LIBERATED: A fully in-browser client and server web application debug and test environment

Derrell Lipman
University of Massachusetts Lowell

Abstract

Traditional web-based client-server application development has been accomplished in two separate pieces: the frontend portion which runs on the client machine has been written in HTML and JavaScript; and the backend portion which runs on the server machine has been written in PHP, ASP.net, or some other "server-side" language which typically interfaces to a database. The skill sets required for these two pieces are different.

In this paper, I demonstrate a new methodology for web-based client-server application development, in which a simulated server is built into the browser environment to run the backend code. This allows the frontend to issue requests to the backend, and the developer to step, using a debugger, directly from frontend code into backend code, and to debug and test both the frontend and backend portions. Once working, that backend code is moved to a real server. Since the application-specific code has been tested in the simulated environment, it is unlikely that bugs will be encountered at the server that did not exist in the simulated environment.

I have implemented this methodology and used it for development of a live application. All of the code is open source.

1 Introduction

Web-based client-server applications can be difficult to test and debug. Disparate development environments on the client and server sides, distinct skill sets for each, and a network that precludes easy synchronous debugging all impede debugging at the client side. Sometimes, the server environment provides little debugging and testing infrastructure.

I will describe here an architecture and framework that allows writing both the frontend code that runs on the client machine (i.e., in the browser) and the backend code that typically runs on a server machine, in a single language. Furthermore, this architecture allows debugging and testing the entire application, both frontend *and* backend, within the browser environment. Once the application is tested, the backend portion of the code can be moved to the production server where it operates with little, if any, additional debugging.

1.1 Typical web application development

There are many skill sets required to implement a modern web application. On the client side, initially, the user interface must be defined. A visual designer, in conjunction with a human-factors engineer, may determine what features should appear in the interface, and how to best organize them for ease of use and an attractive design.

The language used to write the user interface code is most typically JavaScript [6]. There need be at least a small amount of HTML to load the JavaScript code. Many applications are written using a JavaScript framework such as jQuery, ExtJS, or qooxdoo. Developers must therefore be fluent with both the language and the framework.

Debugging is generally accomplished using a debugger provided by the browser, or a plug-in to the browser.

The backend software includes the web server and database engine. Recent statistics [7] show that PHP and ASP.NET are the most popular languages for writing the backend code. Each provides a mechanism for receiving requests in the agreed upon application communication protocol (encoding) from the frontend. These languages also provide a means of communicating with a separate database server, or to an embedded database.

The application-specific backend code, or "business logic," is usually initiated by a web server which may or may not provide mechanisms for easy debugging of the application code. When a debugger is not available, the developer must rely on `print` or `log` statements to ascertain the code location of problems.

With the differing coding language and operating en-

vironment comes unique debugging methodologies. The skill sets required for debugging at the client and server are different, so any debugging session may require the availability of multiple people. Making debugging even more difficult is the asynchronous nature of the client-server interaction. Request messages are sent via the transport, and at some future time, response messages are returned. This separation of client and server means that it is not possible to use a debugger at the browser to step into code which is running on the server, nor even set a breakpoint that would allow stopping at the server-side handler for a key or button press at the user interface.

1.2 Research question

With the afore-mentioned problems in mind, I ask:

Is it feasible to design an architecture and framework for client-server application implementation that allows:

1. *all application development to be accomplished primarily in a single language;*
2. *application frontend and backend code to be tested and debugged within the browser environment; and*
3. *debugged and tested application-specific backend code to be moved, unchanged, from the browser environment to the real server environment, and to run there?*

In order to accomplish this, we first need a language that can be used both in the browser and on the server. For cross-browser use, the only viable choice is JavaScript. We therefore need a JavaScript implementation of the backend code that could run both in the browser and on the server, which can talk to whatever server-side database is to be used. The desired architecture is depicted in Figure 1.

Additionally, we need some form of abstraction that encompasses the set of database operations that are performed. The mechanism must map to a particular database on the server, and to a simulation of the database in the browser.

Two new questions arise out of such an architecture:

1. *How much of a compromise does this architecture impose, i.e., what common facilities become unavailable or more difficult to use?*
2. *Does this new architecture create new problems of its own?*

2 Introducing LIBERATED

LIBERATED is an architecture and JavaScript library upon which full web applications can be built. **LIBERATED** allows a web application to be debugged and

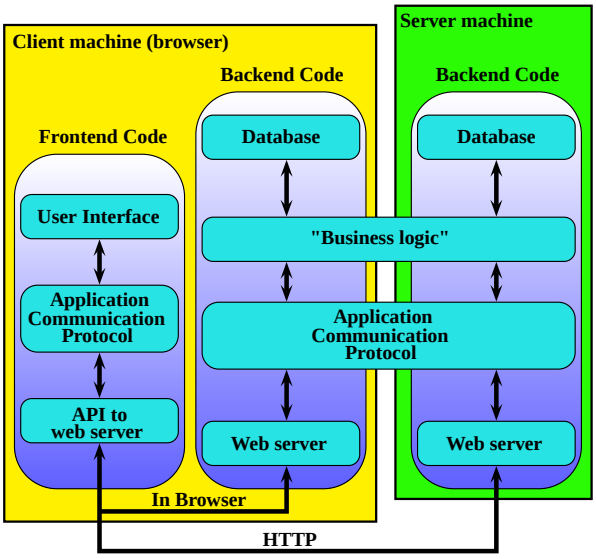

Figure 1: Desired architecture

tested, fully within the browser environment. Once all code is working, that same code can be moved to a real server, and run there. **LIBERATED** truly lives up to its name, liberating the developer from many of the hassles of traditional web application debugging.

LIBERATED is extensible. At present, it provides the following components:

- Database abstraction, used by an application
- Mapping from the database abstraction to the App Engine datastore
- Mapping from the database abstraction to SQLite[1]
- Mapping from the database abstraction to a simulated database which runs in the browser
- JSON-RPC Version 2.0 server
- Web server interface for App Engine
- Web server interface for the Jetty web server[2]
- Transport simulator to talk to an in-browser web server
- Hooks into the qooxdoo JavaScript framework, to allow use of the transport simulator in addition to its standard transports[3]

The following sections will discuss the overall architecture of **LIBERATED** and provide additional details of important components.

2.1 Architecture

In the backend, when using **LIBERATED**, an application's "business logic" code interacts with the database

[1] `http://sqlite.org`
[2] `http://jetty.codehaus.org/jetty/`
[3] `http://qooxdoo.org` (pronounced ['kuksdu:])

using a database abstraction provided by **LIBERATED**. Using the database abstraction allows the actual database to be *real* or *simulated*. A real database is the App Engine datastore, SQLite, MySql, etc, whereas a simulated database runs in the browser. Similarly, the backend receives requests from the frontend via a **transport** that can be either *real*, communicating across a network, or *simulated*, communicating solely within the browser.

LIBERATED handles requests which arrive via the selected **transport**. With a real web server such as provided by App Engine or Jetty, requests arrive via the HTTP protocol. When requests arrive via the simulated transport, they are placed on a queue, and handled in sequence from there, by a simulated web server.

The web server, whether *real* or *simulated*, determines which handler should process a request. A handler for the JSON-RPC server is currently implemented. Others, such as for REST could be added.

2.2 Development environment

The JavaScript framework upon which **LIBERATED** is implemented is qooxdoo. The qooxdoo framework provides a traditional class-based object programming model, and a wealth of additional functionality including classes to assist communicating over a network. There is nothing qooxdoo-specific, however, to this technology, and **LIBERATED** can be used in a non-qooxdoo-based application.

2.3 Database abstraction

In a common SQL-accessed relational database, data is organized into **tables** with names that identify the type of data that is stored in the **table**. A **table** contains **rows** of data, each with a common set of **columns** or **fields**. Each **row** is uniquely identified by a **key** value contained in one or more of its columns.

The database abstraction in **LIBERATED** is built upon a class called `liberated.dbif.Entity`. Each "table" can be thought of as being defined as a separate subclass of `liberated.dbif.Entity`. An instance of one of those subclasses, referred to as an **entity**, represents a row from that table. Each subclass of `liberated.dbif.Entity` defines a unique **entity type**.

`liberated.dbif.Entity` contains a method for registering the **properties** (like column names and types) which are members of each entity of that **entity type**.

To add a new object to the database, an entity of the proper subclass of `liberated.dbif.Entity` is instantiated, its property values set, and its `put()` member method called. When instantiating the subclass, the **key field(s)** of the **entity type** can be provided, to retrieve a specific existing object from the database. The `liberated.dbif.Entity.query()` function is used to retrieve specified sets of objects of an entity type from the database.

At present, relationships among entity types must be maintained by the application. Future plans include improvements in this area.

2.4 JSON-RPC server

The JSON-RPC server accepts incoming requests and returns responses in the format specified by the JSON-RPC Version 2.0 standard. [3] Remote procedure call methods are registered as a tuple consisting of the name of the method, a function that implements the remotely-accessible method, and an array that lists the names of the parameters. The latter allows requests to use either positional parameters or named parameters.

3 Example use of LIBERATED

To demonstrate, in part, how **LIBERATED** is used, consider a database entity which implements a counter. This simple entity type is shown in Listing 1.

Listing 1: Entity type definition for a simple counter

```
1  qx.Class.define("example.ObjCounter",
2  {
3    extend : liberated.dbif.Entity,
4
5    construct : function(id)
6    {
7      // Pre-initialize field data
8      this.setData({ "count" : 0 });
9
10     // Call the superclass constructor
11     this.base(arguments, "counter", id);
12   },
13
14   defer : function()
15   {
16     var Entity = liberated.dbif.Entity;
17
18     // Register the entity type
19     Entity.registerEntityType(
20       example.ObjCounter,
21       "counter");
22
23     // Register the properties
24     Entity.registerPropertyTypes(
25       "counter",
26       {
27         "id"    : "String",
28         "count" : "Integer"
29       },
30       "id");
31   }
32 });
```

The key field for this entity type is a string, referred to as *id*. As soon as this class has been loaded, the `defer()` function is called, which registers the entity type, so it is immediately available for use once the entire application has been loaded. The name of this class (`example.ObjCounter`) and the entity type name ("counter") are provided in the entity type registration, as

shown on lines 19–21. This entity type has two properties: the counter's *id* and its *count*, which are registered on lines 24–30.

When a new object of this class is instantiated, default data is provided for the *count* field: it is initialized to zero, by line 8.

Listing 2 shows how remote procedure calls are implemented. Line 6 begins the registration of the remote procedure named "countPlusOne". Line 7 maps that name to the `countPlusOne` method which begins at line 13. Line 8 shows the list of parameters that are expected or allowed to be passed to the "countPlusOne" RPC. In this case, a single parameter, a counter ID, is expected.

Listing 2: RPC to increment a counter

```
1  qx.Mixin.define("example.MCounter",
2  {
3    construct : function()
4    {
5      // Register the 'countPlusOne' RPC
6      this.registerService("countPlusOne",
7                           this.countPlusOne,
8                           [ "counterId" ]);
9    },
10
11   members :
12   {
13     countPlusOne : function(counter)
14     {
15       var             counterObj;
16       var             counterDataObj;
17
18       liberated.dbif.Entity.asTransaction(
19         function()
20         {
21           // Get the counter object
22           counterObj =
23             new example.ObjCounter(counter);
24
25           // Get the application data
26           counterDataObj =
27             counterObj.getData();
28
29           // Increment the count
30           counterDataObj.count++;
31
32           // Write it back to the database
33           counterObj.put();
34
35         }, [], this);
36
37       // Return new counter value
38       return counterDataObj.count;
39     }
40   }
41 });
```

The implementation of `countPlusOne()` begins a database transaction to ensure that all manipulation of the database is accomplished based on a consistent database state. The function passed as the first parameter to `asTransaction()` will be called once a transaction has been established. When that function completes, the transaction will be ended.

The function to be run as a transaction begins at line 19. It first obtains the current counter object based on the specified counter ID, at line 22, and then retrieves that object's data map, at line 26. The data map contains the values of the two fields in this entity type (*id* and *count*).

The *count* field is incremented, and then the counter object is written back to the database with line 33.

The return value of this function, the counter's new value, is returned by `asTransaction()` after ending the transaction.

4 Discussion

One of the clear benefits of the **LIBERATED** architecture is that key portions of debugging and testing can be easier to handle than with traditional client-server applications. In this section, I will discuss some techniques that are now available, and our experience using them.

4.1 Debugging

The frontend and backend are traditionally initially debugged in isolation. They are often written in different languages, may be developed by different teams, and may not be able to run on the same machine. The interface between them may be implemented solely to a service API specification, with little ability for the frontend and backend to interact until both are nearly completed. There is often no easy way to use a single debugger to step through the code. It may be possible to have separate frontend and backend debuggers, but some server environments do not provide any easy means of debugging, and developers resort to `print` or `log` statements in the code.

With an application developed with **LIBERATED**, debugging of frontend and backend code need not be accomplished in isolation, both are written in the same language, and the service API can be exercised easily during development. This allows early and iterative debugging during the development process. The developer can use a debugger running in the browser to step from frontend code into backend code, or set breakpoints in backend code and then interact with the user interface to cause a request to be sent to the backend... and immediately have the debugger stop at that breakpoint.

4.2 Debugging Experience

During the course of developing the App Inventor Community Gallery, a complete application built upon **LIBERATED**, the **LIBERATED** architecture time and again proved itself to be a highly efficient and easy to use development and debugging environment. Instead of developing the frontend and backend code in isolation, we implemented and tested new user interface features and any corresponding backend changes concurrently. With **LIBERATED**, when new code doesn't work as intended, our typical debug cycle is:

1. Set a breakpoint in the remote procedure call implementation in the backend code. Run the program.
2. If the breakpoint in the RPC is hit, review the received parameters to ensure they are as expected. Step through the RPC implementation, noting variable changes, return values from functions, etc., until the problem is identified.
3. If the breakpoint in the RPC implementation is not hit, this indicates that there is likely a problem in the way the RPC is called. Set a breakpoint in the new frontend code, where the remote procedure call is initiated.
4. Run the program again, and at the breakpoint, ensure that the proper remote procedure call is being requested, and that the parameters have the expected values. If not, fix the problem, and repeat the process.
5. If, upon running the program in the previous step, the breakpoint is not hit, normal frontend debugging procedures are used to ascertain where the code is faulty.

5 Related work

I have been unable to find any literature or related projects which accomplish all of my goals set forth in Section 1.2. Although there is work in progress on the various sub-pieces described here, there appear to be none that would allow an application to be written in a single language, debugged and tested in the browser, and allow debugged, tested code to then be moved to the real server. Significant work which encompasses or relates to portions of my goals is described here.

5.1 Server-side JavaScript

The three JavaScript engines in common use are V8, used in the Chrome browser; SpiderMonkey, embedded in a number of Mozilla products; and Rhino, an implementation of JavaScript written in Java, also from the Mozilla Foundation. Each engine allows adding scripting to an application, so it is easy to build products around the engine. A plethora of such products have shown up in the last few years [8].

5.2 Web standard database interfaces

Work is progressing on a standard database interface for local storage of data at the browser. The proposal gaining acceptance for a browser database interface is *Indexed Database API* [5]. The Indexed Database API provides a programmatic database interface somewhat similar to the database abstraction in **LIBERATED**. Once it is widely available, the Indexed Database API could be used for an improved client-side simulated database in **LIBERATED**.

5.3 Reducing the distinction between client and server

The problem of different languages for client and server development is being tackled in different ways by various projects. The following sections describe some current work in progress.

5.3.1 Google Web Toolkit

Google's answer to unifying the client and server languages for web application development is called the Google Web Toolkit [1]. GWT allows the developer to write client-side code in Java, which is then translated into JavaScript to run in the browser. GWT is essentially backend-agnostic. GWT allows writing frontend applications in Java, and optionally also writing backend applications in Java, to accomplish the language unification.

5.3.2 Plain Old Webserver

Plain Old Webserver (POW) is a browser add-on that provides a web server that runs in the browser. The server "uses Server Side Javascript (SJS), PHP, Perl, Python or Ruby to deliver dynamic content." [4] Using Plain Old Webserver allows cross-platform, consistent access to a single server implementation. It runs on Firefox, on Linux, Mac, or Windows. It does not, however, provide the ability to step from frontend code into backend code.

5.3.3 Wakanda

Wakanda [2] provides a datastore and HTTP server, a Studio to visually design both the user interface and the data models which define how the datastore is organized, and a high-integrated code editor. It also provides the communications mechanism between frontend and backend, and data binding of user interface components to the datastore. The server-side language is JavaScript. Wakanda comes close to meeting the requirements of my research question, but it lacks **LIBERATED**'s critical ability to debug round trip operations, e.g., to trace into backend code upon initiation of a request via a frontend user action. It is also not fully cross-platform. The Wakanda Studio works only on Mac OS X and Windows, not on Linux. (The Wakanda Server, however, does run on Linux.)

6 Conclusions

The implementation of **LIBERATED** shows that an architecture that meets my research questions from Sec-

tion 1.2 is feasible. **LIBERATED** allows both the frontend and backend of the application to be coded in JavaScript. With the simulated server running the backend code in the browser, all of the code can be debugged purely within the browser, with no need for an external server to run the backend code. Breakpoints can be set in backend code, within the browser, or the developer can step directly from frontend code into backend code. Finally, as has been shown with the Google App Engine and Jetty/SQLite interfaces of **LIBERATED**, the working application-specific backend code can be moved to a real server environment and run there.

The answers to my follow-up questions in Section 1.2 are not as clear cut, however.

6.1 Compromises of this approach

Although the architechture of **LIBERATED** is easy to work with and accomplishes the goals set out by my research question, a number of open issues remain, and it is yet to be determined how much impact these might have. These mostly pertain to the database abstraction. To wit:

- Testing a large web app often requires a substantial database. The current simulation database in **LIBERATED** is not adequate for complete testing of an application.
- **LIBERATED** does not yet provide for automated operations based on relations between entities.
- The complete set of property types which an application may use is defined by **LIBERATED**. The target database may allow other types.
- Some datastores, e.g., Google App Engine, do not require a pre-defined schema, but **LIBERATED** requires one.

6.2 New problems of this approach

There have been few new problems seen as a result of using this approach. The most obvious one is that server-side JavaScript is still young, and plentiful libraries of code are not yet available. Even now, though, **Node.js** is building a large library of code, easily `require()`'d (included) from custom code.[4] As server-side JavaScript matures, it appears likely that this problem may simply evaporate.

7 Recommendations

LIBERATED is a working implementation that is being used in a significant application. There is ample related and continuation work that can be done, however.

[4] `http://nodejs.org/`

The most urgent need is a rigorous evaluation of the benefits of **LIBERATED** vs. one or more traditional development paradigms. At present, my conclusions are based only on the development of App Inventor Gallery by one team of developers.

Additionally, there are some obvious improvements that can be made.

- Relationships between objects in **LIBERATED** are ad hoc, maintained exclusively by the application. Object relationships should be defined in the **LIBERATED** database abstraction, allowing for such things as automatic retrieval of related records or cascading deletes.
- The simulation database driver could use the HTML5 Indexed Database for a more capable simulated database.
- Query operators other than "and" should be supported.

Acknowledgments

The inspiration for **LIBERATED** was App Inventor Community Gallery, which was developed under a grant from Google to Professor Fred Martin at UMass Lowell.

Availability

The fully-open-source **LIBERATED** and App Inventor Community Gallery are available from their respective github repositories:

`https://github.com/liberated/liberated`
`https://github.com/app-inventor-gallery/aig`

References

[1] Google Web Toolkit Overview. `http://code.google.com/webtoolkit/overview.html`.

[2] 4D. Wakanda JS.everywhere(). `http://www.wakanda.org/features`.

[3] JSON-RPC WORKING GROUP. JSON-RPC 2.0 specification. `http://jsonrpc.org/spec.html`, Mar. 2010.

[4] KELLOGG, D. Plain Old Webserver. `http://davidkellogg.com/wiki/Main_Page`.

[5] W3C. Indexed Database API. `http://dvcs.w3.org/hg/IndexedDB/raw-file/tip/Overview.html`.

[6] WEB TECHNOLOGY SURVEYS. Usage of client-side programming languages for websites. `http://w3techs.com/technologies/overview/client_side_language/all`, January 2012.

[7] WEB TECHNOLOGY SURVEYS. Usage of server-side programming languages for websites. `http://w3techs.com/technologies/overview/programming_language/all`, January 2012.

[8] WIKIPEDIA. Comparison of server-side JavaScript solutions. `http://en.wikipedia.org/wiki/Comparison_of_server-side_JavaScript_solutions`.