RC24471 (W0801-084) January 22, 2008
Computer Science

IBM Research Report

Zazen: A Mediating SOA between Ajax Applications and Enterprise Data

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

Zazen: A Mediating SOA Between Ajax Applications and Enterprise Data

Avraham Leff
IBM T.J. Watson Research Center
POB 704
Yorktown Heights, NY, USA
avraham@us.ibm.com

James T. Rayfield
IBM T.J. Watson Research Center
POB 704
Yorktown Heights, NY, USA
jtray@us.ibm.com

Abstract

One reason that enterprises are adopting service-oriented architectures (SOA) is to develop applications more quickly by packing – and then reusing – applications and data assets as services. Service encapsulation of implementation details is an important feature, and contributes to the loosely-coupled nature of a SOA. From this perspective, SOA data-services seem incompatible with AJAX *frameworks which presume a great degree of client-side control of an application's data. For their part,* AJAX *frameworks promise to increase web-application performance by reducing the number of interactions between the browser and server. Caching server data on the web-client is a well-known technique for achieving this goal, but implies that enterprise data is exposed to client-side developers.*

This paper presents ZAZEN*, a SOA that mediates between the need to encapsulate enterprise data as a service and the needs of* AJAX *developers who want more control of their application's data. We describe* ZAZEN*'s server-side architecture and discuss two APIs to the data-service: a REST API, and an implementation of the* DOJO *data APIs for relational databases.*

1. Introduction

Much of the benefits of service-oriented architectures (SOA) can be attributed to its emphasis on using service interfaces to encapsulate applications and data assets [7] [18]. In the context of data-services, a web-service interface is defined with technologies such as WSDL, and then implemented to provide access to a back-end data systems such as a relational or XML database [2]. However, the emphasis on SOA encapsulation – coupled with traditional enterprise resistance to exposing data assets to web-clients – would seem to preclude the use of enterprise data in AJAX applications.

AJAX frameworks [13] [12] promise to increase web-application performance by reducing the number of interactions between the browser and server. Caching server data on the web-client is a well-known technique for achieving this goal [20], since the application can access data locally rather than having to access the server. Because relational data is such an important part of enterprise applications, many frameworks have been devised to integrate relational data in AJAX applications [21] [22] [4]. This capability allows web-applications to be dynamically composed in a web-browser – so-called "Web 2.0" applications [23] – rather than being composed on the server ("Web 1.0"). These AJAX frameworks, however, presume that enterprises are willing to expose their database assets directly to web-developers, rather than being encapsulated as a SOA data-service.

We believe that, in certain contexts, a data-service SOA *can* provide AJAX applications with the enterprise data they need, in a way that fits well with the AJAX programming style. Our data-service SOA, called ZAZEN, is designed for use-cases in which AJAX applications only need access to a:

- *single* source of data (unlike data-services that provide integration of multiple data-sources such as ALDSP [1]),

- and where the enterprise provides a *relational* access layer to the data.

In such contexts, ZAZEN enables AJAX applications to access server-side relational data while still providing enterprises with the SOA benefits of encapsulation and loosely coupled systems.

In Section 2 we take a closer look at the concerns that SOA data-services have with AJAX applications seeking greater access to enterprise relational data. The core of the paper is Section 3, where we show how the ZAZEN SOA enables enterprise to project data to web-applications in an AJAX style, while satisfying key concerns of SOA data-services architecture. We discuss two client APIs to the ZAZEN SOA: a REST API and a higher-level DOJO API. Section 4 presents an example of using the ZAZEN data-service to build an application, and Section 5 evaluates how successfully ZAZEN encapsulates enterprise data as a service. We conclude with a discussion of ZAZEN's current status and ways that we may extend ZAZEN in the future.

2. Enterprise Concerns

To see why enterprises resist projecting relational data to AJAX applications, consider the well-known ODBC approach [14]. (Although the ODBC approach was originally written for non-web programming languages and environments, its design extends naturally to the web and AJAX applications.) To pick a simple example, assume that a web-application is used to display "all employees in a given department". In the ODBC approach, web-developers use the following steps to access the required server-side relational data:

1. The developer specifies the required data in terms of the corresponding SQL statement, e.g., SELECT * FROM DEPARTMENT.

2. The sql statement is passed to a client-side API, typically written in JavaScript, to be executed by the database server.

3. A client-side JavaScript library converts the API call into an XML or JSON message that is transmitted to the server in an *XmlHttpRequest* invocation.

4. This message is interpreted on the server and a server-side API is invoked to execute the SELECT * FROM DEPARTMENT against the server-side database (after authenticating the client's credentials).

5. The server-side API packages the result of the SQL statement in XML or JSON format and sent to the client.

6. The client-side library passes the result to the application that initiated the request.

7. Either the client-side library or the application can cache the employee data so that subsequent requests – e.g., to sort the data in different ways – can be accommodated without another round-trip to the server.

The advantages of the ODBC approach (e.g., [21] [22] [4]) are two-fold: (1) it is powerful, allowing any SQL statement to be executed and (2) it is a well-known approach that has been incorporated in many language specific implementations (e.g., JDBC for Java and PDO for PHP). However, important concerns are typically expressed with using this approach.

First – and most important from a SOA perspective – the ODBC approach forces an enterprise to expose much detail about the employee data. For example, the web-developer has to know the name of the database, the name of the employee table, and the schema used in the employee table (e.g., column names and types). This requirement runs counter to the SOA requirement of service encapsulation. In addition to this "generic" SOA concern, database administrators are specifically concerned about the security of ODBC-based data-services.

At first glance, it is hard to understand why the authentication scheme used by ODBC (often just a *userid* and *password*) for desktop application access to a database server, should not also suffice – at least in an intranet environment – for web-client access to the same database server. Even in an internet environment, where insecure communication is definitely an issue, technologies such as SSL can be used to encrypt client-server communication as necessary. Similarly, concerns that the ODBC approach requires that developers be provided with important details about database table schema, are hard to understand given that desktop developers are provided with precisely this information.

A closer look shows that the key difference between desktop and web applications is the security issue of "trusted code". Authentication schemes prove only that a trusted person is *executing* the code. They do not prove that a trusted person *wrote* the code. Compared to a server-based application, it is much easier to inject malicious code into a Web 2.0 client application, and enterprises are therefore very wary about letting client-side business logic execute directly against their databases. In addition, database servers do not usually have fine-grained access-control mechanisms. Typically the database does not have a userid defined for each end-user of the system, but only a userid for each role that might access the database. Also, access to tables is typically granted on a per-table basis, and not on a per-row or per-column basis. Thus the application code is typically heavily involved in verifying that only authorized users have access to only the data they are authorized to see (in addition to the access control provided by the database manager).

3 Zazen

3.1. Architecture

To address these valid enterprise concerns, we designed ZAZEN (Figure 1) as a data-service SOA that mediates between AJAX applications and enterprise data.

In contrast to the ODBC approach, ZAZEN uses a *labeled* SQL approach. The idea is to label an SQL statement such that:

- Web-developers supply the SQL statement's label. The data returned by ZAZEN is the result-set generated by executing the corresponding SQL statement.
- Database administrators optimize the SQL for their particular environment, and validate it using their enterprise's security policies.
- As shown in Figure 1, the ZAZEN server mediates between web-client requests – which specify a given label – and the database server – that executes the SQL associated by the web-client's label.

ZAZEN addresses an enterprise's encapsulation concerns since the labeled SQL is a higher level abstraction than SQL itself. The label – e.g., "all employees in my department" – in effect names the service, with the associated SQL providing the service implementation. (We examine this claim in more detail in Section 5.) The SQL statements are not limited to providing rows from a single database table; they can provide JOIN results from multiple tables or from operations of arbitrary complexity. The result-set sent by ZAZEN to the web-client can be cached locally in the browser, so web-developers exploit the benefits of the AJAX approach. ZAZEN addresses an enterprise's security concerns because web-developers don't even see the SQL that they're executing; database administrators continue to solely responsible for constructing and validating all SQL that executes in their system. Clients can be prevented from knowing even the column names through SQL that maps real names to virtual names.

3.2. REST API

By "black-boxing" a chunk of server-side relational database logic as a function that can be called by applications, a labeled SQL statement is, in effect, a stored procedure. Stored procedures have advantages compared to ODBC APIs, and database administrators often prefer that they be used even in a desktop application environment. Stored procedures integrate data validation and access control into the database, and allow multiple SQL statements, together with business logic, to be combined in a single package. ZAZEN must therefore provide an API for web-clients to parameterize a labeled SQL statement. For example, if the invoked SQL is SELECT * FROM EMPLOYEES WHERE SALARY < :MIN_SALARY, the API must allow clients to specify a value for the labeled parameter MIN_SALARY. We do this with a synchronous REST [9] [16] protocol. As with other REST protocols, ZAZEN clients construct a URI which is passed to the server in an HTTP GET method. Thus, as shown in Figure 1, whenever a web-application needs relational data it constructs the appropriate REST message based on the API requirements and send the message to the ZAZEN server.

Every ZAZEN URI has the form: `https:`

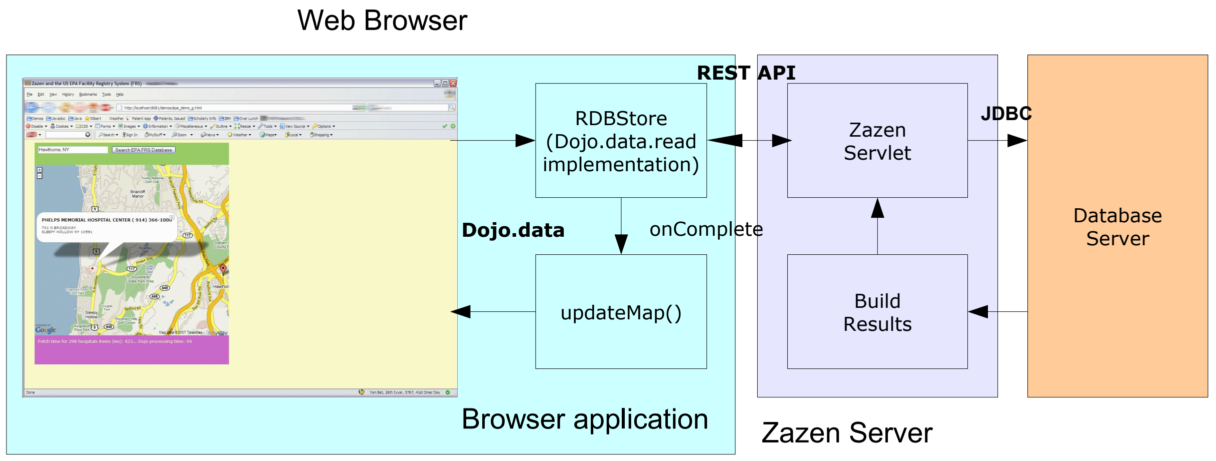

Figure 1. Zazen Architecture

`//.../statement_label?parameter1=value1[&...&]parameterN=valueN`. Some parameters are mandatory: e.g., the name of the database against which the labeled SQL statement is applied, and the user name and password used to authenticate the web-client. Depending on the requirements of a given labeled SQL statement, the URI will also include a parameter name/value pair for every labeled parameter in the corresponding SQL statement.

Other optional URI parameters allow the client to specify filter parameters that sub-set the contents of the result-set in various ways. This ability addresses a major weakness of the labeled SQL approach, namely that it's too inflexible for environments with rapidly changing application requirements. For example, assume that an enterprise has determined that clients with suitable credentials may see the set of employees in a given department via a labeled SQL statement that invokes SELECT * FROM DEPARTMENT. What if an application needs only manager *Smith's* employees rather than the *entire* department's employees? Requiring the enterprise to create a new labeled SQL statement to match such application-specific needs is unrealistic. The alternative forces web-developers to invoke the more general statement, and then filter "by hand" to get the desired subset of data. Doing this correctly and efficiently is hard. The ZAZEN solution exploits the fact that – assuming that a more general SQL statement is secure – security is not compromised by allowing clients to issue a query that returns a subset of the more general query. We do this with filter operations that reduce the number of rows returned by the query, and/or subset the number of columns returned in each row.

In the ZAZEN API, therefore, a web-client specifies a "base" query (*via* the labeled SQL), and can also specify a set of filters that ZAZEN applies to the result set of the base query. Continuing the previous example, the client can specify these filters:

- WHERE MANAGER = 'SMITH', to reduce the number of rows returned by the base query for the entire department
- COLUMNS DEPTNO and DEPTNAME, to eliminate the MANAGER column which is already known to be *Smith*

We use a similar approach to enable clients to specify that certain columns be used to perform an ascending or descending sort of the result-set. Thus, to specify that values of the department number be used to sort the results in ascending order, the client inserts SORTATTRIBUTE0=DEPTNO and ORDERATTRIBUTE0=ASC into the URI. ZAZEN implements other filters (such as COUNT – instructing ZAZEN to return only "count" rows from the result set – and START – instructing ZAZEN to discard a preliminary set of rows) itself.

Importantly, ZAZEN's approach of packaging a data-service as a labeled SQL statement works for data-services other than query. The semantics of the service are determined by the SQL itself. As part of the process of validating the SQL, database administrators determine which REST "verb" (or HTTP method [16]) will be associated with a given labeled SQL statement. Our SELECT * FROM DEPARTMENT example is associated with a GET method; SQL such as UPDATE EMPLOYEES SET DEPT = 'R56' WHERE LAST_NAME ='SMITH' would be associated with a POST method. Similarly, DELETE FROM EMPLOYEES WHERE SSN = '012-34-5679' is invoked with a DELETE method, and INSERT INTO EMPLOYEES (NAME, SSN) VALUES ('JOHN DOE', '012-34-5678' is associated with a PUT method.

3.3. DOJO API

The REST API described in the previous section has the benefit of being a well-understood approach for client access to web-services. It's disadvantage lies in that, from the perspective of a web-developer, it requires some effort to translate the high-level request for data into the correct REST call. We therefore provide a second, higher-level API to the ZAZEN SOA. This is a relational-datastore implementation of the datastore-agnostic DOJO.READ API, and is structured as a mapping between DOJO's higher-level API and ZAZEN's lower-level REST API.

We chose to implement the DOJO.READ API (rather than designing our own) for two reasons. First, the DOJO API is well-integrated into the framework's widget [5] libraries. Second, "Dojo.data is a uniform data access layer that removes the concepts of database drivers and unique data formats. All data is represented as an item or as an attribute of an item. With such a representation, data can be accessed in a standard fashion." [6]. This representation is consistent with ZAZEN's representation of a result-set as a JSON [19] object containing *metaData* (an array of column names) and *resultSet* objects. The resultSet is an array (of DOJO.READ "items"), in which each item's property names are the metaData column names (DOJO.READ "attributes"). An item's attribute values are thus a result set row's column values. The bulk of our DOJO.READ implementation is a thin wrapper that manipulates the "eval-ed" JavaScript object returned by the ZAZEN API. For example, `store.getAttributes(someItem)` returns the column names for a given row; `store.getValue(someAttribute, someItem)` returns the row's value for the specified column.

DOJO already provides implementations for XML, JSON, and other data-stores; ZAZEN's *RDBStore* is an implementation for relational data-stores. An *RDBStore* instance is a client-side handle to a relational database, and is initialized with the server-side ZAZEN URL. The `fetch()` method is an event-driven, asynchronous, method which may specify – in addition to the query itself – the following optional parameters:

- an `onBegin` function, invoked immediately before processing the query's items,
- an `onItem` function, invoked individually on each item,
- an `onComplete` function, invoked after all the items have been processed,

- `onError` function, invoked if an error is invoked.

In addition, optional *start*, *count*, and *sort* parameters can also be specified.

The DOJO.READ "does not specify the syntax or semantics of the query itself": these are supplied with the *query* parameter. *RDBStore* requires that the *query* parameter include the database name, user name, password, and a label that specifies a ZAZEN labeled SQL statement. If the labeled SQL includes named parameters, the *query* parameter includes two arrays that specify the i_{th} parameter name and value. The *RDBStore* instance uses the *query* parameter to construct the corresponding server-side ZAZEN URI, and asks the server to return the specified result-set. The end-to-end flow between web-application and the database server is shown in Figure 1).

4. Mashup Example

To get a sense of how the ZAZEN SOA is used, we'll describe how we built a sample application that integrates data from the Environmental Protection Agency's Facility Registry System into a map web-application. We'll describe the steps used to build the application from the perspective of an AJAX developer.

"The Facility Registry System (FRS) is a centrally managed database that identifies facilities, sites or places subject to environmental regulations or of environmental interest."[11] The FRS database stores data such as a facility's name, phone number, and geographic coordinates. The AJAX developer wants to build a web-application in which facilities are layed out in a "zoomable" map, and through which users can click on a given facility to get more information about that facility – i.e., a classic "map mashup".

1. The AJAX developer constructs an HTML form into which users specify an address or geographic coordinates.

2. The form is linked to JavaScript code which uses a map API (e.g., Google Maps [15], or ESRI [8]) to build a zoomable map that's centered on the user-specified coordinates.

3. Because FRS allows its data to be exported in CSV format [10], large amounts of data are easily imported into a relational database. A database administrator loads the FRS data into three database tables: *hospitals*, *frs*, and *schools*.

4. The ZAZEN data-services SOA is now used to bridge the gap between the client portion of the sample (the left-most portion of Figure 1) and the database server (the right-most portion of Figure 1).

   A database administrator first validates SQL such as SELECT * FROM HOSPITAL WHERE ((LATITUDE $>$:MINLAT AND LATITUDE $<$:MAXLAT) AND (LONGITUDE $>$:MINLONG AND LONGITUDE $<$:MAXLONG)). This SQL provides a service API through which users request all hospital facilities located in a geographical area defined by max/min latitude and max/min longitude parameters. Next, the administrator associates this SQL with a label such as "epa_hospitals_query".

5. The AJAX developer adds an *RDBStore* instance to her application, configuring it to access the ZAZEN server. The map selected by the user is accessed to determine the bounds of an area that will plausibly be large enough to hold the current map plus another zoom factor (so the application can cache data to satisfy future requests without another request to the server). Code-sample 1 shows how the DOJO.READ `fetch()` method is invoked on the *RDBStore*, which in turn, invokes the ZAZEN data-service.

   This is done by passing the SQL statement label (e.g., "epa_hospitals_query"), together with the database name, user name, and password to the *RDBStore*. The *RDBStore* invokes the ZAZEN data-service asynchronously to populate the map with markers corresponding to the set of facility items returned by the service. The markers are positioned on the latitude and longitude specified by the facility data, with the bulk of the information (e.g., telephone number) cached in the *RDBStore*. When users click on a given facility marker, the web-application accesses the *RDBStore* to display a popup containing this cached information.

Code Sample 1 Using the DOJO.READ API to Access the ZAZEN Data-Service

```
/** Invokes the zazen data-service with a query against the EPA database.
 *
 * @param epaSample instance of this class.
 * @param epaType one of 'hospitals', 'schools', 'frs'
 * @param sw south-west GLatLng of the rectangle for which we want data
 * @param ne north-east GLatLng of the rectangle for which we want data
 */
this.fetchEPAData = function(epaSample, epaType, sw, ne)
{
  var keywordArgs = initKeywordArgs(epaType);
  keywordArgs.bindParams(sw.lat(), ne.lat(), sw.lng(), ne.lng());
  keywordArgs.startFetchTime = new Date().getTime();
  keywordArgs.epaType = epaType;

  keywordArgs.onBegin = function (size, request) {
    request.stopFetchTime = new Date().getTime();
    request.startProcessingTime = new Date().getTime();
  };

  keywordArgs.onComplete = function(items, request) {
    var epaType = request.epaType;
    epaSample._numberOfItems[epaType] = items.length;
    epaSample._fetchMillis[epaType] = request.stopFetchTime - request.startFetchTime;

    for (var i=0; i<items.length; i++) {
      var item = items[i];
      var position = buildLatLng(epaSample.getRDBStore(), item);
      var marker = epaSample.createMarker(position, epaSample, item, epaType);
      epaSample._markers.push(marker);
    }

    markerManager.addMarkers(epaSample._markers, minZoom);
    markerManager.refresh();
  };

  keywordArgs.onError = function(errData, request) {
    setErrorPage('Problem fetching data from EPA Database table '+epaType+
      ': '+errData);
  };

  epaSample.getRDBStore().fetch(keywordArgs);
};
```

5. Evaluation

ZAZEN differs from other data-service SOAs in that its services are not formally defined (e.g., using WSDL), and because it's missing features such as integration of multiple back-end data-stores(e.g., [1]). ZAZEN is designed for rapid-development of departmental-sized applications whose data is typically stored in a single relational database. Importantly, such environments often lack the resources for IT architects to formally define, and implement, a service. We contend that, in such environments, the productivity benefits offered by ZAZEN's light-weight service definition outweigh its disadvantages.

ZAZEN is unusual in that the service implementation is wholly specified in SQL, unlike other services that access relational databases only for data, and provide business logic in a programming language such as

Java or PHP. Although relational algebra can be very powerful [3], from this perspective, ZAZEN may be viewed as providing a lower-level service than the typical SOA. However, although this paper has focused on ZAZEN's encapsulation of *single* SQL statements, ZAZEN can be used – using exactly the same configuration – to invoke a stored procedure [17] – *via* standard APIs such as JDBC. This means that server-side developers can use ZAZEN to provide AJAX developers with access to encapsulations of arbitrary amounts of business logic, written in a high-level programming language, and interleaved with any number of SQL statements.

6. Status & Future Work

ZAZEN is currently implemented using a Java Servlet as the front-end of the ZAZEN server, and uses JDBC to access the database-server tier. We provide a JavaScript implementation of the DOJO.READ API that AJAX developers can load into their web-application. As shown in Figure 1, the JavaScript library translates `dojo.data.api.read.fetch()` calls into invocations of the corresponding ZAZEN REST API.

Although the REST API (Section 3.2) enables developers to both read and write data, our DOJO implementation only provides a read-data capability (Section 3.3). The next step is to implement DOJO.WRITE as well, taking care to ensure that security concerns continue to be addressed in this more sensitive context.

References

[1] M. Carey. Data delivery in a service-oriented world: the bea aqualogic data services platform. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 695–705. ACM, 2006.

[2] E. Cerami. *Web Services Essentials Distributed Applications with XML-RPC, SOAP, UDDI & WSDL.* O'Reilly, 2002.

[3] C. Date and H. Darwen. *Databases, Types and the Relational Model (3rd Edition).* Addison-Wesley, Boston, MA, 2006.

[4] Database connectivity for javascript. `http://w3.alphaworks.ibm.com/techs/overview.jsp?tech=dbcjs`, 2007.

[5] Dojo, the javascript toolkit. `http://dojotoolkit.org/`, 2007.

[6] Using dojo.data. `http://dojotoolkit.org/book/dojo-book-0-9/part-3-programmatic-dijit-and-dojo/data-retrieval-dojo-data-0`, 2007.

[7] T. Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design.* Prentice Hall, 2005.

[8] Arcweb explorer javascript. `http://www2.arcwebservices.com/v2006/develop/awx.jsp`, 2007.

[9] R. Fielding. Representational state transfer (rest). `http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`, 2007.

[10] Facility registry system: Ez query. `http://www.epa.gov/enviro/html/fii/ez.html`, 2007.

[11] Facility registry system (frs). `http://www.epa.gov/enviro/html/fii/index.html`, 2007.

[12] J. J. Garrett. Ajax: A new approach to web applications. `http://www.adaptivepath.com/publications/essays/archives/000385.php`, 2005.

[13] J. Gehtland, D. Almaer, and B. Galbraith. *Pragmatic Ajax: A Web 2.0 Primer.* Pragmatic Bookshelf, 2006.

[14] K. Geiger. *Inside ODBC.* Microsoft Press, 1995.

[15] Google maps api documentation. `http://www.google.com/apis/maps/documentation/`, 2007.

[16] J. Gregorio. How to create a rest protocol. `http://www.xml.com/pub/a/2004/12/01/restful-web.html`, 2004.

[17] G. Harrison and S. Feuerstein. *MySQL Stored Procedure Programming.* O'Reilly, Sebastopol, CA, USA, 2006.

[18] M. N. Huhns and M. P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005.

[19] Json in javascript. `http://json.org/js.html`, 2007.

[20] A. Leff and J. T. Rayfield. Programming model alternatives for disconnected business applications. *IEEE Internet Computing*, 10(3):50–57, May/June 2006.

[21] Oat: Openajax alliance compliant toolkit. `http://ajaxian.com/archives/oat-openajax-alliance-compliant-toolkit`, 2007.

[22] Opentoro: Database publishing for the web. `http://opentoro.sourceforge.net/`, 2007.

[23] T. O'Reilly. What is web 2.0. `http://www.oreilly.com/go/web2`, September 2005.