HTTP DB Specification

Version DRAFT 1, July 31, 2001

Leeland Artra, CSI UW

1 - Background

When the programmers at CSI were originally asked to write a Java application that managed and interacted with a data persistence mechanism for the lab software research, they initially created a persistence mechanism directly in the prototype applications that used files on the local machine. However what was needed was something that would read/write data from/to a shared repository. The repository requirements were simply that it had to be shared amongst a group of applications and clients and reasonably responsive. So it could have been anything at all including flat files in a centralized location using a shared file system.

But, when engineering a scaleable high performance repository design based on a shared file scheme there where three major inconveniences:

1. querying the stored data was an extremely cumbersome task;
2. users had to have direct access to the shared files to read and write from/to the centralized location; and
3. remote users may not have direct access to the shared files or data, for example, those outside of the facility's computer network.

The beast approach was to design an application that resolved these problems from the start.

The initial design was to create an application that accessed a SQL database via a JDBC layer with possibly some connection and data caching scheme built in. This solved both problems because data mining a SQL database is trivial, and the application would be easily loaded and run locally or by some shared file system. Then the user could easily access a database that is installed on any IP addressable node on the network. (See Figures 1 and 2.)

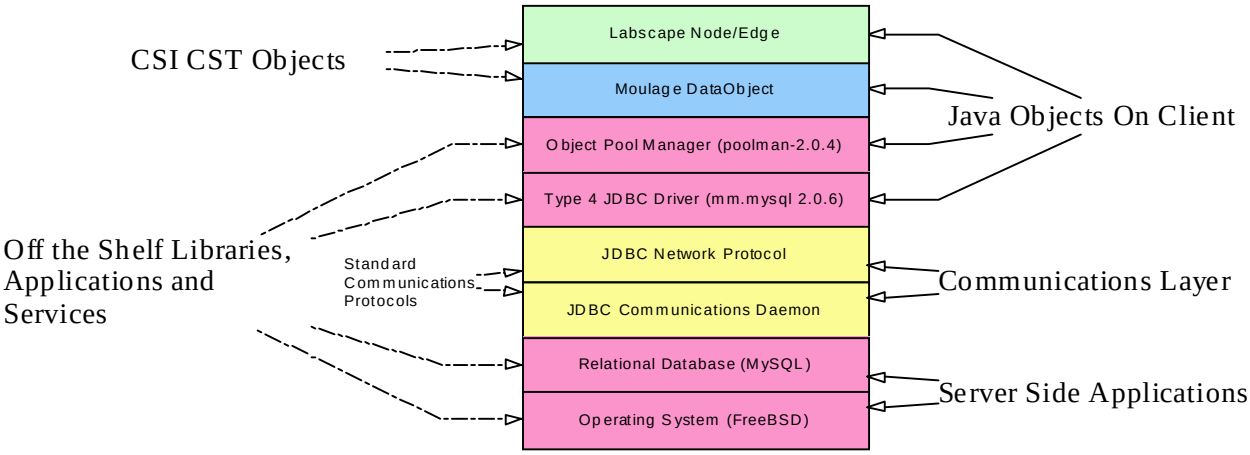

Figure 1, Data Repository Design Layer Diagram v1

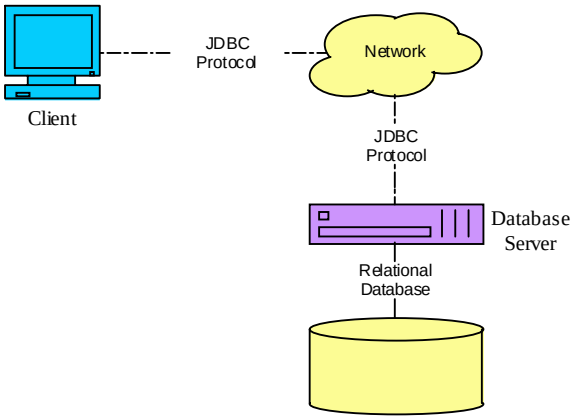

Figure 2, Data Repository Design Usage Diagram

While fully engineering the system's the data access layer it was realized that this architecture was not a scalable or even viable solution for clients. In this approach each client would be making JDBC calls directly, hence each client would require the JDBC and caching drivers be directly installed, making for an extremely bulky client. (JDBC drivers can increase the size of application almost 4 MB, plus downloading such a client can take at least 20 minutes on a 28 Kbits/sec connection.) Therefore, the initial approach was not a reasonable solution because the requirements were to keep the keep the size of the client relatively small, keep the complexity of setup down to a minimum and something that could scale.

Another Java technology, Java servlets solved these design problems. Servlets are Java programs that reside on a server and are executed by the server's servlet engine. Thus providing a gateway to central (high speed) systems that would be the key point of installation and configuration. The Java servlet is essentially the middle tier of this design. The clients communicate with the servlet through HTTP, and the servlet communicates with the database through JDBC. Because the servlet is directly communicating with the database the is no need to include bulky extra drivers on the client systems. Instead, the drivers are install and configured on the server keeping the client requirements very lightweight at just under 11K.

2 - Design Summary

The server runs the Apache HTTP server which is a general purpose web server. It is used to distribute our documentation and classes to the clients. Apache also has a servlet engine called Apache Tomcat (available from www.apache.org), which is the reference implementation for the Java Servlet 2.2 technology. Tomcat is the container that manages our servlets. Apache HTTP Server and Apache Tomcat can be easily integrated to communicate with each other, making the servlets easily accessible by the HTTP client.

The HTTP DB system passes data as plain text (rather than as serialized Java objects or XML). This technique is simple to understand, implement and maintain. The database requests are formatted by client code into key-value pairs and sent directly to the servlet.

A few Java classes have been made that simplify the client/server communication. Developing objects to do the communication creates a simple API where an object's data is quickly formatted and passed on to the database while the job of sending the data to the server is hidden within the implementation. This simplifies the procedure of reading/writing from/to the database down to a couple of simple method calls. The classes that are responsible for this communication

are CommunicationHandler, QueryRequest, and QueryResponse. These classes are portable to almost any project because they are generic. (See Figure 3.)

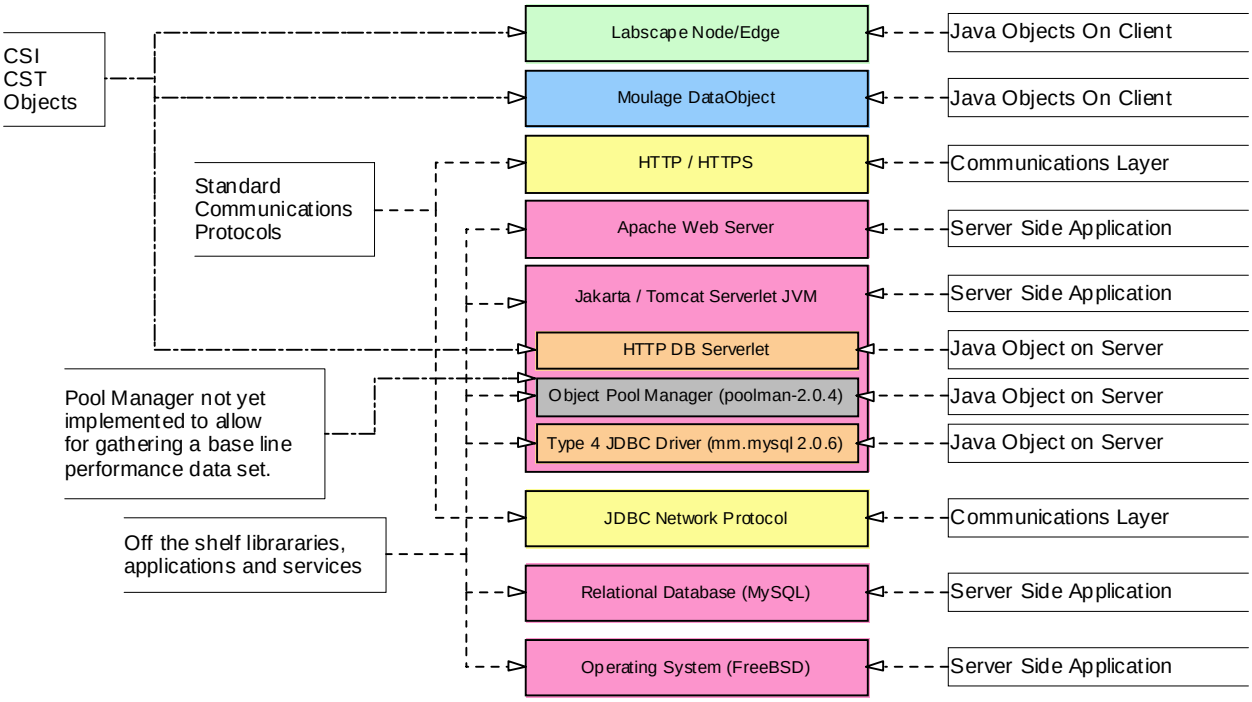

Figure 3, Data Repository Design Layer Diagram v2

The servlet extends javax.servlet.http.HttpServlet. There are two methods in the HttpServlet abstract class that are implemented in the servlet: the doGet() method and the doPost() method. The servlet's doGet() method is used to retrieve data from the database, and the doPost() method is used to write to the database. The methods are split since HTTP GET requests are a natural way of retrieving content from a server. While HTTP POST requests are used to send content to a server.

3 - HTTP DB Serverlet

The HTTP DB service is run primarily by a Java serverlet class that provides an access layer between HTTP requests and one or more database systems. Its main function is to play the role of the "middleman" in the communication of client(s) with a database. (See Figure 4.)

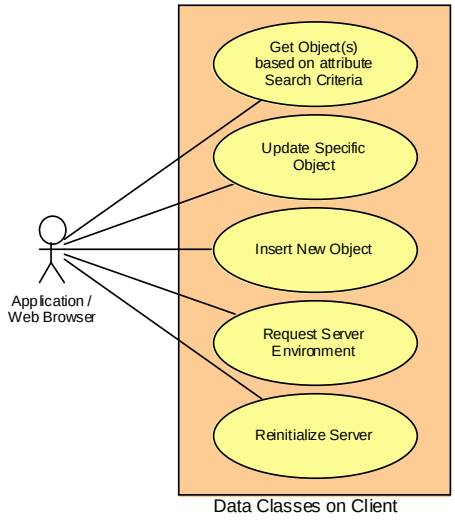

Figure 4, Data Repository Serverlet Use Case Diagram

3.1 Serverlet Configuration

The database connection data is retrieved from (in order, last setting wins) a default setting, file, and finally server environment under the property names "*httpdb.**NAME***". *httpdb.propertiesFile* is the only exception as it can only be set in the serverlet environment.

1. Properties:
   - The configuration properties are named value pairs of strings.

Configuration Properties	
Property	**Default**
httpdb.propertiesFile	**none**
httpdb.allowShowProperties	**NO**
httpdb.jdbcDriver	**org.gjt.mm.mysql.Driver**
httpdb.url	**jdbc:mysql://localhost/test?user=test&password=test**
httpdb.user	**none**
httpdb.password	**none**

2. **Property Translation:**

If *httpdb.user* or *httpdb.password* are set the strings "USER_NAME" and "USER_PASS" in the "*httpdb.url*" property are replaced (if present) with the appropriate environment data.

3.2 Serverlet Usage

Once running the serverlet responds to HTTP POST and GET requests. In both forms a minimum set of parameters must be specified using the HTTP GET URL encoding. At the minimum an **ACTION** HTTP GET command is required.

3.2.1 Usage Notes

1. If **TABLE_NAME** is not set the table ‘httpdb_default’ is used.

2. Each table under HTTP DB management contains the unique key text column named ID, the indexed date column named CREATION_DATE, and the automatically tracked indexed date column LAST_MODIFIED.
3. All attribute names will be translated to uppercase before comparison to the database.
4. Any POST request with out an ID parameter will cause a new row to be inserted with a new unique ID for that table.
5. IDs are base 64 encoded cryptographically generated 1024 bit keys. They are only checked for uniqueness against the table the new ID is being inserted into.
6. If no TABLE_NAME parameter is provided the default table name of 'http_db' is used.
7. Table names are not translated to uppercase and are case sensitive.
8. All requests may receive a WARNING parameter on the return which is any language warning messages from the underlying database system.
9. All requests may receive an ERROR parameter which will explain why a particular request was not fulfilled. The ERROR message may come from the underlying database or from the serverlet.

3.2.2 HTTP GET Requests

1. GET_DATA

Example URL:
*http://hostname/serverlet/HTTPDB?**ACTION**=GET_DATA[&**TABLE_NAME**=TABLE[&VAR=VAL...]]*

For the GET_DATA action command the serverlet will create a list of all the HTTP GET values (except for **ACTION** and **TABLE_NAME**). It will then create a database query command like "select * from **TABLE_NAME**" with the values assembled as "WHERE Variable='Value' [AND WHERE Variable2='Value2' ...]".

2. REINIT

Example URL:
*http://hostname/serverlet/HTTPDB?**ACTION**=REINIT*

When the serverlet receives this **ACTION** command it will close all database connections, reread the *httpdb.propertiesFile* (if set), drop all cached data and reconnect to the database.

3. SHOW_PROPERTIES

Example URL:
*http://hostname/serverlet/HTTPDB?**ACTION**=SHOW_PROPERTIES*

When the serverlet receives this **ACTION** command it will, if *httpdb.allowShowProperties* is set to "YES", return a properties list in the body of the HTML document containing the named value list for all of its environmental settings.

3.2.3 HTTP POST Requests

1. UPDATE

Example URL:
*http://hostname/serverlet/HTTPDB?**ACTION**=UPDATE[&**TABLE_NAME**=TABLE[&**ID**=ID_VAL]])*

When the serverlet receives this **ACTION** command it will first check to see if the named table exists. If the named table does not exist it will create a new skeleton table by that name.

Next the HTTB DB serverlet will make sure that all the named values have an associated text column in the above table. If any columns do not exist it will automatically add the needed text column.

Once the table has been check the serverlet will check if an object or row ID was given. If not it will automatically generate a new ID and insert a blank row into the table.

Finally the serverlet will create a properties list from all other HTTP POST values. Then it performs a database update operation on the appropriate ID identified row in the named table. The update operation command will look something like "UPDATE TABLE " followed by the POST variables assembled as "SET Variable='Value' [, SET Variable2='Value2' ...] WHERE ID='<ID>'".

4 - Servlet and Database Performance

Server-side performance is important, and the serverlet improves performance. The most expensive code in the servlet is the code that sets up the database connection. Performance is greatly improved by the creation of the connection only once, when the servlet is instantiated in the init() method.

The servlet implements the interface SingleThreadModel. Because it creates the connection only once, in the init() method of the servlet, the connection is available to all threads that access the servlet. Implementing this interface guarantees that two threads will not access the servlet, and hence the connection object, simultaneously.

Although this approach produces a performance gain, scalability can be increased even further, since the servlet has been built thread-safe, by later incorporating a database connection pooling technology. The problem with opening and closing the connection in the init() and destroy() methods is that it does not allow for multithreading. Connection pooling will solve this problem by obtaining a connection from the pool every time a servlet is executed. Hence the first thing the servlet would do, upon entering the doGet() and doPost() methods would be to obtain a connection from the connection pool. This would allow for the reuse of physical connections and reduce the overhead involved with opening and closing the connections each time the servlet is instantiated or destroyed.

Another expensive task in the servlet is the SQL query of the database. The use of nonprepared statements is useful for infrequent queries because they are not precompiled for efficiency, as are prepared statement. The SQL statements used in the serverlet must be compiled each time they are executed. Hence A prepared statement has advantages over this method because it is created with a parameterized SQL statement. Most databases allow for the definition of user-defined functions called stored procedures that are actually stored in the database. Using stored procedures may provide the system with significant performance enhancements. But, they will also make the server code harder to maintain and far less portable, because stored procedures are usually database specific. For this reason, this serverlet is design with out the use of stored procedures.

This implementation just barely touches one of the major components of Java 2 Enterprise Edition (J2EE) servlets. The application can be made more sophisticated by using other J2EE components, such as Enterprise Java Beans, to allow for even greater scalability.

5 - Reference Client

In designing any client the first question that must be answered is “what does the client want to do?” The following UML Use Case Diagram (Figure 5) answers this question for the reference client implementation. Essentially the client needs to retrieve data from the database, or write data to the database, or find out the status of the database.

Figure 5, Reference Implementation Client Use Case Diagram

The code in Listing 1, from the reference client, demonstrates how to perform the GET request using the CommunicationHandler class. In this case, we are communicating with a servlet that will query the database and return all of the data associated with employee ID 5555. We first create the Properties object that will be used to package the query parameters. Then we set the parameters that will be used by the servlet and the database. The ACTION parameter specifies the action to be executed by the servlet on the database, and the EMPLOYEE_ID parameter specifies the database query parameter. Parameters are packaged as key-value pairs when sent and are similarly retrieved on the server side. (For more information on similar experiments, see "eMobile: A Sample End-to-End Application Using the Java 2 Platform, Enterprise Edition," in the Resources section.)

We then send the GET request to the servlet using CommunicationHandler's query(...) method. The query(...) method may perform either the GET or POST method by specifying the operation as the first argument. The second and third arguments of the query(...) method are the URL of the servlet and the packaged Parameters object. The final argument of the query(...) method is a QueryResponse object. QueryResponse implements an interface, CommunicationHandler.ResponseHandler, that is passed to the CommunicationHandler to process the content that is retrieved from the database. Because the database content is also returned by the servlet as a set of key-value pairs, when the GET request returns, the QueryResponse object will have retrieved the content of the database query as a Properties object.

Now let's take a look at sending content using an HTTP POST request from the client. The code to execute a POST request is more complicated than that of a GET request, because not only will we send parameters, but the content that we write to the database, as well (see Listing 2).

Just as we did in the GET request, we package our parameters into a Properties object. But here in the POST request, we additionally package the content that we would like to write to the database in its own Properties object, Content. We serialize this object by sending it directly to the server in the content of the HTTP POST request. In Listing 2, we are just sending plain text as our content type, so we could have packaged our content as key-value pairs in with the Parameters object. Why? Because if our content type were of another data format—such as XML—the POST request could easily handle it.

We do not plan to receive any content from the database, but we still define and use a QueryRequest object for error handling purposes. Error messages from the server are retrieved

from the QueryResponse object just as they were in the GET request. (The details of the client/servlet communication are documented in the CommunicationHandler code.)

6 - Bibliography

1. "Tomcat User's Guide", Gal Shachor, Alex Chaffee and Rob Slifka, "http://jakarta.apache.org/tomcat/tomcat-3.3-doc/tomcat-apache-howto.html".
2. "Working With mod_jk", Gal Shachor, Mike Braden, Mike Bremford, and Chris Pepper, "http://jakarta.apache.org/tomcat/tomcat-3.3-doc/mod_jk-howto.html"
3. "Tomcat and JServ", OOP-Research Group, "http://www.oop-research.com/tomcat_3_1/".
4. "Tomcat - Apache HOWTO", Gal Shachor, "http://jakarta.apache.org/tomcat/tomcat-3.3-doc/tomcat-ug.html".
5. "Java™ Serverlet Specification, v2.2 Final Release", James Duncan Davidson and Danny Coward, Sun Microsystems.
6. "JavaServer Pages™ Specification, v1.1", Java Software, Sun Microsystems.
7. "Mastering The Details With Technical Charts", Leeland Artra, Version 1 June 2001.
8. "eMobile: A Sample End-to-End Application Using the Java 2 Platform, Enterprise Edition", MDE Enterprise Java Team, "http://developer.java.sun.com/developer/technicalArticles/whitepapers/javaone00/eMobile.pdf".
9. "Growing JSP and Serverlet Sites To EJB-Based Services", by Patrick Sean Neville, Java Developer's Journal, June 2001, Volume 6 Issue 6 pgs 44-52.
10. "JDBC Developer's Guide and Reference", Brian Wright and Thomas Pfaeffle, "http://technet.oracle.com/docs/products/oracle8i/doc_library/817_doc/java.817/a83724/toc.htm".
11. "JDBC 2.0 Standard Extension API", Seth White and Mark Hapner, "http://java.sun.com/products/jdbc/jdbc20.stdext.pdf".