Leveraging a web server as a database broker using XML

By Imran Hussain

Almost all the relational databases require some libraries on the client side. These libraries are responsible for transporting client's request to the server and other low-level network communication between the two ends. Since these libraries use their own propriety protocol, it is virtually impossible to avoid installing all the client binaries that are required. XML promises a solution that can eliminate this dependency. All you need is an XML parser that can be used to filter the database records out from the text. However, be careful, you don't want to over estimate this technology and completely eliminate the client.

The goal of this article is to show how to leverage a web server to accept SQL queries sent by different clients, process them through server-side programs and send the result back to the clients.
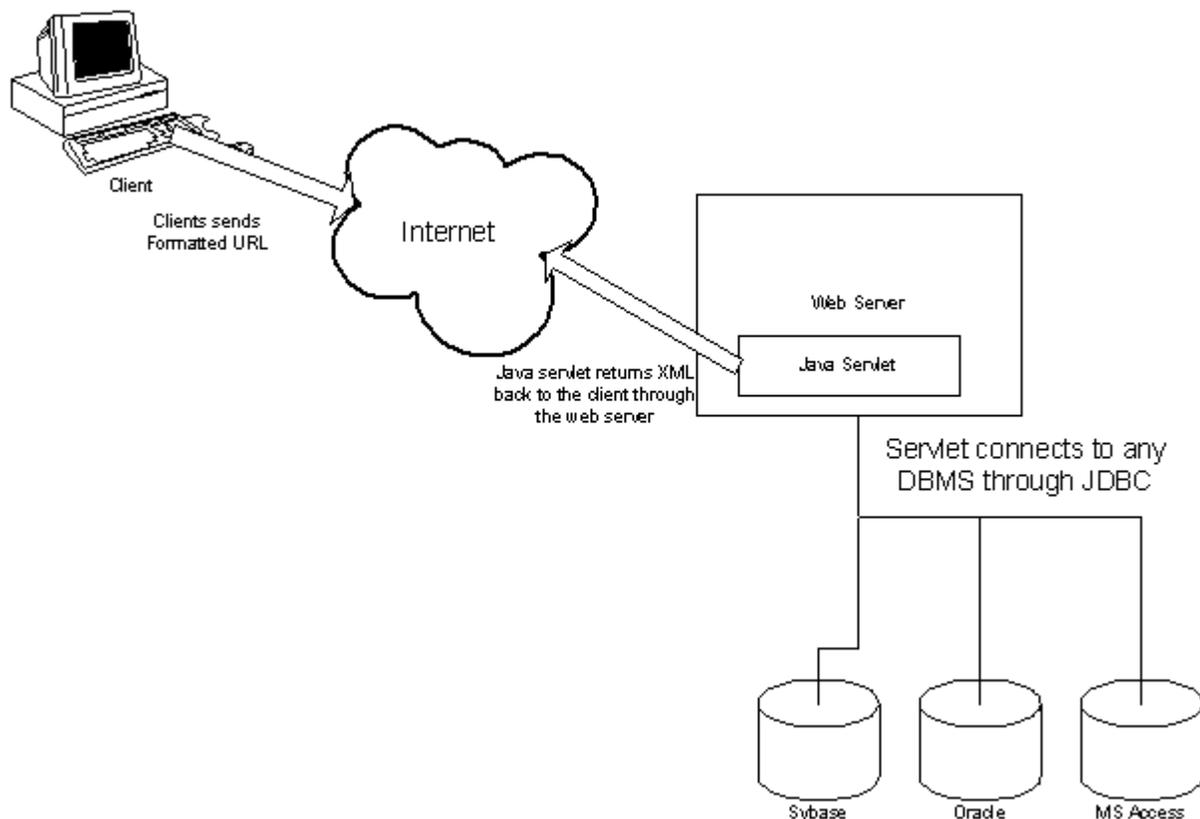
System architecture

Figure 1 shows an overview of the architecture. The client sends an SQL query embedded into a URL to the web server using the HTTP protocol. The web server forwards this message to a Java servlet that connects to any DBMS using JDBC. Which then, runs the query against the server,

formats the result into XML, hands it over to the web server and finally, the client receives it. The key point is that a servlet can connect to any database that is registered on the server by passing a different connection parameter. Client in this case is not necessarily a web browser. It could be any application that can talk HTTP to a web server. In this article, I have used two different mechanisms as an example. A simple HTML form and an executable written in Microsoft Visual Basic 6.0. The client web browser must be capable of parsing XML. At the time of this article only MS Internet Explorer 5.0 qualifies for this requirement.

Tools & technologies used
Below is a list of tools that I used to come up with this example. You can change any of these depending on your need

Server side

- Any web server (I used Apache Web Server 1.3.6)
- Servlet engine (I used Apache JServ 1.0)
- JDK 1.1.8
- Database driver (I used JDBC-ODBC bridge that comes with JDK 1.1.8)

Client side

- Any programming tool capable of either calling DLLs or OLE objects (I used Microsoft Visual Basic 6.0)
- An XML parser (I used Microsoft XML DOM that come with Internet Explorer 5)

Why XML

Before I go deep into the mechanism of generating XML, I'd like to discuss a few words about the technology itself. The number of articles that gets published about XML in different magazines is itself an advocate of its strength. According to some people, it is "the wave of the future".

XML is a subset of SGML (Standard Generalized Markup Language) for defining markup languages to represent structured data. Users can create their own tags to suite their needs based on the rules specified by the W3C consortium. Once a document is created, it can be sent to any other application that would extract the data out of these tags and further process it.

So why XML? Aren't the database libraries much faster and more network friendly than sending a textual document that contains unnecessary tags?

XML was created so that richly structured documents could be shared over the Internet. Therefore, once the document is created, all you need is a parser on the client end. This not only releases the client with the bondage of their specific database libraries but also, to a certain extent, gives a chance to a desktop database, like MS Access, to work like a client/server database. Besides, if necessary, additional business logic can be added on the middle tier. Since you can use the Internet, SSL can be used for secure communication.

Lets take an example where you have a typical order processing database with about 25,000 customer records and about 150,000 orders for these customers. This database is located in New York, which is your main distribution center. There are about 5 other local offices where a user can enter orders and check the status of their previous orders. The system uses a proprietary database that is extremely fast but requires client libraries to access the system. The management decides to use the desktop version of the database as it is relatively less expensive.

One solution is to put this database on a WAN where all the local offices can share a single copy of the data. This will not only generate lot of network traffic but will also require every client to have appropriate libraries. A better solution is to send selective data; in this case only the records that belong to a particular client being queried. A service broker that can send only a subset of records would make perfect sense in this scenario.

Server-side processing

On the server end, we need to write an application that acts like a service broker. This application should be able to accept SQL queries from different clients, run them against the requested database server, process the results and send them back to the individual clients.

For this task, I decided to use a Java servlet. This choice allows us to use different web servers on multiple platforms. However, depending on your environment, you might find other technologies like MS Active Server Pages, PHP or simple CGI scripts to do a similar task.

Although a discussion about servlets is out of the scope for this article, I'll try to explain it in a few sentences. Servlet can be thought of as a Java program similar to an applet that runs on the same machine as the web server. It does not have any user interface and is a way of extending a web server. They can be invoked either from an HTML page by using the <SERVLET> tag or directly by a passing a formatted URL string to the web server. Since the code is in Java, it has access to the entire JDK API including JDBC, which is used in this example.

Connection pools

Creating a new database connection is a relatively expensive task. It would be a waste of resources if the servlet tried to establish a new connection to a DBMS every time a new client request comes in. In the example code, I have used a hash table to store multiple instances of the connection object, each connecting to a different database. Every time a new request comes in, the program checks if a connection already exists in the table. In case if it does not exist, the program establishes a new connection and stores the object instance in the pool. For all subsequent client requests, same connection is reused. Notice that I only create one connection per database. This should be fine as long as the number of client request is limited to a small number. As the number of client increases it would overload this connection and you might have to change the code to accommodate this situation. Besides, if a client wants to use transactions, you will have to isolate that particular client in separate connection and also maintain its state.

Once the connection is established, the servlet opens a statement that is used to send the SQL query. Depending on the nature of the SQL, the statement may or may not return a result set.

Generating XML

XML, unlike its cousin HTML, has strict rules. One small error in the document can cause the parser generate errors, hence making the document unusable. Therefore, it is important the document is well formed.

The structure used to generate the document is as follows.

Document Root
      Catalog Information
            Field Information

Result Set
        Row Value

Notice that the catalog information for each field is generated only once as supposed tagging that information with each column in the result set. This approach reduces the size of the document. Listing 1 shows the output of a query that has two columns and two rows.

```
Listing 1
---------
<?xml version="1.0"?>
<QueryResult>
    <Catalog>
            <Field>
                    <Name>LOCNUM</Name>
                    <DataType>CHAR</DataType>
                    <MaxLength>10</MaxLength>
            </Field>
            <Field>
                    <Name>CITY</Name>
                    <DataType>CHAR</DataType>
                    <MaxLength>20</MaxLength>
            </Field>
    </Catalog>
<ResultSet>
            <Row>
                    <Column>1004</Column>
                    <Column>Belleville</Column>
            </Row>
            <Row>
                    <Column>1005</Column>
                    <Column>Carteret</Column>
            </Row>
</ResultSet>
<ErrMsg></ErrMsg>  <!--This line does not appear unless there is an error -->
</QueryResult>
```

Now lets rollup up the sleeves and do some coding. The major functions are listed below:

- service
- establishDBConnection
- prepareXML
- runSelectStatement
- runDMLStatement

Web server gets a request from a client.

service function gets invoked.

Call the establishConnection function.

Is connection already in the pool table?

Use existing connection. — Yes

Create a new connection and add it in the pool. — No

Success?

Yes — Determine query type:

No

Select — Call runSelectStatement.

DML — Call runDMLStatement.

Display Error.

Open a JDBC statement.

Success?

Yes — Generate XML

No

Send data back to client

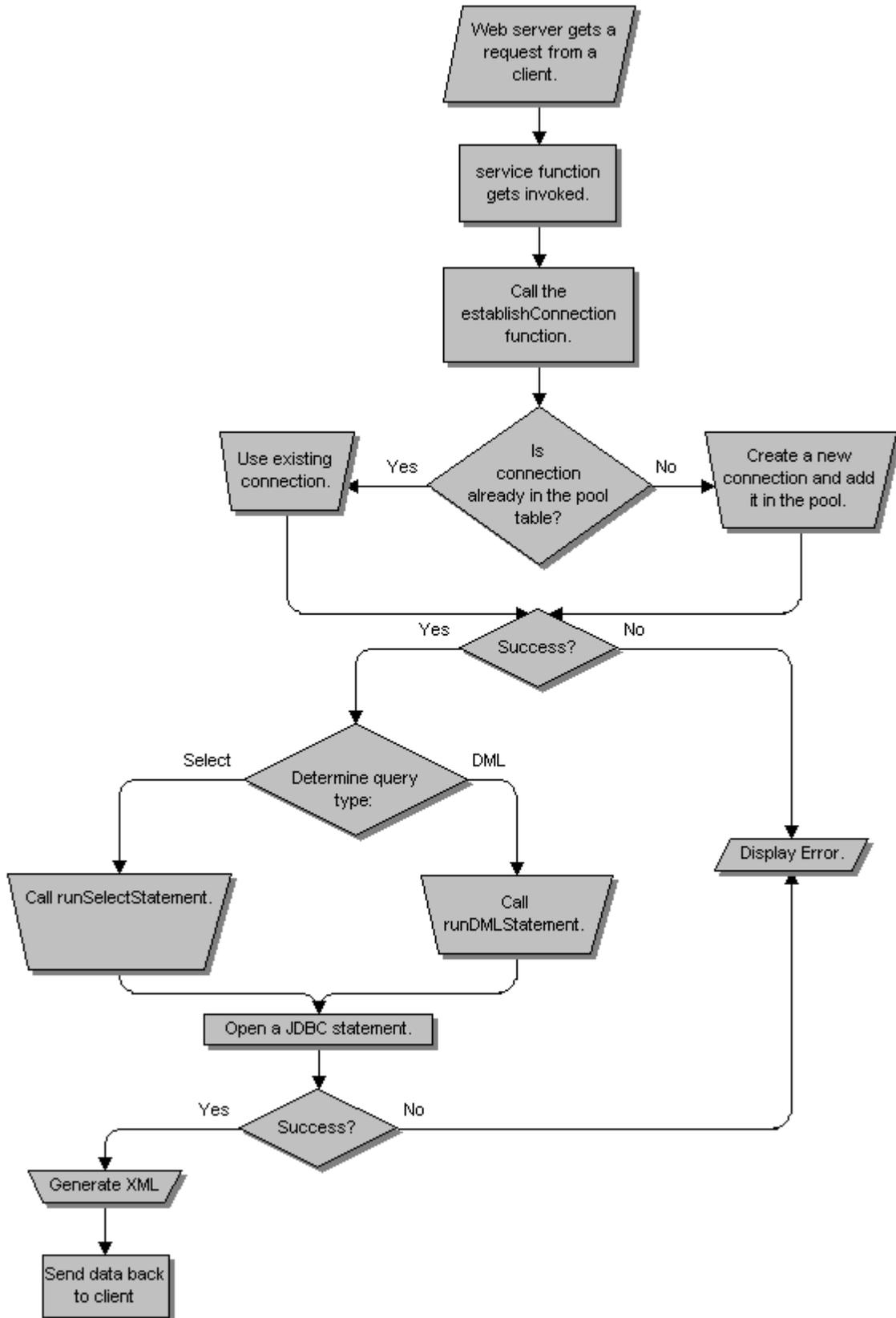Figure 2

Refer to figure 2 for the program flow on the server side.

```
Listing 2


public void service (HttpServletRequest request,
                                    HttpServletResponse response)
                          throws ServletException, IOException{

        // get the communication channel with the requesting client
        out = response.getWriter();


            if(!establishDBConnection(request.getParameter("ConnStr"),

    request.getParameter("UID"),

    request.getParameter("PWD"))){

                    return;

            }


            if(request.getParameter("isSelect").equals("1")){
                    runSelectStatement(request.getParameter("ConnStr"),

    request.getParameter("Query"));
            }
            else{
                    runDMLStatement(request.getParameter("ConnStr"),
                                            request.getParameter("Query"));
            }

    }
```

The service function, Listing 2, is the main entry point of the servlet. It first checks whether a database connection is established. If not than connects to the database and then depending on the nature of the query calls either runSelectStatement or runDMLStatement.

The two "runXXX" functions are responsible for generating XML and their structure is very similar to each other. First, they generate the XML header:
```
<?xml version="1.0"?>
```
Although this line is not required, it explicitly identifies the document as an XML document and indicates the version of XML to which it was authored. If your database uses UNICODE or any other character set, you would add that here. Please refer to XML specification at http://www.w3.org/TR/REC-xml for all the possible combinations.

Refer to Listing 3 for the source code of "runXXX" functions.

```
private void runDMLStatement(String connString, String sqlString){
    Statement       stmt = null;
    int             result ;
    Connection      con;

            //Create a statement
            try{

                    //First get the connection object for the hash table.
                    con = (Connection)connectionTable.get(connString);

                    if(con == null){
                            showError("Connection is invalid");
                            displaySQLError("Connection is invalid");
```

```java
                                return;
                        }

                        showMsg("Creating SQL Statment...");
                        stmt = con.createStatement(); //Create a statement
                        showMsg(sqlString);

                        try{
                                result = stmt.executeUpdate(sqlString);
                        }catch(SQLException e) {
                                //This must be an error in the query.
                                displaySQLError(e.getMessage()); //Process error messages.
                                showMsg(e.getMessage());
                                return;
                        }

                        showMsg("Rows affected = " + result);

                        showHTML("<?xml version=\"1.0\"?>");//This line is not required
                                                                //but you can
specify the character
                                                                //set used
here.

                        showHTML("<QueryResult>");     //Docuemnt Root
                                showHTML("\t<Catalog>");       //Start catalog tag

                        showHTML("\t\t<Field>");       //Since the catalog will only have
                                                                //the number
of rows affected, the
                                                                //information
is hard coded.
                        showHTML("\t\t\t<Name>Row(s) affected</Name>");
                        showHTML("\t\t\t<DataType>Integer</DataType>");
                        showHTML("\t\t\t<MaxLength>15</MaxLength>");
                        showHTML("\t\t</Field>");

                        showHTML("\t</Catalog>");
                        showHTML("<ResultSet>");


                        showHTML("\t<Row>");   //This will contain the actual data.
                        showHTML("\t\t<Column>" + result + "\t\t</Column>");
                        showHTML("\t</Row>");

                        showHTML("</ResultSet>");
                        showHTML("</QueryResult>");
                        stmt.close();


                }catch(SQLException ex) {
                        System.err.println("SQLException: " + ex.getMessage());
                        return;
                }



        }
        //=========================================================================
        private void runSelectStatement(String connString, String sqlString){
        Statement      stmt = null;
        ResultSet      result = null;
        int            totalCols, i, curRow;
        String         line;
        Connection     con;


                //Create a statement
                try{
```

```java
                        //First get the connection object for the hash table.
                        con = (Connection)connectionTable.get(connString);

                        if(con == null){
                                showError("Connection is invalid");
                                displaySQLError("Connection is invalid");
                                return;
                        }

                        showMsg("Creating SQL Statment...");
                        stmt = con.createStatement();
                        showMsg(sqlString);
                        try{
                                result = stmt.executeQuery(sqlString);
                        }catch(SQLException e) {
                                //This must be an error in the query.

                                displaySQLError(e.getMessage());
                                showMsg(e.getMessage());
                                return;
                        }

                        totalCols = result.getMetaData().getColumnCount();
                        showMsg("Total columns = " + totalCols);

                        showHTML("<?xml version=\"1.0\"?>");//This line is not required
                                                                            //but you can
specify the character
                                                                            //set used
here.

                        showHTML("<QueryResult>");      //Document root
                                showHTML("\t<Catalog>");//First create the catalog tag.

                        //This loop will get the meta data from the resultset and send
                        //it to the client.
                        for(i = 1; i <= totalCols; i++){
                                showHTML("\t\t<Field>");

                                //First get the name of the column
                                showHTML("\t\t\t<Name>" +
                                                        result.getMetaData().getColumnLabel(i) +
                                                        "</Name>");
                                //Second get the data type
                                showHTML("\t\t\t<DataType>" +
                                                        result.getMetaData().getColumnTypeName(i) +
                                                        "</DataType>");
                                //Finally, get the size required to display the data.
                                showHTML("\t\t\t<MaxLength>" +
                                                result.getMetaData().getColumnDisplaySize(i) +
                                                        "</MaxLength>");
                                showHTML("\t\t</Field>");
                        }
                        showHTML("\t</Catalog>");
                        showHTML("<ResultSet>");

                        curRow = 0;

                        //Fetch the records until end and send the result to the client.
                        while(result.next()){
                                showHTML("\t<Row>");
                                //For every row, fill the columns
                                for(i = 1; i <= totalCols; i++){
                                        line = result.getString(i);
                                        if(line == null) //This if is necessary to
                                                                //avoid a runtime error.

                                                        showHTML("\t\t<Column> </Column>");
                                        else
                                                        showHTML("\t\t<Column>" +
```

```
                                prepareXML(line.trim()) + "</Column>");
                    }
                    showHTML("\t</Row>");
                    showMsg("Processing row number: " + ++curRow);
                }

                showHTML("</ResultSet>");
                showHTML("</QueryResult>");

                //Finally close the statement.
                stmt.close();

        }catch(SQLException ex) {
                System.err.println("SQLException: " + ex.getMessage());
                return;
        }

    }
```

Notice that while generating XML, I did not define the document types. This is because our clients already know what to expect. However, if you want to make you servlet more generic, an addition of a DTD would be good idea.

Next, the document tags are generated. A resultset of an SQL query not only has the data but also the metadata that defines the characteristics of its columns. Using the JDBC getMetaData function, I retrieve the column header labels, the data types and their display length. This information is put in the Catalog node of the document. Alternatively, you could also put this information as a list of attributes in the "Columns" tag. However, this would increase the size of the document since this information would be repeated for every row.

After the catalog information, the function generates the actual data nodes. There must be a one-to-one correspondence between the catalog node and the columns in all the rows. If this mapping is incorrect, the client will not be able to interpret the data correctly.

Other side of the story

Once the web server sends the dynamically generated document, client must take this input and parse it. Our target here is to use multiple clients; this could be either a web browser or any custom application.

In case of a browser, there is nothing to write except for a small HTML form capable of submitting a URL to the server and as long as the browser is XML enabled, the result will be displayed in the Window. Listing 4 shows the HTML.

```
Listing 4

<html>
<head>
    <title>Web Connection</title>
</head>

<body>

<b><font size="+1"><font size="+1">XML Query Demo</font></font></b>

<form action="http://hussaini/xmlQuery/dbServlet">
<table cellspacing="2" cellpadding="2" border="0">
<tr>
    <td>Connection String:</td>
    <td><input type="text" name="ConnStr" value="jdbc:odbc:Northwind" size="45"></td>
```

```
</tr>
<tr>
    <td>User ID:</td>
    <td><input type="text" name="UID" value="admin" size="15"></td></td>
</tr>
<tr>
    <td>Password:</td>
    <td><input type="text" name="PWD" value="admin" size="15"></td></td>
</tr>
<tr>
    <td>Query:</td>
    <td><textarea cols=40 rows=10 name="Query"></textarea></td>
</tr>
<tr>
    <td>Query Type:</td>
    <td><input type="radio" name="isSelect" value="1" checked>Select Statement <br>
        <input type="radio" name="isSelect" value="0">DML Query</td>
</tr>
</table>

<br>
<input type="submit">

</form>


</body>
</html>
```

The fun starts when you want to use a custom application. Here not only you need to parse the document but also have to talk to a web server using HTTP. I chose Microsoft Visual Basic to do this task and used the Microsoft DOM parser to do parsing. Again, you can easily use any other parser to do the job.

Document Object Model (DOM)

According to the MS XML DOM user guide, "With the XML Document Object Model (DOM), you can load and parse XML files, gather information about those files, and navigate and manipulate those files". You can find a complete specification of DOM at:
http://www.w3.org/TR/REC-DOM-Level-1.

A detail explanation of DOM is out of the scope of this article. However, a few lines are necessary. It is a language neutral API that allows a programmer to build a hierarchical tree of nodes based on an XML document. Once a document is opened, the user can use this API to navigate, add, modify or delete elements. The design of this API is specified by an OMG IDL and can be found at http://www.w3.org/TR/REC-DOM-Level-1/idl-definitions.html. Figure 3 shows the DOM-tree representation of the sample query in listing 1
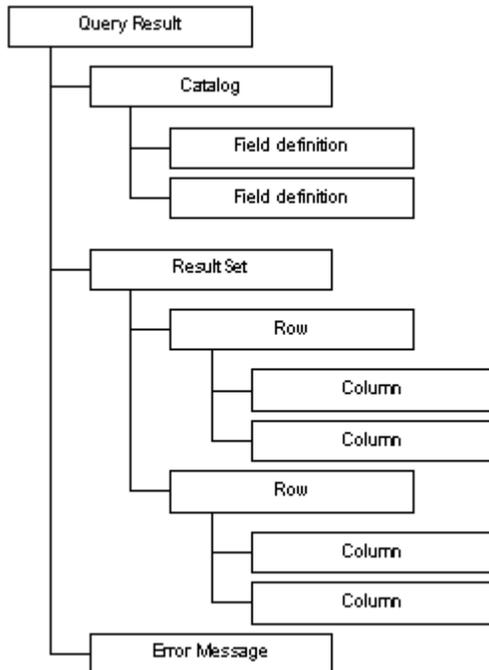
Figure 3

Here the QueryResult tag is the root of the document and has three children. The first one represents the Catalog of the resultset, second represents the data and the last one contains any error messages that get generated.

Sending and receiving data

The processing on the client side can be divided in to two parts. First, it has to format the URL string and send it to the server. Second, parse and populate the window control(s) once the server sends the resultset back.

Both the steps require that the client must talk to the server using HTTP on a specified port. This itself can be quite a bit of programming if you used plain socket calls. The best solution is to leverage someone else's code (and tell your boss that you did it). Since I used Visual Basic and XML DOM by Microsoft, I was able to reduce all the TCP/IP related code to just two lines. Listing 5 shows magic code.

```
Listing 5
---------

    xml_document.async = True
    xml_document.Load prepareURL  'This function prepares the URL for the web server
```

The prepareURL function is responsible for formatting the input URL. It replaces all the characters that are not allowed in a URL and replaces them with their appropriate values.

Notice that the xml_document variable is of type DOMDocument and is declared using WithEvents. This will generate two events: ondataavailable and onreadstatechanged. For the sake of this example, we are only interested in the latter one. When the state of the document object becomes COMPLETED, the program can first parse the catalog and then the resultset.

Parsing and filling the data grid

Before the grid receives data, it must be formatted to hold enough columns with appropriate width and headers labels. This information is embedded in the catalog node and is done by the GetColumnInfo. Listing 6 shows a pseudo code for this function

```
Listing 6
---------

totalColumnNumber = TotalChildren for the column node.
Grid.totalColumn = totalColumnNumber

For all the columns
   Col.width = width
   Col.Label = column label
Next column
```

Listing 7 shows the second function, FetchRows, which is responsible for filling the grid rows. First it checks whether the result contains an error message or the actual data. In case of a resultset, it parses the values for each row and column and populates the grid.

The most important object used in the code is an XML Node. In fact, the DOM Document object is also inherited from a Node. First, a reference to the document node is resolved, which is then used to navigate through the remaining children nodes.

Summary

The example under discussion uses a combination of tools and technologies that are integrated with each other to bring a final solution. Some of these concepts, like XML, DOM, Java Servlets, deserves a book on their own and therefore the discussion about them in this article is definitely not enough. This article is an attempt to give you an overview of using a web server as a database broker.

Even though this solution has many benefits, you don't want to over estimate its capabilities. When you use a client library, the network communication is usually optimized and only relevant data gets sent through the wire. Therefore, if you are trying to write a decision support program with lots of reports you might want to install the client libraries and use the native method. However, if you like to send small amount of data, especially over the Internet to many clients, XML is a very good answer.

About the Author

Imran Hussain works as IT Architect for Sun Microsystems Inc. He holds a bachelor degree in Computer Engineering and is a Microsoft certified solution developer. He can be reached at imranh@imranweb.com or you can visit his web site at http://www.imranweb.com