RC24217 (W0703-060) March 14, 2007
Computer Science

IBM Research Report

DBC-JS: Middleware for Secure Web-Client Access to Relational Data

Avraham Leff, James T. Rayfield
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

DBC-JS: Middleware for Secure Web-Client Access to Relational Data

Avraham Leff and James T. Rayfield

IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, N.Y. 10598
{avraham,jtray}@ibm.com

Abstract. DBC-JS (*Database Connectivity for JavaScript*) is middleware that allows web-clients to directly access relational data without compromising enterprise security. On the client, DBC-JS consists of a JavaScript™ API and library that can be used by web-applications without special browser-plugins. On the server, the DBC-JS gateway, written in PHP, is an adaptor layer that mediates between DBC-JS and relational databases and provides functions such as operation forwarding and security. Web 2.0 applications can thus use DBC-JS to access relational data as a first-class construct instead of through *ad-hoc* protocols. We explain the advantages of direct web-client access to relational data, discuss how developers benefit from using the DBC-JS client-side libraries, and show that enterprises can use DBC-JS to make relational data assets available to web-applications in a secure and robust fashion.

1 Introduction

DBC-JS (*Database Connectivity for JavaScript*) is middleware that allows web-clients to directly access relational data without compromising enterprise security. One motivation for DBC-JS is the trend for web-applications to be dynamically composed in a web-browser – so-called "Web 2.0" applications [1] – rather than being composed on the server ("Web 1.0"). DBC-JS is specifically interested in enabling the potential Web 2.0 benefits of increased application responsiveness and the ability to flexibly combine information from various sources on the client. Web 2.0 access to server-side data, however, is currently characterized by REST-like [2] APIs which are typically application specific. For example, as catagorized by ProgrammableWeb (`http://www.programmableweb.com/`), 51% of "Mashups" use Google Maps, 10% use Flickr, and 8% use Amazon APIs. For many applications, a more general-purpose data-access API would be more appropriate. For example, ODBC [3] and its various language-dependent incarnations (e.g., JDBC for Java or PDO for PHP) is a popular application-independent API for accessing relational data on servers. ODBC is powerful – allowing any SQL statement to be executed – and simple, in the sense that developers are required to understand only a few abstractions. With DBC-JS, we propose an "ODBC for web-clients", so that web-developers can benefit from a general-purpose API for accessing relational data. We provide web-developers with a

JavaScript library that implements the API without the need for browser plug-ins. Developers can therefore easily integrate relational data with other parts of a web-application including its business logic or interact with widget frameworks such as Dojo [4]. In general, using DBC-JS middleware instead of the current *ad-hoc* approaches to server-side data access provides well-known middleware benefits. Developers do not need to be concerned with wire formats, message parsing, exception handling, and security, as these functions are implemented by the client-side and server-side DBC-JS middleware. In addition, DBC-JS specifically makes available to client-side developers the same powerful relational model that has traditionally been available for server-side developers.

In addition to facilitating more powerful Web 2.0 applications, DBC-JS is motivated by the trend towards simplifying client access to server-side data. Previously, client access was mediated by many layers of software such as the Enterprise JavaBeans (EJBs) and Service Data Objects (SDOs). The complexity of such programing models at least partially derived from the fact that they provided useful function: e.g., the need to provide structured access to server-side data and the need to mediate data access to unsecured clients. Increasingly, however, client developers prefer a simpler style of programming, and DBC-JS is consistent with this trend. DBC-JS seeks to combine the benefits of the old and new programming styles. Its client-side API provides an ODBC-like API that provides structured relational data access for web-clients, while its server-side gateway provides function such as security and data schema evolution. As with ODBC, the DBC-JS gateway implementation is decoupled from the client library implementation, and may be in a different language.

After discussing how DBC-JS relates to other work in this area, in Section 2 we explain how DBC-JS changes the current web-application relational data architecture, and how its middleware approach benefits web-developers. In Section 3 we discuss aspects of the DBC-JS programming model, including transactions and security. In Section 4 we show how web-developers use DBC-JS's client-side API to access relational data. Section 5 summarizes the status of the DBC-JS project and the contributions of DBC-JS middleware for web-application access to relational data.

1.1 Related Work

From an implementation perspective, DBC-JS fits squarely into the large number of Web 2.0 projects that use *AJAX* (Asynchronous JavaScript and XML) techniques [5] to add more independent web-client function in browser applications. DBC-JS adds data-access capabilities to web-clients, specifically relational data capabilities. However, to date, most Web 2.0 applications have focused on acquiring server-side data through REST-like [2] APIs, so that, even if the data are actually relational, that structure is not exposed as such to web-clients.

Some projects such as OAT [6] and Opentoro [7] do provide relational data access for web-applications, but use a approach that differs from DBC-JS. Unlike these projects, DBC-JS does not provide GUI widgets that present data in a way that deliberately hides the data's relational structure. DBC-JS uses a minimalistic

approach that is concerned only with relational data access and is not concerned with widget integration. More importantly, by using an ODBC-like approach, DBC-JS requires that web-client developers use an API that is (more-or-less) the same as the server-side API. We think it preferable to leverage knowledge about the mature ODBC technology and the large number of server-side developers that are familiar with this approach.

Finally, it is important to note the limited scope of DBC-JS's client-side caching of server-side data. DBC-JS is designed only for *connected* web-clients, and (unlike some approaches [8]) does not work for disconnected clients. While connected to the server, as clients issue queries against a server-side database, DBC-JS pre-fetches the query results from the server and makes the data available to the web-client. However, this data is only used to satisfy requests for that query. Subsequent query requests are sent to the server, and are not executed against the pre-fetched data from earlier queries. By not using cached data across multiple queries, DBC-JS passes up opportunities to further improve performance [9] – but benefits from a simpler implementation and programming model.

2 Relational Data & Web-Application Architectures

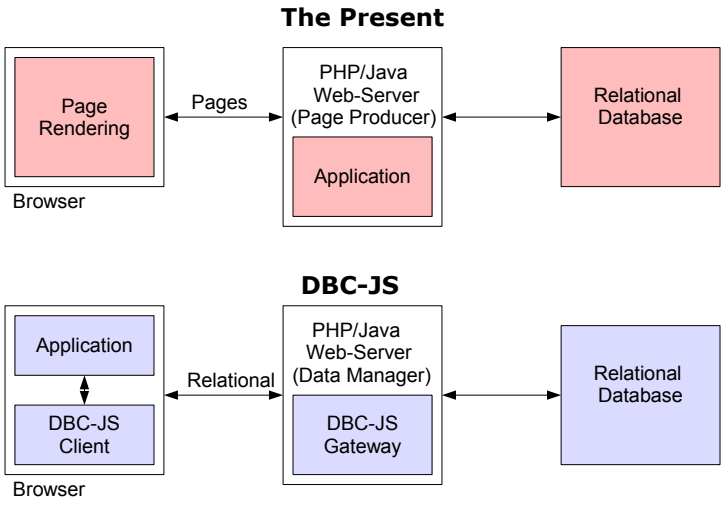

Fig. 1. Contrasting Architectures for Web-Application Use of Relational Data

Figure 1 contrasts the system architecture typically used to integrate relational data into web-pages with the DBC-JS architecture. Currently, web-pages that require relational data are statically built by the web-server. Technologies such as PHP and Java are used to interleave the code that generates the browser presentation layer (e.g., an HTML table) with the code that accesses server-side data (e.g., the data that populates the HTML table). In this Web 1.0 approach, the web-server plays a *page-producer* role. Although the DBC-JS approach certainly "allows" the server to continue playing the page-producer role, it also enables Web 2.0 applications to actively and dynamically create web-pages somewhat independently of the web-server. In this *data-manager* approach, the web-server mediates between web-clients and the database server to regulate access to server-side relational data. Rather than simply rendering the server's web-page, the web-application now interacts with the DBC-JS client library to build a web-page that integrates relational data as necessary. The DBC-JS client library interacts with the DBC-JS gateway on the server to access the relational data. Web-clients are thus empowered to build web-pages *via* first-class support for relational data access; at the same time, database servers are assured that only authenticated clients are permitted access to a specified subset of the server-side data.

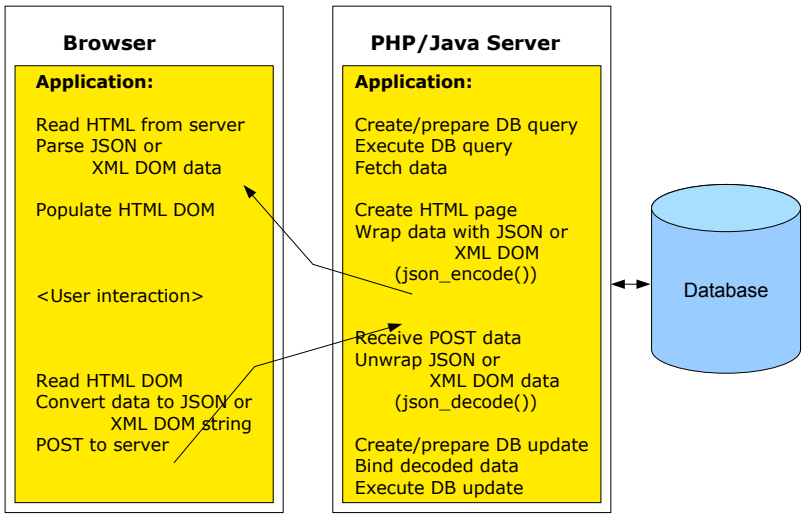

Fig. 2. Current Structure of Web-Application Relational Data Access

While Figure 1 contrasts static and dynamic architectures, Figures 2 and 3 both assume a Web 2.0 architecture. These figures illustrate the benefits of using the DBC-JS middleware-based approach for integrating relational data into web-applications. Using current approaches (Figure 2) a Web 2.0 application using relational data actually consists of two, fairly large, sub-applications: one residing on the client and one residing the server. Both portions must – on a per-application basis – agree on a message format and implement message-marshalling and message-demarshalling code. The client-side portion must write code to populate the GUI with server-side data, and the server-side portion must write code that executes queries against the database and embed the data into messages. As shown in Figure 3, using DBC-JS moves the application-independent code into middleware, thus reducing the size of the client and server portions that are specific to the application.

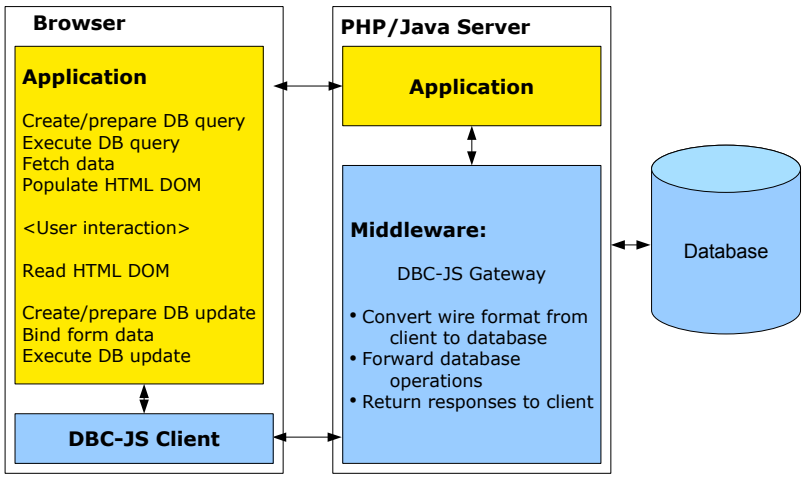

Fig. 3. Web-Application Relational Data Access Restructured With DBC-JS

3 Programming Model

The DBC-JS programming model presents a "direct-to-relational" API for web-developers that (1) is familiar to PHP and Java developers and (2) fits naturally into a JavaScript application. However, the reality of the underlying web-

client/web-server environment does affect some DBC-JS design decisions, and are visible in some aspects of the its programming model.

3.1 Function-Shipping

The DBC-JS implementation uses a "function shipping" approach in contrast to actively processing a user's SQL on the client. In this approach, user invocations of the client-side API result in this generic sequence of operations (see the lower-half of Figure 1):

1. The DBC-JS client library converts the API call to an XML message.
2. The DBC-JS client library transmits the message to the DBC-JS gateway on the server.
3. The DBC-JS gateway parses the client's message and constructs an invocation of the corresponding server-side API.
4. The server-side API is invoked, and the results are evaluated.
5. A XML message containing these results (e.g., the result set for a query, an error condition, or an exception) is constructed and transmitted to the client.
6. The DBC-JS client library parses the server's message, and returns the API's corresponding result to the user.

The DBC-JS client library, in other words, is not a "smart" client: it doesn't, for example, try to validate whether a user's SQL is valid. If the SQL is invalid, the server will detect this, throw an exception or raise an error condition, and this result is returned to the user.

3.2 Security

Database administrators are understandably nervous about exposing data assets to web-clients. DBC-JS uses several techniques to address such security concerns.

- The DBC-JS API requires that clients authenticate themselves to the server with a name and password when first establishing a connection to the server. These credentials are passed on each message from the client to the server which authenticates the client on a per-message basis. (We expect that technologies such as SSL will be used to encrypt communication between clients and the server as necessary.)
- DBC-JS prevents SQL injection attacks by using a prepared statement API. Since the parameter values are never parsed as SQL, injection attacks cannot occur.
- Clients can execute only pre-audited SQL. All SQL statements that the client passed to the DBC-JS gateway are validated against a "white-list" of validated SQL statements.
- Clients can be prevented from even seeing the SQL statement with an indirection technique in which clients use "named SQL" statements. The client provides the name of the SQL statement, and the server maps the name onto the actual SQL. This allows the server to withhold information about the database design from the clients.

Basic access rights are handled by the database manager (DBM). That is, after user authentication, the DBM is responsible for deciding which tables and stored procedures the user may access, and whether write access is allowed. For some DBMs, this may not provide sufficient granularity for access control. For example, some systems may not provide a userid for each unique user of the system. Instead, userids may be assigned to application roles, such as *external user*, *internal user*, *administrator*, etc. On such systems, it will not be possible to choose different access permissions for users sharing the same role. For such cases, the DBC-JS gateway could be extended to create and manage its own set of userids.

DBC-JS could also enforce access control at the "named SQL" level. That is, certain users could be granted permission to use certain named SQL statements. Since named SQL could refer to calls to stored procedures, this would be equivalent to stored-procedure access control. Finally, database views may have access control enforced by the database, and so may be used to provide security.

3.3 Transaction Model

Because all server-side relational database APIs include a transactional API, ideally we'd like to also project this behavior to the DBC-JS API. Unfortunately, the large latency in client/server communication imply that a lock-based implementation will result in unacceptable performance on the database server. Similarly, the large latency implies that a timeout-based approach to releasing locks (e.g., when an application is deadlocked) will not perform well. Non-locking protocols do exist for long-running transactions [10], but we consider them to be far too "heavy-weight" for browser-based applications. Also, concerns about whether these protocols are really interoperable or easy-to-use remain to be addressed. Most importantly, in practice, users of client-based applications appear to be satisfied without traditional transactional guarantees. For these reasons, DBC-JS uses the less powerful, but simpler, *auto-commit* transaction model in which an independent server-side transaction is associated with each client-side SQL statement.

4 Web-Developer View

In this section we show how web-developers use DBC-JS's client-side API to access relational data. The API is based on three types of objects:

1. **rdb**, corresponding to a database (or connection) handle.
2. **rdbstatement**, corresponding to a statement handle.
3. **rdbexception**, thrown when DBC-JS detects an error.

Because constructing a **rdb** instance requires that the database authenticate the user, the web-application typically uses an HTML form to collect the necessary information from the user (see Figure 4).

```
var submitLogin = function() {
  try {
    var dsnEntry = document.getElementById(dsnID);
    dsn = dsnEntry.value;
    var userNameEntry = document.getElementById(userNameID);
    userName = userNameEntry.value;
    var passwordEntry = document.getElementById(passwordID);
    password = passwordEntry.value;

    var validPassword = false;
    try {
      var unused = new RDB(dsn, userName, password);
      validPassword = true;      // no exception
    }
    catch (e) {
      var loginStatusEntry = document.getElementById(
          loginStatusID);
      loginStatusEntry.firstChild.data = "Login failed: " + e;
    }

    if (validPassword)
       continueWithApplicationBusinessLogic();
  }
  catch (e) {
    alert("Exception: " + e);
  }

  return false;
}
```

Listing 1.1. Accessing a Relational Database

Using code similar to Listing 1.1, the application reads the form data and tries to connect to the specified database. If an exception is thrown (e.g., the database doesn't exist, or the user/password is invalid), an error message is displayed. Otherwise, the application can now use the **rdb** instance.

Web-applications invoke *rdb.prepare(sql)* to create an executable form of an SQL statement (a *prepared statement*) from a character-string form of the statement. For security reasons (Section 3.2), DBC-JS allows "sql" to actually be a description of the SQL which, on the server, gets mapped to the actual SQL. As shown in Listing 1.2, *prepare()* returns a **rdbstatement** if successful. If the SQL contains place-holder parameters, the *bindValue()* method – not shown in Listing 1.2 – binds a given value to a place-holder.

The *execute()* is used to run the SQL against the database, and returns a boolean indicating whether the operation succeeded. (The **rdbexception**'s *errorInfo()* and *getCode()* methods are used to determine the underlying cause of

```
function executeSQL(sql)
{
  try {
    removeAllChildren(resultTable);
    setStatus("Processing...");
    var rdb = new RDB(dsn, userName, password);
    var stmt = rdb.prepare(sql);
    var success = stmt.execute();
    if (!success)
      setStatus("EXECUTE failed: " + stmt.errorInfo()[2]);
    else {
      setStatus("OK");

      if (stmt.isQuery()) {
        var colNames = stmt.columnNames();
        var tableHead = createTableHead(colNames);
        resultTable.appendChild(tableHead);
        var tableBody = createTableBody(stmt, colNames);
        resultTable.appendChild(tableBody);
        }
      else
        removeAllChildren(resultTable);
      }
    }
  catch (e) {
    removeAllChildren(resultTable);
    if (e instanceof RDBException) {
      setStatus("SQL Error code: " + e.getCode() + ": " +
                e.errorInfo()[2]);
    }
    else
      setStatus("Exception: " + e);
  }
}
```

Listing 1.2. Executing SQL

an exception.) If the SQL inserted or deleted a row in a database table, no further processing is necessary. Otherwise (*isQuery()*), an application can use the DOM APIs to create an HTML that displays the results of the query. Listing 1.2 creates the table heading with the result set's *columnNames()*, and fills in the HTML table's cells in Listing 1.3.

As shown in Listing 1.3, successive rows from a query's result-set are retrieved through the *fetch()* method. This advances the cursor of the result set, and returns the contents of the current row from the result set result as an associative array indexed by column name. The value of a specific column in that row can thus be retrieved as *row[columnNames[i]]*. An alternative version, *fetchAll()*,

```
function createTableHead(colNames)
{
  var thead = document.createElement("thead");
  var tr = document.createElement("tr");
  thead.appendChild(tr);
  for (var i = 0; i < colNames.length; i++) {
    var th = document.createElement("th");
    var thText = document.createTextNode(colNames[i]);
    th.appendChild(thText);
    tr.appendChild(th);
  }

  return thead;
}

function createTableBody(stmt, colNames)
{
  var tbody = document.createElement("tbody");
  for (;;) {
    var row = stmt.fetch();
    if (row == null)
      break;

    var tr = document.createElement("tr");
    tbody.appendChild(tr);

    for (var i = 0; i < colNames.length; i++) {
      var td = document.createElement("td");
      tr.appendChild(td);
      var value = row[colNames[i]];
      var tdText = document.createTextNode(value);
      td.appendChild(tdText);
    }
  }

  return tbody;
}
```

Listing 1.3. Displaying Query Results

returns an array containing all of the remaining rows in the result set. Figure 5 shows the results of the application's executing a sample query in the web-browser.

Figure 6 shows how the DBC-JS gateway returns the server-side error codes and messages if the client attempts to execute invalid SQL.

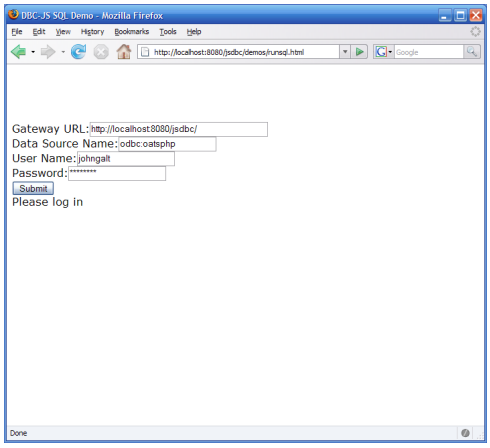

Fig. 4. Sample Login Page

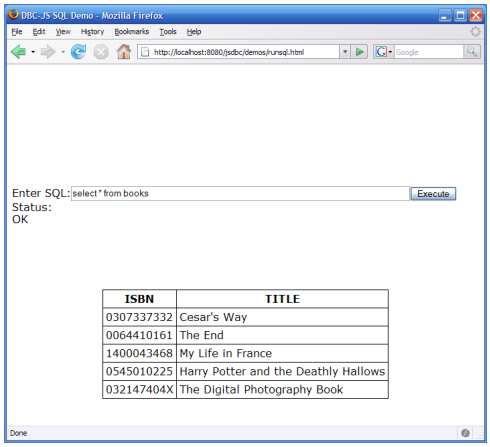

Fig. 5. Sample Query Results Page

5 Status & Conclusions

The current version of DBC-JS includes the API and client-side JavaScript library described in this paper, and a server-side gateway implemented in PHP. The gateway converts client-side messages to corresponding invocations of the "PDO" (PHP Data Object) extension. By providing a data-access abstraction layer, PDOs provide a consistent interface for PHP developers to access relational databases for a large set of different database vendors. We are considering creating a DBC-JS gateway for accessing databases hosted on a Java container.

We think that DBC-JS can play an important role in enabling Web 2.0 applications through its middleware that provides relational database access without compromising enterprise security. By using DBC-JS web-developers can replace

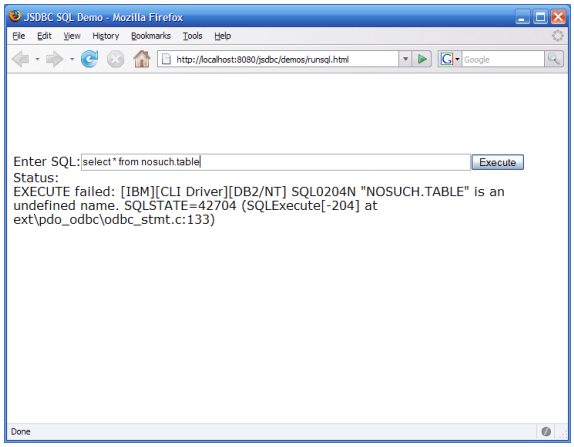

Fig. 6. Sample Error Report Page

ad-hoc approaches to relational data with a framework that hides the underlying message flows and security issues. Because the DBC-JS client library is written in JavaScript it executes natively in web-browsers without the need for special browser-plugins.

References

1. T. O'Reilly. What is web 2.0. `http://www.oreilly.com/go/web2`, September 2005.
2. Wikipedia. Representational state transfer. `http://en.wikipedia.org/w/index.php?title=Representational_State_Transf%er&oldid=109299419`, 2007.
3. Kyle Geiger. *Inside ODBC*. Microsoft Press, 1995.
4. Dojo, the javascript toolkit. `http://dojotoolkit.org/`, 2007.
5. Justin Gehtland, Dion Almaer, and Ben Galbraith. *Pragmatic Ajax: A Web 2.0 Primer*. Pragmatic Bookshelf, 2006.
6. Oat: Openajax alliance compliant toolkit. `http://ajaxian.com/archives/oat-openajax-alliance-compliant-toolkit`, 2007.
7. Opentoro: Database publishing for the web. `http://opentoro.sourceforge.net/`, 2007.
8. A. Leff and J. T. Rayfield. Programming model alternatives for disconnected business applications. *IEEE Internet Computing*, 10(3):50–57, May/June 2006.
9. Avraham Leff and James T. Rayfield. Enterprise javabeans caching in clustered environments. *Concurrency - Practice and Experience*, 17(7-8):1027–1051, 2005.
10. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, CA, USA, 1993.