Using JTOpen with Android to access IBM i

Two new offerings enable mobile applications!

John W. Eberhard
Jesse Gorzinski

December 05, 2013

This article provides an overview of the two most recent additions to the IBM Toolbox for Java and JTOpen offering. These new Java™ packages allow you to write native applications for Android-based devices that need to interact with IBM i servers.

Introduction

If you've written Java applications that communicate with IBM i, you already know that the IBM Toolbox for Java allows you to access and interact with your server's data and resources from desktop and web applications. But did you know that the Toolbox for Java can bring the same capability to Android devices? With the latest set of offerings, it can!

The IBM Toolbox for Java has been around for quite some time. In fact, it has been around since V4R2M0 (1998). As you can imagine, in its 15 years of existence, the toolbox has evolved and grown to suit a wide variety of needs. Naturally, some of these needs have been in the mobile device world. For instance, when Java ME was driving market demand for Java on mobile devices, the toolbox released a .jar file compatible with ME applications. Now, with the growth of Android operating systems, the toolbox has expanded with new offerings to enable application development on that platform.

You might recall that the IBM Toolbox for Java can be acquired in a couple ways:

- In V6R1, the most important JAR files are included in an installable licensed program product (LPP) on IBM i. This LPP is named 5761-JC1 and is completely free. In V7R1, the JAR files are now included with option 3 of 5770-SS1. Updates to these JAR files are delivered through interim fixes.
- The open-source package (known as JTOpen) can be downloaded from sourceforge.net.

To get the complete set of deliverables, you need to download the entire JTOpen package from sourceforge.net. This package includes a number of JAR files. Currently, that list includes the following files:

- composer.jar

- jt400.jar
- jt400android.jar
- jt400Micro.jar
- jt400Native.jar
- jt400Proxy.jar
- jt400Servlet.jar
- jtopenlite.jar
- jui400.jar
- outputwriters.jar
- reportwriter.jar
- tes.jar
- uitools.jar
- util400.jar

Don't be alarmed by the number of JAR files. Many of these JAR files have specific purposes and will be unutilized by the average user. This article will not attempt to explain the usefulness of each JAR file. Instead, we will focus on the two newest additions, which enable application development for the Android operating system: jt400android.jar and jtopenlite.jar. We will compare and contrast these offerings with jt400.jar, the most well-known of the bunch.

jt400.jar ("jt400")

This JAR file has existed since the inception of the IBM Toolbox for Java, as one might guess from the name similar to IBM AS/400®. It is the tried-and-true mechanism for remotely accessing your IBM i, and it serves as the backbone for many IBM products, including IBM Navigator for i, IBM i Access Client Solutions, and IBM i Access for Web. It is by far the most popular and well-known deliverable, and its name has often become synonymous with the toolbox itself. However, this JAR file is not compatible with the runtime environments of modern Android devices.

jt400android.jar ("jt400-android")

The jt400.jar file is not compatible with Android because it is built using Java classes that have no counterparts in the Android runtime. The jt400android.jar deliverable was provided to directly address that issue. It did so by taking the jt400.jar codebase and making use of "stub" classes to take the place of missing-on-Android classes. Adjustments were also made to accommodate other shortcomings (for instance, Android runtimes do not have Java Swing or AWT classes). In a nutshell, the code is entirely derived from jt400.jar but with minor tweaks to make it Android-ready.

What does that mean for developers? It means that code can be easily migrated from using an Android-incapable jt400.jar to using an Android-capable jt400android.jar. We'll walk through a code example later, but this is often a small task. Though this package is the easiest for migrating existing code, it might still be too heavyweight to perform optimally on some devices.

jtopenlite.jar ("JTOpenLite")

Finally, there's JTOpenLite, delivered in jtopenlite.jar. JTOpenLite was designed to be a performance-savvy, lightweight member of the toolbox family. Unlike jt400android.jar, the code

is not derived from the classic toolbox classes from jt400.jar. For example, there are no AS400 objects used for creating new connections! Migrating existing jt400-using code to JTOpenLite takes a bit of refactoring.

JTOpenLite contains only a subset of the functions provided by jt400.jar and jt400android.jar, but it still enables you to perform these important tasks:

- Access the integrated file system (IFS).
- Run commands.
- Call programs or service programs.
- Access your database with distributed data management (DDM) or Java Database Connectivity (JDBC).

Its lightweight nature means that it can perform these operations more quickly, and that the toolbox can run at maximum efficiency on many devices, including Android phones and tablets. While JTOpenLite is the fastest-performing solution for cellular phones, keep in mind that **JTOpenLite is not "just for mobile devices."** Desktop and web applications have also shown to have better performance with some of the features in JTOpenLite. In fact, JTOpenLite was created with the intent of providing faster database access for enterprise applications.

JTOpenLite also has the advantage of a small disk footprint. As of today, the entire JAR file sizes up at a mere 560 KB. Meanwhile, jt400.jar and jt400android.jar weigh in at about 4.5 MB and 3.3 MB, respectively. This might be important during Android development because this contributes to the size of your application package.

Of course, these performance and footprint benefits come with some tradeoffs. For instance, the included JDBC driver lacks several advanced features. Examples of missing features are ResultSet positioning, SQLArray types, and bidirectional Coded Character Set Identifier (CCSIDs).

Code example – running a command

Let's examine how you can use these packages to perform a very basic task: run a command on the IBM i and print the resulting messages. In all examples, the code writes to a `PrintStream` object called `outputPrintStream`.

jt400.jar

This is an example of how the `CommandCall` class has been used for the past 15 years. Start with an AS400 object. This AS400 object represents a connection, and is needed to instantiate an object of type `CommandCall`. The `CommandCall` object is then used to run commands. Again, this code is not Android-ready because it relies on the non-Android-ready jt400.jar.

Listing 1a: The imports used for running a command with jt400.jar

```
import java.beans.PropertyVetoException;
import com.ibm.as400.access.AS400;
import com.ibm.as400.access.AS400Message;
import com.ibm.as400.access.CommandCall;
```

Listing 1b: The code to run a command with jt400.jar

```
    AS400 as400 = new AS400("MySystem", "myUser", "myPassword");
    try {
        as400.setGuiAvailable(false);
    } catch (PropertyVetoException e) {
        log(e);
    }
    CommandCall cc = new CommandCall(as400);
    try {
        boolean isSuccessful = cc.run("CRTLIB FRED");
        outputPrintStream.println("Success? "+isSuccessful);
        // getMessageList returns an array of AS400Message objects
        for(AS400Message msg : cc.getMessageList()){
            outputPrintStream.println(
             "  msg: "+msg.getID() +": "+msg.getText());
        }
    } catch (Exception e) {
        log(e);
    }
```

jt400android.jar

Now, let's do the same task with jt400android.jar. You'll note that the implementation code is exactly the same as the jt400.jar implementation! The only difference is in the required imports. Remember, jt400.jar relies on some classes that aren't present in the Android runtime, so jt400android.jar created stubs for these classes. These stub implementations are in the com.ibm.as400.jtopenstubs package. After you import the classes from the new location, your code looks the same. Most modern development environments will automatically do this for you. Alternatively, you can choose to just catch "Exception," which would use the `java.lang.Exception` class and eliminate the need for awareness of the stub classes.

Listing 2a: The imports used for running a command with jt400android.jar

```
import com.ibm.as400.access.AS400;
import com.ibm.as400.access.AS400Message;
import com.ibm.as400.access.CommandCall;
import com.ibm.as400.jtopenstubs.javabeans.PropertyVetoException;
```

Listing 2b: The code to run a command with jt400android.jar

```
    AS400 as400 = new AS400("MySystem", "myUser", "myPassword");
    try {
        as400.setGuiAvailable(false);
    } catch (PropertyVetoException e) {
        log(e);
    }
    CommandCall cc = new CommandCall(as400);
    try {
        boolean isSuccessful = cc.run("CRTLIB FRED");
        outputPrintStream.println("Success? "+isSuccessful);
        // getMessageList returns an array of AS400Message objects
        for(AS400Message msg : cc.getMessageList()){
            outputPrintStream.println(
             "  msg: "+msg.getID() +": "+msg.getText());
        }
    } catch (Exception e) {
        log(e);
    }
```

jtopenlite.jar

Finally, let's examine how we can accomplish this task with JTOpenLite. You will notice that the code is entirely different. For one, there's no `CommandCall` class, but instead a `CommandConnection` class. This class represents a connection and also provides the framework for running commands. The JTOpenLite classes are imported from the `com.ibm.jtopenlite` package.

More JTOpenLite examples are included with the project's source code in the `com.ibm.jtopenlite.samples` package.

Listing 3a: The imports used for running a command with jtopenlite.jar

```
import com.ibm.jtopenlite.Message;
import com.ibm.jtopenlite.command.CommandConnection;
import com.ibm.jtopenlite.command.CommandResult;
```

Figure 3b: The code to run a command with jtopenlite.jar

```
try {
        // This can throw an IOException, so put it inside a try block
        CommandConnection cc = CommandConnection.getConnection("MySystem", "myUserid", "myPassword");

        // no GUI owned by jtopenLite, so there's no equivalent to
        // setGuiAvailable(false)

        // CommandConnection does both program calls and command calls.
        // commands are done with the execute() method. Instead
        // of returning a set of AS400Message objects, it returns
        // a CommandResult.
        CommandResult result = cc.execute("CRTLIB FRED");

        // CommandResult contains exit code, success/fail indication,
        // and messages (as Message objects, not AS400Message).
        outputPrintStream.println("Success? "+result.succeeded());
        for(Message msg : result.getMessages()){
            outputPrintStream.println(
             "  msg: "+msg.getID() +": "+msg.getText());
        }
    } catch (Exception e) {
        log(e);
    }
```

Code example – query the database using JDBC

As another example, let's take a look at the code behind a simple JDBC call. In these examples, the code runs a simple `SELECT` statement and writes the output to a `PrintStream` object, with the assumption that some mechanism would show the output to the user. Instead, you could easily use a construct specific to your platform. On Android, for instance, you could easily insert the results into a ListView object. In both examples, the only required imports come from the `java.sql` package (so the import statement can become the wild card "`import java.sql.*;`").

jt400.jar or jt400android.jar

Again, the code for using jt400.jar and jt400android.jar is identical. The `Class.forName()`call causes the JDBC driver class to be initialized, and a connection can then be retrieved from the `DriverManager` class.

Listing 4: The code to use JDBC with jt400.jar or jt400android.jar

```
Class.forName("com.ibm.as400.access.AS400JDBCDriver");
Connection connection = DriverManager.getConnection("jdbc:as400://MYSYSTEM", user, password);
Statement statement = connection.createStatement();
ResultSet rs = statement.executeQuery(
    "SELECT CURRENT USER FROM SYSIBM.SYSDUMMY1");
while (rs.next()) {
    outputPrintStream.println(rs.getString(1));
}
connection.close();
```

jtopenlite.jar

For jtopenlite.jar, there are just two simple changes that need to be made:

- The `jtopenlite` class must be specified for `Class.forName()`.
- The URL has changed from the "`jdbc:as400`" syntax to "`jdbc:jtopenlite`".

Listing 5: The code to use JDBC with jtopenlite.jar

```
Class.forName("com.ibm.jtopenlite.database.jdbc.JDBCDriver");
Connection connection = DriverManager.getConnection("jdbc:jopenlite://MYSYSTEM", user, password);
Statement statement = connection.createStatement();
ResultSet rs = statement.executeQuery(
    "SELECT CURRENT USER FROM SYSIBM.SYSDUMMY1");
while (rs.next()) {
    outputPrintStream.println(rs.getString(1));
}
connection.close();
```

What's right for me?

In general, check out JTOpenLite if you are writing Android applications from scratch or where performance is a concern. Use jt400android.jar if you are trying to port existing Java applications to an Android application. You might run into cases where JTOpenLite seems to be the right answer but doesn't contain all the functionality that you're looking for. In that event, fall back to using jt400android.jar as needed.

Also, don't forget that JTOpenLite isn't restricted to mobile devices. It might be worth evaluating in any case where performance is critical, even in desktop or enterprise applications.

While jt400.jar continues to be the reliable workhorse behind many modern Java applications, two new additions bring the JTOpen family to the realm of phones and tablets. They are definitely worth checking out!

Resources

- Sourceforge project page
- The JTOpen source code
- IBM i developerWorks forum