With the 9.1 driver, column meta data is reported as "NAME", "SQLTYPE", "SIZE" while in 9.2 it is reported as "COLUMN_NAME", "DATA_TYPE", "COLUMN_SIZE".) The newer names are more consistent with other JDBC drivers, but if your application might encounter either driver you must detect the driver version and accommodate the differences. (In our case, LabKey Server uses a different SQL dialect for each driver version.) Since JDBC drivers are usually backwards compatible, the easiest solution here is to **always use the most recent 9.2 JDBC driver**, even against a 9.1 SAS installation.

- 9.2 driver versions. SAS released several versions of the 9.2 JDBC driver. The first release had a critical bug in its handling of database meta data; a call to Connection.getMetaData().getColumns() returned information claiming that every variable in the data set was type "double" (even character and date variables)! Fortunately, SAS Institute has released newer versions of the driver that fix this problem. As with the above issue, **always use the most recent 9.2 JDBC driver**. At this time, the most recent version is called 9.22-m2; the driver lists a version number of 9.2.902200.2.0.20090722190000_v920m2.

- No database name. Unlike most database servers, SAS/SHARE doesn't use a database name. JDBC clients simply connect to a port and see all the published libraries. General purpose clients that expect a database name will need to change to address this difference.

- DatabaseMetaData not completely implemented. The DatabaseMetaData JDBC interface is used by clients to determine capabilities of the current driver and database. The SAS JDBC driver does not fully implement this interface; 18 of the standard methods throw InvocationTargetException. Many of the unimplemented methods are fairly obscure, but several are important for customizing behavior based on driver and SAS versions: getDatabaseMajorVersion(), getDatabaseMinorVersion(), getJDBCMajorVersion(), getJDBCMinorVersion(). In place of these more convenient methods, **use getDriverVersion() and getDatabaseProductName()** and parse the strings they return.

- ResultSet.getRow() method not implemented. Calling getRow() always throws a SQLException ("Method not yet supported"); **code that needs to track the current row must do so manually**.

- Paging. Many applications provide "paging" of large data sets; they display (for example) only the first 1,000 rows and provide a means to skip to other pages containing 1,000 rows each. Most database servers provide "limit" and "offset" predicates in their SQL syntax to enable efficient paging implementations. The SAS/SHARE SQL syntax does not support these commands, which makes implementing paging tedious and inefficient. Paging is important for our application, so we implemented paging manually. If a limit of 1,000 rows is requested, we select all rows but display only the first 1,000 rows. If we're displaying the third page of 1,000 rows, we select all the rows, skip through the first 2,000 rows, then display rows 2,001 to 3,000. This approach incurs extra database work and network traffic, but it gets the job done.

- Dates. A JDBC PreparedStatement allows for parameterized SQL queries; SQL is written with placeholders (?) that take their values from Java objects that are added to the PreparedStatement. We discovered that, unlike other database servers, the SAS/SHARE driver does not support java.sql.Timestamp objects as PreparedStatement parameters; using a Timestamp object as a parameter in a WHERE clause resulted in completely unpredictable results. When setting a date parameter, **use a java.sql.Date object** instead. SAS/SHARE SQL also supports a date literal syntax; the following SQL statement is legal:

```
SELECT * FROM example.hosp WHERE AdmitDate = '12JUL05'd
```

However, including hard-coded date strings in SQL is rarely useful.

- No support for in-line comments. Most of the SQL statements sent to the database server by LabKey Server are generated by code. We include comments inside the SQL of some of the more complex queries to help with debugging. Most database servers ignore comments inside SQL statements, but SAS/SHARE does not; any comments result in an exception. To resolve this issue, we added a supportsComments() method to the SQL dialect; our code checks this method before adding comments to a SQL statement.
- Alias names. SAS/SHARE required some small changes to code that generated alias names for columns and tables. We had used dollar signs ($) in alias names, but these were rejected by SAS/SHARE; we replaced them with underscores (_).
- Formats. One major feature we have not yet been able to support is retrieving formatted values from SAS/SHARE. Retrieving raw data is useful in many scenarios, but in other cases formatted values would be very helpful. The SAS/SHARE SQL syntax supports the PUT function, which applies a format to the given column, and the format name associated with a variable can be retrieved from JDBC meta data. The problem is associating the right format library with each dataset. One can set a format library search path when establishing a SAS/SHARE session, but a single search path is too restrictive for most SAS repositories. Our client, for example, stores datasets in hundreds of studies, each of which has a unique format library. A single format library search path does not work at all and the alternative, creating separate SAS/SHARE services for each protocol, would be an operational nightmare. We are searching for a solution to this issue; please contact the author if you discover one.

GENERAL RECOMMENDATIONS

Integrating SAS/SHARE with Java requires substantial development effort. Here are a few general recommendations for developers embarking on such a solution:

- Configure your development machine as a complete, standalone test environment. Install SAS, configure SAS/SHARE, and run your Java application all on your development machine. Development and testing will require frequent restarts and reconfiguration of both your application and SAS/SHARE; you certainly don't want to test against a production SAS installation. Having complete control over all the components will speed the development process.
- Implement general data and meta data browsers. As you develop your solution, you'll want an easy way to display both the data and the meta data returned for each data set. Invest in general-purpose browsing UI that displays this information. Even simple logging routines will provide great value.
- Design your security approach early in the process. How will you protect the production SAS/SHARE server? How does your Java application appropriately limit access to data sets? How will an administrator control which users can access each data set?

CONCLUSION

The SAS/SHARE driver for JDBC gives Java applications direct access to SAS data. This bridging technology supports a robust set of database functions and performs well in high-throughput environments. Using the SAS/SHARE driver for JDBC, and learning to work around its limitations, provides a powerful tool for sharing valuable SAS data throughout an organization.

ACKNOWLEDGMENTS

We would like to thank Rana Bonnice, Iraj Mohebalian, Roy Obenchain, Sarah Ramsay, Geoff Snyder, and Julie Stofel of The Statistical Center for HIV/AIDS Research and Prevention (SCHARP) at Fred Hutchinson Cancer Research Center for all their help in designing, testing, and deploying this solution.

This work was funded in part by the Microbicide Trials Network under National Institute of Allergy and Infectious Disease (NIAID) grant 5 U01 AI068615-04.

RECOMMENDED READING

SAS® 9.2 Drivers for JDBC Cookbook, SAS Institute Inc.

CONTACT INFORMATION

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

Name:	Adam Rauch
Enterprise:	LabKey Software
Address:	2226 Eastlake Ave. East, #101
City, State ZIP:	Seattle, WA 98102
Work Phone:	206/667-7012
Fax:	206/667-4831
E-mail:	adam@labkey.com
Web:	http://www.labkey.com

APPENDIX A: SAS/SHARE JDBC TEST APPLICATION

```
import java.sql.*;

// Simple test of querying SAS data sets using the SAS/SHARE driver for JDBC
public class Main
{
   public static void main(String[] args)
   {
      Connection conn = null;

      try
      {
         Class.forName("com.sas.net.sharenet.ShareNetDriver");
         String url = "jdbc:sharenet://localhost:5010?appname=JdbcTest";
         conn = DriverManager.getConnection(url);

         DatabaseMetaData dma = conn.getMetaData();
         System.out.println("Connected to " + dma.getURL());
         System.out.println("Driver " + dma.getDriverName());
         System.out.println("Version " + dma.getDriverVersion());

         logSchemas(dma);
         logTables(dma, "sample");

         Statement stmt = conn.createStatement();

         executeAndLog(stmt, "SELECT * FROM sample.empinfo");
         executeAndLog(stmt, "SELECT * FROM sample.leave");
         executeAndLog(stmt, "SELECT * FROM sample.empinfo INNER JOIN
            sample.leave ON empinfo.idnum = leave.idnum ORDER BY Name");
         executeAndLog(stmt, " SELECT * FROM sample.empinfo e LEFT OUTER JOIN
            sample.leave l ON e.idnum = l.idnum  WHERE Location <> 'Cary'");
         executeAndLog(stmt, "SELECT * FROM sample.leave RIGHT OUTER JOIN
            sample.empinfo ON empinfo.idnum = leave.idnum");
      }
      catch(Exception e)
      {
         e.printStackTrace();
      }
      finally
      {
         try
         {
            if (null != conn)
               conn.close();
         }
         catch (SQLException e)
         {
            e.printStackTrace();
         }
      }
   }
```

```
   private static void logSchemas(DatabaseMetaData dmd) throws SQLException
   {
      logResultSet(dmd.getSchemas());
   }

   private static void logTables(DatabaseMetaData dma, String schemaName)
      throws SQLException
   {
      ResultSet tables = dma.getTables(null, schemaName, null, null);
      logResultSet(tables);
   }

   private static void executeAndLog(Statement stmt, String sql) throws
      SQLException
   {
      ResultSet rs = stmt.executeQuery(sql);
      logResultSet(rs);
      stmt.close();
   }

   private static void logResultSet(ResultSet rs) throws SQLException
   {
      try
      {
         ResultSetMetaData md = rs.getMetaData();
         int columnCount = md.getColumnCount();

         for (int i = 1; i <= columnCount; i++)
            System.out.print(md.getColumnName(i) + "(" +
               md.getColumnType(i) + ") ");

         System.out.println();

         while (rs.next())
         {
            for (int i = 1; i <= columnCount; i++)
            {
               Object o = rs.getObject(i);
               System.out.print(null == o ? "" : o.toString() + " ");
            }

            System.out.println();
         }
      }
      finally
      {
         if (null != rs)
         {
            rs.close();
         }
      }
   }
}
```