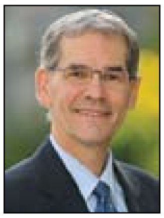
# Barista Uses Your RDBMS of Choice

Barista® is an application development and runtime framework written in BBj® and, like thousands of applications developed with BASIS products, was initially created to utilize the extremely fast and efficient native BBx® file system. Used to create new GUI applications, refresh old GUI applications, or transform existing CUI applications to GUI, this data-driven framework takes over an enormous amount of the development effort by simply describing the data model in Barista's data dictionary. BASIS realized this metadata concept could be as revolutionary to the widespread market of SQL or relational database applications. To facilitate this, BASIS decided to "open up" Barista to allow any JDBC 2.0-compliant relational database to act as the user data repository! To accomplish this large goal, BASIS broke it down into smaller tasks.



***By Ralph Lance***
*Software Engineer*

## The Road Ahead

The first challenge was for BASIS to change all data I/O from the native BBx file system to database independent SQL in order to work with any relational database. While this would otherwise be a sizable challenge for any software house, Barista's modular construction reduced it to a manageable project since most of the data I/O was reasonably contained.

The next challenge was retrieving all the required information from the foreign database. Since database architects have created their databases in SQL, it was important to use that metadata to build the necessary Barista table and data element descriptions while accommodating incompatibilities with the architecture of the BASIS data dictionary. For example, SQL databases may allow table and column naming conventions incompatible with Barista, so it was necessary to modify these names without losing track of their original values.

Integrating with third party databases also presented Barista with data layout challenges. BASIS modified Barista to translate the external data to and from the required standard BASIS record templates to resolve the data inconsistency. This solution made the least impact on Barista's existing code base.

One challenge was already satisfied by Barista. It has, since its debut, used SQL for the table inquiry function. The only additional need was to ensure that the SQL commands were standardized.

Lastly, Barista and the external database may both have their own authentication mechanisms, such as user name and password, so the connection information and user credentials needed to be stored in Barista tables.

## Interfacing With the Third Party Database

BBj and the BASIS SQL Engine offer a myriad of possibilities and functionality targeted at solving specific SQL-related tasks, while abstracting the underlying complexities. This makes it easy for BBj developers to work with a RDBMS database either by using SQL commands or using objects like the BBjRecordSet. However, Barista needed to interface with databases at a deeper level than these constructs provided. For example, Barista needed to obtain metadata, control

>>

the volume of data being processed, and use result set navigation to avoid constructing individual database-specific SQL statements for every data I/O operation.

A new BBj API method came to the rescue – getJDBCConnection(). It returns a BBjCollapsableJDBCConnection object that represents a connection to the specified database. This object is an implementation of the **`java.sql.Connection`** interface that opens up access to the complete functionality of the JDBC API. This method is the cornerstone of a BBj custom class called BBjdbo (BBj JDBC Database Object) found in the Barista program **`bsq_bbjdbo.bbj`**. The BBjdbo class wraps many of the java.sql classes available and handles all of the interfacing to the SQL database via the JDBC 2.0-compliant driver.

Instantiation of the BBjdbo object occurs when connecting to a database where a native file open would normally have been performed. Connection and user credential information from Barista tables are used to make a connection to one or more databases. To keep the connection overhead under control, the class takes advantage of BBj's connection pooling feature to cache database connections. The class also handles result sets on a per table basis and manages their "channels" in a HashMap also stored in the group namespace.

When importing table description information from an existing database, metadata is interrogated for table, column, and index information. During the import, table and column names are modified to fit into Barista's data dictionary as needed. The original names, along with their catalog and schema, are stored in the Barista table and column records. Barista automatically generates data dictionary keys and their segments from the obtainable index information. The import also does its best to identify primary and foreign key relationships and sets up these relationships as validation tables in Barista's data element definitions. For more information about importing, see the link at the end of this article.

After describing tables and their data elements in Barista, the developer can easily generate the standard data maintenance forms and immediately use them to query and maintain the data in the external database.

## Implementation Details

The actual data access either occurs from result set navigation (first(), last(), next(), previous()) or via BBjdbo methods like getRecordByKey(). Keep in mind that an SQL "key" translates to a WHERE condition that may be composed of more than one segment. To accomplish this, Barista constructs a HashMap of <column name>, <operator>, <value> conditions from the field-based key segments defined for a table such as **`cust_num, =, 000012`**. The column name and operator are used to build the WHERE clause for a prepared statement, e.g. **`WHERE cust_num=?`**. Values are passed to a method that sets the query parameters. The use of the Statement.setObject() method eliminates the burden of dealing with different data types.

The SELECT statement built by Barista for inquiries is simply passed to an execute() method that returns the SQL result set as a vector of BBjTemplatedStrings. Barista uses this vector, instead of the READ RECORD loop traditionally used to access the native file system. The maximum number of result set rows returned to the user has a configurable global default value to prevent memory (or user!) overload.

## Dealing with Database Differences

Barista queries the database metadata for, among other things, the characters that wrap identifiers in SQL statements to avoid conflicts with reserved words. This is a good example of using what the database itself offers as metadata to avoid hard coding for idiosyncrasies in different databases. Although the JDBC 2.0 API is a "standard," it is up to the driver developers to implement the API interfaces and to determine to what extent they adhere to the standard. Many drivers also have database-specific extensions to this standard. The Barista SQL integration is based on the JDBC standard and in only a couple of places does the database product name play a part in the program logic.

External databases may or may not support SQL transactions and may have different capabilities for constraints, stored procedures, triggers, and the like. As a result, BASIS does not offer explicit support for these in Barista. Unsuccessful SQL operations caused by constraints will notify the user that Barista is unable to complete the process.

Handling large character and binary objects (CLOBs and BLOBs) are usually application-specific and currently handled as non-editable "attachments." Barista displays the first 400 bytes in forms and in the future, will make a call to an attachment viewer.

## Summary

BASIS has tested the SQL integration with a number of the more popular relational databases; Oracle, MySQL, SQL Server, JavaDB/Derby, and BASIS' own ESQL relational database. Test for yourselves by checking out some of the sample demo databases and JDBC drivers available for download from the various vendors. Discover first-hand how SQL integration in Barista now gives more users more choices when using BASIS products. 🟩

For more information about importing, see the tutorials at
www.basis.com/products/devtools/barista/documentation

Read more about Java implementation at
java.sun.com/javase/6/docs/api/java/sql/Connection.html