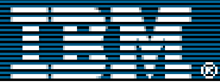


The data melting pot
Building a business-to-business application with XML

Brett McLaughlin
Enhydra strategist, Lutris technologies
May 2000

In this demonstration of how to exchange data with other companies through XML, find solutions to some of the problems inherent in B2B communication. Code samples include a means to translate an XML representation of an SQL query to a JDBC-usable query and how to convert the resulting data back into an intuitive XML format.

Contents:

What's behind that door?
GotCredit? requirements
The game plan
The handoff...
And the pitchback
The plumbing
Summary
Resources
About the author

The commercial world has grabbed onto the concept of electronic business with a feverish grip. As a result, an application is rarely the sum of its parts, but instead the central component of a larger system, often involving data from alien applications. Data has become the commodity in the 21st century, and the original source of that data no longer matters to most companies. Legacy systems, applications in COBOL, FORTRAN, and C, and databases from countless vendors all act as gatekeepers of data that companies must access. Often monumental format, storage, and language differences obstruct communication across the entities. Managers and executives don't care how you solve these problems, but you must come up with a way to manage an increasing variety of data formats in your applications. Enter the extensible markup language (XML).

What's behind that door?

By providing a standards-based data format, XML allows communication between heterogeneous systems, while allowing both endpoints of the exchange to remain ignorant of the other application's specifics. As long as requests come in XML format, they can be interpreted. And since XML results are returned, you can achieve interoperability without having to write code to handle the hundreds of varieties of clients that may need to interact with an application. In fact, the identity of the client becomes, in a sense, a mystery: bachelor number three sits completely hidden from view, speaking only in occasional bursts of XML.

While this uncoupling of applications across companies is fairly simple to envision, implementation is not trivial. Central to business-to-business communication is the exchange of data, usually stored within large corporate databases, and some of the highest hurdles to overcome are found there. Providing a client program, especially a fairly anonymous one, access to a corporate database is unthinkable; however, the data must be made available, without adding undue complexity to the client (or they may choose another, friendlier, vendor), and without exposing too much information about sensitive application architecture to clients. To illustrate these concepts further, and to look at how they can be addressed with Java technology and XML, consider the case of a fictional credit reporting agency, GotCredit?, Inc.

GotCredit? requirements

GotCredit? specializes in providing online, rapid-response credit evaluations of consumers at banks, automobile dealers, and credit agencies. The company maintains a database of creditors, debtors, and debts held by individuals, which all must be available for their clients. The agency must be able to provide this information on demand, in a fast, scalable way (in typical e-business fashion). Because GotCredit? is an Internet-savvy company, it has outlined some general constraints, which apply to many business-to-business applications:

- Disallow direct JDBC (or other language-specific) access to their database
- Require secure channels for communication
- Allow easy consumption of data by client applications

Because GotCredit? doesn't want to allow alien applications to directly access its data (and rightly so), the company developers need to build an abstraction layer over the database. They need the abstraction layer to allow them to transparently switch data sources, or even data source types. While all their data is currently in a database, it may also at some point be located in an LDAP directory server. They also want to allow secure communication to their data, while maintaining a standards-based, flexible means of communication.

To fulfill these constraints, GotCredit? developers have decided to write a servlet that will accept HTTP requests, both on secure and nonsecure ports (using SSL). The servlet will receive a request for data in the form of an SQL query, proxy the query through to the database, and then return the results, again via HTTP. The ease of using HTTP in various languages makes this an attractive solution. To allow simple communication, the SQL query and results will be formatted as XML. This also allows easy transformation and filtering of XML results between application components. Finally, by not accepting SQL directly, later changes to the data store result in a different translation from the XML-formatted SQL request, instead of forcing clients to change their request format.

GotCredit? developers must implement several components:

- Decide which portions of the database to expose

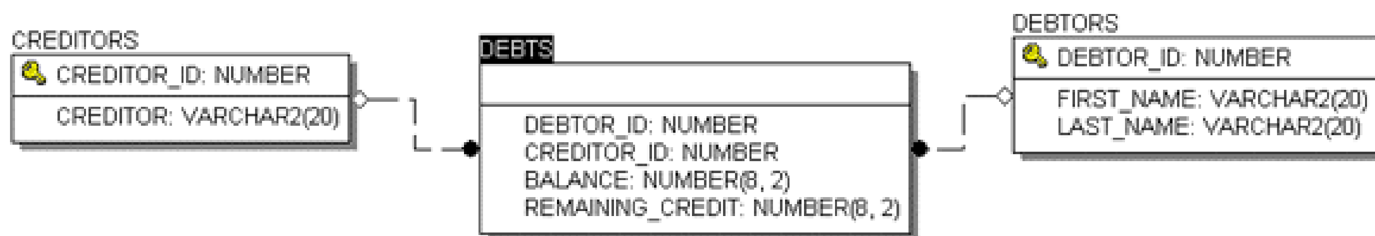
- Develop a servlet to accept requests and write responses over HTTP
- Create a utility to convert XML-formatted SQL queries to JDBC
- Create a utility to convert JDBC result sets to XML-formatted responses

Additionally, GotCredit? plans to supply an example to its potential clients that demonstrates the ease of communication with their server and data. This involves:

- Providing a utility to allow POST requests to be submitted to the GotCredit? servlet
- Demonstrating a simple client (servlet-based) that communicates using XML to the GotCredit? application

The game plan
With the plan laid out, you can start building the GotCredit? system. Although this article only covers a small example portion of the application, it should give you the concepts and tools to use in your own applications. First, create three tables to store the relational data that GotCredit? needs for creditor and debtor records. The resulting table structure should look similar to Figure 1:

Figure 1. ER diagram for GotCredit? database schema



The SQL script to create this structure can be viewed in HTML form or can be downloaded. The script is specifically for an Oracle database, but you can easily modify it for a different vendor. For sample data, you can enter data into the tables yourself or use the `populate_xmlsql_schema.sql` script shown in HTML form (or you can download it). The CREDITORS table contains creditors -- companies such as credit card lenders and banks. The DEBTORS table holds debtors -- individuals or companies that have lines of credit with the creditors. The tables can store general information about the two types of entities; the example tables hold the creditor name and the debtor name. The third table, DEBTS, lists debtor, creditor, current balance, and remaining credit. The DEBTS table and its data will do most of the work for the GotCredit? clients.

The handoff...
With the data model set (and presumably made public in some form to GotCredit?'s clients), we can look at what it takes for a client to communicate with GotCredit? Consider the following, a very typical SQL query that would retrieve the names of debtors and creditors, as well as the balance, for all individuals in the GotCredit? data store:

Listing 1. SQL query to list all debtors and their debts

```
SELECT FIRST_NAME, LAST_NAME, CREDITOR, BALANCE
   FROM CREDITORS C, DEBTORS D, DEBTS B
 WHERE C.CREDITOR_ID = B.CREDITOR_ID
       AND D.DEBTOR_ID = B.DEBTOR_ID;
```

Assuming that GotCredit? had done some homework and supplied the client with a suitable DTD, the XML representation of this query is constructed easily:

Listing 2. XML version of SQL query

```
<?xml version="1.0"?>

<query type="select">
  <resultField>FIRST_NAME</resultField>
  <resultField>LAST_NAME</resultField>
  <resultField>CREDITOR</resultField>
  <resultField>BALANCE</resultField>
  <table>CREDITORS C</table>
  <table>DEBTORS D</table>
  <table>DEBTS B</table>
  <criteria>C.CREDITOR_ID = B.CREDITOR_ID</criteria>
  <criteria>D.DEBTOR_ID = B.DEBTOR_ID</criteria>
</query>
```

This simple XML file can be understood by anyone with general familiarity with SQL, and also serves as a simple means of communicating the query to the GotCredit? application.

As an example, look at a simple servlet that takes an SQL query (that it has stored in a member variable in the form of a string of XML content) and then sends the XML data as a POST request to the GotCredit? application. While the Java Servlet API does not provide a convenient means to perform this task, Jason Hunter's `com.oreilly.servlet.HttpMessage` class (available free of charge from servlets.com -- see Resources) provides this functionality. Using this class, you can create a simple `Properties` object, pass the XML

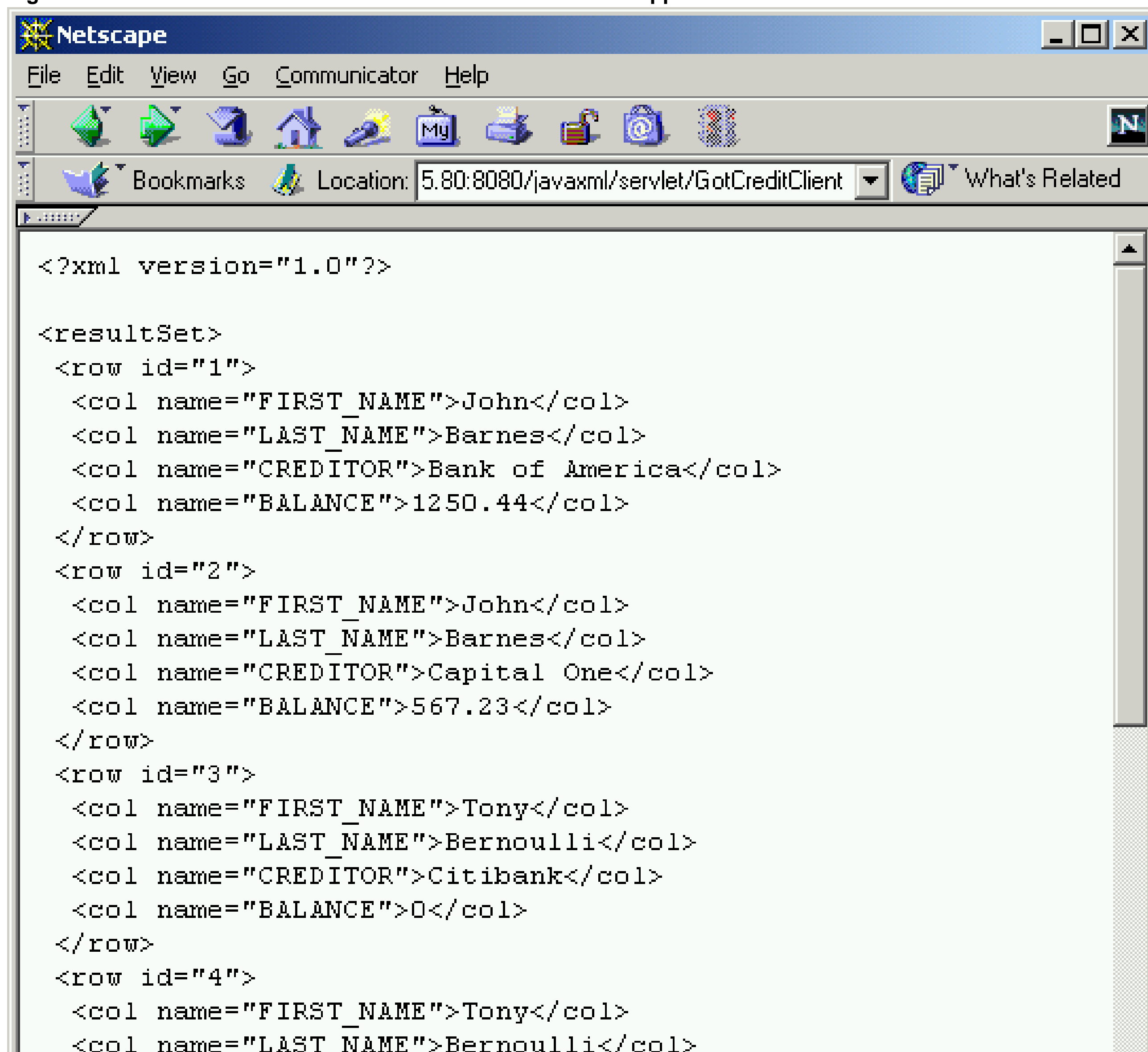
in as the value of the QUERY parameter, and send the data as an HTTP POST request on to the GotCredit? servlet. The lines of interest involving creating and sending the request are shown here:

Listing 3. Getting the results from GotCredit?

```
PrintWriter out = res.getWriter();
res.setContentType("text/plain");

try {
    URL url = new URL(GOTCREDIT_URL);
    HttpMessage msg = new HttpMessage(url);

    Properties p = new Properties();
    p.put("QUERY", XML_SQL_REQUEST);

    InputStream in = msg.sendPostMessage(p);

    // read InputStream and output

    in.close();
    out.close();
} catch (Exception e) {
    e.printStackTrace(out);
}
```

The complete GotCreditClient servlet code can be viewed in HTML form and can be downloaded in its pure Java code format. Once the request is sent, the response is captured from the GotCredit? application as an `InputStream`, shown above. For simplicity, you can then output this result to the screen, demonstrating that you correctly received an XML result set with the data requested. The screen capture here shows the result of the simple query constructed above:

Figure 2. Screen Shot of XML result set from GotCredit? client application



```
  <col name="CREDITOR">Discover</col>
  <col name="BALANCE">1809.87</col>
 </row>
 <row id="5">
  <col name="FIRST_NAME">Bobby</col>
  <col name="LAST_NAME">Sikes</col>
  <col name="CREDITOR">Citibank</col>
  <col name="BALANCE">2000</col>
```

Document: Done

The result displays the individuals' names and then lists the creditor and balance with that creditor. With this information in an XML format, it is simple to output as XHTML, WML, or another markup language, transform and pass the result on to another application component, or use the result directly within the client application. All of these techniques can quickly utilize the XML data, without having to convert from a proprietary textual format, a JDBC result set, or any other nonstandard object type.

And the pitchback
With the input coming in as XML through a POST request, the GotCredit? servlet that receives this request can be very simple. Later I'll introduce the utility class that handles the conversion of XML to an SQL query (and from a result set to XML again); for now, assume that the utility class exists and can focus on receiving a request, making the requested JDBC call, and returning the results. In essence, this servlet receives the handoff from the client, determines what to do, and tosses the request right back to the client; the cycle can then repeat. The servlet can improve performance by opening an initial connection that is used by all instances and closing that connection at the end of the servlet's life cycle. This is done in the `init()` and `destroy()` methods, which are not listed here but can be examined in the complete `com.oreilly.xml.XmlSqlServlet` code listing (in HTML format or downloadable.

The key code lies in the main `service()` method, which accepts the POST request and dispatches the request to several helper methods, and the helper methods themselves:

Listing 4. The core of the XmlSqlServlet class

```
public void service(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {

    // Create a builder instance
    builder = new XmlSqlBuilder();

    // Handle errors from initialization

    try {
        // Get the SQL Query from the XML input
        String sqlQuery = getQuery(req);

        // Output results as XML
        out.println(getXmlResultSet(sqlQuery));
    } catch (Exception e) {
        out.println("Error in execution occurred: " + errorMessage);
    } finally {
        out.close();
    }
}

private String getQuery(HttpServletRequest req) throws IOException {
    return builder.getQuery(
        new StringBufferInputStream(req.getParameter("QUERY")));
}

private String getXmlResultSet(String query) {
    try {
        Statement st = conn.createStatement();
        ResultSet rs = st.executeQuery(query);

        return builder.buildXML(rs);

    } catch (SQLException e) {
        StringBuffer results = new StringBuffer();

        results.append("<?xml version=\"1.0\"?>\n\n")
               .append("<resultSet>\n")
               .append("  <error type=\"sql\">")
               .append(e.getMessage())
               .append("</error>\n")
               .append("</resultSet>");

        return results.toString();
    }
```

```
}
```

Here, the request is fielded and passed on to the `getQuery()` method. This method converts the String parameter supplied by the client application to an `InputStream`, which is then passed on to the `com.oreilly.xml.XmlSqlBuilder` class (which I'll discuss next). This returns the JDBC-ready SQL query, which bubbles back up to the main `service()` method. Then, the second helper method, `getXmlResultSet()`, is invoked with the query. The JDBC query is executed, and the JDBC `ResultSet` is supplied to the `XmlSqlBuilder` class, which then returns the results in XML format. Finally, the XML is sent back to the client through HTTP.

By separating the HTTP request and response from the JDBC calls, and separating those calls from the conversion to and from XML, the application is very modular, allowing the developer team to change components easily. For example, moving from using JDBC connectivity with a database to LDAP calls to a directory server would not affect the main `service()` method at all.

Finally -- the plumbing

So far, I've discussed mainly the components that enable the communication between businesses, without detailing the actual conversion of SQL to XML and back again. This is, in fact, intentional -- the concepts behind B2B communication are much more complex than coding XML or Java applications on their own. Additionally, quite a few good resources on developerWorks help get you into XML and Java technologies quickly. For this reason, treat the `XmlSqlBuilder` class as just another utility, one that helps you communicate to clients using languages other than Java programming language, or clients that prefer to use the more flexible XML format for data exchange.

However, for all those out there just dying to see the water flow through those pipes, here's a brief explanation of the `XmlSqlBuilder` helper class. This class has two main methods: `getQuery()` and `buildXML()`. The first, `getQuery()`, converts an XML representation of an SQL query into a format usable in a JDBC statement. Using SAX (2.0), the method parses the input and lets a registered `org.xml.sax.ContentHandler` instance build up the query. You can view `com.oreilly.xml.XmlSqlServlet` the complete helper class listing online or download it (see Resources). The body of the method is shown below:

Listing 5. The getQuery() method of the XmlSqlBuilder class

```
public String getQuery(InputStream in) throws IOException {
    try {
        // Create class for handling content
        XmlSqlHandler handler = new XmlSqlHandler();
        parser.setContentHandler(handler);

        // Parse the document
        parser.parse(new InputSource(in));

        return handler.getQuery();
    } catch (SAXException e) {
        throw new IOException(e.getMessage());
    }
}
```

The guts of the logic are in the registered class, which is too lengthy to reprint here. However, you can see how simple it is to use SAX to parse XML content and use the data within to generate SQL statements (or any other useful data).

The counterpart to this method, `buildXML()`, takes the JDBC set of results (presumably generated from the SQL just created) and converts it back into XML. In that method there is no need to use SAX or DOM -- creating a string output of XML is sufficient:

Listing 6. The buildXML() method of the XmlSqlBuilder class

```
public String buildXML(ResultSet rs) {
    StringBuffer xml = new StringBuffer();

    // Set up XML declaration and root element
    xml.append("<?xml version=\"1.0\"?>\n\n")
       .append("<resultSet>\n");

    try {
        ResultSetMetaData metaData = rs.getMetaData();
        int counter = 0;

        while (rs.next()) {
            xml.append(" <row id=\"")
               .append(++counter)
               .append("\">\n");

            // Iterate through columns
            for (int i=1, numCols = metaData.getColumnCount(); i<=numCols; i++) {
                xml.append("  <col name=\"")
                   .append(metaData.getColumnName(i))
                   .append("\">")
                   .append(rs.getString(i))
                   .append("</col>\n");
            }

            xml.append(" </row>\n");
```

```
            }
        } catch (SQLException e) {
            xml.append("  <error type=\"sql\">")
               .append(e.getMessage())
               .append("</error>\n");
        }

        xml.append("</resultSet>");

        return xml.toString();
    }
```

The method uses the `ResultSetMetaData` class to determine the names of the columns within the JDBC `ResultSet` and then iterates through each row, building up the XML as it goes. At the end of this simple loop, the XML's root element (`<resultSet>`) is closed up, and the XML is returned to the GotCredit? servlet. Nice, simple, and easily reusable.

Summary

So with some business-to-business concepts and XML, servlet, and JDBC code under your belt, you're ready to tackle the world, right? Well, maybe not quite the world, but you should now have some better ideas about how XML can allow businesses to interact, particularly with data sources like relational databases, often without ever making a single JDBC call. You should also be getting a sense of how important modular design is; you can swap and modify all of the components created for GotCredit? with minimal effects on the remaining components. The modular design ensures that your application can function throughout all the changes that businesses and applications you communicate with may go through. Try the code out, modify it, add support for other query types and data stores, and you may quickly find yourself chattering up a storm with businesses that speak entirely different languages.

Resources

- Download JDOM, an API with a different approach, developed by a team including the author of this article, to make using XML from Java applications simple and intuitive.
- Order the author's new book *Java and XML*.
- Find out more about The Simple API for XML (SAX), the API used in this example.
- View the `XmlSqlBuilder` helper class or download it.
- Download Jason Hunter's `com.oreilly.servlet.HttpMessage` class, available free of charge from servlets.com.

About the author

Brett McLaughlin works as Enhydra strategist at Lutris Technologies and specializes in distributed systems architecture. He is author of the upcoming books *Java and XML* (O'Reilly) and *Enterprise Applications in Java* (O'Reilly). He is involved in technologies such as Java servlets, Enterprise JavaBeans technology, XML, and business-to-business applications. Along with Jason Hunter, he recently founded the JDOM project, which provides a simple API for manipulating XML from Java applications. He is also an active developer on the Apache Cocoon project, EJBoss EJB server, and a co-founder of the Apache Turbine project. Brett can be reached at brett@newInstance.com.

What do you think of this article?

Killer! Good stuff So-so; not bad Needs work Lame!

Comments?

Privacy | Legal | Contact