

***LIBERATED: A FULLY IN-BROWSER CLIENT AND SERVER
WEB APPLICATION DEBUG AND TEST ENVIRONMENT***

BY

DERRELL LIPMAN
B.S., UNIVERSITY OF THE PACIFIC (1983)

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF SCIENCE
COMPUTER SCIENCE
UNIVERSITY OF MASSACHUSETTS LOWELL

Author
November 28, 2011

Certified by.....
Fred Martin
Associate Professor
Thesis Supervisor

Certified by.....
Mark Sheldon
Lecturer
Thesis Reader

Accepted by
Jie Wang
Professor and Chair

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

by

Derrell Lipman

Abstract of a thesis submitted to the faculty of the
Computer Science
in partial fulfillment of the requirements
for the degree of
Master of Science
University of Massachusetts Lowell
2011

Thesis Supervisor: Fred Martin
Title: Associate Professor

Thesis Reader: Mark Sheldon
Title: Lecturer

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, meaning that sometimes the frontend and backend are developed and tested completely independently, based purely on an interface specification. More recently, server-side JavaScript has begun to gain momentum, allowing for more overlap of skill set, but still requiring separate development and testing of the frontend and backend pieces.

In this thesis, I propose 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 code to issue requests to the backend in either a synchronous or asynchronous fashion, step, using a debugger, directly from frontend code into backend code, and to completely test both the frontend and backend portions. That exact same backend code, now fully tested in the simulated environment, is then moved, unaltered, to a real server. Since the application-specific code has been fully tested in the simulated environment and moves unchanged to the server, it is unlikely that bugs will be encountered at the server that did not exist in the simulated environment.

To show that this proposal is more than just theory, I will discuss an implementation that has proven this concept in practice.

Acknowledgments

My work on this project over the past year, and in fact, all of my graduate work to date, has been quite enjoyable, but very time consuming. I wish to thank my wife and kids for allowing me the time to pursue this endeavor.

Professor Fred Martin, my advisor, has been continually encouraging, while still ensuring that I am always both being challenged externally, and am challenging myself. I sincerely thank him for his efforts on my behalf.

Professor Martin and Professor Mark Sheldon, my thesis reader, provided invaluable input to this thesis.

Google provided the grant for developing the App Inventory Community Gallery which provided the impetus to create **LIBERATED**.

Reed Spool, a key contributor to the App Inventor Community Gallery, has offered innumerable suggestions that have changed the direction of this work for the better.

This entire body of work would likely never have come into being were it not for the existence of the qooxdoo project. The qooxdoo team has developed a superb product, and demonstrated an ideal methodology to organizing and running an open source project.

Contents

Preface	1
Introduction: Defining the problem	2
1 Typical development environment	4
2 The research question	8
Methodology: Designing a reference implementation	10
3 Introducing LIBERATED	12
3.1 Switchable transports	13
3.1.1 Existing transports	13
3.1.2 Adding a simulation transport	16
3.2 Required backend elements	19
3.2.1 Server-side JavaScript	19
3.2.2 Means of application communication	20
3.2.3 Database models	23
3.3 Glue code	26
3.3.1 Frontend/backend communication	27
3.3.2 Mapping the database abstraction to a database	27
3.4 LIBERATED implementation details	27
3.4.1 The qooxdoo JavaScript framework	28
3.4.2 Interface for remote communication	34

3.4.3	Transport simulator	35
3.4.4	Simulation web server	42
3.4.5	Portable JSON-RPC server	44
3.4.6	Per-backend remote procedure call interface	54
3.4.7	Database manipulation	60
Results: Testing the reference implementation		103
4	App Inventor Community Gallery: An example application	104
4.1	Background	105
4.2	Destination: Google App Engine	105
4.3	Remote procedure calls	106
4.4	Database schema	107
Discussions		110
5	Application development benefits	110
5.1	Debugging	110
5.2	Automated tests	111
5.3	Debugging Experience	112
6	Related work	114
6.1	Testing methodologies	114
6.2	Server-side JavaScript	115
6.3	Web standard database interfaces	116
6.3.1	Web SQL Database	116
6.3.2	Indexed Database API	117
6.4	Reducing the distinction between client and server	117
6.4.1	Program Mobile Robots in Scheme	118
6.4.2	Google Web Toolkit	118
6.4.3	Dart	119

6.4.4	Plain Old Webservice	120
6.4.5	CouchDB and PouchDB	120
6.4.6	Wakanda	120
Conclusions		123
	Revisiting the Research Question	123
	Compromises of This Approach	124
	New Problems Created By This Approach	125
Recommendations		127
Epilogue		129
Appendix – LIBERATED source code		135
1	rpcjs.sim.remote.Transport	135
2	rpcjs.sim.remote.MRequest	141
3	rpcjs.sim.remote.MExchange	142
4	rpcjs.sim.remote.MRpc	145
5	rpcjs.sim.Simulator	147
6	rpcjs.rpc.Server	151
7	rpcjs.AbstractRpcHandler	161
8	rpcjs.sim.Rpc	166
9	rpcjs.appengine.Rpc	169

10	rpcjs.dbif.Entity	170
11	rpcjs.appengine.Dbif	182
12	rpcjs.sim.Dbif	194
13	aiagallery.dbif.Constants	204
14	aiagallery.dbif.DbifAppEngine	206
15	aiagallery.dbif.DbifSim	209
16	aiagallery.dbif.Decoder64	213
17	aiagallery.dbif.Entity	216
18	aiagallery.dbif.MDbifCommon	218
19	aiagallery.dbif.MApps	223
20	aiagallery.dbif.MComments	243
21	aiagallery.dbif.MFlags	250
22	aiagallery.dbif.MLiking	254
23	aiagallery.dbif.MMobile	256
24	aiagallery.dbif.MSearch	264
25	aiagallery.dbif.MTags	268
26	aiagallery.dbif.MVisitors	270
27	aiagallery.dbif.MWhoAmI	275
28	aiagallery.dbif.ObjAppData	277
29	aiagallery.dbif.ObjComments	280

30 aiagallery.dbif.ObjDownloads	282
31 aiagallery.dbif.ObjFlags	283
32 aiagallery.dbif.ObjLikes	285
33 aiagallery.dbif.ObjSearch	286
34 aiagallery.dbif.ObjTags	288
35 aiagallery.dbif.ObjVisitors	290
36 aiagallery.dbif.MSimData	292

Figures

1-1	The client/server environment	4
1-2	Skill sets required	6
1-3	Language use	7
2-1	Desired architecture	9
3-1	Transports	15
3-2	Adding a simulation transport	17

Listings

3.1	Class definition	29
3.2	Function called to send a request	36
3.3	Binding a method to an object	37
3.4	Post function called by simulation server, with response	39
3.5	Patching qooxdoo to use the simulation transport	41
3.6	Finding a handler for a request	42
3.7	Parsing the input text	48
3.8	Testing for a batch of requests	48
3.9	Testing for a single request	48
3.10	Mapping an array of requests to their corresponding responses	49
3.11	Ensuring the method name contains only valid characters	50
3.12	Ensuring the method name contains no double dots	50
3.13	Obtaining the service method from the service factory	50
3.14	Converting named- to positional parameters	51
3.15	Schedule or run the service function	52
3.16	Filtering out notification results	53
3.17	Returning a batch or single response	53
3.18	Constructor for the Abstract RPC Handler	54
3.19	Service factory	55
3.20	Service registration	57
3.21	Processing a JSON-RPC request arriving via a transport	58
3.22	Simulation remote procedure call handler constructor	58
3.23	Request to process an incoming message from the simulation server	59

3.24	Entity type registration function	63
3.25	A map specifying database properties and their types	64
3.26	A map specifying how to canonicalize the <i>value</i> field	65
3.27	How an entity type registers itself	66
3.28	How a database driver registers itself	68
3.29	Replacing data property names with their canonical peer	73
3.30	An entity class for dogs	75
3.31	Adding some dogs to the database	77
3.32	Query for all dogs in the database	77
3.33	Sort dogs by breed	78
3.34	Sort dogs by breed then age	78
3.35	Dogs at least three years old	78
3.36	Dogs at least three years old, no poodles or chihuahuas	79
3.37	App Engine search criteria processing: specified key	81
3.38	Default function to build a default key	82
3.39	App Engine search criteria processing: query	83
3.40	App Engine result criteria processing	84
3.41	Preparing and issuing a query to App Engine	85
3.42	Retrieving entity properties	86
3.43	Determining the key value	86
3.44	Automatically generating unique keys	87
3.45	Creating an App Engine key from a provided key	87
3.46	Building and storing an App Engine entity	88
3.47	Deleting an entity from App Engine	90
3.48	Initializing the environment for a query	93
3.49	Handling by-key queries	94
3.50	Dynamically building a search predicate function	95
3.51	Selecting and returning queried entities	100
3.52	Writing entity data to the simulated database	101
3.53	Deleting an entity from the simulated database	102

rpcjs/sim/remote/Transport.js	135
rpcjs/sim/remote/MRequest.js	141
rpcjs/sim/remote/MExchange.js	142
rpcjs/sim/remote/MRpc.js	145
rpcjs/sim/Simulator.js	147
rpcjs/rpc/Server.js	151
rpcjs/AbstractRpcHandler.js	161
rpcjs/sim/Rpc.js	166
rpcjs/appengine/Rpc.js	169
rpcjs/dbif/Entity.js	170
rpcjs/appengine/Dbif.js	182
rpcjs/sim/Dbif.js	194
aiagallery/dbif/Constants.js	204
aiagallery/dbif/DbifAppEngine.js	206
aiagallery/dbif/DbifSim.js	209
aiagallery/dbif/Decoder64.js	213
aiagallery/dbif/Entity.js	216
aiagallery/dbif/MDbifCommon.js	218
aiagallery/dbif/MApps.js	223
aiagallery/dbif/MComments.js	243
aiagallery/dbif/MFlags.js	250
aiagallery/dbif/MLiking.js	254
aiagallery/dbif/MMobile.js	256
aiagallery/dbif/MSearch.js	264
aiagallery/dbif/MTags.js	268
aiagallery/dbif/MVisitors.js	270
aiagallery/dbif/MWhoAmI.js	275
aiagallery/dbif/ObjAppData.js	277
aiagallery/dbif/ObjComments.js	280
aiagallery/dbif/ObjDownloads.js	282

aiagallery/dbif/ObjFlags.js	283
aiagallery/dbif/ObjLikes.js	285
aiagallery/dbif/ObjSearch.js	286
aiagallery/dbif/ObjTags.js	288
aiagallery/dbif/ObjVisitors.js	290
aiagallery/dbif/MSimData.js	292

Preface

The methods described in this thesis are immediately applicable to industry. I have therefore explicitly chosen to write in a style somewhat more conversational than is typical of dissertations. The collective “we” is often used to include me, the writer, you, the reader, and in many cases, also the code we are discussing. It is my hope that this thesis is very approachable and easy for web application designers and engineers of varying experience levels to read and understand. I do assume here a basic knowledge of web technologies and architecture, but try to provide web or publication references for all important topics with which the reader may be unfamiliar.

Introduction:

Defining the problem

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. Often, the server environment provides little debugging and testing infrastructure at all.

In this thesis, I propose an architecture, framework, and reference implementation 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 fully tested, the backend portion of the code can be moved to the production environment on the real server *with no changes*, where it operates without any additional changes or debugging.

Chapter 1

Typical development environment

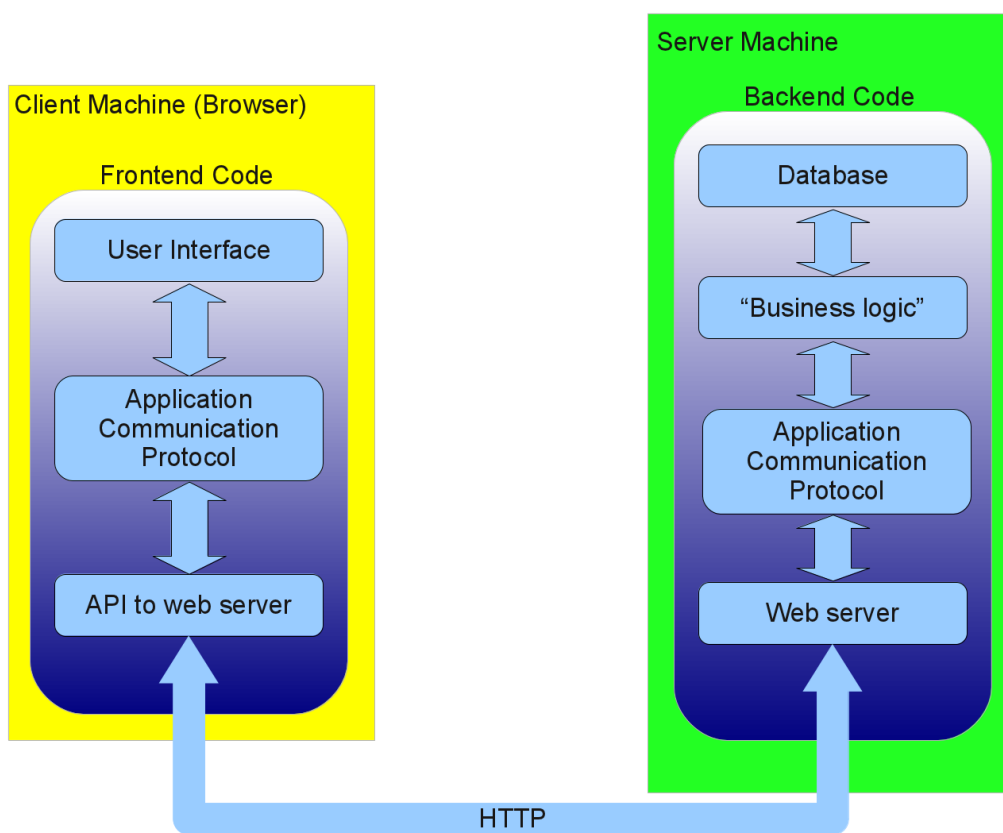


Figure 1-1: The client/server environment

A client/server web application is generally implemented in two distinct pieces. Frontend code, that runs in the browser, is typically written in HTML and JavaScript.

The other piece of the client/server web application, the backend code, runs on the server machine. It provides the **web server** component that serves the application to the client browser. It also often interacts with a database, and, via a communication channel between the frontend and backend, lets the frontend application issue requests to the backend that cause data to be stored in, or retrieved from, a database. Recent statistics [1] show that PHP and ASP.NET are most commonly used for writing the backend application.

Figure 1-1 shows the architecture of the environment that is in common use. The frontend is software running in the client (browser). It provides a user interface with which a person may interact. The server machine runs backend software known as a **web server** and additional backend software for manipulating a database. In order for the frontend application to interact with the backend, it encodes a request (probably using a library that implements some application communication protocol) and sends that data to the backend via some communication path often referred to as a **transport**, most typically HTTP.

We might think about annotating Figure 1-1 to depict the many skill sets required to implement such an architecture.

Figure 1-2 illustrates many of them. 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 [2], as shown in Figure 1-3(a). There need be at least a small amount of HTML to load the JavaScript code. Some 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. Users of Firefox often install the Firebug plug-in. The Chrome browser has a built-in debugger, as do recent versions of Internet Explorer.

The backend software includes the web server and database engine. Designing a

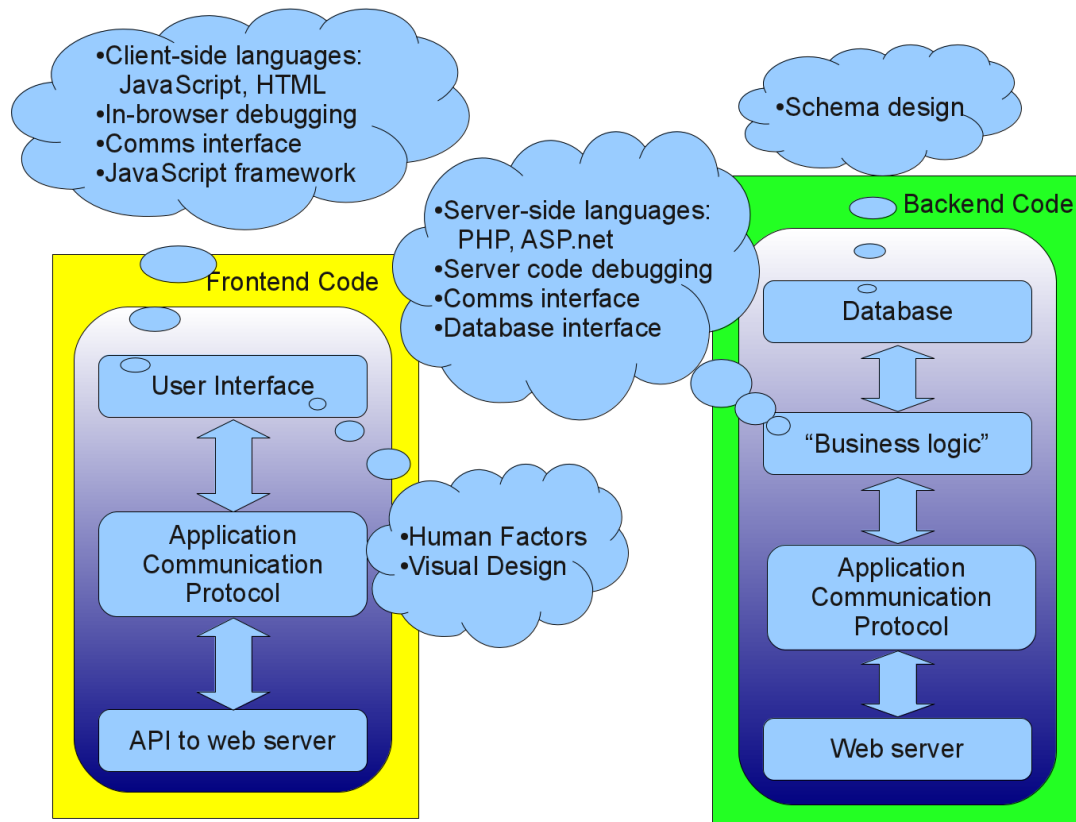


Figure 1-2: Skill sets required

good schema for a database is a specialized skill. As mentioned, PHP and ASP.NET are currently the most popular languages for writing the backend code [1], as shown in Figure 1-3(b). Each of these 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 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 environment comes unique debugging methodologies. The skill sets required for debugging at the client and server

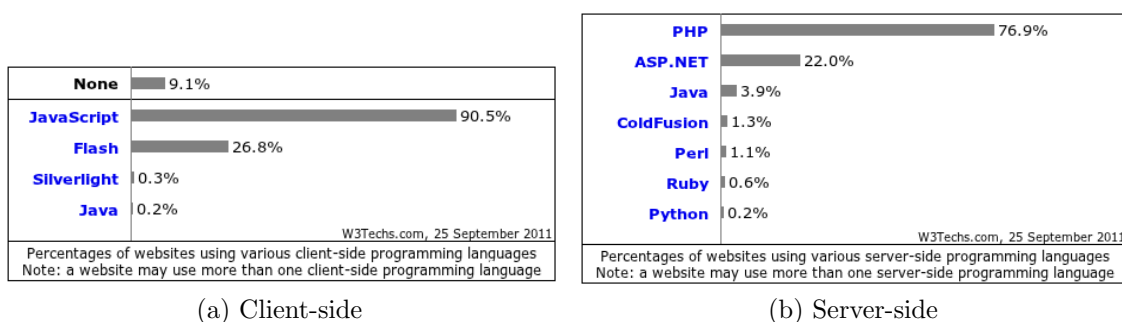


Figure 1-3: Language use

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.

Chapter 2

The research question

With the afore-mentioned problems in mind, I ask my primary research question:

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 entirely tested and debugged within the browser environment; and*
- 3. fully-tested application-specific backend code to be moved, entirely 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. There is little choice for the browser side. For cross-browser use, unless one wanted to require a plug-in, 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. That JavaScript backend code would need the ability to talk to whatever server-side database was to be used. The desired architecture, then, is depicted in Figure 2-1.

Additionally, we would need some form of abstraction that encompasses the set of database operations that are performed. We would need some mechanism that could

be mapped to a particular database on the server, and which could be mapped to some simulation of the database in the browser.

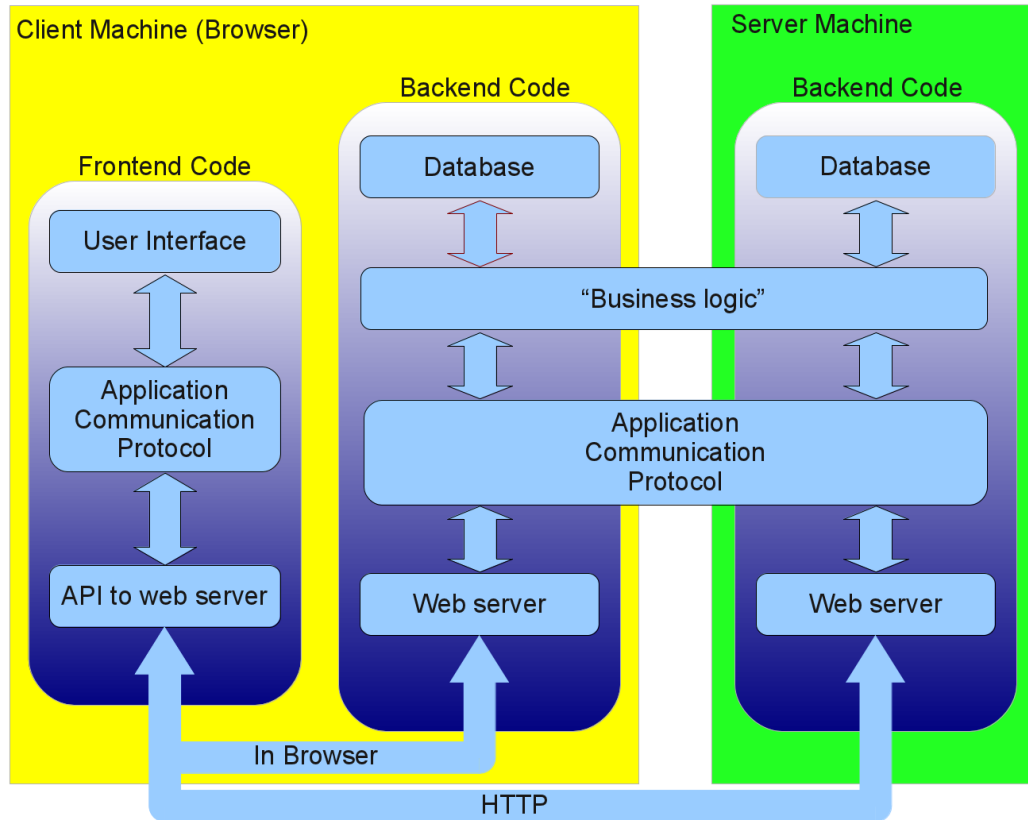


Figure 2-1: Desired architecture

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?*

Methodology:

Designing a reference implementation

I describe, here, the components necessary to achieve the desired architecture, in enough detail that one experienced in the art could implement the architecture and framework from scratch. I also discuss some specifics of my reference implementation, that brought this architecture into the realm of reality.

Few applications, these days, are written from scratch. Instead, they make use of existing libraries, modifications of code written previously, and frameworks that offer a common core applicable to numerous applications.

For my work, I chose to start with qooxdoo,¹ a JavaScript framework that provides a traditional class-based object programming model (as opposed to JavaScript's built-in prototype-based model)², and a wealth of additional functionality including classes to assist communicating over a network.

¹<http://qooxdoo.org>, (pronounced ['kuksdu:])

²Readers unfamiliar with prototype-based programming will likely find the Wikipedia article on the topic [3] to be educational.

Chapter 3

Introducing LIBERATED

To achieve the desired architecture and framework, a number of components are required:

1. Switchable transports. Recall that a **transport** is a communication path between frontend and backend software. We must, then:
 - add a local, **simulation** transport, to communicate with the in-browser backend,
 - and ensure that there is a means of selecting either the normal transport that routes to a real server, or the simulation transport.
2. Backend elements to run in the browser or on the real server. This requires:
 - server-side JavaScript,
 - an application communication protocol server,
 - and a database operation abstraction.
3. Glue code that is unique to each environment: running in the browser, or on a real server. This includes:
 - an interface from incoming requests to backend code, i.e., the arrows between the *Web server* and *Application Communication Protocol* blocks in Figure 2-1;
 - and an interface between the database abstraction and the simulated

in-browser, and real databases, as indicated by the arrows between the “*Business logic*” and *Database* blocks.

I have built a reference implementation to demonstrate the feasibility of the desired architecture. The reference implementation is called **LIBERATED (Lipman’s In-Browser EnviRonment for Application TEsting and Development)**. It contains all of the above components, that will now each be discussed in turn.

3.1 Switchable transports

A **transport** is a means of communicating data between two endpoints. In the web environment, the Hyper-Text Transfer Protocol (HTTP) is the typical transport between a frontend and a backend. In the context of web applications, transports refers not only by the protocol over which the communication is accomplished, but also the means by which that protocol is used. This includes one or more layers on top of a low-level protocol, that encodes a particular style of interactions, as discussed in Section 3.2.2.

3.1.1 Existing transports

The typical web browser provides facilities for sending requests to a server. Most commonly, in modern AJAX-based web designs,¹ HTTP requests are sent “in the background,” i.e., without refreshing or reloading the currently-displayed page, by issuing a JavaScript request to the method `XMLHttpRequest()`. This method provides a synchronous or asynchronous mechanism to issue background requests to the server from which the original web page was loaded. `XMLHttpRequest()`, therefore, is one possible **transport**.

`XMLHttpRequest()` is restricted, for security reasons, by the **same origin policy**, which allows the request to be issued only to the server from which the original web

¹AJAX is an acronym for Asynchronous JavaScript And XML. The term is typically used, however, to describe many forms of background requests and responses between the frontend application and its backend peer.

page was loaded.²

In qooxdoo, there are two other available transports: **script** and **iframe**. These are used when `XMLHttpRequest()` can not be used due to the **same origin policy**, or as described below, when a form submission is being simulated.

When a request must be made to a server other than the one from which the original web page was loaded (known as a **cross-domain request**), other options exist. A request may be encoded in a `<script>` tag, which may be sent to any server. The means with which the frontend will process the result must be agreed on with the backend, since the desired result is not actually a script to be run, but rather some data. A standardized means of handling responses, called JSONP (JavaScript Object Notation with Padding), is becoming popular.³ In JSONP, a function name (typically) is provided in the request, and the response will be a script that contains a call to the specified function, with the result data as a parameter to that function. The client browser is easily able to execute the returned script, which calls the function, which can process the result parameter.

A final request method to be mentioned here uses the `<iframe>` tag. In this mechanism, parameters to be passed to a server are placed in fields of a form inside of an `<iframe>`, and the form is submitted. Forms may be submitted only to the same origin from which the original page was loaded. As with the `<script>` method, there must be agreement between frontend and backend, on the exact format of the passed and returned data.

Figure 3-1 shows these three transports, `XMLHttpRequest`, `Script`, and `IFrame`, in context. An application decides which transport to use, either on an application-wide basis, or on a per-request basis. The message to be sent, encoded in an agreed-upon **Application Communication Format** is provided to the selected transport.

All three of the discussed transports are provided by qooxdoo, and each descends from an abstract class called `qx.io.remote.transport.Abstract`. The abstract class provides all of the functionality common to any transport, and each individual

²`XMLHttpRequest()` Level 2 adds the capability of communicating with servers in other domains, by use of special HTTP headers.

³<http://en.wikipedia.org/wiki/JSONP>

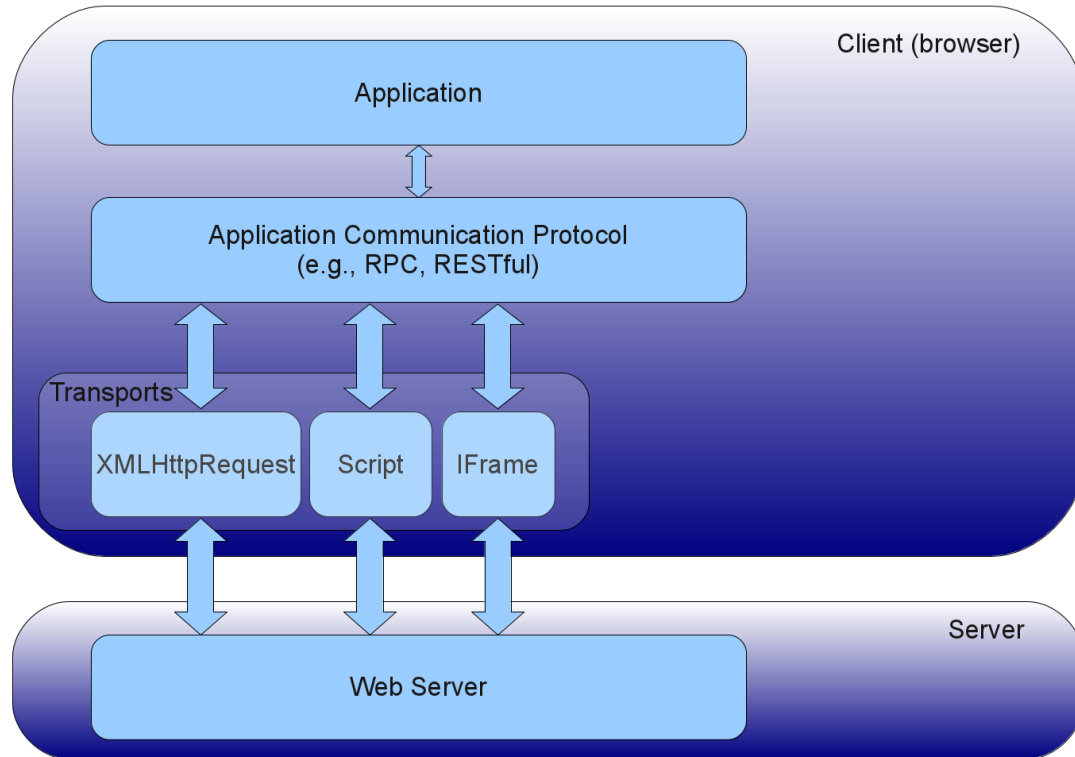


Figure 3-1: Transports

transport class extends the abstract class with the specifics need to implement the required functionality.

Developers of qooxdoo applications which make use of remote communication need not know the details of the various transports. Instead, the application specifies its **needs**, and an appropriate transport is selected automatically. Different transports handle different needs, so based on the user's requirements, the appropriate transport is selected. **Needs** include whether:

- a synchronous or asynchronous request should be issued,
- a cross-domain request is required
- a file upload must be accomplished
- the user has specifically requested **form fields**
- the desired response type is provided by the transport

Each transport handles various needs. As qooxdoo searches for a transport to support a particular request, it tries, in the default order, the XMLHttpRequest

transport, followed by the Script transport, and finally, the IFrame transport. The first transport that supports all of the specified **needs** is used.

The three transports handle the **needs** as follows:

	XMLHttpRequest	Script	IFrame
Synchronous	yes	no	no
Asynchronous	yes	yes	yes
Cross Domain	no	yes	no
File Upload	no	no	yes
Form Fields	no	no	yes
Response Types	text/plain, text/javascript, application/json, application/xml, text/html	text/plain, text/javascript, application/json	text/plain, text/javascript, application/json, application/xml, text/html

From the table, it is clear that if the user issued an asynchronous request for the same origin as the original web page (i.e., not cross-domain), that does not require a file upload nor form fields, and desired a **text/javascript** response, the request could be handled by any of the transports. On the other hand, if the user changed only one parameter, requiring a synchronous request, it could be handled only by the XMLHttpRequest transport. If the request were asynchronous but required cross-domain, it could be handled by the Script transport.⁴

3.1.2 Adding a simulation transport

In order to implement a simulated backend in the browser, the first task is to add a new transport whose function is to send requests to our simulated server in the browser, rather than to a real server running elsewhere. What we desire, then, is depicted in Figure 3-2.

⁴All requested needs must be met by a transport, for that transport to be selected. If no transport

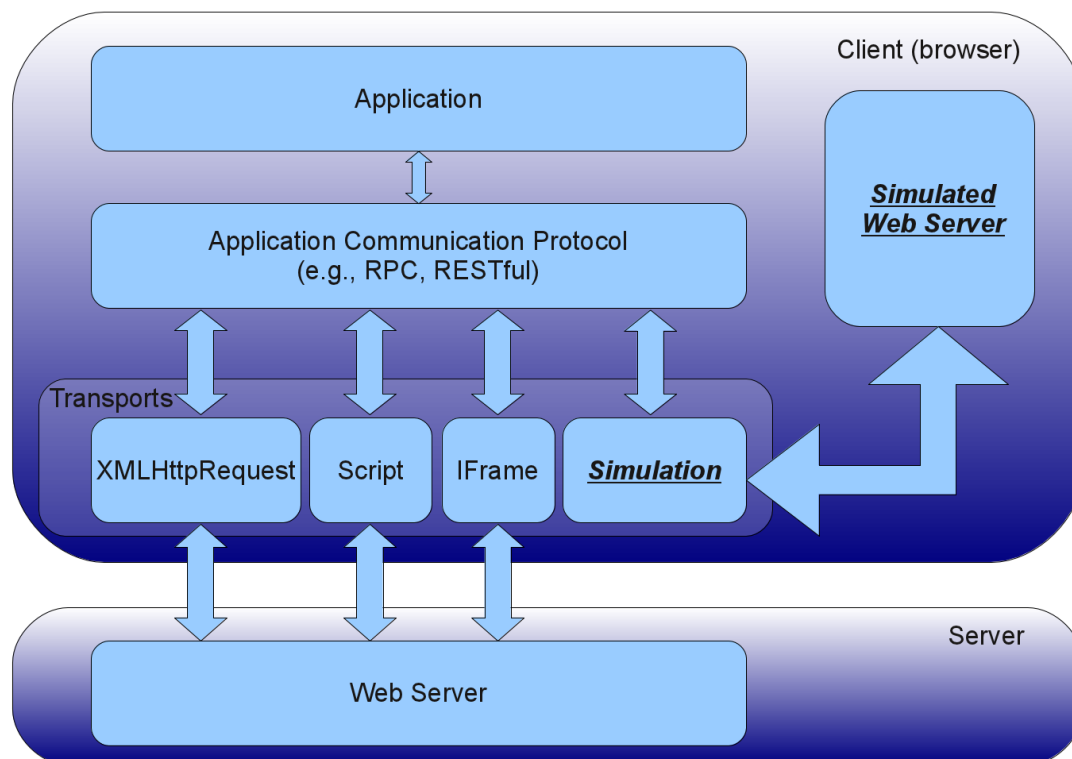


Figure 3-2: Adding a simulation transport

Here, we have added a new **Simulation** transport which communicates, within the browser, to some simulated web server. Implementing this additional transport within the qooxdoo environment involved creating a new transport class that extended `qx.io.remote.transport.Abstract`. I chose to implement a request queue, and let the requests be pulled from the queue by the simulated server as the server was ready for them. This allowed implementing both asynchronous server activity as would be seen with a real server, and, importantly, a synchronous facility that allows stepping easily from frontend code into backend code in a debugger.

The simulation transport supports all of the **needs** except for file upload. In order to switch between the simulation transport and one of the standard transports, a new need, **simulate**, was added. If the application specifies that **simulate** is needed, none of the other transports will be able to oblige, and thus the simulation transport will be selected. It is therefore very easy for an application to switch back and forth between transports.

meets all of the requested needs, an error is signaled.

The simulation transport collects all of the parameters of the request that might otherwise be placed in HTTP headers or elsewhere, and places them in a JavaScript object. This is the object that is enqueued for the simulation server to pick up. Among the important fields of the object, are:

url

The URL to which the request is to be issued

parameters

Parameters to the request. Were this an HTTP GET request, these parameters would be encoded in the URL.

formFields

Field data added programmatically to the request, as if they were fields in a form to be submitted. This is used to simulate a form submission.

data

Additional data to be submitted with the request. If this were an HTTP POST request, this data would be sent in the body of the request.

post

A function reference, i.e., a callback function. Defined within the simulated transport, this function will be called by the backend to post the response back to the frontend.

The object, containing these and other fields, is provided to the simulated server, which enqueues it for processing. Once the server has completed processing the request, it will call the transport's `post()` function with the response. The transport saves the data and sets its state to completed or failed, which signals the application. It then awaits the application to retrieve the response.

3.2 Required backend elements

The second major component required to enable the proposed architecture is a set of portable backend elements that run equally well on a real server, and simulated in the browser. This chapter discusses the issues involved in meeting this challenge.

3.2.1 Server-side JavaScript

In order to run a web server and database both in the browser and on the real server machine, we need a way to run server-side JavaScript. Server-side JavaScript has become increasingly popular in recent years, with three popular JavaScript engines that may be run outside of a browser: V8, SpiderMonkey, and Rhino.

The example application which makes use of **LIBERATED** uses Rhino's compiling capability, described below. This allows running the backend JavaScript code in Google App Engine's Java environment.⁵

3.2.1.1 V8

V8 [4] is Google's JavaScript engine, which is used in their Chrome browser. It is written in C++, and is designed to be embedded in other applications. V8 is the JavaScript engine used in the popular Node JavaScript environment [5]. When the V8 engine is to be used to provide server-side JavaScript, applications generally use Node. Most references to V8, therefore, are therefore indirect, via Node. That convention is generally followed here as well.

3.2.1.2 SpiderMonkey

The SpiderMonkey [6] JavaScript engine is used in a number of Mozilla products, including Firefox. SpiderMonkey is written in C and C++, and can be embedded into, and interact with, programs that compile and link in the normal C fashion. Although

⁵Google App Engine is discussed in the context of an example application built using **LIBERATED**, in Section 4.2.

older than Node, SpiderMonkey seems to be less prevalent at present. Node is said to run somewhat faster. [7]

3.2.1.3 Rhino

Rhino [8] is an implementation of JavaScript written in Java. It allows not only normal JavaScript to run, but provides access to the rich Java API, allowing JavaScript applications to interact with the numerous Java classes. Rhino was also developed by the Mozilla Foundation.

Rhino provides two distinct environments. In its more typical use, Rhino acts as a JavaScript interpreter, reading and executing source files. Rhino also has the ability to compile JavaScript source files into .class files, similarly to how the Java compiler compiles Java source files into .class files. This allows JavaScript to run in any environment in which compiled Java would run.

3.2.2 Means of application communication

Web application developers must select a means of communicating requests from the frontend to the backend, and for the backend to reply with responses to those requests. Older web sites often issued requests by submitting an HTML form to the server. A normal form submission uses the HTTP protocol, and the server replies with an entirely new web page for the browser to display. Developers of modern applications generally prefer sending requests and receiving responses in the background, and so typically choose a transport such as those described in Section 3.1.

Having selected a background transport, developers must also choose a data format and/or protocol in which to send requests to, and receive replies from, the server. The data format can be designed specifically for the application. There are, however, a few approaches which have come into general use.

3.2.2.1 Representational state transfer

Representational State Transfer (REST) is an architectural style that allows identification of **resources** by a **resource identifier**. It “is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and hypermedia as the engine of application state.” [9] A resource can be any concept (in the normal English sense of the word) one might want to offer information about. That concept can be depicted or represented in a number of ways. A web page, for example may be one “representation” of a resource (a concept). There are many resources and resource types that can be accessed via their individual resource identifiers, which often take the form of a URL. URLs “tell the browser that there’s a concept somewhere. A browser can then go ask for a specific representation of the concept. Specifically, the browser asks for the web page representation.” [10]

Resources need not be static. In the context of client/server web application, a resource can be, for example, some data in a database. A frontend, then, could request the current value stored in a database, by sending a request to the backend that includes the resource ID of that database entry. Resource IDs need not specify a single resource, however. They may also specify a collection of resources, for example, all records from a database which match a specific criterion.

3.2.2.2 Remote procedure calls

A remote procedure call (RPC) is a client/server mechanism that allows the frontend to issue a request to the backend in a manner similar to directly calling a local procedure or function. The concept of remote procedure calls was discussed as early as 1976 in RFC 707 [11], and its applications defined in great detail during the period 1981–1984 [12, 13]. Use of remote procedure calls involves the frontend calling what may appear to it as a normal (local) function call. Library or manually- or automatically-generated code then initiates a communication path from the frontend to the backend. A representation of the request, including the function to be called

and each of the parameters, is packaged into a message, and sent to the backend. In the early use of remote procedure calls, it was expected that the frontend application would block, as in normal, local procedure calls, awaiting the completion of, and returned result from the RPC. In current use, both synchronous and asynchronous use of remote procedure calls are popular.

Remote procedure calls are often used as a means of communicating between a frontend and backend in the web application arena. Two popular protocols have been standardized and are in common use: XML-RPC and JSON-RPC.

XML-RPC [14] has existed since 1999. As its name implies, XML-RPC encodes remote procedure calls as Extensible Markup Language (XML) [15]. The specification details a strict set of rules by which requests and responses are formatted. Parameters may be of various data types. Responses indicate success or failure, and can provide a single result value.

JavaScript Object Notation, JSON, is “a lightweight, text-based, language-independent data interchange format. It was derived from the ECMAScript Programming Language Standard. JSON defines a small set of formatting rules for the portable representation of structured data” [16]. JSON, then, can be used to represent data using an EcmaScript (JavaScript)-like syntax, which is very easy to both generate and parse, and is also easily human-readable.

JSON-RPC [17] is a specification for communicating remote procedure call requests and responses. Similarly to XML-RPC, JSON-RPC specifies the format in which specific remote methods are requested, how parameters are passed, and, to allow for multiple requests to be outstanding at one time, how responses can be matched to their associated request by means of a unique identifier per RPC request.

3.2.2.3 The REST/RPC battle

As a result of its long lifetime, RPC implementations are prevalent and popular. REST has gained substantial use as a means of communicating between a frontend and its backend. There are strong detractors of each [18, 19], with centrists explaining that there are appropriate times to use each [20]. Some have even gone so far as to say

that REST is actually an implementation of RPC [21], which has prompted responses to the contrary [22]. The argument will likely wage on. Clearly, there are multiple means of client/server communication.

Although REST is not limited to use over HTTP, it typically uses HTTP-style identification of request types. A GET request is expected to retrieve information; a POST request, to change information; a PUT request, to add a new piece of information; etc. A strict mapping of request type to expected result or purpose is therefore encouraged.

Remote procedure calls have the same expectations as local procedure calls: that the documented function of the called procedure is completed when the procedure returns. This allows somewhat more flexibility for side effects within a single request than does REST's strict mapping. For example, a remote procedure call to add a new item to an inventory might add a record to an Inventory table in a database. If this were the first item in a new category of items, the procedure might also add a new record to a Category table. Although not entirely disallowed by REST, that sort of secondary operation is discouraged there.

I selected JSON-RPC for **LIBERATED**'s communication between the frontend and backend.

3.2.3 Database models

In order to implement the backend in a simulated server that runs in the browser, we require a simulated database. It is unlikely that a true, full-blown database management system will be implemented in JavaScript to run in the browser, as the current technology available to the browser for reading and writing local files is still fairly primitive.⁶ That technology is, by definition, restrictive, to avoid web sites or downloaded scripts of evil intent from damaging local files. The simulated database, therefore, must be of a different design than any real server-side database. We need an abstraction on which application code operates, which can interface equally easily with our simulation database and various types of real, server-side databases. Let us

⁶See the discussion in Section 6.3.2 of Indexed Database API, however, for a viable possibility.

first examine the architectures of some common server-side database models. We'll follow that up, in Section 3.4.7, with a design for a simple simulation database, and discuss the abstraction that allows operating on any of them.

Over the years, a number of models have been defined, by which data may be stored and retrieved. Some are extremely simple. Others are quite sophisticated, and each provides benefits that others lack. There are many database models [23]. The characteristics of five common models will be discussed: flat, hierarchical, network, relational, and object.

3.2.3.1 Flat model

The flat model is so simple that one might not even think of it as a database model at all. In essence, the flat model is a two-dimensional table of information: a grid. Its component parts are rows, and cells within each row. A spreadsheet is a common example of a flat-model database. In many spreadsheet applications, data can be read from, or written to cells. Cells, or groups of cells in a column, can have data types assigned to them, including formatting of dates, or number of decimal places or whether to prepend a dollar sign for numeric values, etc. Rows, entities of related data, can be sorted based on the individual values in a column of cells.

3.2.3.2 Hierarchical model

The hierarchical database model organizes data as a tree structure. Each node, or entity, may have a series of children, and each entity knows its **parent** entity. This model is therefore very efficient for handling immediate parent/child relationships, but loses efficiency when an entity must reference an unrelated entity. A complete path from the root to the referenced entity must be stored.

The organization of XML and JSON data is representative of the hierarchical model, but few modern database management systems are built using this model.

3.2.3.2.1 Network model

The network model is similar to the hierarchical model, in that an entity is connected by branches to one or more other entities. In the network model, though, as opposed to the hierarchical model, there need not be an exclusive parent/child relationship between entities. Any entity may be pointed to by multiple other entities. This creates the network model's concept of sets, which define one-to-many relationships. An entity may be a member of one or more sets, and may be the **owner** of one or more sets.

3.2.3.3 Relational model

The relational model was developed by E.F. Codd in 1970 [24]. A collection of entities in the relational model is called a **table** and each entity or **record**, is a **row** within that table. All rows have a fixed set of **columns**.

Relational tables have several properties [23]:

- Values are atomic
- Each row is unique
- Column values are of the same kind
- The sequence of columns is insignificant
- The sequence of rows is insignificant
- Each column has a unique name.

A characteristic of the relational model is that one or more columns of a table may be defined to be **keys** such that no two rows in the table have the same key value(s). A composition of the values in the key columns is called the **primary key**. Each row is therefore uniquely identified by its primary key. A specific row of the same or another table can be referenced by its primary key, to create virtual pointers from one record to another (possibly in a different table).

With primary key values and virtual pointers used as references to specific other rows, it is possible to **join** records together. The concept of joining allows retrieving the related rows from various tables, along with requested rows, creating a result which is a composite of multiple database table rows.

Relational databases often use the SQL data definition language, with which tables may be manipulated and queried. Many currently-popular database management systems such as MySQL, PostgreSQL, and Microsoft SQL Server are SQL-based relational databases.⁷

3.2.3.4 Object model

Modern programming languages are based on an **object-oriented** paradigm, in which objects of a certain type are instantiated, and each instance is provided with the set of member variables for the class. The concept of inheritance allows a new type to inherit the characteristics of another type.

The object database model can help alleviate the problem of mapping the objects within a programming language to the table structure of the relational database model. Known as the **object-relational impedance mismatch**, the problem is that object oriented languages and the relational paradigm are inherently different, the object-oriented paradigm “based on proven software engineering principles,” and the relational paradigm “based on proven mathematical principles. . . With the object paradigm you traverse objects via their relationships whereas with the relational paradigm you join the data rows of tables.” [25]

3.3 Glue code

Most of the code in **LIBERATED** is common to all backends. Whether the backend is on a real server, or is running in the browser, the identical code is used.

⁷Apparently these systems deviate somewhat from Codd’s original principles — deviations against which Codd argued fiercely. These arguments are said to be in his book, *The Relational Model for Database Management*, Addison-Wesley Publishing Company, 1990. I was unable to obtain this book for reference.

There are some portions of code which are unique to an environment, however. This section describes, in general terms, those differences. The specific implementations will be discussed in Section 3.4.

3.3.1 Frontend/backend communication

The transport used for communication between the frontend and a backend running on a real server must certainly be different than one used for communicating with a backend running in the browser. In the former case, most likely, HTTP requests and replies are being sent across a physical network. The backend must retrieve these requests and send the replies using some mechanism provided by the operating system or a network access library. In the latter case, on the other hand, requests are being sent to backend code running locally in the browser, so instead of issuing HTTP requests, messages can be simply added to a queue. The queue could be processed via a timer for asynchronous requests, or immediately, for synchronous requests. Responses can be returned via a callback function.

3.3.2 Mapping the database abstraction to a database

Similarly, a real database on a real server machine must be manipulated differently than a simulation of a database running in the browser. The real database is manipulated by issuing SQL commands via a defined interface, or by interacting with the database-specific API. Each database with which **LIBERATED** is to operate will need a unique database interface driver to handle those details. Similarly, a database interface driver is required to handle the simulation database which runs in the browser.

3.4 **LIBERATED** implementation details

In this section, we will discuss **LIBERATED**, my reference implementation of code that allows running a JSON-RPC and database server in a simulation mode,

within the browser environment, and running essentially the same code on Google App Engine.⁸ As we will see, only a few files differ by the specific backend. We will look first at the generic code library which may be used by an application. This code need not be modified on a per-application basis.

This implementation is developed using the qooxdoo JavaScript framework. Although qooxdoo, with its rich widget set, is most frequently used for frontend (user interface) functionality, its mechanism for writing object-oriented code that is well organized, documented, and easy to read, makes it ideal for backend functionality as well. Since the topic of this thesis is about debugging and testing frontend and backend functionality within a browser, qooxdoo is the ideal framework to base the totality of this work on. There is a qooxdoo-provided library which is intended specifically for this purpose, allowing easy use of the qooxdoo object model for non-GUI applications.

We will begin with a brief introduction to the object-oriented features of qooxdoo, and its remote procedure call interface. From there we will delve into the implementation of the transport simulator and the ancillary mixins required to modify standard qooxdoo transport and RPC features to add our new, simulation transport and in-browser remote procedure call server.⁹ We'll then move up to the implementation of the JSON-RPC server as well as the little bit of RPC server code that differs depending on whether we are in the simulated environment or on a real server. Having understood how remote procedures get called, we can then look at the abstraction for database manipulation.

3.4.1 The qooxdoo JavaScript framework

As described at the qooxdoo web site, qooxdoo “is a universal JavaScript framework that enables you to create applications for a wide range of platforms. With its object-oriented programming model you build rich, interactive applications (RIAs), native-like apps for mobile devices, light-weight traditional web applications or even applications to run outside the browser.” It is “entirely class-based and tries to leverage the

⁸Only the start-up and glue code differs. All application-specific code is identical.

⁹Section 3.4.1.5 provides an introduction to mixins and how they are used in **LIBERATED**.

features of object-oriented JavaScript. It is fully based on namespaces and does not extend native JavaScript types to allow for easy integration with other libraries and existing user code.” [26]

In order for the reader to more easily understand the **LIBERATED** details to be discussed, I will begin with an introduction to some features of qooxdoo that are used pervasively throughout, or required by, **LIBERATED**.

3.4.1.1 Object orientation

Common languages such as Java and C++ provide what is known as **classical inheritance**. A **class** is declared, providing a constructor, and defining the class’ static and/or member variables. The class can then be **instantiated** to create an object of that class. Furthermore, another new class can be declared, which **extends** the first class, thereby providing all of the functionality and features of the first class, with specified modifications. This has become a very familiar inheritance model for computer programmers.

JavaScript natively provides a different model called **prototypal inheritance**. Prototypal inheritance means that objects inherit properties from other objects, not from classes. In a pure prototypal environment, there are no classes. “Prototypal inheritance is powerfully expressive, but is not widely understood,” [27] writes Douglas Crockford, one of the gurus of JavaScript. “Few classical programmers found prototypal inheritance to be acceptable.”¹⁰ JavaScript’s prototypal model is sufficiently powerful that it can map (with a few limitations) to the classical style. Let us then look at how a class is defined, using qooxdoo syntax.

Listing 3.1: Class definition

```

1 qx.Class.define("rpcjs.sim.Rpc",           // class being defined
2 {
3     type      : "singleton",             // special class types
4     extend   : rpcjs.AbstractRpcHandler, // superclass

```

¹⁰To be fair, this latter quote is taken out of context. Crockford is actually describing how “classically inspired syntax obscures JavaScript’s true prototypal nature.” The quote rings true, however, and classically-trained programmers find an environment which provides classical inheritance easier to grasp, initially. qooxdoo’s classical use of JavaScript’s inheritance model, being described here, thus makes sense.

```

5   implement  : [ ... ],           // interfaces
6   include    : [ ... ],           // mixins
7   construct  : function() { ... }, // constructor function
8   properties : { ... },           // object properties
9   statics    : { ... },           // static members
10  members    : { ... },           // instance members
11  events     : { ... },           // dispatched events
12  destruct   : function() { ... }, // destructor function
13  defer      : function() { ... } // "after load" function
14 });

```

Each class has a fully-qualified name which consists of the namespace in which the class resides, and the name of the class itself. A namespace can be thought of (and in fact, is implemented as) a directory path to get to the class. Instead of using the traditional slash as the separator for each component of the path, a period (“.”) is used. This is the same mechanism as is used by Java.

Listing 3.1 shows use of `define()`, a static method of the `Class` class in the `qx` namespace. `qx.Class.define()` takes two parameters to define a new class: the name of the class to be defined, and a map describing the configuration characteristics of the class being defined. None of the configuration members are required, although a class which specified none of them would have no functionality.

The fully-qualified name of the class defined in Listing 3.1 is `rpcjs.sim.Rpc`.

Each of the configuration map keys shown in the listing will be explained here. This is not a complete list of available keys, but includes all that are used within **LIBERATED**.¹¹

3.4.1.2 type

Normal class definitions need not specify a `type`. Special types of classes, however, can indicate their type, which allows qooxdoo to provide extra capabilities to deal with the class. The first of these special types is for an **abstract** class, i.e., one that is never expected to be instantiated on its own, but rather is expected to be referenced as the superclass of some set of other classes which complete the functionality provided by the abstract class. Abstract classes specify a value of “abstract” for `type`.

¹¹The complete list is keys is documented at <http://demo.qooxdoo.org/current/apiviewer/#qx.Class> (click on the “+” to expand the documentation for the `define()` method) and are further explained in the qooxdoo manual, at http://manual.qooxdoo.org/1.5/pages/core/oo_feature_summary.html

A second commonly-used special type is **singleton**. A singleton is a class which is designated to have only a single instance. When a class is defined of type **singleton**, qooxdoo automatically generates a `getInstance()` method in the class, and prevents the class from being instantiated in the normal fashion with the `new` keyword.

A final special value of type is **static** which tells qooxdoo that the class is never intended to be instantiated, i.e., that it contains no member variables or functions. qooxdoo is thus able to flag errors if any member variables or functions are defined within the class. The **static** type value is rarely explicitly specified. A properly-written static class operates identically whether or not the `type` is specified as **static**.

When no type is specified, a normal, non-static class is defined.

3.4.1.3 **extend**

Most classes are defined to subclass, or **extend**, some existing class. In the qooxdoo environment, in order to get full benefit of qooxdoo capabilities, all classes descend, ultimately, from `qx.core.Object`, but the subclass chain may be arbitrarily long.

3.4.1.4 **implement**

An **interface** is a definition of features that must be implemented by any class that claims to **implement** that interface. Interfaces are additionally used in qooxdoo to provide development-time testing of such things as the types of passed parameters and other run-time, call-time checking that need not be included in a deployed product.

3.4.1.5 **include**

Any single qooxdoo-defined class must be defined in a single source file, as is required of Java source files. Splitting a large class into multiple smaller pieces can help keep the code organized, however. qooxdoo provides for the definition of **mixins** which can be included into a class. By defining a series of mixins as the afore-mentioned smaller pieces of a large class, the class definition can **include** the mixins, thus creating the composite, larger class, while keeping the source code in

separate, meaningful pieces.¹²

3.4.1.6 **construct**

Each new class may have a constructor which initializes a new instance of the class, e.g. by setting initial values of member variables, or setting property values based on parameters passed to the constructor. The constructor is a function specified by the **construct** key to the configuration map. A constructor need not be specified. If one is not specified, the superclass' constructor will be called automatically. If a constructor is specified, in qooxdoo's inheritance model, superclass constructors are not called automatically. If the new subclass desires that the superclass' constructor be called (which is required in the vast majority of cases), the subclass constructor must explicitly call the superclass constructor. This is done with the following incantation:

```
    this.base(arguments);
```

If parameters are to be passed to the superclass constructor, they follow the word "arguments":¹³

```
    this.base(arguments, "hello world", 42);
```

3.4.1.7 **properties**

One of the wonderful capabilities provided by the qooxdoo object system is its property system. Getter and setter functions are created automatically for **properties**. Through the definition of properties, the developer can specify a property's initial value and whether the property may take on the null value. If the developer declares a specific type for property values, e.g., "String," "Date" or "Number") the property

¹²Mixins may also be included into a class externally, i.e., via a function call issued after a class exists rather than via the **include** key in the configuration map while defining a class. This allows the capability for a developer to alter the behavior of a pre-existing class by mixing in additional functionality to that pre-existing class. We will see the use of that means of invoking mixins, in Section 3.4.3.

¹³**arguments** is a keyword in the JavaScript language. It is an array-like object that contains the list of parameters to the current function, a reference to the current function itself, and some other special characteristics that the **base()** method is able to make use of to efficiently locate and call the superclass constructor.

system will automatically check that values provided to the setter function meet that requirement. Arbitrarily-sophisticated checks can be accomplished by providing a function to check the values passed to the setter. The developer may dictate that a specified function be called whenever the value is being changed, and properties may even be defined to have events fired automatically upon value change. There are many other useful features of the property system. The intrigued reader is encouraged to examine the qooxdoo API documentation to discover the plethora of capabilities.¹⁴

3.4.1.8 statics

A class may have both instance member variables and functions, and variables and functions which are per-class. The `statics` key is a map which contains those variables and functions which exist on a class-wide basis. These variables and functions are referenced via their fully-qualified name, i.e., including the name space.

3.4.1.9 members

Variables and functions which vary per instance of the class, i.e., are unique to each object which is instantiated from the class, are called member variables and functions, and are defined in the `members` map.

3.4.1.10 events

The `events` map declares any events which are to be fired by this class. An event has a name and a type. All of the pre-defined event types are defined in classes within the `qx.event.type` namespace. The two most common are `qx.event.type.Event`, which is used when only an indication that something has occurred is required, but no data specific to the event must be provided; and `qx.event.type.Data`, in which some data specific to the event is passed along with the event.

¹⁴<http://demo.qooxdoo.org/current/apiviewer/#qx.core.Property>

3.4.1.11 `destruct`

JavaScript does a pretty good job of garbage collecting, freeing memory that is no longer in use. It is important for the application to ensure that there are no pending references to any objects, to allow garbage collection to occur, however.

When an object has saved references to other objects, it can help the garbage collector by cleaning up (removing) its saved references. This is particularly important when there are references to objects in the browser's DOM. The `destruct()` method must remove such references, typically by setting the member variables containing those references to null.

3.4.1.12 `defer`

It is sometimes necessary to execute some initialization code that requires that a class be fully loaded beforehand. For this purpose, the `defer` function can be used. This function will be called immediately after its containing class has been fully loaded (defined).

3.4.2 Interface for remote communication

The qooxdoo communication infrastructure uses a set of inter-related high-level classes, plus a series of low-level classes which implement the particular transports that are available, to send and receive data.¹⁵ Here, we will discuss the overall architecture of the qooxdoo remote communication infrastructure. Note that we are referring to the means in which a *frontend* application issues requests and receives responses using the qooxdoo framework. How these requests and responses are handled by the backend in the simulated server is treated later.

When an application wishes to send data to a server, it instantiates an object of class `qx.io.remote.Request`. The URL to which the request is to be sent is specified, as is the HTTP method (on the assumption that an HTTP transport may be used),

¹⁵There is a new communication infrastructure in development, in the `qx.io.request` namespace. **LIBERATED** was developed before the new infrastructure was sufficiently far along, so continues to use the original infrastructure.

and the expected response type. Various other property values (**needs**, as described earlier) can be specified, such as whether a cross-domain request is to be issued, if file upload capability is required, if data is to be sent in the request, username and password (possibly for HTTP Basic authentication) are needed, etc.

Once all of the information about the request has been specified, the `send()` method of the `Request` object is called. This places the request object onto a queue in the `qx.io.remote.RequestQueue` class, associates the request with an instance of `qx.io.remote.Exchange`, and calls the `send()` method of the `Exchange` object. From that point on, the `RequestQueue`'s job is to watch for timeouts on requests. The job of the `Exchange` object is to determine the appropriate transport to use, based on the **needs** that have been specified and the capabilities that each transport can handle (see Section 3.1.1), and to use the selected transport to send the request data.

When a response is received via the selected transport, the `Exchange` object arranges to issue an event of type `qx.io.remote.Response`, containing the status, headers, and content that were received.

Noted for later discussion is the mechanism used for selecting a transport. The `Exchange` class has an array of names of transports, `typesOrder`, which specifies in what order the transports are tested to determine if the capabilities they handle meet the specified needs. The `send()` method uses this array, in conjunction with the `Request` object's **needs** properties (e.g., asynchronous, synchronous, cross-domain, etc.), to loop through each of the transports in search of one that will handle the needs. The first transport that is discovered that handles the specified needs is selected for use.

3.4.3 Transport simulator

Recall the desired architecture depicted in Figure 2-1: the goal is to allow selecting a transport that routes, via HTTP, to some real web server someplace... or, selecting one that routes to a simulated server running in the browser. Our first step, then, in **LIBERATED**, is to create a transport that routes to an in-browser simulated server.

The simulation transport implementation is provided in full in Appendix 1. We

will discuss specific portions of it, and when necessary, include snippets of code.

The simulation transport implementation will handle the following capabilities:

- simulate (this is a new capability implemented by this transport)
- synchronous
- asynchronous
- cross-domain
- form fields

We will support the full range of standard response types:

- text/plain
- text/javascript
- application/json
- application/xml
- text/html

With this set of **needs** handled and **response types** available, this transport will meet any normal requirements. It will therefore be selected whenever the **need** for **simulate** is specified. We will see shortly how that need is added to the search list.

The `send()` method is of particular interest. Recall from Section 3.4.2 that after the instance of `qx.io.remote.Exchange` has identified the appropriate transport to use, it calls that transport's `send()` method. Let's look at the implementation of this method in the simulation transport. Refer to Listing 3.2.

Listing 3.2: Function called to send a request

```

80     send : function()
81     {
82         // The request data, as retrieved from the various properties
83         this.__request =
84         {
85             url           : this.getUrl(),
86             method        : this.getMethod(),
87             asynchronous   : this.getAsynchronous(),
88             username      : this.getUsername(),
89             password       : this.getPassword(),
90             parameters    : this.getParameters(),
91             formFields    : this.getFormFields(),
92             requestHeaders : this.getRequestHeaders(),
93             data           : this.getData(),

```

```

94
95     // Function to post the response. The signature is:
96     //   post(responseData, responseHeaders);
97     post      : qx.lang.Function.bind(this.post, this)
98   };
99
100   // Initialize responses data and headers
101   this.__responseData = null;
102   this.__responseHeaders = {};
103
104   // Simulate initial states
105   this.setState("created");
106   this.setState("configured");
107
108   // Post this request to the simulator's request queue
109   rpcjs.sim.Simulator.post(this.__request);
110
111   // Simulate sending state, now that we've "initiated" sending to our
112   // peer. (In reality, it's already arrived.)
113   this.setState("sending");
114 },

```

Lines 83–98 create a map containing all of the information to be passed to the simulation server. Each of the member values has been saved in the transport by the `Exchange` instance. In a “real” transport, these values would be placed into headers of a request to the server, or in the case of the `url` field, used to specify which server was to be used. In this simulation transport case, we simply want to package all of the values and pass them on to the simulation server.

In order to allow for both synchronous and asynchronous requests, a callback mechanism is used. When the server has completed processing a request, it will call the function provided in the `post` member of the request map. This member is set on line 97. The function that should be called is the `post()` method of the simulation transport, i.e., `this.post`. The server, however, has no reference to the simulation transport object, so would not be able to call the `post()` method in the context of the simulation transport. In other words, when the method was called, `this` would not be the simulation transport object. To ensure that the `post()` method knows its context when it is called by the simulation server, the method is initially **bound** to the transport simulation. The function `qx.lang.Function.bind()` has additional capabilities, but effectively implements the small piece of code shown in Listing 3.3.

Listing 3.3: Binding a method to an object

```

1 function bind(func, obj)
2 {

```

```
3   return function()
4   {
5     return func.apply(obj, arguments);
6   }
7 }
```

The `bind()` method creates a function which returns the results of calling the argument function with the argument object as its context. This works via **closure**, meaning that the values of `func` and `obj` that are passed as arguments to `bind()`, retain their values in the returned function. Therefore, when the returned function is later called, it will call the bound function, provided by the parameter `func`, in the context of the parameter object `obj`. The `arguments` keyword is used in the call to `func` so that any arguments passed to the function returned by `bind()` are included in the argument list to `func`.

Returning to the `send()` function in Listing 3.2, lines 101–102 initialize the response fields. These will be filled in by the `post()` method, which receives, as is documented on line 96, the response data and any simulated response headers.

As a transport proceeds through a full request lifetime, it flows through a series of states:

- created
- configured
- sending
- receiving
- completed

That flow of states can be interrupted at any time by an error, time limit exceeded, or user action. Any time after entering the *created* state, then, it is possible that instead of proceeding through the normal flow, one of these states is entered:

- aborted
- timeout
- failed

Any of the states, *completed*, *aborted*, *timeout*, or *failed* indicates the termination of the request. Upon state *completed*, termination was successful. In any other case, the request terminated because of an error.

Changes of states cause events to be fired, for any registered listeners.

A “real” transport would be first be created, i.e., it would enter the state *created*, it would then be *configured*, and finally, the request would be sent, so the state would go to *sending*. In the case of the simulation transport, creation and configuring are the same thing, so lines 105–106 simulate these by setting the state to *created* and then *configured*. (Events are fired for each state change, so setting the state to *created* just before resetting it to *configured* is not superfluous.)

At this point, the request is ready to send, which for this simulation transport, means posting it to the simulated server, `rpcjs.sim.Simulator` by calling the Simulator’s `post()` method at line 109. The single parameter is the request object that has been built. The transport’s state is then set to *sending* even though the entire send operation has already been accomplished. The next state that we enter will be *receiving*, and that has not yet begun, so *sending* is the most appropriate simulated state.

The `post()` method is shown in Listing 3.4. As described above, the values from the two parameters, which provide the response data and response headers, are saved into their respective private variables. Portions of these values can be retrieved by the user of this class later, with the getter methods, `getStatusCode()`, `getStatusText()`, `getResponseText()`, and `getResponseXml()`.

Since the `post()` method has been called, we are now *receiving*, so the state is set to indicate such.

Listing 3.4: Post function called by simulation server, with response

```

127     post : function(responseData, responseHeaders)
128     {
129         this.__responseData = responseData;
130         this.__responseHeaders = responseHeaders;
131
132         // The transport of the request and response is complete
133         this.setState("receiving");
134
135         // Was this a successful result?
136         if (responseHeaders.status == 200)

```

```

137     {
138         this.setState("completed");
139     }
140     else
141     {
142         this.setState("failed");
143     }
144 },

```

The simulated server uses an HTTP-style header value to communicate completion status back to the simulation transport. If the response shows a `status` value of **200**, the `success` code for the HTTP protocol, we are being told that the request completed successfully. Any other code indicates an error of some sort has occurred, so the state is set to `failed`. The response headers also include a `statusText` field which provides a textual description of any error that occurred.

Attaching the simulated transport

So far, we have discussed the simulated transport in operation, on the assumption that it had somehow been attached into the qooxdoo communication infrastructure. It is now time to see how that is accomplished.

The transport simulator handles a `need` called `simulate`. This need is not included in the array `typesOrder` that we discussed previously. Nor does the `Exchange` class' `send()` method know to look at this new need to ascertain whether the simulation transport is a better pick than one of the standard transports. In order to correct these issues for the simulated transport, mixins are provided to modify the behavior of the `Request` and `Exchange` classes.

The mixin `rpcjs.sim.remote.MRequest` simply adds a new property to the class to which it is included. The new property is `simulate` which is defined to have an initial value of `false`. The source to this mixin is provided in Appendix 2.

The mixin `rpcjs.sim.remote.MExchange` is somewhat longer, but adds only a little new functionality. What it really wants to do is modify the `Exchange` class' `send()` method, to check whether the `simulate` need was specified, and if so, ensure that a transport that handles `simulate` is selected. Unfortunately, there is no way to modify a small portion of a method, and the `send()` method is not written in such a

way as to easily allow adding a new need, so this mixin is used to patch `Exchange` to entirely replace the original `send()` method with the one in this mixin. The source is shown in Appendix 3.

These mixins are included by the defer section of `rpcjs.sim.remote.Transport`, shown in Listing 3.5.

Listing 3.5: Patching qooodoo to use the simulation transport

```

348 defer : function()
349 {
350     // Patch qx.io.remote.Exchange with our own send() method that
351     // supports looking at the "need" for a simulated transport.
352     qx.Class.patch(qx.io.remote.Exchange, rpcjs.sim.remote.MExchange);
353
354     // Similarly, for qx.io.remote.Request, except it can be a simple
355     // include since it's only adding a property.
356     qx.Class.include(qx.io.remote.Request, rpcjs.sim.remote.MRequest);
357
358     // Patch qx.io.remote.Rpc with our own createRequest() method that
359     // supports setting the simulate property of the request.
360     qx.Class.patch(qx.io.remote.Rpc, rpcjs.sim.remote.MRpc);
361
362     // Register ourselves with qx.io.remote.Exchange
363     qx.io.remote.Exchange.registerType(rpcjs.sim.remote.Transport,
364                                     "rpcjs.sim.remote.Transport");
365
366     // Add ourselves as the last tried transport
367     qx.io.remote.Exchange.typesOrder.push("rpcjs.sim.remote.Transport");
368 }
369 });

```

Mixins are attached to existing classes using one of the methods `qx.Class.include()` or `qx.Class.patch()`. The difference is that the former is considered “safe,” i.e., it only allows new functionality; it does not allow altering any existing properties or methods. The latter is intended only for those developers with a deep understanding of the code being patched, and does not prevent one from shooting himself in the foot. Existing methods can be replaced at will.

Line 352 applies the `MExchange` mixin just described, to the `Exchange` class. Recall that this mixin overwrites, entirely, the `send()` method, and thus it must use `qx.Class.patch()` to apply the mixin.

The mixin that adds the *simulate* property to the `Request` class is not altering existing behavior — it is adding a new property — so `qx.Class.include()` can be used to apply this mixin.

Next, on line 360, the remote procedure request class, `qx.io.remote.Rpc`, the class

with which applications issue remote procedure calls, is patched to make use of the Transport's new *simulate* property. The `createRequest()` method which normally just instantiates a `Request` object, is overridden to set the *simulate* property of the `Request` object, if a static boolean flag is true.

Finally, lines 362–268 register the new simulation transport with the `Exchange` class, and add the new simulation transport onto the list of transports, as the very last one to be selected. Only if no other transport handles all of the specified needs will the simulation transport be selected (assuming that it handles the specified needs).

3.4.4 Simulation web server

The simulation web server operates by allowing other classes to register a handler with it, and allowing each of those handlers to decide whether it handles a given request. The handlers are called in the order in which they are registered, and may use any of the values in a `request` such as URL, method, etc., to determine whether it will process the request.

The simulation web server is shown in Appendix 5. The loop which locates a handler for a request is depicted in Listing 3.6.

Listing 3.6: Finding a handler for a request

```

134     for (i = 0; i < Simulator.__handlers.length; i++)
135     {
136         // Assume that the handler will not be able to process the request
137         responseHeaders =
138             {
139                 status      : 501,
140                 statusText : "Not supported by this handler"
141             };
142         // Call the handler's processRequest function.
143         response = Simulator.__handlers[i](request, responseHeaders);
144
145         // See what the handler did with this request.
146         // status=200 means it handled the request successfully
147         // status=501 means we need to try another handler;
148         // else: handled the request but an error occurred
149         if (responseHeaders.status == 200)
150         {
151             // Yes it did! Restore the transport's post method to
152             // the request object and send the response.
153             request.post = transportPost;
154             request.post(response, responseHeaders);
155             break;
156         }
157         else if (responseHeaders.status == 501)
158         {

```

```

159         // Try the next handler. Nothing to do here.
160     }
161     else
162     {
163         //
164         // The request got handled, but was not successful.
165         //
166
167         // Restore the transport's post method to the request
168         // object and send the error response.
169         request.post = transportPost;
170         request.post(null, responseHeaders);
171         break;
172     }
173 }

```

All handlers that registered themselves with this simulated web server were placed in the `_handlers` list, a static variable of this `rpcjs.sim.Simulator` class. At line 115 (not shown), the variable `Simulator` was initialized to `rpcjs.sim.Simulator`, so acts as a shortcut to the class.

The loop depicted in lines 134–173 shows how the appropriate handler is selected. A handler is expected to return, in the `responseHeaders` map, a status value of **501** if it does not support a request. A status value of **200** indicates that the handler successfully processed the request. Any value other than **501** and **200** indicates that the handler handled the response, but was somehow unsuccessful. In lines 137–141, the response headers are initialized with the **501** status, so that any handler which is tried but can not handle the request can simply return, and the **not handled** status is pre-set. This will cause the code at lines 157–160 to be executed (i.e., do nothing), and, with no further code to execute before the end of the loop, the loop will increment to the next possible handler.

When the handler does choose to process the request passed at line 143, before returning, the handler is required to set the `status` and `statusText` fields of the `responseHeaders` map. Upon return, if the status was set to **200**, line 154 will be executed. This calls the simulation `Transport`'s `post()` method that was discussed in Section 3.4.3. The response that was returned by the handler, and the response headers, are passed as parameters to the `post()` method.

If the handler indicates that the request was not successful, by setting a status value other than either **501** or **200**, line 170 is executed. No response is available, so

`null` is passed in its stead. The response headers are passed as the second parameter, as they were in the success case.

At present, there is only a single handler registered in **LIBERATED**: the remote procedure call server.

3.4.5 Portable JSON-RPC server

Up until this point, we've been discussing code that is used only by the server simulator that runs in the browser. Now, however, we begin to see code that can (and does) run both in the simulation environment, and on a real server. We now turn our attention to the portable JSON-RPC server.

3.4.5.1 The JSON-RPC version 2.0 protocol

The current version of the JSON-RPC protocol is 2.0, and the protocol is defined in a specification at [17]. JSON-RPC is a fairly simple protocol.

Request

Requests have four fields: `jsonrpc`, `id`, `method`, and `params`. A brief description of the fields follows.¹⁶ In the context of JSON-RPC, there is assumed to be a JSON-RPC *client* issuing requests to a JSON-RPC *server*. In a web client/server application, the JSON-RPC client is typically in the frontend, and the JSON-RPC server is typically in the backend.

`jsonrpc`

The string “2.0” which indicates the version of the JSON-RPC protocol.

`id`

An identifier provided by the client, and returned to the client by the server in the response to the request. The client may issue multiple concurrent

¹⁶Many details of the specification are excluded here, including, but not limited to, handling of errors during parsing a request, required types for various fields, etc. The reader is encouraged to review the specification itself for these details.

remote procedure calls, and the responses may come back in any order. The client may therefore use the `id` field to match a response to its corresponding request.

Some remote procedure calls require no response. In JSON-RPC parlance, these are called **notifications**. If the client wishes to receive no response to a particular request, it simply omits the `id` field, indicating that this request is really a notification. No response will be returned.

method

A string containing the name of the method to be invoked. In common use, method names are often **namespaced**, i.e., they contain a series of dot-separated entries.

params

The parameter values to be used during the invocation of the method. This field may be provided as an array of positional parameters, or it may be an object containing named parameters. If no parameters need be passed, this field may be omitted.

Response

Responses also have four possible fields, three of which are used on any particular response. All responses contain the two fields `jsonrpc` and `id`. If the requested method completed successfully, the third field is `result`. If the requested method was not successful, the third field is instead `error`.

jsonrpc

The string “2.0” which indicates the version of the JSON-RPC protocol.

id

This `id` value will be the same as was provided in its corresponding request.

result

If the requested method completed successfully, the `result` field contains

the return value from that method. If the requested method did not complete successfully, the `result` field is omitted.

error

If the requested method did not complete successfully, the `error` field will contain an object which contains details of the error. If the requested method did complete successfully, the `error` field is omitted.

Error

When a method does not complete successfully, an object is provided that contains the details of the error. This object is defined to contain three fields: `code`, `message`, and `data`.

code

An integer that indicates which error occurred. There are a number of pre-defined error codes, all in the range of -32000 – -32700. Other error codes are defined by the application designers.

message

A short description of the error. This is intended as a hint to the developer, more than as a string intended to be displayed to the user of the GUI. GUI developers instead generally use the `code` to map to a meaningful error message.

data

Additional optional details about the error.

Requests can be sent to a Version 2 JSON-RPC server in batches. When a batch request is issued, an array of requests is passed. The server responds to an entire batch at one time, providing an array of responses. As expected, since notifications provide no response, any notification requests in the batch will have no corresponding entry in the response array.

3.4.5.2 qooxdoo's modified version 1 protocol

For many years, long pre-dating the release of Version 2 of the JSON-RPC protocol, qooxdoo has supported remote procedure calls. The initial, Version 1 specification was deemed at the time to be imperfect, and with little standardization anyway, qooxdoo provided “improvements” in its implementation of the frontend remote procedure call code. The JSON-RPC server that will be discussed shortly supports both the current Version 2 protocol, and qooxdoo's modified Version 1 protocol, referred to as **qx1**. As the server implementation is discussed, only the Version 2 portion will be described. The equivalent **qx1** protocol implementation will be summarily ignored, as it is expected to be deprecated in the near future.

3.4.5.3 Implementation

With knowledge of the protocol being implemented, let us look at the JavaScript JSON-RPC server in **LIBERATED**. The complete code for this server can be found in Appendix 6.

The architecture of this server involves a **service class factory** which is provided to the server upon instantiation. The service class factory is a function which is passed a namespaced method name, and it must return a function that implements that method. The service class factory is also passed an error object. If the requested method is not available, the error object is filled in to indicate the details of the error, and the factory function returns **null**.

The bulk of the JSON-RPC server code is in the method called **processRequest()**. This method is passed JSON text as input, which is supposed to be a valid JSON-RPC request. Much of the method is devoted to validating that the input is both in proper form and that it makes no attempt to access functionality outside of the remote procedure calls provided at the server. There is a considerable amount of code purely for error handling.

Selected pieces of the **processRequest()** method will now be examined. We will assume here that valid JSON-RPC input is received, and that none of the error code

need be executed. We will look, though, at each of the tests that are accomplished in the process of validating the input.

The first task in processing a request is to parse the JSON input text, as shown on line 107 of Listing 3.7. The static `qx.lang.Json.parse()` method from `qooxdoo` is used to parse the JSON input. It accepts a JSON-encoded string, and returns the object or array corresponding to that encoded string. If there is a parsing problem, the parser throws an error which will be caught and the `catch` portion of the `try/catch` block will be executed.

Listing 3.7: Parsing the input text

```

104     try
105     {
106         // Parse the JSON
107         requests = qx.lang.Json.parse(jsonInput);
108     }
109     catch(e)
110     {

```

With a fully parsed request, we now have either an object, if this is a single request, or an array, if this is a batch of requests. We must therefore determine which it is, in order to act accordingly.

Listing 3.8: Testing for a batch of requests

```

129     if (qx.lang.Type.isArray(requests))
130     {

```

Listing 3.9: Testing for a single request

```

153     else if (qx.lang.Type.isObject(requests))
154     {
155         // It's normal mode
156         bBatch = false;
157
158         // Create an array as if it were batch mode
159         requests = [ requests ];
160     }

```

Listings Listing 3.8 and Listing 3.9 show these tests. Note that if a single request object is found, line 159 converts the single request into an array of requests containing only that one, for common processing later in the code.

We now have an array of requests. We wish to map each of these requests to a response, and we use the JavaScript Array’s `map()` method to do so. This method is called in the context of an array, and returns a new array where each element of the new array is created by mapping the corresponding element of the context array via a specified function. In our case, the context array is the array of requests, and the returned array is the array of responses to each of those requests. The mapping function is quite long. It begins at line 181, shown in Listing 3.10, and continues all the way to line 575. This function implements the details of handling each request.

Listing 3.10: Mapping an array of requests to their corresponding responses

```

181     responses = requests.map(
182         function(request)
183         {
184             var         ret;
185             var         id;
186
187             // Get the id value to use in error responses
188             id = typeof request.id == "undefined" ? null : request.id;

```

Each request may be a normal function call requiring a response, or it may be a notification requiring no response. Line 188 determines which it is. Later, those responses with a `null` id will be filtered from the response array.

Much validation and error checking ensues. The request object being examined is tested to ensure that it is, in fact, an object, since arrays of requests may exist only at the top level, not recursively. Next, the `jsonrpc` field is tested to ensure that it is the string “2.0”. The `method` field is tested to ensure that it is a string, as required by the protocol. The request object is examined to ensure that the `params` field is either omitted, or if provided, is of a correct type: either an object or an array. Listings of these are not shown, as the code is similar to code previously described.

During that validation, the variable `fqMethod` was set to the fully-qualified method name (which for the “2.0” protocol, is the method name provided in the request object). The method name is then validated. The required form of a method name is:

- It must begin with a letter. Subsequent characters may include an underscore, dot (period), letter, or digit.

- No two adjacent characters may be dots.

Method name validation is shown in Listings 3.11 and 3.12.

Listing 3.11: Ensuring the method name contains only valid characters

```
341     if (! /^[a-zA-Z][_a-zA-Z0-9]*$/ .test(fqMethod))
342     {
```

Listing 3.12: Ensuring the method name contains no double dots

```
368     if (fqMethod.indexOf("..") != -1)
369     {
```

With a known safe method name, the service factory is now called to obtain a reference to the requested method, as shown in Listing 3.13.

Listing 3.13: Obtaining the service method from the service factory

```
396     service = this.getServiceFactory()(fqMethod, protocol, error);
```

The service factory function is passed the method name, the protocol (“2.0” for the JSON-RPC Version 2 protocol), and an error object. It returns either a function that implements the requested method, or null if an error occurred in locating the requested method. In the latter case, it sets details of the error in the passed error object.

The JavaScript language has no means of introspection of a function, to ascertain, for example, what parameters the function accepts or expects to be passed, or the names of any of the parameters. JSON-RPC Version 2, however, allows parameters to be passed in a map containing parameter names and associated values,¹⁷ on the expectation that the server implementation will be able to map these named parameters appropriately to positional parameters, as required. One possible implementation of service methods, then, would be for each method to always accept a map which contains each of the parameters. That mechanism, however, makes it less clear in the code that implements a service function, what parameters the function expects.

¹⁷similar to Python’s keyword arguments

In **LIBERATED**, the service function uses normal, positional parameters. Additionally, though, the function may have a *property* called *parameterNames*, which is an array of parameter names, in the order they are expected by the service function. Using this array, named parameters can be placed into a positional parameter array in the positions expected by the service function. Listing 3.14 shows how this is accomplished.

Listing 3.14: Converting named- to positional parameters

```

424     // Were we given a parameter array, or a parameter map, or none?
425     if (qx.lang.Type.isArray(request.params))
426     {
427         // Use the provided parameter list
428         parameters = request.params;
429     }
430     else if (qx.lang.Type.isObject(request.params))
431     {
432         // Does this service allow a map of parameters?
433         if (service.parameterNames)
434         {
435             // Yup. Initialize to an empty parameter list
436             parameters = [];
437
438             // Map the arguments into the parameter array. (We are
439             // forgiving of members of request.params that are not
440             // in the formal parameter list. We just ignore them.)
441             service.parameterNames.forEach(
442                 function(paramName)
443                 {
444                     // Add the parameter. If it's undefined, so be it.
445                     parameters.push(request.params[paramName]);
446                 });
447         }
448     }
449     else
450     {
451         // No provided parameters is equivalent to an empty list
452         parameters = [];
453     }

```

Line 425 tests whether we were passed an array of parameters. If so, we already have a positional list, so it is simply assigned to `parameters`.

Otherwise, we test in line 430 whether named parameters are provided. The *parameterNames* property of the service function is iterated, beginning on line 441, with the value of each named parameter added to the `parameters` array in the order in which it is required.

If neither of the above cases occurs, we have no provided parameters, so the parameter array is initialized to empty, on line 452.

Consider the case where a batch of requests is provided. The batch contains all notifications except for one request in the batch. The client, then, desires to await the result of that single non-notification request. The notifications, however, may be long-running functions, and there is no need to block the client just to run those notifications for which it desires no response anyway. As an optimization, we therefore allow notifications to run after having returned results to the client. Lines 455–506 (not depicted as a Listing here) create a function that will run each single request that we are processing. The created function is stored in a variable called `run`.

Listing 3.15: Schedule or run the service function

```

508     // Is this request a notification?
509     if (typeof(request.id) == "undefined")
510     {
511         // Yup. Schedule this method to be called later, but ASAP. We
512         // don't care about the result, so we needn't wait for it to
513         // complete.
514         setTimeout(
515             function()
516             {
517                 run(request);
518             },
519             0);
520
521         // Set result to the timer object, so we'll ignore it, below.
522         result = timer;
523     }
524     else
525     {
526         // It's not a notification. Run requested service method now.
527         result = run(request);
528     }

```

If the request is a notification, we'll schedule the created function to be called “soon” but after we have completed processing this batch of requests. If the request is not a notification, then we call its service function in-line, in order to obtain a result for this request. This is shown in Listing 3.15. The `id` field of the request is undefined if this request is a notification, and if so, beginning on line 514, we schedule the service function to run `0` milliseconds later. Since JavaScript is single-threaded, that will occur immediately after we finish processing in this function and control returns to the JavaScript main loop. If the `id` is not undefined, we run the service function immediately, as shown on line 527.

The processing of this single request from the batch has then completed. Assuming

no error, the result from the service function is returned. Recall that the current function is the one being used to map requests to responses. This result, then, is placed into the responses array, in the position corresponding to the requests position in the requests array.

Finally, just before returning all of the responses, the array is filtered to remove any (undefined) results from notifications. The JavaScript Array's `filter()` method is used for this. This method is called in the context of an array, and returns a new array where elements of the new array are identical to the elements of the context array, except that some elements may have been removed. A function is provided, which accepts as its parameter a single element from the context array, and returns true if the element should be copied to the new array, or false if the element should not be copied to the new array.

Listing 3.16 shows how this is implemented.

Listing 3.16: Filtering out notification results

```

577     // Remove any responses that were for notifications (undefined ones)
578     responses = responses.filter(
579         function(response)
580         {
581             return typeof(response) !== "undefined";
582         });

```

At last, we have a full set of responses. If this was a batch request, the set of responses is returned. If this was not a batch request, we had simulated it as a batch request in order to process an array of elements, but the client expects only a single response. We therefore return the first response from the array. This is shown in Listing 3.17.

Listing 3.17: Returning a batch or single response

```

591     // Give 'em the response(s)
592     return (bBatch
593         ? qx.lang.Json.stringify(responses)
594         : qx.lang.Json.stringify(responses[0]));

```

3.4.6 Per-backend remote procedure call interface

The JSON-RPC server discussed in Section 3.4.5 expects as input, a JSON-encoded string containing a valid JSON-RPC request. That code runs at the server, simulated or real. The means by which the JSON-encoded string is received by the server is dependent on the server itself. The base functionality is likely to be the same regardless of server, but each has its own details to be worked out.

3.4.6.1 Abstract handler for remote procedure calls

We shall now look at the abstract class in **LIBERATED** which provides the service map and service factory — a piece of code that can be common between all of the servers. The service factory, in addition to simply returning a function reference, also allows an authorization function to be provided. If an authorization function is specified, it will be called to ensure that the current user is allowed to run the requested method.

3.4.6.1.1 Creating a handler for RPC services

The full Abstract RPC Handler is shown in Appendix 7. Its constructor is detailed in Listing 3.18.

Listing 3.18: Constructor for the Abstract RPC Handler

```

51  construct : function(rpcKey)
52  {
53      var          i;
54      var          services = {};
55      var          part;
56
57      // Call the superclass constructor
58      this.base(arguments);
59
60      // Initialize the services
61      rpcjs.AbstractRpcHandler._services = services;
62
63      // Add each of the RPC keys
64      for (i = 0, part = services; i < rpcKey.length; i++)
65      {
66          part = part[rpcKey[i]] = {};
67      }
68
69      // Store the final part, where registered services will go
70      rpcjs.AbstractRpcHandler._servicesByKey = part;
71
72      // Get an RPC Server instance.
73      this.__rpcServer = new rpcjs.rpc.Server(

```

```

74     rpcjs.AbstractRpcHandler._serviceFactory);
75 },

```

As seen on line 54, the constructor creates a map of service methods. This map acts as a tree to locate a particular service method. The full, dot-separated path to a method defines the path within the service map, i.e., to the actual service method (function). For example, if the requested method name is the string “sys.fs.read” then for this method to be found, the service map must contain entries like this:

```

{
  sys :
  {
    fs :
    {
      read : function()
      {
        // implementation of the "sys.fs.read" method
      }
    }
  }
}

```

The constructor takes a single parameter, an array, that defines the initial, shallow-end portion of the services map tree. In the above example, the parameter to the constructor would be the array,

```
[ "sys", "fs" ]
```

which specifies that for this handler, any methods that are registered will be in the “sys.fs” namespace. Lines 64–67 create that initial portion of the services tree.

The RPC Server which was discussed in Section 3.4.5 is instantiated at lines 73–74. In instantiating the JSON-RPC server, this code provides a reference to its own service factory method, `_serviceFactory`. That service factory method is shown in Listing 3.19. As was previously described, the service factory takes three parameters: the method name, the JSON-RPC protocol version, and an error object.

Listing 3.19: Service factory

```

116   _serviceFactory : function(fqMethodName, protocol, error)
117   {

```

```

118     var                method;
119
120     // Append the fully-qualified method name to the services map and
121     // evaluate it in hopes of getting a method reference.
122     //
123     // We have a dot-separated fully-qualified method name. We want to
124     // access that entry in the map of maps in _services. Using the
125     // index notation won't work, since it's multiple levels deep
126     // (i.e. fqMethodName might be something like "a.b.c").
127     //
128     // fqMethodName was sanitized by rpcjs.rpc.Server:processRequest()
129     // so this eval is reasonably safe. (I know... Famous last words!)
130     method = eval("rpcjs.AbstractRpcHandler._services" +
131                 "." + fqMethodName);
132
133     // We might have just gotten null, which also means no such method
134     if (! method)
135     {
136         // No such method.
137         error.setCode(
138             {
139                 "qx1" : qx.io.remote.RpcError.qx1.error.server.MethodNotFound,
140                 "2.0" : qx.io.remote.RpcError.v2.error.MethodNotFound
141             }[protocol]);
142         error.setMessage("No such method");
143         return null;
144     }
145
146     // Validate allowability of calling this function
147     if (rpcjs.AbstractRpcHandler.authorizationFunction &&
148         !rpcjs.AbstractRpcHandler.authorizationFunction(fqMethodName))
149     {
150         // Permission denied
151         error.setCode(
152             {
153                 "qx1" : qx.io.remote.RpcError.qx1.error.server.PermissionDenied,
154                 "2.0" : qx.io.remote.RpcError.v2.error.PermissionDenied
155             }[protocol]);
156         error.setMessage("Permission denied.");
157         return null;
158     }
159
160     // Give 'em the reference to the method they can call.
161     return method;
162 },

```

3.4.6.1.2 Service factory

The key to the service factory is finding a method based on a dot-separated name, within the services map. Lines 130–131 accomplish this, using JavaScript's `eval()` method. Using `eval()` is potentially unsafe, particularly if user input is ever directly passed to it. In this case, though, the fully-qualified method name has been vetted by the RPC server, to ensure, we hope, that it contains nothing that could be dangerous. (See 3.11 and 3.12.)

If the requested name exists in the services map, `method` will reference the proper

function. If not, `method` will be `null` or `undefined`. In the latter case, we simply return a “No such method” error.

This class has a static variable, `authorizationFunction`, which defaults to `null`. The application may modify this variable to reference a function that will be called to ensure that the current user is allowed to run a particular method. Lines 147–148 are checking whether the application has provided an authorization function, and if so, that function is called, and passed the requested method name. The authorization function is required to return `true` if the current user is authorized to run the requested method; `false` otherwise.

If the authorization function returns `false`, then a “Permission denied” error is returned. Otherwise, we have the requested method and the current user is allowed to run it, so the method reference is returned.

3.4.6.1.3 How RPC services are registered

Remote procedure call methods are registered as a tuple consisting of the name of the method (not including the namespace that was passed to the constructor), a function that implements the remote method, and optionally an array that lists the names of the parameters, which allows JSON-RPC Version 2’s named parameter capability. The function is often required to be called in the context of some object, so the registration also includes a context parameter. Listing 3.20 shows how service registration takes place.

Listing 3.20: Service registration

```

179   registerService : function(serviceName, fService, context, paramNames)
180   {
181     var          f;
182
183     // Use this object as the context for the service
184     f = qx.lang.Function.bind(fService, context);
185
186     // Save the parameter names as a property of the function object
187     f.parameterNames = paramNames;
188
189     // Save the service
190     rpcjs.AbstractRpcHandler._servicesByKey[serviceName] = f;
191   },

```

First, line 184 binds the specified service function to the context. Next, on line

187, the bound function object has a property attached to it, to hold the parameter names array. Finally, line 190 attaches the bound function to the services map, based on its specified name.

When a new request comes in from a transport, it is provided to the RPC handler by a call to the `processRequest()` method. This method is shown in Listing 3.21.

Listing 3.21: Processing a JSON-RPC request arriving via a transport

```

269     processRequest : function(jsonData)
270     {
271         // Call the RPC server to process this request
272         return this.__rpcServer.processRequest(jsonData);
273     }

```

The constructor instantiated the RPC Server and stored the instance in its private `__rpcServer` variable (see Listing 3.18, line 73). Processing a new inbound request, then, is simply a matter of passing the request to the RPC Server's `processRequest()` method, as shown on line 272.

3.4.6.2 Handler for remote procedure calls: simulated server

The previous section detailed an abstract class to handle many of the common issues pertaining to a remote procedure call handler. Here, we look specifically at the RPC handler for the simulated server, which extends that abstract class.

Recall from Section 3.4.4 that the simulation web server tries each of its registered handlers in turn, in search of one that is willing to handle a particular inbound request. It initializes the `status` field of the `responseHeaders` parameter to **501**, a code that means that the request is not supported.

In the handler for remote procedure calls in the simulated server, then, we must decide whether to process a request. This is based on a URL which was provided to the constructor, shown in Listing 3.22, and is accomplished in the `__processRequest()` method shown in Listing 3.23.

Listing 3.22: Simulation remote procedure call handler constructor

```

54     construct : function(rpckey, url)
55     {
56         // Call the superclass constructor

```

```

57     this.base(arguments, rpckey);
58
59     // Save the URL
60     this.setUrl(url);
61
62     // Register ourselves with the simulation transport, as a handler
63     // for the specified URL.
64     rpcjs.sim.Simulator.registerHandler(
65         qx.lang.Function.bind(this.__processRequest, this));
66 },

```

Listing 3.23: Request to process an incoming message from the simulation server

```

97     __processRequest : function(request, responseHeaders)
98     {
99         var         jsonData;
100        var         result;
101
102        // Make sure we can handle this request
103        if (request.url != this.getUrl())
104        {
105            // We don't support this one. Response header status was preset.
106            return null;
107        }
108
109        // For the moment, we support only POST
110        if (request.method != "POST")
111        {
112            responseHeaders.status = 405;
113            responseHeaders.statusText =
114                "Method " + request.method + " not allowed.";
115            return null;
116        }
117
118        // Retrieve the JSON-RPC data
119        jsonData = request.data;
120
121        // From here on out, we'll have a successful result (even if the RPC
122        // sends back an error response).
123        responseHeaders.status = 200;
124        responseHeaders.statusText = "";
125
126        // Call the RPC server to process this request
127        result = this.processRequest(jsonData);
128        return result;
129    }

```

The constructor is given the URL that will be used for handling remote procedure calls with this instance of the handler, and that URL is saved in the *url* property. The constructor then registers the `__processRequest()` method as a handler, with the simulation transport.

This handler's `__processRequest()` method will be called when a request arrives via the simulation transport. It is passed the details of the request in the `request` parameter and in the pre-initialized `responseHeaders` map. This method first tests, at line 103, whether the URL in the request is the one which was assigned to this

handler when this handler was instantiated. If not, then `null` is returned. There is no need to modify the response headers, as they have been pre-initialized to a status value of **501** (request not supported by this handler).

Once it is determined that the URL matches, the code tests, at line 110, whether the request uses method **POST** and if not, returns an error indication.

Having gotten this far, from a simulated HTTP point of view, the request is valid and we set it up to return a **200** (success) status value regardless of the outcome of the remote procedure call (lines 123–124).

Line 119 retrieves the JSON-RPC request, and at line 127, the JSON-RPC request is passed to the simulation server, via the superclass abstract handler's `processRequest()` method, where the majority of work for handling the request is accomplished.

Although much of the class has been included in listings here, it is provided in its entirety in Appendix 8.

3.4.6.3 Handler for remote procedure calls: App Engine

Since the abstract handler accomplishes all of the common work for any remote procedure call handler, there is little for a concrete handler to do. This was seen in the previous section, where only a small bit of processing was done before calling the abstract handler's `processRequest()` method.

In the remote procedure call handler for App Engine, there is no additional work required at all, over what the abstract handler implements. The App Engine RPC handler extends the abstract handler, but no additional functionality is provided, as shown in the listing of this class in Appendix 9.

3.4.7 Database manipulation

*One Entity to rule them all, One Entity to query,
One Entity to bring them all, and in the server bind them.*¹⁸

¹⁸Clearly, Tolkien I am not.

In Section 3.2.3, a number of database models were discussed. In this section, we'll look at an abstraction that could potentially be mapped to any of a number of different database models, and then inspect two specific implementations: one that maps the abstraction to a browser-based simulated database, and one that maps to the Google App Engine database.

As we discussed the various database models, in each case, we referred to a basic element of information that could be stored in a database of that model, and we called it an **entity**. An entity, then, is a single unit of data that can be stored individually to a database, and later retrieved individually from the database. The mechanism by which an entity is found in a database varies between models, but in each case, some piece of **key** data uniquely identifies a single entity.

Various types of data that are stored in the database require differing sets of **columns** or **cells** or **fields** or **data properties** (different terms for the same concept, in different database models) within an entity. In a flat model database, each type of data requires a new table-like structure, with potentially different column headings. Similarly, a relational database defines **tables** of different types, each of which can be assigned different sets of **column** labels. An object database can have objects with different members, also known as **properties**, when it needs to refer to different types of data.

An abstraction at the **entity** level, then, allows mapping to databases of various models. In **LIBERATED**, an abstract class called `rpcjs.dbif.Entity` is used to model these elements and types of elements stored in and retrieved from the database. Each concrete subclass of `rpcjs.dbif.Entity` defines the data properties of a particular **entity type**, and an instance of one of those specific concrete subclasses allows storing data of that type into the database. Each concrete subclass of `rpcjs.dbif.Entity` also defines the **key** field(s) which are used to uniquely identify a particular entity within the database, so an instance of the class can efficiently retrieve a particular entity of data from the database.

Let's look at the requirements for the Entity class, from the perspective of **LIBERATED**. We can then look at the code to see how those requirements are met.

Requirements of the implementation:

1. We need a way to store the data properties of a subclass that implements a particular **entity type**.
2. Since this Entity class is generic, i.e., not tied to any particular database model, we require database-specific code to implement the actual interface to a database. This database-specific **driver** must therefore be able to register itself with the Entity class, so that operations on entities can be passed off to the driver.
3. There must be a means of managing (setting and getting) the cell data for an entity instance.
4. When we instantiate an entity instance, it should be pre-populated with data from the database, based on a **key** value.
5. Some entity types will have no inherently unique data property values. There must be a mechanism for automatically adding a **key** value that will be unique, and is independent of any of the data property values in the entity.
6. Other entity types will require that two or more data property values, in combination, be unique. There must therefore be a mechanism for specifying **composite keys** which are composed of multiple data property values.
7. After having specified the data for an entity, there must be a method to write that data to the persistent database.
8. Data need sometimes be removed from the database, so there must be a method to remove an entity from persistent storage.
9. Sometimes the application will know specifically the key of the entity it requires. For other cases, a more general search or query mechanism is needed.
10. Certain data property values may be stored in the database in a form specified by some user, e.g., in mixed case, but must be allowed to be queried based on a canonical representation of the values.
11. Very large objects to be stored in the database may cause inefficiencies or problems, if stored directly in normal entities. A means to write, retrieve, and remove these large objects (known as **blobs**) from the database is therefore required.

3.4.7.1 Entity type registration

Each subclass of `rpcjs.dbif.Entity` registers its **entity type** by calling the static function `rpcjs.dbif.Entity.registerEntityType()` which has the signature shown in Listing 3.24.

Listing 3.24: Entity type registration function

```

247     registerEntityType : function(classname, entityType)
248     {
249         // Save this value in the map from classname to entity type
250         rpcjs.dbif.Entity.entityTypeMap[classname] = entityType;
251     },

```

The parameters to this function are the class name of the subclass, and a (typically) shorter entity type name that can be used by the database driver. (The entity type name might, for example, be used as the name of a table in a SQL database.) The `rpcjs.dbif.Entity.registerEntityType()` function creates a mapping from the class name to the entity type, that can be used later for converting from a known class name to its associated entity type.

3.4.7.2 Data properties of an entity

Each subclass of `rpcjs.dbif.Entity` implements a specific type of object that is stored in the database. Each type of object, i.e., each subclass of `rpcjs.dbif.Entity` defines a set of data properties which each instance of that object will contain. Every object instance of that subclass, and thus every entity in the database of this same entity type, contains the set of data properties specified by that subclass.

These data properties may take on various somewhat generic data types, which are mapped as necessary to the particular database in which the data is stored. These data types are:

String is an ordinary character string, which may be queried.

LongString is also a character string, but may not be queried. Some databases have a length limit on queryable strings, or can make retrieval more efficient by storing long data fields differently or separately from shorter data. Defining a

property as a `LongString` type allows the database implementation flexibility to store these more efficiently.

Date is a date and time field.

Integer and Float are stored in their respective numeric formats.

Key is an automatically-generated key field in the database. In most cases, it is an auto-incrementing integer value which is assigned to a new object being stored in the database, and is then incremented so that the next stored object gets the subsequent value. Some databases may implement automatic key values differently, so no assumptions can be made about the type. Key field values can be used to reference other objects.

KeyArray, **StringArray**, **LongStringArray**, **IntegerArray**, and **FloatArray** are fields which contain a list of items of the specified type. A query for a value in that field will return the object if any element of the list matches the specified query value.

Data properties are defined in a map, with the map's member names being the names of data properties in the entity type, and the member values being one of the data types shown above. An example database property map is shown in Listing 3.25:

Listing 3.25: A map specifying database properties and their types

```

1   {
2     value : "String",
3     type  : "String",
4     count : "Integer"
5   }
```

Three data properties are defined in this database property map: a `value` field, which will contain a string, a `type` field, which will contain a string, and a `count` field, which will contain an integer.

3.4.7.3 Canonical values of properties

At times, the value to be stored in the database is in a form meaningful to a user, but a form that is not convenient for which to search. For example, text may

be entered by the user in mixed case, yet a search for that text should always be accomplished in a case-insensitive fashion. When the object is located, however, the mixed case that had been entered by the user should be returned.

For this purpose, an Entity subclass may define fields that are to be converted to a **canonical** form, and stored both in their original and canonical forms. When specifying data properties that are to be canonicalized, a new field name is provided in which the canonical form is stored. The canonical form need not be of the same data type as the form of the data property being canonicalized, so the data type of the canonical form is specified as well. Finally, a function that converts a property value to its canonical form is given.

In a fashion similar to definition of the database property map, fields to be canonicalized are also defined in a map. Each member name is the name of a data property defined in the database property map. Each member value is itself a map containing three fields: the data property name for this property's canonical value, the type of the canonical value, and a function to be used to convert the property value to its canonical value. Listing 3.26 shows an example.

Listing 3.26: A map specifying how to canonicalize the *value* field

```

1      {
2          value :
3          {
4              // Data Property in which to store the canonicalized value.
5              // Since we are converting the value to lower case, give it
6              // a name suffix that reflects that.
7              prop : "value_lc",
8
9              // The canonical value will be a string
10             type : "String",
11
12             // Function to convert a value to lower case
13             func : function(value)
14             {
15                 return value.toLowerCase();
16             }
17         }
18     }

```

Here, the map specifies that the *value* data property is to be stored both in its original form, and in its canonical form. The canonical form will be saved in a new data property named *value_lc*, which is to hold a string. To convert from the original form of *value* to its canonical form, the value will be converted to lower case by the provided

function.

3.4.7.4 The entity key property

Each entity stored in the database has a unique **key** value. Through use of the key, a particular entity can be quickly and efficiently retrieved from the database. Where the key value comes from is determined by the subclass implementing the entity type. Some entity types contain a data property that is always unique and whose value can be used as the key. Other entity types contain some combination of data properties that together are unique and whose values, concatenated by some means, can be used as the key. These are called **composite** keys, as they are composed of the values of multiple data properties from the object. Finally, there are entity types which have no data properties that are unique. For the final case, an automatically-generated “unique identification” value, a *uid*, is used as the key.

3.4.7.5 Registering data property types

Each entity type subclass registers itself with the `Entity` class, so that entities of that type can be stored to, or retrieved from, the database. The static function `rpcjs.dbif.Entity.RegisterPropertyTypes()` is called for this purpose. The function is shown in Listing 3.27.

Listing 3.27: How an entity type registers itself

```

324     registerPropertyTypes : function(entityType,
325                                   propertyTypes,
326                                   keyField,
327                                   canonicalize)
328     {
329         var                pn;        // property name
330
331         // If there's no key field name specified...
332         if (! keyField)
333         {
334             // Add "uid" to the list of database properties.
335             propertyTypes["uid"] = "Key";
336             keyField = "uid";
337         }
338
339         // Add the canonicalize properties to the property list
340         if (canonicalize)
341         {
342             for (pn in canonicalize)
343             {

```

```

344         // Add the property in which to store the canonicalized value
345         // to the list of database properties.
346         propertyTypes[canonicalize[pn].prop] = canonicalize[pn].type;
347     }
348 }
349
350 rpcjs.dbif.Entity.propertyTypes[entityType] =
351 {
352     keyField      : keyField,
353     fields        : propertyTypes,
354     canonicalize  : canonicalize
355 };
356 },

```

`rpcjs.dbif.Entity.registerPropertyTypes()` may have four parameters passed to it:

1. `entityType` is the short name for the entity type.
2. `databaseProperties` defines the names and types of data properties for this entity type. It is a map such as that shown in Listing 3.25
3. `keyField` is the field or set of fields to be used to determine the key of an entity written to the database. It may be in any of the forms discussed above: a string representing a single data property name, an array of strings to indicate a composite key, or null or undefined to indicate that there is no field or set of fields in the object that can be used as the key, so a *uid* should be used instead.
4. `canonicalize` is a map such as that shown in Listing 3.26 and describes the data properties that are to have canonical values stored in the database and the means of creating those canonical values.

The function first checks, at line 332, whether a key field was specified. If none was, a *uid* data property is added to the `propertyTypes` map. Its type is “Key” to indicate that it is automatically generated according to the rules of the database being used.

If a `canonicalize` map is provided, the code at lines 340–348 adds each of the data properties in which canonical values will be stored, to the `propertyTypes` map. The type of each new data property is as defined in the `canonicalize` map.

Finally, at lines 350–355, the key field, property types, and canonicalization information is saved in a map indexed by the entity type.

3.4.7.6 The database-specific driver interface

Let’s move on to the requirements of a database-specific driver, and how such a driver may register itself with the `Entity` class. Sections 3.4.7.9 and 3.4.7.10 will

detail the implementations of those drivers. Here, we discuss the common interface that each driver must implement when registering as a driver.

Listing 3.28: How a database driver registers itself

```

97     registerDatabaseProvider : function(query, put, remove,
98                                   getBlob, putBlob, removeBlob)
99     {
100       // Save the specified functions.
101       rpcjs.dbif.Entity.__query = query;
102       rpcjs.dbif.Entity.__put = put;
103       rpcjs.dbif.Entity.__remove = remove;
104       rpcjs.dbif.Entity.getBlob = getBlob;
105       rpcjs.dbif.Entity.putBlob = putBlob;
106       rpcjs.dbif.Entity.removeBlob = removeBlob;
107     }

```

Listing 3.28 shows the function used by a database driver to register itself with the `Entity` class. The database driver provides six function references, and the `registerDatabaseProvider()` method simply stores those references. The six functions implement the following functionality:

query

Retrieve one or more objects from the database.

put

Write an object into the database’s persistent storage.

remove

Remove an object from the database’s persistent storage.

getBlob

Retrieve a “large object” from the database.

putBlob

Write a “large object” to the database.

removeBlob

Remove a “large object” from the database.

The `Entity` class provides an interface definition, or signature, for each of these functions, and each database driver’s functions must conform to those signatures. Let’s examine those interfaces.

3.4.7.6.1 query

The `query()` function takes three parameters. The first is the class name of the **Entity** subclass being queried. The other two parameters describe the search criteria (i.e., which entities of the specified entity class are eligible to be returned), and the result criteria, which can be used to restrict or alter the returned entities. The return value is always an array, regardless of how many results are returned.

searchCriteria

A query requests that one or more entities of a specified entity type be located and retrieved from the database. The query is provided with search criteria to identify the set of particular entities that should be retrieved. The resulting set could yield no results, a single result, or more than one result, depending upon the query and the data that exists in the database.

When issuing a query, if all entities of the specified entity type are allowed to match, then the search criteria parameter may be entirely omitted.

When only a single entity is desired and its key is known, the search criteria can be the string, number, or array (consisting of the composite key values) that specifies the key value of the particular desired entity.

In all other cases, the search criteria for a query consists of an arbitrarily deep tree composed of JavaScript objects. Each node in the tree is of one of two types, as determined by its `type` field. If the type is the string **element** then the node specifies that a particular data property in the database must have a given value. In this case, there must also be a `field` member which contains a string specifying which data property is to be matched, and a `value` member to indicate what value must be in that field. Optionally, there may be a `filterOp` member that specifies how the `value` is compared against each entity's `field` data. By default, when no `filterOp` member is present, the operation is **equal to**. When specified, the value for `filterOp` may be one of: “<”, “<=”, “=”, “>”, “>=”, or “!=”, corresponding to **less than**, **less than or equal to**, **equal to**, **greater than**, **greater than or equal to**, and **not equal to**, respectively.

The other type of node is an **operation**, indicated by the **type** member being the string “op”. An operation node must contain a **method** member which contains the logical operation to be performed. At present, the only supported operation is “and”. There must also be a **children** member, which is an array of the criteria to which the specified operation is applied.

resultCriteria

All entities that match the search criteria are eligible to be returned by the query. That result set can be restricted or altered based on the **resultCriteria** parameter. This parameter is expected to be an array of maps. Every element in the array further restricts or alters the entities to be returned, and the elements are processed in their order within the array. Each map in the array contains a **type** field which may be one of the strings “offset”, “limit”, “sort”, or “option”.

When the type is “offset” or “limit”, there must be a **value** member which indicates, respectively, the offset into the list of entities eligible to be returned at which to begin the actual set of returned entities; and the limit of how many entities to actually return.

The type may be “sort”, in which case there must be a **field** member which specifies the data property within the specified entity type on which to sort the result set. There must also be an **order** member with the value “asc” to indicate an ascending sort, or “desc” to indicate a descending sort. There may be multiple maps in the array with a type of “sort”, in which case the earlier sort specification is the overriding field of the sort, the next specification causes a sort by its field within each same value of the initial sort specification, etc.

Finally, the type can be “option” which indicates that an option **name** member will be present, and if necessary, an option **value** member which is of the correct type for the specified name. The only supported option name at present is “stripCanon” which requires a boolean **value**. This specifies whether the data properties which were added for storing canonical values should be removed from each entity in the result set.

3.4.7.6.2 put

The database driver interface for writing an entity to the database is quite simple. The `put()` method is expected to take an entity as the one and only parameter, and write it to the database. The `put()` method uses the properties of the entity object, e.g. the *entityType* and *data* properties, to ascertain what to write to the database, and where.

3.4.7.6.3 remove

Similarly to `put()`, the `remove()` method is given an entity as its one and only parameter, and must remove the element with the given entity's key from the database.

3.4.7.6.4 getBlob

Large objects are identified by a database-specific unique identifier. The one and only parameter to the `getBlob()` method is that unique identifier, known colloquially as the “blob id.” The return value is the blob data, of type “LongString.” When a blob id is provided that is not found in the database, this method returns the JavaScript value `undefined`.

3.4.7.6.5 putBlob

To add a blob to the database, the `putBlob()` method is used. This method accepts as its parameter data of type “LongString”, writes that data to the database, and returns the unique “blob id” identifying that data.

3.4.7.6.6 removeBlob

Removing a blob from the database is accomplished by providing the identifier of a blob as the one and only parameter to the `removeBlob()` method. If the blob with the specified id exists, it is removed. If it does not exist, the request is silently ignored. There is no return value from this method.

3.4.7.7 Public interface for database operations

The previous section discussed the interface provided by the database-specific drivers. The functions in database-specific drivers are not called directly by the application developer. Instead, there are static and member methods in `rpcjs.dbif.Entity` that the developer directly accesses.

3.4.7.7.1 Properties and setters/getters

As discussed in Section 3.4.1.7, qooxdoo automatically generates setters and getters for defined **properties** of classes. Two properties of the `rpcjs.dbif.Entity` class are particularly important for making use of entities.

data

The **data** property contains a map with member names being the registered data property names, and the data values for those data properties, specific to this entity, as the member values. The setter method is `setData()`, and the getter method is `getData()`.

brandNew

When a new entity is instantiated, the constructor will issue a query for the specified key value, as described in the next section. The *brandNew* property will then contain a boolean value indicating, if **false**, that the key value was found in the database and that the *data* property contains the entity data values that were found in the database. If **true**, this property indicates that the specified key was not found, and therefore this is a brand new entity. The property value may be obtained with the `getBrandNew()` getter method.

3.4.7.7.2 Constructor

The constructor for the `rpcjs.dbif.Entity` class is typically called only by its subclass' constructors. It takes two parameters: **entityType**, the entity type (short name) of the subclass, and **entityKey**, an entity key value which will cause a query to be issued to initialize the data in the instantiated entity object to what is found in

the database. If the key field, specified when the property types for this entity type were registered via a call to `rpcjs.dbif.Entity.registerPropertyTypes()`, is one containing a canonical value of some other field, the `entityKey` value is converted to its canonical form before issuing the query.

The `rpcjs.dbif.Entity.query()` method is called with the provided or canonicalized `entityKey` value. If results are returned, i.e., the specified key exists in the database, the data from the query is stored in the `data` property of the new instance, and the `brandNew` property is set to **false**. If no results were returned, the key does not exist, so the `brandNew` property is set to **true**.

3.4.7.7.3 Static method `rpcjs.dbif.Entity.query`

The `rpcjs.dbif.Entity.query()` method provides a wrapper around the database-specific driver's `query()` method. The primary purpose of the wrapper is to modify the query provided by the caller into one that references the canonical fields instead of their non-canonicalized peers. Listing 3.29 shows how this is accomplished.

Listing 3.29: Replacing data property names with their canonical peer

```

444     // Gain easy access to the canonicalize map.
445     canonicalize =
446         rpcjs.dbif.Entity.propertyTypes[entityType].canonicalize;
447     if (canonicalize)
448     {
449         // Get a list of the fields to be mapped
450         canonFields = qx.lang.Object.getKeys(canonicalize);
451     }
452
453     // Are there any canonicalization functions for this class
454     if (canonicalize && searchCriteria)
455     {
456         // Yup. Rebuild the search criteria, replacing non-canonicalized
457         // fields with their peer canonical fields. Start with a clone
458         // of the original criteria, so we don't modify the caller's map.
459         searchCriteria = qx.util.Serializer.toNativeObject(searchCriteria);
460
461         // Recursively descend through the search criteria, replacing
462         // non-canonicalized field names with their canonical peer.
463         (function replaceCanonFields(criterion)
464         {
465             // Null or undefined means retrieve all objects.
466             if (! criterion)
467             {
468                 // Nothing for us to do
469                 return;
470             }
471
472             // element criterion (type="element" or no type field)?
473             if (! criterion.type || criterion.type == "element")
474             {

```

```

475     // Is this field name one to be canonicalized?
476     if (qx.lang.Array.contains(canonFields, criterion.field))
477     {
478         // Replace the value with the canonical version
479         criterion.value =
480             canonicalize[criterion.field].func(criterion.value);
481
482         // Replace the field name with its canonical peer
483         criterion.field = canonicalize[criterion.field].prop;
484     }
485 }
486 else if (criterion.children)
487 {
488     // If there are children, deal with each of them.
489     criterion.children.forEach(
490         function(child)
491         {
492             replaceCanonFields(criterion[child]);
493         });
494     }
495 })(searchCriteria);
496 }

```

First, the list of canonicalized fields is obtained, at line 450. If there were any canonicalized fields, line 459 creates a deep copy of the search criteria.¹⁹ This avoids altering the search criteria object provided by the caller. The copy of the search criteria map is then passed to the recursive function `replaceCanonFields()` where the `type` member is examined. Nodes of type “element” indicate fields to be searched on. If the requested field is one that has a canonical peer, the field name in the node is replaced with that peer’s name. This is accomplished by lines 472–485.

If the type was not “element” then a “children” member is examined. If it exists, the `replaceCanonFields()` function is called recursively with each child from the array as a parameter, as shown in lines 486–494.

After completing all recursion of `replaceCanonFields()`, a fully-modified search criteria is available, and the database-specific driver’s query method is called.

If result criteria were provided to the query request, the option value “stripCanon”, discussed in Section 3.4.7.6.1, is now handled. If the option is provided, and was specified with a value of `true`, then the result set is iterated, and the added canonical value data properties are removed.

¹⁹A **deep copy** means that arrays and maps containing, to an arbitrarily deep level, arrays and maps, are recursively copied. There are many serializers available in `qoxxdoo`. They all recursively traverse the parameter object and copy its data into a new form, e.g., JSON. This one serializes into a native JavaScript object.

3.4.7.7.4 Member method `put`

Writing an entity to the database is accomplished with the member method, `put()`. After the map in the `data` property has been provided with all of the data properties to be stored in the database, the `put()` method is called. No parameters are required, as the data to be written is taken directly from the `data` property map.

3.4.7.7.5 Member method `removeSelf`

Removing an entity from the database is accomplished by instantiating an entity of the appropriate subclass of `rpcjs.dbif.Entity`, setting the key value, and calling the member method `removeSelf()`.

3.4.7.7.6 Static method `rpcjs.dbif.Entity.getBlob`

The static method `rpcjs.dbif.Entity.getBlob()` directly calls the database-specific driver method of the same name.

3.4.7.7.7 Static method `rpcjs.dbif.Entity.putBlob`

The static method `rpcjs.dbif.Entity.putBlob()` directly calls the database-specific driver method of the same name.

3.4.7.7.8 Static method `rpcjs.dbif.Entity.removeBlob`

The static method `rpcjs.dbif.Entity.removeBlob()` directly calls the database-specific driver method of the same name.

3.4.7.8 An example entity type subclass

For purpose of illustration, let us define an example entity class which we can create some instances of, manipulate, write to and read from the database, etc.

Listing 3.30 defines a new class of entities which provide the details about specific dogs. The class is, of course, a subclass of `rpcjs.dbif.Entity`.

Listing 3.30: An entity class for dogs

```

1 qx.Class.define("animals.ObjDogs",
2   {

```

```

3     extend : rpcjs.dbif.Entity,
4
5     construct : function(ownerSurname, ownerGivenName, dogName)
6     {
7         // Pre-initialize the data
8         this.setData(
9             {
10                dogName      : dogName,
11                ownerSurname  : ownerSurname,
12                ownerGivenName : ownerGivenName,
13                breed         : null, // not yet known
14                age           : 0     // not yet known
15            });
16
17        // The key is a composite of the owner's surname, owner's
18        // given name, and dog's name, on the assumption that that
19        // tuple will always be unique.
20        this.setEntityKeyProperty([
21            "ownerSurname",
22            "ownerGivenName",
23            "dogName"
24        ]);
25
26        // Call the superclass constructor
27        this.base(arguments,
28            "dogs",
29            [
30                ownerSurname,
31                ownerGivenName,
32                dogName
33            ]
34        ),
35
36        defer : function(clazz)
37        {
38            // The short entity name for this entity type is "dogs"
39            rpcjs.dbif.Entity.registerEntityType("animals.ObjDogs",
40                "dogs");
41
42            // Specify the data properties for this entity type
43            var databaseProperties =
44            {
45                // The dog's name
46                dogName      : "String",
47
48                // The owner's surname
49                ownerSurname  : "String",
50
51                // The owner's given name
52                ownerGivenName : "String",
53
54                // The dog's breed
55                breed         : "String",
56
57                // The dog's age
58                age           : "Integer"
59            };
60
61            // Register our property types. The key is a composite of
62            // the owner's surname, owner's given name, and dog's name,
63            // on the assumption that that tuple will always be unique.
64            rpcjs.dbif.Entity.registerPropertyTypes("dogs",
65                databaseProperties,
66                [
67                    "ownerSurname",
68                    "ownerGivenName",
69                    "dogName"
70                ]
71            );

```

```

71     }
72   });

```

We can now add a few dogs to the database, as shown in Listing 3.31:

Listing 3.31: Adding some dogs to the database

```

1  [
2    {
3      surname   : "Lipman",
4      givenName : "Derrell",
5      dogName   : "Stasha",
6      breed     : "Siberian Husky",
7      age       : 7
8    },
9    {
10     surname   : "Marks",
11     givenName : "Joel",
12     dogName   : "Snoopy",
13     breed     : "Purebreed Mutt",
14     age       : 2
15   },
16   {
17     surname   : "Binks",
18     givenName : "Jar Jar",
19     dogName   : "Curly",
20     breed     : "Poodle",
21     age       : 11
22   }
23 ].forEach(
24   function(data)
25   {
26     // Instantiate a new dog and provide its composite key values
27     var dog = new animals.ObjDogs(data.surname,
28                                   data.givenName,
29                                   data.dogName);
30
31     // Retrieve the data property map
32     var dogData = dog.getData();
33
34     // Give this dog its additional data properties
35     dogData.breed = data.breed;
36     dogData.age = data.age;
37
38     // Write this dog to the database
39     dog.put();
40   });

```

Finally, we can query the database. First, we look at the most basic of queries: one that requests all entities of a specified entity type that exist in the database. Listing 3.32 shows a request to retrieve all of the objects of class `animals.ObjDogs` from the database.

Listing 3.32: Query for all dogs in the database

```

1  var results = rpcjs.dbif.Entity.query("animals.ObjDogs");

```

That result set will contain the dog entities in an arbitrary order. Listing 3.33 returns the same result set, but sorts the results in ascending order, by each dog's breed.

Listing 3.33: Sort dogs by breed

```

1  var results = rpcjs.dbif.Entity.query(
2      // The class name of the entity type to be queried
3      "animals.ObjDogs",
4
5      // Search criteria. In this case, there are none.
6      null,
7
8      // Result criteria. Sort by breed, ascending
9      [
10     {
11         type : "sort",
12         field : "breed",
13         order : "asc"
14     }
15 ]);

```

We could further sort the results, so that within the group of results for each breed, the dogs are sorted by their age, in descending order, as shown in Listing 3.34.

Listing 3.34: Sort dogs by breed then age

```

1  var results = rpcjs.dbif.Entity.query(
2      // The class name of the entity type to be queried
3      "animals.ObjDogs",
4
5      // Search criteria. In this case, there are none.
6      null,
7
8      // Result criteria. Sort by breed ascending, age descending
9      [
10     {
11         type : "sort",
12         field : "breed",
13         order : "asc"
14     },
15     {
16         type : "sort",
17         field : "age",
18         order : "desc"
19     }
20 ]);

```

We may want to see only those dogs who are at least three years old. For this, we need to use a search criterion, as shown in Listing 3.35.

Listing 3.35: Dogs at least three years old

```

1  var results = rpcjs.dbif.Entity.query(
2      // The class name of the entity type to be queried

```

```

3     "animals.ObjDogs",
4
5     // Search criteria.
6     {
7         type      : "element",
8         field     : "age",
9         value     : 3,
10        filterOp  : ">="
11    },
12
13    // Result criteria: none. We can pass null, or omit entirely.
14    null);

```

But Listing 3.35 includes poodles and chihuahuas, and we may not want them. Listing 3.36 shows how we would exclude those breeds from the results.

Listing 3.36: Dogs at least three years old, no poodles or chihuahuas

```

1  var results = rpcjs.dbif.Entity.query(
2  // The class name of the entity type to be queried
3  "animals.ObjDogs",
4
5  // Search criteria. Age greater than 3 and
6  // no poodles and no chihuahuas
7  {
8      type      : "op",
9      method    : "and",
10     children  :
11     [
12         {
13             type      : "element",
14             field     : "age",
15             value     : 3,
16             filterOp  : ">="
17         },
18         {
19             type      : "element",
20             field     : "breed",
21             value     : "Poodle",
22             filterOp  : "!="
23         },
24         {
25             type      : "element",
26             field     : "breed",
27             value     : "Chihuahua",
28             filterOp  : "!="
29         }
30     ]
31 },
32
33 // Result criteria: none. We can pass null, or omit entirely.
34 null);

```

3.4.7.9 App Engine database driver

Google's App Engine provides a few language environments in which applications may run: Python, Go, and Java. The App Engine database interface of **LIBERATED**

makes use of the Java environment. The App Engine Java library classes can be directly accessed and instantiated from within a JavaScript program.

The Java package library is available to JavaScript applications via the `Packages` namespace, and referenced from within that namespace by the Java package's normal fully-namespaced identification. The App Engine datastore service factory class, for example, is referred to in Java as `com.google.appengine.api.datastore.DatastoreServiceFactory`. That same Java class can be accessed from a JavaScript program by prefixing that name with `Packages.`, for example, `Packages.com.google.appengine.api.datastore.DatastoreServiceFactory`.

The **import** mechanism in Java, whereby a package may be imported and then referenced simply by the last component of its fully-namespaced name, does not exist in the JavaScript environment. As a common replacement, a variable can be used to hold a namespace (which is really a tree composed of JavaScript maps), to create a shorthand reference to the class. For example, a shorthand reference to the `DatastoreServiceFactory` class shown above could be created as:

```
var DatastoreServiceFactory =
    Packages.com.google.appengine.api.datastore.DatastoreServiceFactory;
```

Methods of the `DatastoreServiceFactory` could then be accessed using the shorthand reference:

```
var datastoreService = DatastoreServiceFactory.getDatastoreService();
```

We'll now discuss the implementations of the six functions required of a database driver. The code is shown in its entirety in Appendix 11. Portions are excerpted here, as needed.

3.4.7.9.1 query

The query function is passed the name of the class to be searched. Recall that entity types have two name forms: their class name, and a short entity type name, and that `rpcjs.dbif.Entity.entityTypeMap` provides a mapping from the class name

to the short entity type name. The `query()` method first converts from class name to entity type name, using that map. It also retrieves a list of the registered data property names for this entity type, from the `rpcjs.dbif.Entity.propertyTypes` map, for easy access throughout the code.

Listing 3.37 shows how the search criteria are then processed. First, at lines 90–91, we retrieve an App Engine datastore service object with which we can query for and retrieve datastore elements, write to the datastore, etc.

Listing 3.37: App Engine search criteria processing: specified key

```

90     Datastore = Packages.com.google.appengine.api.datastore;
91     datastore = Datastore.DatastoreServiceFactory.getDatastoreService();
92
93     // If we've been given a key (single field or composite), just look up
94     // that single entity and return it.
95     switch(qx.lang.Type.getClass(searchCriteria))
96     {
97     case "Array":
98         // Build the composite key
99         searchCriteria =
100             rpcjs.appengine.Dbif._buildCompositeKey(searchCriteria);
101
102         // fall through
103
104     case "Number":
105     case "String":
106         // We've been given a key. Try to retrieve the specified entity.
107         try
108         {
109             result =
110                 datastore.get(Datastore.KeyFactory.createKey(type, searchCriteria));
111         }
112         catch(e)
113         {
114             // Entity not found
115             return [];
116         }
117
118         dbResults = [];
119         dbResults.push(result);
120
121         // Make dbResults look like a Java array, so we can convert its
122         // contents to JavaScript types in code which is common with a Query
123         // result set.
124         dbResults.hasNext = function() { return this.length > 0; };
125         dbResults.next = function() { return this.shift(); };
126         break;

```

A query is accomplished most efficiently when the search criterion is a key value of the entity type, identifying a specific entity. Beginning at line 95, we determine what type of search criteria have been provided. The types that we could have received are: an array, indicating that we've been provided with the values to create a composite

key; a number or string, indicating a unique key value; or a normal `searchCriteria` object tree, with a complete query.

The code, then, first checks to see if the provided criteria is an array, and if so, creates a composite key based on the values within the array by calling, in lines 99–100, the static method, `rpcjs.appengine.Dbif._buildCompositeKey()`. This function, by default, concatenates the values in the array into a string, with each of the individual values converted to strings (if necessary), and separating each string value with an ASCII 31, referred to as the **ASCII Unit Separator**. This static function may be replaced on a per-application basis, to build composite keys as desired. The default function is shown in Listing 3.38.

Listing 3.38: Default function to build a default key

```

35     _buildCompositeKey : function(keyArr)
36     {
37         return keyArr.join(String.fromCharCode(31));
38     },

```

Referring back to Listing 3.37, whether the criteria type was an array, in which case a composite key was built, or it is found to be a number or string, the key value is used, as shown at lines 109–110, to locate the specific object identified by that key. If the key was not found, line 115 returns an empty array. Otherwise, the (one and only) result is pushed onto an array.

A normal query, as we shall soon see, returns a Java iterator containing the results, and there is code that iterates that result set to process any `resultCriteria` that are provided and produce the final result array. In order to make common use of that code, lines 124–125 make the `dbResults` array look like a Java iterator, by adding `hasNext()` and `next()` methods to it.

Normal queries are created, in App Engine, by limiting the full set of objects of a particular entity type by applying **filters** to the query prior to actually issuing the query. The code beginning at line 128, shown in Listing 3.39, handles such a normal query.

Listing 3.39: App Engine search criteria processing: query

```

128     default:
129         // Create a new query
130         Query = Datastore.Query;
131         query = new Query(type);
132
133         // If they're not asking for all objects, build a criteria predicate.
134         if (searchCriteria)
135         {
136             (function(criterion)
137             {
138                 var                filterOp;
139
140                 switch(criterion.type)
141                 {
142                     case "op":
143                         switch(criterion.method)
144                         {
145                             case "and":
146                                 // Generate the conditions specified in the children
147                                 criterion.children.forEach(arguments.callee);
148                                 break;
149
150                             default:
151                                 throw new Error("Unrecognized criterion method: " +
152                                                 criterion.method);
153                         }
154                         break;
155
156                     case "element":
157                         // Map the specified filter operator to the db's filter ops.
158                         filterOp = criterion.filterOp || "=";
159                         switch(filterOp)
160                         {
161                             case "<=":
162                                 filterOp = Query.FilterOperator.LESS_THAN_OR_EQUAL;
163                                 break;
164
165                             case "<":
166                                 filterOp = Query.FilterOperator.LESS_THAN;
167                                 break;
168
169                             case "=":
170                                 filterOp = Query.FilterOperator.EQUAL;
171                                 break;
172
173                             case ">":
174                                 filterOp = Query.FilterOperator.GREATER_THAN;
175                                 break;
176
177                             case ">=":
178                                 filterOp = Query.FilterOperator.GREATER_THAN_OR_EQUAL;
179                                 break;
180
181                             case "!=":
182                                 filterOp = Query.FilterOperator.NOT_EQUAL;
183                                 break;
184
185                             default:
186                                 throw new Error("Unrecognized comparison operation: " +
187                                                 criterion.filterOp);
188                         }
189
190                         // Add a filter using the provided parameters
191                         if (fields[criterion.field] == "Integer" ||
192                             fields[criterion.field] == "Key")
193                     {

```

```

194         query.addFilter(criterion.field,
195                         filterOp,
196                         java.lang.Integer(criterion.value));
197     }
198     else
199     {
200         query.addFilter(criterion.field,
201                         filterOp,
202                         criterion.value);
203     }
204     break;
205
206     default:
207         throw new Error("Unrceognized criterion type: " +
208                         criterion.type);
209     }
210 })(searchCriteria);
211 }

```

Lines 130–131 create a new query, using the App Engine `Query` class. A recursive function is then called, with the search criteria as its parameter, to create the query filters. If a type “op” node is found in the search criteria, the function is recursively called for each of its children, as shown at line 147. For type “element” nodes, the code first ascertains the filter operator to be used, in lines 158–188. The Java query object’s `addFilter()` method is called. Regardless of the data property type being filtered on, the first two parameters are the data property, or field name being filtered, and the filter operator that was previously ascertained. The third parameter depends on the data property type. It requires that a Java type be passed. Strings are automatically converted, but JavaScript integers must be manually converted to Java integers before being passed. Line 196 shows this.²⁰

With the search criteria fully established, result criteria can now be processed, as shown in Listing 3.40. App Engine provides default **fetch options**, which are retrieved with a `Builder` class, as shown at line 215.

Listing 3.40: App Engine result criteria processing

```

218     if (resultCriteria)
219     {
220         // ... then go through the criteria list and handle each.
221         resultCriteria.forEach(
222             function(criterion)
223             {
224                 switch(criterion.type)
225                 {
226                     case "limit":
227                         options.limit(criterion.value);

```

²⁰The namespace `java` is an automatic shorthand for `Packages.java`.

```

228         break;
229
230     case "offset":
231         options.offset(criterion.value);
232         break;
233
234     case "sort":
235         query.addSort(criterion.field,
236                       {
237                         "asc" : Query.SortDirection.ASCENDING,
238                         "desc" : Query.SortDirection.DESENDING
239                       }[criterion.order]);
240         break;
241
242     default:
243         throw new Error("Unrecognized result criterion type: " +
244                         criterion.type);
245     }
246     });
247 }

```

The result criteria array is processed in array order, in lines 221–247. The options returned by the `Builder` may be amended by setting limit or offset values, as shown in lines 227 and 231. When `sort` options are provided in the result criteria, these are added directly to the query object, as shown in lines 235–239.

At this point, the query can be issued. With App Engine, the query is “prepared” and then there are various methods of issuing the query that specify how the results should be returned. In this case, the method that returns the results as a Java iterator is used. This is shown in Listing 3.41.

Listing 3.41: Preparing and issuing a query to App Engine

```

249     // Prepare to issue a query
250     preparedQuery = datastore.prepare(query);
251
252     // Issue the query
253     dbResults = preparedQuery.asIterator(options);

```

The results of the query are then iterated, converting each of the Java result types into their JavaScript equivalents. Each Java string is converted to a JavaScript string, each Java number is converted to a JavaScript number, and arrays of each type are properly adjusted as well.

Once the results are in proper form, the result array is returned to the caller.

3.4.7.9.2 put

The `put()` method accepts an entity as its parameter. The data to be stored in the database is in the entity object's `data` property, and the key field is retrieved from the object's `entityKeyProperty` property. Listing 3.42 shows how this information is retrieved from the entity object.

Listing 3.42: Retrieving entity properties

```

346     var          entityData = entity.getData();
347     var          keyProperty = entity.getEntityKeyProperty();
348     var          type = entity.getEntityType();

```

Listing 3.43 shows how the key value is ascertained (if one has yet been created or specified). When the key is composite, the array of field data composing the key is built, as shown in lines 368–372, and then the composite key is generated by calling the `_buildCompositeKey()` method discussed in Section 3.4.7.9.1. When the key is not composite, the key is retrieved directly from the entity data, at line 378.

Listing 3.43: Determining the key value

```

364     // Are we working with a composite key?
365     if (qx.lang.Type.getClass(keyProperty) == "Array")
366     {
367         // Yup. Build the composite key from these fields
368         keyProperty.forEach(
369             function(fieldName)
370             {
371                 keyFields.push(entityData[fieldName]);
372             });
373         key = rpcjs.appengine.Dbif._buildCompositeKey(keyFields);
374     }
375     else
376     {
377         // Retrieve the (single field) key
378         key = entityData[keyProperty];
379     }

```

An App Engine key is generated from a user-provided key, or can be automatically generated. Allocation of automatically generated keys (uid values) must be handled carefully in the App Engine environment, because there can be multiple instances of the backend code running concurrently. In order to ensure that an automatically-generated key value is allocated atomically on a global basis, a special method must be used to allocate the key: the database service's `allocateIds()` method. This method

takes two parameters: the type of the entity being created,²¹ and the number of id values to allocate, and returns a range of id values. Once this service has allocated and returned id values, it is guaranteed that those same id values will not be used for this same entity type by this or any other backend instance.

Listing 3.44: Automatically generating unique keys

```

404     switch(fields[keyProperty])
405     {
406     case "Key":
407     case "Number":
408         // Obtain a unique key that no other running instances will obtain.
409         keyRange =
410             datastoreService.allocateIds(entity.getEntityType(), 1);
411
412         // Get its numeric value
413         dbKey = keyRange.getStart();
414         key = dbKey.getId();
415         break;
416
417     case "String":
418         // Obtain a unique key that no other running instances will obtain.
419         keyRange =
420             datastoreService.allocateIds(entity.getEntityType(), 1);
421
422         // Get its numeric value
423         dbKey = keyRange.getStart();
424         key = dbKey.getName();
425         break;
426
427     default:
428         throw new Error("No way to autogenerate key");
429     }

```

Allocation of unique id keys is shown in Listing 3.44. Lines 409–410 and 419–420 allocate a range of id values, but request the range be of size one. Lines 413 and 423 obtain the first (only) key returned. The code then differs depending on whether the key property is numeric or a string. There are two ways of representing and storing a key: by its id, a number; or by its name, a string. If the key property is numeric, then, the key is retrieved by its id in order to save a number in the key field. If the key property must be a string, the key is retrieved by its name.

If the key was not automatically generated, we must create an App Engine key from the provided key. This is shown in Listing 3.45. The entity type and the provided key are passed as parameters, and an App Engine key is returned.

²¹optionally modified to indicate a parent entity


```

506         conv = function(val)
507         {
508             return val;
509         };
510     }
511
512     jArr = new java.util.ArrayList();
513     for (i = 0; value && i < value.length; i++)
514     {
515         jArr.add(arguments.callee(conv(value[i]),
516                                 type.replace(/Array/, "")));
517     }
518     return jArr;
519
520     default:
521         throw new Error("Unknown property type: " + type);
522     }
523     })(entityData[fieldName], fields[fieldName]);
524
525     // Save this result
526     dbEntity.setProperty(fieldName, data);
527 }
528
529 // Save it to the database
530 datastoreService.put(dbEntity);

```

This code loops through each of the fields to be stored. An anonymous function, defined beginning at line 468, is called with each of the field names and values to be stored. (The parameters are passed to the function at line 523.) Some values such as strings and floating point numbers can be placed directly in an App Engine entity property. Lines 477–479 handle this case. In all other cases, the data value must be manipulated.

Since there is not a clear distinction between integer and floating point values in JavaScript, integer values are explicitly converted to Java `java.lang.Long` objects, as shown at line 484. Non-null “LongString” values are converted to App Engine’s `Text` type, at lines 487–488.

All of the array types must recursively do the same sort of conversion. In addition, the arrays themselves must be converted from JavaScript arrays to Java arrays. This latter step occurs in lines 512–517.

Once a value has been converted as necessary to its Java counterpart type, the App Engine entity’s property corresponding to that field’s name is set, at line 526. After all fields have been stored as properties of the App Engine entity, the App Engine entity is saved to the database with a call to the database service’s `put()` method, shown at line 530.

3.4.7.9.3 remove

The `remove()` method accepts an entity as its only parameter. The App Engine key is ascertained in essentially the same way as was shown in Listing 3.43, except that it is expected that a key value is provided in the parameter entity.

After the key is determined, the datastore service's `delete()` method must be called, but as shown in Listing 3.47, this is complicated by the fact that `delete` is a reserved word in JavaScript.

Listing 3.47: Deleting an entity from App Engine

```
574 // Remove this entity from the database
575 datastore = Datastore.DatastoreServiceFactory.getDatastoreService();
576 datastore["delete"](dbKey);
```

If this method had had another name such as `remove` then line 576 could simply have been

```
datastore.remove(dbKey);
```

but we can't do that with a method name of `delete`. Instead, we use JavaScript's array-like object referencing, which allows using any string value as the member name. In this case, it is the string "delete."

3.4.7.9.4 getBlob, putBlob, and removeBlob

The methods `getBlob()`, `putBlob()`, and `removeBlob()` make use of the App Engine Java API in a fashion similar to the `query()`, `put()`, and `remove()` methods. Although different APIs are used, the issues are the same, principally, that the data stored in the database is of Java types, and must be converted back and forth to and from JavaScript types. The interface is further complicated by size limitations on how much data can be read from a blob at a single time. The `getBlob()` function therefore has a loop to retrieve portions of large blobs at a time until the entire blob has been retrieved. The reader interested in further details should refer to Appendix 11.

3.4.7.10 Simulation database driver

The simulation database driver has many fewer complications than has the App Engine driver. There is no need to be converting back and forth to Java types, and no unusual limitations based on any API. The simulation database driver, however, need not just access some existing database API; rather it must implement the entirety of the database functionality itself. The full code is shown in Appendix 12.

The implementation is kept fairly simple. The database is modeled as a JavaScript map, where each entity type is a top-level entry in that map. The entity type itself is the member name, and the member value for each entity type is itself a map, where the member names are the key values for entities of that type, and the member values are maps containing each of the data properties of the entity with that key.

In Section 3.4.7.8, an example entity type subclass was shown, for the maintenance of information about dogs. Assume now that there is a similar entity type for cats. Having added each of the example dogs from that section, plus some cats, the simulation database might look something like this:²²

```
{
  "dogs" :
  {
    "Lipman+Derrell+Stasha" :
    {
      surname   : "Lipman",
      givenName : "Derrell",
      dogName   : "Stasha",
      breed     : "Siberian Husky",
      age       : 7
    },
    "Marks+Joel+Snoopy" :
    {
      surname   : "Marks",
      givenName : "Joel",
      dogName   : "Snoopy",
      breed     : "Purebreed Mutt",
      age       : 2
    },
  },
}
```

²²For the purpose of clarity, the composite key here uses a plus sign between the field values which are concatenated to form the composite key, instead of ASCII 31 as is actually used by default in both the App Engine database driver and the simulation database driver.

```

    "Binks+Jar Jar+Curly" :
    {
        surname    : "Binks",
        givenName  : "Jar Jar",
        dogName    : "Curly",
        breed      : "Poodle",
        age        : 11
    }
},

"cats" :
{
    "Rosenbaum-Lipman+Shelley+Darth Vader" :
    {
        surname    : "Rosenbaum-Lipman",
        givenName  : "Shelley",
        catName    : "Darth Vader",
        breed      : "Domestic Shorthair",
        trait      : "Vicious",
        age        : 15
    },

    "Axelrod+Fran+Tootsie" :
    {
        surname    : "Axelrod",
        givenName  : "Fran",
        catName    : "Tootsie",
        breed      : "Rag doll",
        trait      : "Lap cat",
        age        : 9
    },

    "Glass+Gil+Sunshine" :
    {
        surname    : "Glass",
        givenName  : "Gil",
        catName    : "Sunshine",
        breed      : "Siamese",
        trait      : "Receives signals from home planet.",
        age        : 3
    }
}
}

```

The simulation database map's initial state is provided to the static method `rpcjs.sim.Dbif.setDb()` which stores it in static variable `rpcjs.sim.Dbif.Database`.

Whenever needed, the database (map) is then accessed via this static variable.

When entities are written to this database, if the browser supports the HTML5 **localStorage** feature, the database is serialized and written to the `simDB` member of local storage. At the application's option, it can read and deserialize the stored database upon startup, and pass the resulting map to `rpcjs.sim.Dbif.setDb()`, which provides a poor-mans persistent database that works with many modern browsers.

We shall now examine the implementation of this simulation database driver. As with the App Engine database driver, the simulation database driver implements the six required functions: `query()`, `put()`, `remove()`, `getBlob()`, `putBlob()`, and `removeBlob()`.

3.4.7.10.1 query

Listing 3.48 shows the initialization phase of the `query()` method. It first ascertains its entity type from the provided class name, at line 93. That entity type is used to locate the entity-type-specific portion of the database, at line 100. In the dogs and cats example, the provided class name would have been `animals.ObjDogs` or `animals.ObjCats`, resulting in an entity type of either “dogs” or “cats”, and `dbObjectMap` referencing either the dogs map or the cats map from the database.

Listing 3.48: Initializing the environment for a query

```

92     // Get the entity type
93     type = rpcjs.dbif.Entity.entityTypeMap[classname];
94     if (! type)
95     {
96         throw new Error("No mapped entity type for " + classname);
97     }
98
99     // Get the database sub-section for the specified classname/type
100    dbObjectMap = rpcjs.sim.Dbif.Database[type];
101
102    if (qx.core.Environment.get("qx.debug"))
103    {
104        if (! dbObjectMap)
105        {
106            throw new Error("Type '" + type + "' " +
107                "was not found in the simulation database.");
108        }
109    }
110
111    // Initialize our results array
112    results = [];

```

The results array is initialized at line 112. Results will be deep copies of data from the database, not references into the database, allowing the caller to manipulate the data at will without fear of the database's data being altered.

In Listing 3.49, direct by-key access to a specific entity is handled. First, if the search criteria indicate that composite key fields are provided, a composite key is generated using the static method `rpcjs.sim.Dbif._buildCompositeKey()`. As with the App Engine database driver, this method concatenates the fields with an ASCII 31 separator. The method may be replaced by the application if a different composite key function is preferred.

Listing 3.49: Handling by-key queries

```

114     // If we've been given a key (single field or composite), just look up
115     // that single entity and return it.
116     switch(qx.lang.Type.getClass(searchCriteria))
117     {
118     case "Array":
119         // Join the field values using a known field separator
120         searchCriteria = rpcjs.sim.Dbif._buildCompositeKey(searchCriteria);
121
122         // fall through
123
124     case "Number":
125     case "String":
126         if (typeof dbObjectMap[searchCriteria] !== "undefined")
127         {
128             // Make a deep copy of the results
129             result =
130                 qx.util.Serializer.toNativeObject(dbObjectMap[searchCriteria]);
131             results.push(result);
132         }
133     return results;
134 }

```

Given a number or string key, or the string built from the values composing a composite key, line 126 determines whether an object with this key exists, and if so, lines 129–131 make a deep copy of the object and add it to the result set. Since the key of a specific entity was provided as the search criteria, the caller is requesting only a single object, so the query is complete. Line 133 returns the array of results which contains, as its only element, the requested entity.

There are two other search criteria cases to be handled. If no search criteria were provided, then all entities of the specified entity type are to be returned in the result set. If a search criteria map was provided, we must locate the particular entities that


```

186     }
187   }
188   ret += ")";
189   break;
190
191   default:
192     throw new Error("Unrecognized criterion method: " +
193                   criterion.method);
194   }
195   break;
196
197   case "element":
198     // Determine the type of this field
199     propertyTypes = rpcjs.dbif.Entity.propertyTypes;
200     switch(propertyTypes[type].fields[criterion.field])
201     {
202     case "String":
203     case "LongString":
204     case "Date":
205       if (typeof criterion.value != "string")
206       {
207         qx.Bootstrap.warn(
208           "Expected criterion value to be string, " +
209           "got " + typeof(criterion.value));
210         ret += "false";
211       }
212     else
213     {
214       ret +=
215         "entry[\"" + criterion.field + "\"] " + filterOp +
216         "\"\" + criterion.value + \"\" ";
217     }
218     break;
219
220     case "Key":
221     case "Integer":
222     case "Float":
223       if (typeof criterion.value != "number")
224       {
225         qx.Bootstrap.warn(
226           "Expected criterion value to be number, " +
227           "got " + typeof(criterion.value));
228         ret += "false";
229       }
230     else
231     {
232       ret +=
233         "entry[\"" + criterion.field + "\"] " + filterOp +
234         criterion.value;
235     }
236     break;
237
238     case "KeyArray":
239     case "StringArray":
240     case "LongStringArray":
241       if (typeof criterion.value != "string")
242       {
243         qx.Bootstrap.warn(
244           "Expected criterion value to be string, " +
245           "got " + typeof(criterion.value));
246         ret += "false";
247       }
248     else if (criterion.filterOp)
249     {
250       qx.Bootstrap.warn(
251         "Filter operations can not be applied to array types");
252     }
253     else

```

```

254     {
255         ret +=
256             "qx.lang.Array.contains(entry[\"" +
257                 criterion.field + "\", " +
258                 "\"" + criterion.value + "\"]");
259     }
260     break;
261
262     case "IntegerArray":
263     case "FloatArray":
264         if (typeof criterion.value !== "number")
265             {
266                 qx.Bootstrap.warn(
267                     "Expected criterion value to be string, " +
268                     "got " + typeof(criterion.value));
269                 ret += "false";
270             }
271         else if (criterion.filterOp)
272             {
273                 qx.Bootstrap.warn(
274                     "Filter operations can not be applied to array types");
275             }
276         else
277             {
278                 ret +=
279                     "qx.lang.Array.contains(entry[\"" +
280                         criterion.field + "\", " + criterion.value + "\"]");
281             }
282         break;
283
284     default:
285         throw new Error("Unknown property type: " + type);
286     }
287     break;
288
289     default:
290         throw new Error("Unrecognized criterion type: " +
291             criterion.type);
292     }
293
294     return ret;
295 })(searchCriteria);
296
297 // Create a function that implements the specified criteria
298 qualifies = new Function(
299     "entry",
300     "return (" + builtCriteria + ")");
301 }
302 else
303 {
304     // They want all entities of the specified type.
305     qualifies = function(entity) { return true; };
306 }

```

An anonymous recursive function, whose definition runs from line 140–295, is called, initially, with the top (root) criterion of the search criteria tree. The function initializes the variable `ret`, at line 143, to an empty string, and will add to this string the filter indicated by the criterion provided in the current recursive instance of the function. The filter is added to this string in JavaScript source syntax. Each recursive call returns the string that it built, to be incorporated into the result being built by

the caller instance of the function. This allows the recursive operation to process the entire criteria tree, to generate a sophisticated filter in JavaScript source syntax. Once the entire search criteria map has been traversed, and the top-level instance of the function returns, the entire generated JavaScript source string is passed to the `Function` constructor to create a function that can be called on each entity of the specified type. Let's look first at how the generated function will be called, which will make the code within the generated function easier to understand when we examine it.

Lines 298–300 take the text string that our recursive function generated and turn it into a `Function` object, i.e., a callable function. The string built by our recursive function was stored in the variable, `builtCriteria`. That string contains a boolean expression. What our new function should do, then, is return the result of processing the boolean expression, given some entity as input. The parameter name is given as “entry” on line 299. As we examine the implementation of the function that builds the boolean expression, then, we will see that the entity that the function is handling will be referred to in the generated string, as `entry`.

Line 140 shows that a single criterion is passed to this function. Lines 148–170 show how a filter operation is converted to its JavaScript source syntax equivalent. If the `filterOp` value in the criterion is “=” or if no `filterOp` member is present in the criterion at all, the JavaScript `identical` comparator, `===`, is returned (line 164). The `not-equal` filter operator, “!=", is converted to the JavaScript `not identical` operator, `!==` (line 160). The other filter operators are already in JavaScript source syntax, so are returned as is (line 157). The resulting string is stored in the `filterOp` variable for future use.

The criterion's type is then examined. If an operation criterion is found (type is “op”) and its method is “and” then the `ret` string is built recursively. First, line 179 adds an initial left parenthesis to ensure that precedence of operators does not adversely affect the requested operation. The function is then recursively called with each child criterion in turn (line 182). After return from the recursive call, if there are more children, the JavaScript conjunction operator, `&&` is appended before the

next recursive call. Finally, after all children have been processed, the trailing right parenthesis is added at line 188, to match with the left parenthesis that was added initially.

Let's consider an "element" criterion. In this case, the type of the specified field must be determined. This is done at lines 199–200. The registered property types for this entity type are accessed, and the `fields` member is a map of fields and types. This map is then indexed by the field name specified in the criterion, to find the data property type for the specified field.

Lines 202–218 handle the case of a string type. The desired JavaScript filter, then, is a comparison between a provided entity's value and the value specified in the criterion. Recall that the entity passed to this instance of the function will be in a parameter variable called `entry` so the comparison is against some field of the entity `entry`. The filter operator, in JavaScript source syntax, was determined previously. The required comparison, then, is specified by lines 215–216. The criterion's value is surrounded by quotes since in this case, the data property type is some sort of string.

The remainder of data property types are handled in the subsequent cases. The return strings are generated similarly to the String case.

Once the full filter string has been built, it is turned into a function, as previously mentioned. The function is referenced by the variable `qualifies`. In the case where no search criteria were provided, that indicates that all entities of the specified type are requested, so a simple function that always returns `true` is created, and its reference stored in `qualifies`.

In a manner similar to the creation of a function to handle the search criteria, any sort criteria within the result criteria map must be processed. For sort criteria, we wish to be able to pass the result set array to the JavaScript `sort()` method. That method allows a function to be provided, that determines the relationship between two provided elements from the array. The function must return a value less than zero, equal to zero, or greater than zero, depending upon whether its first parameter is deemed to be less than, equal to, or greater than its second parameter, respectively. This function is built in essentially the same way as the function for search criteria

was built, so is not shown here. The resulting sort function is stored in the variable `sortFunction`, which was initialized to null to indicate no requested sort order.

As the result criteria array is iterated, if an “offset” or “limit” entry is found, its value is saved.

With a function to determine which entities qualify for inclusion in the result set, a sort function, and limit and offset values in hand, the entities of the specified entity type can now be iterated in search of ones which qualify. This is shown in Listing 3.51.

Listing 3.51: Selecting and returning queried entities

```

370     for (entry in dbObjectMap)
371     {
372         if (qualifies(dbObjectMap[entry]))
373         {
374             // Make a deep copy of the results
375             result = qx.util.Serializer.toObject(dbObjectMap[entry]);
376             results.push(result);
377         }
378     }
379
380     // Sort the results
381     if (sortFunction)
382     {
383         results.sort(sortFunction);
384     }
385
386     // Give 'em the query results, with the appropriate offset and limit.
387     return results.slice(offset, offset + limit);

```

The variable `dbObjectMap` was previously initialized to the map of entities of the requested entity type. Line 372 calls the qualification function that had been built based on the search criteria, to see if the currently-iterated entity qualifies for inclusion in the result set. If it does, a deep copy of the entity is created at line 375, which is then added to the result set.

After building the entire result set, if a sort function was specified, the results are sorted (line 383). Finally, any specified offset and limit are used to reduce the returned set to only those requested. (The offset and limit variables were initialized to include all results, and only altered if offset or limit entries were found in the result criteria.)

3.4.7.10.2 put

The `put()` method begins similarly to its App Engine database driver counterpart, determining the key of the entity to be written. That code was shown in

Listing 3.43 on page 86. Unlike its App Engine driver sibling, however, there are assumed to be no multiple instances of the simulation database running concurrently, so the next automatically-generated unique value is simply stored in static variable `rpcjs.sim.Dbif.__nextKey`. Each time a unique id is required, the current value of this variable is used, and this variable is incremented in preparation for the next required id value.

Listing 3.52: Writing entity data to the simulated database

```

475     // Create a simple map of properties and values to be put in the
476     // database
477     for (propertyName in entity.getDatabaseProperties().fields)
478     {
479         // Add this property value to the data to be saved to the database.
480         data[propertyName] = entityData[propertyName];
481     }
482
483     // Save it to the database
484     rpcjs.sim.Dbif.Database[type][key] = data;
485
486     // Write it to Web Storage
487     if (typeof window.localStorage !== "undefined")
488     {
489         qx.Bootstrap.debug("Writing DB to Web Storage");
490         localStorage.simDB = qx.lang.Json.stringify(rpcjs.sim.Dbif.Database);
491     }

```

Once a key value is obtained, either from the provided entity data or automatically generated, the entity data to be saved is built. This is shown in Listing 3.52. The list of properties for this entity is iterated by the loop starting at line 477, and the data property values to be stored are copied from the provided entity's data. The entity type and key are then used to index into the database, and store the data object, as shown at line 484. Finally, if the browser supports the HTML5 **localStorage** facility, the database is serialized and written to disk.

3.4.7.10.3 remove

The `remove()` method begins similarly to `put()`, determining the key value. It concludes, as shown in Listing 3.53, by deleting the specified object from the database map, and writing the database to disk if possible.

Listing 3.53: Deleting an entity from the simulated database

```
528     delete rpcjs.sim.Dbif.Database[type][key];
529
530     // Write it to Web Storage
531     if (typeof window.localStorage !== "undefined")
532     {
533         qx.Bootstrap.debug("Writing DB to Web Storage");
534         localStorage.simDB = qx.lang.Json.stringify(rpcjs.sim.Dbif.Database);
535     }
```

3.4.7.10.4 getBlob, putBlob, and removeBlob

The `getBlob()`, `putBlob()`, and `removeBlob()` methods work similarly to what we have just discussed. They are stored as a special entity type called “**BLOB**”, and may be retrieved via their key value which is returned by `putBlob()`.

Results:

Testing the reference implementation

Chapter 4

App Inventor Community Gallery: An example application

LIBERATED is intended to allow a developer to write various types of applications based on its architecture. In this chapter, we shall look at an application, *App Inventor Community Gallery* (AICG), which, although still a work in progress, is beginning to see use at a few universities around the country. The App Inventor Community Gallery has been built upon **LIBERATED**. We will review, here, the overall design of the App Inventor Community Gallery backend. The frontend design will not be discussed, as that is outside the scope of this thesis. Nearly the entire backend is included in Appendices 13 – 35. The only thing not included in its entirety is the simulation database. Appendix 36 contains a very stripped-down version of the database, that shows each of the entity types, but excludes the majority of the data objects, and truncates long image data.

Unlike **LIBERATED** which was implemented almost exclusively by me, the App Inventor Community Gallery has been a team effort, with a number of people's code contributing to the application.

4.1 Background

The App Inventor Community Gallery is a site for sharing of programs written using App Inventor.¹ App Inventor is a programming environment developed by engineers from MIT and Google, in which blocks or puzzle pieces are moved around on the screen and interconnected to create programs, in a manner similar to using Scratch² or LEGO Mindstorms.³ Whereas Scratch programs allow designing animation, games, and stories that run on a typical home computer, and Mindstorms programs control LEGO robots, App Inventor allows writing programs to run on Android-based phones. App Inventor is proving to be popular in the education community, for introductory programming courses for ages ranging from middle-school through college.

A goal of the App Inventor Community Gallery is to allow sharing of these App Inventor programs. Ultimately, it is envisioned that in addition to complete programs, libraries and individual components will be shared, allowing developers to mix and match their own code with code available in the gallery.

4.2 Destination: Google App Engine

App Inventor Community Gallery’s “real server” environment is Google App Engine. Google App Engine allows web applications to run on Google’s infrastructure. As previously mentioned, applications can be written using a number of supported languages: Java, Python, and Go [28].

App Engine’s datastore is object-based, and maps well from the database abstraction of **LIBERATED**. A key difference is that the datastore provided by App Engine is **schemaless**, i.e., there is no requirement that each object of a certain type contain the same properties. The **LIBERATED** database driver for App Engine, however, imposes its schema (registered properties and types) requirements over App Engine’s datastore, so in fact each object of a given type does end up containing the same

¹<http://www.appinventorbeta.com/about>

²<http://scratch.mit.edu>

³<http://mindstorms.lego.com/en-us/Default.aspx>

properties.

4.3 Remote procedure calls

The backend of App Inventor Community Gallery provides JSON-RPC remote procedure call implementations, which query and manipulate a database using the database abstraction of **LIBERATED**. The remote procedure calls are implemented in mixins so that related RPCs can be grouped appropriately. The App Inventor Community Gallery is still in development, but at present, there are mixins for remote procedure calls related to:

applications

Information about each application which has been uploaded to the gallery (Appendix 19)

comments

User-provided comments about applications which have been uploaded to the gallery (Appendix 20)

flags

Indications that users have said that something inappropriate has been uploaded, or an inappropriate comment was made about an application (Appendix 21)

liking

Appreciation of a particular application, as indicated by a user pressing the “Like It!” button. (Appendix 22)

mobile

Special functions for access by the mobile client.⁴ (Appendix 23)

search

Functions to browse or search for applications. (Appendix 24)

⁴The mobile client is an Android phone-based client to access much of the data from the App Inventor Community Gallery. The mobile client requires some massaging of the data returned to it, which is accomplished by the methods in this mixin. <http://www.appinventor.org/mobile-gallery>

tags

Groupings of applications. (Appendix 25)

visitors

The users of the site, and any site administration permissions they have been granted. (Appendix 26)

whoami

Identification of the logged-in user. (Appendix 27)

4.4 Database schema

The remote procedure calls we just discussed search and manipulate the database. The database is composed of a number of entity types, described briefly here. Each entity type is in a file and class prefixed with “Obj” to indicate that it is a database object. Following are descriptions of each:

ObjAppData

An application that has been uploaded to the gallery. (Appendix 28)

ObjComments

A specific comment on a specific application. (Appendix 29)

ObjDownloads

Record of which applications each visitor has downloaded, and when. (Appendix 30)

ObjFlags

Detail of which applications or comments have been flagged, by whom, and when. (Appendix 31)

ObjLikes

Mapping of application and visitor, indicating that that visitor has already *liked* a certain application. This prevents a single visitor from running up the *liked* count on an application. (Appendix 32)

ObjSearch

There is one object of this type for each tuple of word, application, and field (title, description, or tags) in which a word was found. This allows searching for applications based on words that have been entered in each of those fields. (Appendix 33)

ObjTags

The values of tags which have either been entered by users when uploading their applications, or which are in a designated set of **category** tags from which all applications must select at least one. This object type demonstrates use of the canonical value property. In addition to a *value* property, the tag value entered by the user (which may be in mixed case), an additional *value_lc* property is added to each object automatically, containing a canonicalized value. The canonicalization function converts the user-provided value to lower case. (Appendix 34)

ObjVisitors

Information about each visitor (user) of the site is maintained here. This includes the visitor's account information, credentials, and (for future use), searches the user has issued recently, and applications that have been viewed recently. (Appendix 35)

Discussion

Chapter 5

Application development benefits

One of the clear benefits of this new architecture is that key portions of debugging and testing, both during development and for regression identification, become much easier to handle than with traditional client/server applications. In this chapter, we will examine some techniques that are now available.

5.1 Debugging

In a traditional web application architecture, the frontend and backend must be initially debugged in isolation. The frontend and backend are likely written in different languages, may be developed by different teams, and may not even be able to run on the same machine. The interface between the frontend and backend applications is developed solely to a service API specification, and there is little ability for the frontend and backend to interact until both are nearly completed. Even once integration testing of the frontend and backend code begins, there is no easy way to use a single debugger to step through the code. In some environments, it may be possible to have a frontend debugger for watching the code running on the client, and a separate backend debugger, for watching the code running on the server. Other 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**, many of those problems

are reduced or eliminated. 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, allowing early and iterative debugging during the development process.

Of even more importance to the debugging process, the developer can now use a debugger running in the browser to step directly 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.

5.2 Automated tests

During development, it is common practice to write tests for small pieces of code or to test specific features or functionality. These tests are often called **unit tests**. In traditional client/server applications, unit tests would be written for the server side, to test the backend functionality. Separately, tests would be written for the client side, to test the GUI functionality. These are all valid tests, but neglect a key piece of functionality: the round-trip from client to server to client, via the application's selected means of communication between the frontend and backend, e.g., RPC, REST, etc.

When the entire code can be tested purely within the browser, as is the case with an application based on **LIBERATED**, it is feasible to add the round-trip to the set of tests. Not only can backend remote procedure call implementations, for example, be tested in isolation, they can be called by tests via the same remote procedure call invocation that the application would use, to ensure that they work correctly that way too.

Web application testing can be automated with the use of third-party packages such as Selenium,¹ which includes facilities for automating user interaction such as key presses and mouse clicks. Using such a package, in conjunction with a simulated

¹<http://seleniumhq.org>

server running in the browser, complete user actions can be tested automatically.

Applications written using qooxdoo also have a built-in unit test framework that is designed for complete regression testing. A unit test application allows selecting tests to be run, or selecting a complete suite. Once tests are written to test a certain piece of functionality, any change to the code that breaks that functionality will be immediately obvious when the unit tests are run. The test suite can be set up to run automatically, e.g., every evening, to quickly catch code regressions.

The App Inventor Community Gallery project has, to date, over fifty qooxdoo-based unit tests that demonstrate this capability. Remote procedure call implementations are tested directly, for unit-level accuracy. Additionally, there are tests in which the full remote procedure call path is confirmed to be working correctly. These tests issue a remote procedure call request in the same manner as the real application does, which creates the JSON-RPC request and sends it via the selected transport, initiating parsing of the request, which causes the appropriate RPC implementation to be called, which then similarly returns the result. There are not yet any tests that exercise GUI-initiated requests.

5.3 Debugging Experience

During the course of developing the App Inventor Community Gallery, 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 as might be done in a traditional client/server environment, new user interface features and any corresponding backend changes are developed and tested concurrently. When new code does not work as intended, a 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.
6. In the early days of **LIBERATED**, while bugs in the application communication protocol (JSON-RPC generation and parsing) were still being worked out, it was also occasionally useful to step from the frontend code that issued the remote procedure call, through the code that created the JSON-RPC request and queued the message for the simulated server, etc., all the way into the RPC implementation in the backend code. This would be a useful technique again, for an application which desired a different application communication protocol.

Chapter 6

Related work

Prior art related to the architecture proposed here appears to be split into several different camps, each of which will be discussed:

- Articles which discuss improved testing methodologies for web applications.
- The momentum gain of server-side JavaScript.
- Recent work on web standards which include database interfaces that are directly accessible from an application running in the browser.
- Projects intended to reduce the distinction between client and server.

I have been unable to find any literature or related projects which accomplish all of the goals of my research question identified in Chapter 2. Although there is work in progress on the various sub-pieces described here, there appears to be none that would allow an application to be written in a single language, debugged and tested completely in the browser, and allow debugged, tested code to be easily moved, unaltered, to the real server for production use.

6.1 Testing methodologies

The literature in the area of improving testing methodologies for web applications dates back to the early 2000s. Yang et al. discuss an object-oriented architecture for web application testing [29]. Two of the components they discuss, the Test

Management Subsystem (TMS) and the Test Development Subsystem (TDS) describe a means of recording a sequence of test results. Although they don't identify it as such, this describes an early version of today's regression testing suites such as described in Section 5.2.

Elbaum et al., nearly a decade ago, described a means of using session data to track interaction between the client and server [30]. The primary focus of this testing, though, is on ensuring that functional requirements are met, rather than ensuring correctness of the existing code.

Work being done at Tufts University by Professor Samuel Z. Guyer et al. involves implementing assertion testing in the Java garbage collector [31] and as a set of separate threads [32], allowing for very limited impact on program performance while still including contract validation even in production code. If or when a similar system is provided in the JavaScript system in browsers, test capabilities will be enhanced. Until such time, the present means of including in-line run-time assertions in the code, which are removed when the application is built for production (because of their performance impact), will have to suffice.

6.2 Server-side JavaScript

Section 3.2.1 described the three primary JavaScript engines in common use (V8, SpiderMonkey, and Rhino). Each of these engines 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 [33]. These products build on top of the engine to provide an interface that allows the user to develop server-side JavaScript code using the JavaScript engine.

Although most of the products use the three common JavaScript engines, other engines exist as well. The initial JavaScript engine developed for the WebKit-based browsers (e.g., Safari), JavaScriptCore, was rewritten as SquirrelFish, and rewritten again as SquirrelFish Extreme [34]. Each iteration provided increased speed.

Another option for server-side JavaScript is Microsoft's IIS. The ASP environ-

ment allows writing server-side code in multiple languages, including their version of JavaScript called JScript.

6.3 Web standard database interfaces

There has been ongoing work to create a standard database interface for local storage of data at the browser. One of these options could be used to implement a client-side simulated database for use in the **LIBERATED** environment. With a standard interface available at the browser, that same interface might be implementable at the server, allowing code to be tested at the browser, then moved to the server. At present, however, (a) there is no agreed-upon standard, and (b) there do not appear to be any server-side implementation of these proposed standards.

The two proposals for a browser database interface are *Web SQL Database* [35] and *IndexedDatabase API* [36], with parallel examples of each shown in [37].

6.3.1 Web SQL Database

Web SQL Database is a now-abandoned W3C Working Group project. It defines an API for storing data in databases that can be queried using a variant of SQL. It provides asynchronous and synchronous APIs, supports transactions, and limits SQL injection with parameterized queries.

Implementors in the Working Group all used the SQLite database, internally, which made it inappropriate for the standardization path, which requires multiple independent implementations.

Had Web SQL Database become standardized, it might have simplified the sharing of code between a real server and one running in the browser. Database-specific driver code could be used, such that the application would talk either to a Web SQL Database, when running at the browser, or to a server-side SQL database. Used in conjunction with a server-side JavaScript environment, this would be an alternative to the database abstraction discussed in this thesis.

6.3.2 Indexed Database API

The Indexed Database API provides a programmatic database interface somewhat similar to the database abstraction described in Section 3.4.7. Keys can be of various types, including string, numeric, date, and array. They can be compared for equality, less than, and greater than, allowing ordering of records in the database.

Values can be of any type, including both primitive types (string, number, etc.) and objects and arrays. In this regard, values are identical in principle to those in the database abstraction.

The specification for Indexed Database API describes a means of traversing a value which is an object, using a fully-qualified, dot-separated name, to identify a possibly deep portion of the value object. This, too, is equivalent to the database abstraction in the way object values are stored and accessed.

Transactions are supported, as is a versioning feature so that applications can automatically upgrade a database as new object types are added to the application.

Synchronous functions are available. Primary access to the API seems to be via the asynchronous interface, via events and event handler callback functions. Requests for database operations are provided with callback functions at invocation time, and those functions are called when the operation completes, or when an error occurs.

6.4 Reducing the distinction between client and server

There are a number of projects which are trying to solve the problem of different languages being required for client and server development. The problem is being solved in different ways by various projects, described in the next few sections: by converting a typically server-side language to JavaScript to run it on the client; by implementing and then pushing for adoption, a brand new language intended for ultimate use on both the client and server side; or with a framework that helps create applications that use JavaScript on both the client and server side, but hiding some

of the details of the client/server integration.

Some projects also try to solve the problem of separation of client and server, by making a server available at the client machine.

6.4.1 Program Mobile Robots in Scheme

Although the research domain is very different, work done at Cornell University in the early 1990s, “implementing a software environment that permits a small mobile robot to be programmed using the Scheme programming language” [38] involves a highly related development paradigm as that proposed for **LIBERATED**. In the case of the robot environment, both the robot and a development workstation run a system based on Scheme. Remote procedure calls may be used to issue requests either from the robot to the workstation, or from the workstation to the robot, to run Scheme functions on the peer. This allows the robot to make notifications to the workstation, and it allows the workstation to dynamically evaluate behavior on, or alter behavior of the robot, in real time. Were this system intended for web applications instead of robots, it would meet some of the research goals described herein: it allows both the development system (call it the frontend) and the robot (call it the backend) to be written in the same language. Functions can be developed and tested on the frontend (possibly making use of remote procedure calls to low-level primitives on the backend), and when fully tested, moved unaltered to the backend. The only thing it is missing appears to be a robot simulator running at the frontend, which could have code accessible by a debugger, for use during development.

6.4.2 Google Web Toolkit

Google’s answer to unifying the client and server languages for web application development is called the Google Web Toolkit (GWT). The Google Web Toolkit “is a development toolkit for building and optimizing complex browser-based applications. Its goal is to enable productive development of high-performance web applications without the developer having to be an expert in browser quirks, XMLHttpRequest,

and JavaScript.” [39] GWT allows the developer to write client-side code in Java, which is then translated into JavaScript to run in the browser.

The documentation for Google Web Toolkit discusses building client/server applications, but in reality, GWT is really only for writing the client side of the application. The toolkit includes facilities for issuing remote procedure calls and other means of communicating with the server. They provide simple examples of server code running on Google’s App Engine, and their development environment includes a built-in web server and Java run-time, with which server-side code can be tested. There is no specific client/backend interface specified, and in fact, GWT is essentially backend-agnostic. The Google Web Toolkit, then, allows writing frontend applications in Java, and optionally also writing backend applications in Java, to accomplish the language unification.

6.4.3 Dart

Another emerging project from Google is the *Dart* programming language, just announced [40] in October of 2011. Dart aims to:

- Create a structured yet flexible language for web programming.
- Make Dart feel familiar and natural to programmers and thus easy to learn.
- Ensure that Dart delivers high performance on all modern web browsers and environments ranging from small handheld devices to server-side execution.

It will either run natively on a virtual machine, or can be compiled into JavaScript code to run on any JavaScript engine. There is, as of yet, no backend functionality, in particular, no libraries providing access to databases. Dart has potential, however, as a force to unify client and server languages, once it gains more momentum. It will be interesting to watch how this progresses.

6.4.4 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.” [41] Because it is a browser add-on, it is not limited by the security constraints of applications running in the browser, and can in fact execute programs or scripts, read and write files, and make use of a built-in SQLite library for SQL-based database operations.

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.

6.4.5 CouchDB and PouchDB

CouchDB is “a document-oriented database that can be queried and indexed using JavaScript” [42] via a RESTful JSON API. PouchDB is described [43] as “a complete implementation of the CouchDB storage and views API that supports peer-to-peer replication with other CouchDB instances. The browser version is written for IndexedDatabase (part of HTML5).” It intends to provide database consistency guarantees equivalent to those of Apache CouchDB.

By using PouchDB as a simulated backend database on the client, and CouchDB as a real backend on the server, it is possible to write database code that is portable from a pure browser-based development environment to a production server environment.

6.4.6 Wakanda

Wakanda “has three distinct parts” [44]. The Wakanda Server provides a datastore and HTTP server, and houses the **business logic** of the application. The Wakanda Studio provides a means to visually design both the user interface and the data models which define how the datastore is organized. It also contains a code editor which is highly integrated with the data model and associated datastore. Finally, the Wakanda Framework provides the communications mechanism between frontend and backend,

and data binding of user interface components to the datastore.

The server-side language of Wakanda is JavaScript. Wakanda's premise seems to be that with applications designed in their Studio, and run in conjunction with their Framework, much of the often-handwritten code is generated automatically. This eliminates a whole class of bugs, and allows an application to be built quickly.

Wakanda comes close to meeting the requirements of my research question. With Wakanda, a web application is written in a single language, JavaScript, for both the server and client components (via the Wakanda Studio). Code can be developed and tested purely in the client environment, and then once the application works as desired, the code can be moved to the real server machine. That works by having a Wakanda Server instance on the development machine (similarly to the environment described for Plain Old Webserver), and a different Wakanda Server instance on the real server. What is still missing, though, is the critical ability to debug round trip operations, e.g., to trace into backend code upon initiation of a request via a frontend user action. The Wakanda development environment 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.)

Conclusions

Revisiting the research question

Let us look again at the primary research question of this thesis. From Section 2, my question was:

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 entirely tested and debugged within the browser environment, and*
- 3. fully-tested application-specific backend code to be moved, entirely unchanged, from the browser environment to the real server environment, and to run there?*

The implementation of **LIBERATED** shows that such a design is feasible. **LIBERATED** allows the entirety of the application, both frontend and backend, to be coded in JavaScript. With the simulated server running the backend code in the browser, all of the code can be fully tested and 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 glue code of **LIBERATED**, the fully-tested application-specific backend code can be moved to App Engine and run there.

I had also asked some follow-up questions, however. They were:

- 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?*

The answers to these are not as clear cut.

Compromises of this approach

Although the architecture 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:

Limited number of property types

LIBERATED defines the complete set of property types which an application may use in its database schema. These were discussed in Section 3.4.7.2. This limited set may well exclude the data types available in the target database of an application, so that the full capabilities of the database can not be realized.

Required schema

The database abstraction of **LIBERATED** requires that the schema of each entity type be pre-defined. Some databases, however, do not impose this requirement. Google App Engine, for example, allows objects of the same type to contain different properties. Here again, the full capabilities of the database are limited by the database abstraction of **LIBERATED**.

Conversion from native language to JavaScript

If access to the target database is via an API in a language other than JavaScript, it may be necessary to convert query results to JavaScript, and data being written to the database from JavaScript types into the native language types. This is true with the **LIBERATED** driver for Google App Engine. Query results are returned in Java types, which must be converted to JavaScript types before being returned to the caller of the query. In some cases, data to be written must be explicitly converted to Java types. (In many cases, though, implicit conversion in the process of issuing the API call is adequate.) Any explicit conversion is done as code in the database driver, and thus comes at some time penalty. Whether that penalty is even noticeable is an open question.

Difficulty of writing database drivers

Each of the database drivers that have been written to date — the driver for the

simulated database that runs in the browser, and the one for App Engine — have fairly closely matched the **Entity** model of the database abstraction. During development, I considered, as I architected the abstraction, how it would map to other database models such as a SQL-based relational database. I believe that the abstraction should map reasonably easily to SQL, but other database types may prove more challenging. CouchDB, discussed previously, provides a RESTful database API. It will be interesting to see how easily the mapping from the **LIBERATED** database abstraction is applied to CouchDB.

There may be a more general compromise. The most popular server languages, PHP and ASP.net, are interpreted languages like JavaScript. There are a number of JavaScript engines of varying performance. It is unknown whether a server running JavaScript code is slower than one running one of the other popular languages. If the JavaScript code is compiled to Java's .class files, it is unknown whether the compiled code has more or less performance than compiled Java.

New problems created by this approach

There have been few new problems shown to date, as a result of using this approach. The one that makes itself most obvious is that server-side JavaScript is still young, and the plentiful libraries of code to do nearly anything one might want, readily available for PHP or ASP.net, are not yet available for JavaScript. This means that in order to do something as common as, say, sending an email message, requires extra work. With the backend running on App Engine, this would require accessing the Java API classes — not particularly difficult, but not as trivial as simply calling the `mail()` function of PHP [45]. Communicating with an external process would likely involve custom code that would not be required with the more mature environments. Even now, though, **Node.js** is building a large library of code, easily `require()`'d (included) from custom code. As server-side JavaScript matures, it appears likely that this problem may simply evaporate.

Recommendations

LIBERATED is a working implementation that is being used in a real, full-fledged application. There is ample related and continuation work that can be done, however. First, it would be nice to determine exactly the impact of the compromises identified above. This would require, in many cases, implementing a parallel environment in a different language (or in multiple languages) to determine the impact of using the **LIBERATED** environment.

Additionally, there are a few obviously-missing pieces of the implementation, and some improvements that can be made.

Transactions

The biggest current deficiency in **LIBERATED** is its lack of support for database transactions. There are a number of ways to implement them from an API point of view, e.g., (a) a function call to begin a transaction followed by the requests to be issued within the transactions, followed by a function call to commit or roll back the transaction; (b) a function call to begin a transaction that takes as a parameter a function which issues all requests to be processed in the course of the transaction; etc. The database driver implications are a bit more complicated, though. In the simulation database driver, it would be trivial to save database changes in a parallel map until a transaction is committed. The App Engine datastore, on the other hand, does not make it trivial. Transactions for App Engine can only be applied to objects with a common ancestor as part of their entity type. This means that there must be more server-specific knowledge in the **LIBERATED**-based application's schema. Some fair amount of thought will have to go into this.

SQL database driver

Relational databases, with their SQL interfaces for storing and querying for data, are in very common use. For **LIBERATED** to be more generally useful, a database driver which maps to one of the popular database servers such as MySQL should be developed.

Object relations

Relationships between objects in **LIBERATED** are ad hoc, maintained exclusively by the application. It is possible for a database driver (and database server) to impose more strict requirements on the database, e.g., with relations defined in a schema of a SQL-based relational database. That is entirely outside the scope of **LIBERATED**, though. It would be nice if some form of object relationship could be defined in the **LIBERATED** database abstraction, allowing for such things as automatic retrieval of related records.

Better browser-based persistent storage

The simulation database driver could be rewritten to use the HTML5 Indexed Database or some form of persistent storage other than a JavaScript map saved in its entirety to the `localStorage` object, as is currently done.

Additional operators in queries

Currently only “and” is supported as a query operator. Additional operators should be added.

Epilogue

The implementation of **LIBERATED** and the App Inventor Community Gallery have been described here as if **LIBERATED** was created first, and the Gallery project was written as a proof of concept. In fact, development on the App Inventor Community Gallery was the impetus to create **LIBERATED**.

I was the initial developer of the App Inventor Community Gallery code. Its frontend was qooxdoo-based, and the backend was originally written in Java and ran on App Engine. When we were preparing to bring new developers onto the development team, I created the beginnings of **LIBERATED**, with the simple intention of providing a JavaScript backend running versions of the remote procedure calls, rewritten from Java into JavaScript, to allow easier debugging for the new developers. I then realized that the App Engine Java environment didn't really require Java source code; rather it required Java .class files — a Java runtime environment. Rhino's ability to run JavaScript code in any Java environment fulfilled that requirement.

Thus arose the concept of allowing complete debugging and testing in the browser, and fully-debugged and tested code to be moved to the production server environment.

References

- [1] Web Technology Surveys, “Usage of server-side programming languages for websites.” http://w3techs.com/technologies/overview/programming_language/all, September 2011.
- [2] Web Technology Surveys, “Usage of client-side programming languages for websites.” http://w3techs.com/technologies/overview/client_side_language/all, September 2011.
- [3] Wikipedia, “Prototype-based programming.” http://en.wikipedia.org/wiki/Prototype-based_programming.
- [4] Google, “V8 JavaScript Engine.” <http://code.google.com/p/v8/>.
- [5] Joyent, “Node.” <http://github.com/joyent/node/wiki>.
- [6] Mozilla, “SpiderMonkey.” <http://developer.mozilla.org/en/SpiderMonkey>.
- [7] Pepijn de Vos, “JS Server Benchmark: Node.js & Rhino.” <http://pepijndevos.nl/js-server-benchmark-nodejs-rhino/>, August 2010.
- [8] Mozilla, “Rhino.” <http://www.mozilla.org/rhino>.
- [9] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, Irvine, California, 2000.
- [10] R. Tomayko, “How I Explained REST to My Wife.” <http://tomayko.com/writings/rest-to-my-wife>, Dec. 2004.
- [11] J. White, “High-level framework for network-based resource sharing.” RFC 707, Jan. 1976.

- [12] B. J. Nelson, "Remote Procedure Call," Tech. Rep. CSL-81-9, Xerox Palo Alto Research Center, May 1981.
- [13] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, pp. 39–59, February 1984.
- [14] D. Winer, "XML-RPC Specification." <http://www.xmlrpc.com/spec>, June 1999.
- [15] "Extensible Markup Language (XML) 1.0 (Second Edition)." <http://www.w3.org/TR/2000/REC-xml-20001006>, Oct. 2000.
- [16] D. Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)." RFC 4627 (Informational), July 2006.
- [17] JSON-RPC Working Group, "JSON-RPC 2.0 specification." <http://jsonrpc.org/spec.html>, Mar. 2010.
- [18] S. Vinoski, "Convenience Over Correctness," *IEEE Internet Computing*, vol. 12, pp. 89–92, July 2008.
- [19] D. Benjamin, "Ten things that XML-RPC does... that REST leaves unspecified." <http://ramenlabs.com/2008/02/17/ten-things-that-xml-rpc-does-that-rest-leaves-unspecified/>, Feb. 2008.
- [20] E. Newcomer, "Abstraction and control in REST vs RPC." <http://blogs.iona.com/newcomer/archives/000572.html>, July 2008.
- [21] K. Zyp, "REST and RPC Relationship." <http://www.sitepen.com/blog/2008/03/25/rest-and-rpc-relationship/>, Mar. 2008.
- [22] T. Ewald, "Three reasons that REST is not RPC." <http://www.pluralsight-training.net/community/blogs/tewald/archive/2007/04/28/47067.aspx>, Apr. 2007.
- [23] UnixSpace, "Database models." <http://unixspace.com/context/databases.html>.
- [24] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, pp. 377–387, June 1970.

- [25] S. Ambler, “The Object-Relational Impedance Mismatch.” <http://www.agiledata.org/essays/impedanceMismatch.html>.
- [26] “qooxdoo – About.” <http://qooxdoo.org>.
- [27] D. Crockford and C. Douglas, *JavaScript: The Good Parts*. O’Reilly Media, illustrated edition ed., Dec. 2008.
- [28] “What Is Google App Engine.” <http://code.google.com/appengine/docs/whatisgoogleappengine.html>.
- [29] J. tzay Yang, J. long Huang, F. jian Wang, William, and C. Chu, “Constructing an object-oriented architecture for web application testing,” *Journal of Information Science and Engineering*, vol. 18, pp. 59–84, 2002.
- [30] S. Elbaum, S. Karre, and G. Rothermel, “Improving web application testing with user session data,” in *Proceedings of the 25th International Conference on Software Engineering, ICSE ’03*, (Washington, DC, USA), pp. 49–59, IEEE Computer Society, 2003.
- [31] E. E. Aftandilian and S. Z. Guyer, “Gc assertions: using the garbage collector to check heap properties,” in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, PLDI ’09*, (New York, NY, USA), pp. 235–244, ACM, 2009.
- [32] E. Aftandilian, S. Z. Guyer, M. T. Vechev, and E. Yahav, “Asynchronous assertions,” in *OOPSLA*, pp. 275–288, 2011.
- [33] Wikipedia, “Comparison of server-side JavaScript solutions.” http://en.wikipedia.org/wiki/Comparison_of_server-side_JavaScript_solutions.
- [34] W. Team, “SquirrelFish Extreme.” <http://www.webkit.org/blog/214/introducing-squirrelfish-extreme/>.
- [35] W3C, “Web SQL Database.” <http://dev.w3.org/html5/webdatabase/>.
- [36] W3C, “Indexed Database API.” <http://dvcs.w3.org/hg/IndexedDB/raw-file/tip/Overview.html>.

- [37] A. Ranganathan and S. Wilsher, “An early walk-through of IndexedDB.” <http://hacks.mozilla.org/2010/06/comparing-indexeddb-and-webdatabase/>, June 2010.
- [38] J. Rees and B. Donald, “Program mobile robots in scheme,” 1992.
- [39] “Google Web Toolkit Overview.” <http://code.google.com/webtoolkit/overview.html>.
- [40] L. Bak, “Dart: a language for structured web programming,” Oct. 2011.
- [41] D. Kellogg, “Plain Old Webserver.” http://davidkellogg.com/wiki/Main_Page.
- [42] A. S. Foundation, “The Apache CouchDB Project.” <http://couchdb.apache.org/>.
- [43] M. Rogers, “PouchDB (Portable CouchDB JavaScript implementation).” <https://github.com/mikeal/pouchdb>.
- [44] 4D, “Wakanda JS.everywhere().” <http://www.wakanda.org/features>.
- [45] “PHP – Send mail.” <http://www.php.net/manual/en/function.mail.php>.

Appendix
LIBERATED source code

Appendix 1

rpcjs.sim.remote.Transport

```

1  /*
2  * Copyright:
3  *   2011 Derrell Lipman
4  *
5  * License:
6  *   LGPL: http://www.gnu.org/licenses/lgpl.html
7  *   EPL: http://www.eclipse.org/org/documents/epl-v10.php
8  *   See the LICENSE file in the project's top-level directory for details.
9  *
10 * Authors:
11 *   * Derrell Lipman (derrell)
12 */
13
14 /*
15 #asset(rpcjs/*)
16 */
17
18
19 /**
20 * Transports requests to a simulated server
21 *
22 * This class should not be used directly by client programmers.
23 */
24 qx.Class.define("rpcjs.sim.remote.Transport",
25 {
26   extend : qx.io.remote.transport.Abstract,
27
28   construct : function()
29   {
30     // Call the superclass constructor
31     this.base(arguments);
32
33     // Initialize response headers
34     this.__responseHeaders = {};
35   },
36
37   statics :
38   {
39     /**
40     * Capabilities of this transport type.
41     */
42     handles :
43     {
44       simulate           : true, // This is our unique capability
45       synchronous       : true,
46       asynchronous      : true,
47       crossDomain       : true,
48       fileUpload        : false,

```

```

49     programaticFormFields : true,
50     responseType         :
51     [
52         "text/plain",
53         "text/javascript",
54         "application/json",
55         "application/xml",
56         "text/html"
57     ]
58 },
59
60
61 /**
62  * Whether the current browser supports this transport
63  *
64  * @return
65  * true, always. This transport is supported by all browsers.
66  */
67 isSupported : function()
68 {
69     return true;
70 }
71 },
72
73 members :
74 {
75     __request         : null,
76     __responseData   : null,
77     __responseHeaders : null,
78
79     // overridden
80     send : function()
81     {
82         // The request data, as retrieved from the various properties
83         this.__request =
84         {
85             url           : this.getUrl(),
86             method        : this.getMethod(),
87             asynchronous   : this.getAsynchronous(),
88             username      : this.getUsername(),
89             password      : this.getPassword(),
90             parameters    : this.getParameters(),
91             formFields    : this.getFormFields(),
92             requestHeaders : this.getRequestHeaders(),
93             data          : this.getData(),
94
95             // Function to post the response. The signature is:
96             // post(responseData, responseHeaders);
97             post          : qx.lang.Function.bind(this.post, this)
98         };
99
100        // Initialize responses data and headers
101        this.__responseData = null;
102        this.__responseHeaders = {};
103
104        // Simulate initial states
105        this.setState("created");
106        this.setState("configured");
107
108        // Post this request to the simulator's request queue
109        rpcjs.sim.Simulator.post(this.__request);
110
111        // Simulate sending state, now that we've "initiated" sending to our
112        // peer. (In reality, it's already arrived.)
113        this.setState("sending");
114    },
115
116 /**

```

```

117     * A response to a previously-issued request is posted via a call to this
118     * function.
119     *
120     * @param responseData {Any}
121     *   The data that is returned as a result of the previously-issued request
122     *
123     * @param responseHeaders {Map|null}
124     *   If response headers are returned, they will be in a map provided
125     *   here. Otherwise, this parameter will be either null or undefined.
126     */
127     post : function(responseData, responseHeaders)
128     {
129         this.__responseData = responseData;
130         this.__responseHeaders = responseHeaders;
131
132         // The transport of the request and response is complete
133         this.setState("receiving");
134
135         // Was this a successful result?
136         if (responseHeaders.status == 200)
137         {
138             this.setState("completed");
139         }
140         else
141         {
142             this.setState("failed");
143         }
144     },
145
146     // overridden
147     setRequestHeader : function(label, value)
148     {
149         this.getRequestHeaders()[label] = value;
150     },
151
152
153     // overridden
154     getResponseHeader : function(vLabel)
155     {
156         // No response headers in this transport
157         return null;
158     },
159
160
161     // overridden
162     getResponseHeaders : function()
163     {
164         // No response headers in this transport
165         return {};
166     },
167
168
169     // overridden
170     getStatusCode : function()
171     {
172         return this.__responseHeaders["status"];
173     },
174
175
176     // overridden
177     getStatusText : function()
178     {
179         // Since we always assume success, there's no need for status text.
180         return this.__responseHeaders["statusText"];
181     },
182
183
184     // overridden

```

```

185     getResponseText : function()
186     {
187         return this.__responseData;
188     },
189
190
191     // overridden : function()
192     getResponseXml : function()
193     {
194         // If we're expecting XML data in the response...
195         if (this.getResponse_type() == "application/xml")
196         {
197             // ... then give it to 'em.
198             return this.__responseData;
199         }
200
201         // Otherwise, there's no XML response data.
202         return null;
203     },
204
205
206     // overridden
207     getFetchedLength : function()
208     {
209         // We either have complete content, or none at all.
210         return 0;
211     },
212
213
214     // overridden
215     getResponseContent : function()
216     {
217         var          ret;
218         var          text;
219
220         // If we don't yet have a response...
221         if (this.getState() != "completed")
222         {
223             if (qx.core.Environment.get("qx.debug"))
224             {
225                 if (qx.core.Environment.get("qx.ioRemoteDebug"))
226                 {
227                     this.warn("Transfer not complete, ignoring content!");
228                 }
229             }
230
231             // ... there's nothing useful to give 'em.
232             return null;
233         }
234
235         if (qx.core.Environment.get("qx.debug"))
236         {
237             if (qx.core.Environment.get("qx.ioRemoteDebug"))
238             {
239                 this.debug("Returning content for responseType: " +
240                     this.getResponse_type());
241             }
242         }
243
244         // What response type are we expecting?
245         switch(this.getResponse_type())
246         {
247             case "text/plain":
248             case "text/html":
249                 if (qx.core.Environment.get("qx.debug"))
250                 {
251                     if (qx.core.Environment.get("qx.ioRemoteDebugData"))
252                     {

```

```

253         this.debug("Response: " + this._responseContent);
254     }
255 }
256
257 // It's an ordinary string. We can give it to them as is.
258 ret = this._responseContent;
259 return (ret === 0 ? 0 : (ret || null));
260
261 case "application/json":
262     if (qx.core.Environment.get("qx.debug"))
263     {
264         if (qx.core.Environment.get("qx.ioRemoteDebugData"))
265         {
266             this.debug("Response: " + text);
267         }
268     }
269
270     try
271     {
272         text = this._responseData;
273         if (text && text.length > 0)
274         {
275             ret = qx.lang.Json.parse(text);
276             ret = (ret === 0 ? 0 : (ret || null));
277             return ret;
278         }
279         else
280         {
281             return null;
282         }
283     }
284     catch(ex)
285     {
286         this.error("Could not execute json: [" + text + "]", ex);
287         return "<pre>Could not execute json: \n" + text + "\n</pre>";
288     }
289
290 case "text/javascript":
291     if (qx.core.Environment.get("qx.debug"))
292     {
293         if (qx.core.Environment.get("qx.ioRemoteDebugData"))
294         {
295             this.debug("Response: " + text);
296         }
297     }
298
299     try
300     {
301         text = this._responseData;
302         if(text && text.length > 0)
303         {
304             ret = window.eval(text);
305             return (ret === 0 ? 0 : (ret || null));
306         }
307         else
308         {
309             return null;
310         }
311     }
312     catch(ex)
313     {
314         this.error("Could not execute javascript: [" + text + "]", ex);
315         return null;
316     }
317
318 case "application/xml":
319     text = this.getResponseXml();
320

```

```

321     if (qx.core.Environment.get("qx.debug"))
322     {
323         if (qx.core.Environment.get("qx.ioRemoteDebugData"))
324         {
325             this.debug("Response: " + text);
326         }
327     }
328
329     return (text === 0 ? 0 : (text || null));
330
331     default:
332         this.warn("No valid responseType specified (" +
333             this.getResponse() + ")!");
334         return null;
335     }
336 }
337 },
338
339
340 destruct : function()
341 {
342     this.__request = null;
343     this.__responseData = null;
344     this.__responseHeaders = null;
345 },
346
347
348 defer : function()
349 {
350     // Patch qx.io.remote.Exchange with our own send() method that
351     // supports looking at the "need" for a simulated transport.
352     qx.Class.patch(qx.io.remote.Exchange, rpcjs.sim.remote.MExchange);
353
354     // Similarly, for qx.io.remote.Request, except it can be a simple
355     // include since it's only adding a property.
356     qx.Class.include(qx.io.remote.Request, rpcjs.sim.remote.MRequest);
357
358     // Patch qx.io.remote.Rpc with our own createRequest() method that
359     // supports setting the simulate property of the request.
360     qx.Class.patch(qx.io.remote.Rpc, rpcjs.sim.remote.MRpc);
361
362     // Register ourselves with qx.io.remote.Exchange
363     qx.io.remote.Exchange.registerType(rpcjs.sim.remote.Transport,
364         "rpcjs.sim.remote.Transport");
365
366     // Add ourselves as the last tried transport
367     qx.io.remote.Exchange.typesOrder.push("rpcjs.sim.remote.Transport");
368 }
369 });

```

Appendix 2

rpcjs.sim.remote.MRequest

```
1  /*
2  * Copyright:
3  *   2011 Derrell Lipman
4  *
5  * License:
6  *   LGPL: http://www.gnu.org/licenses/lgpl.html
7  *   EPL: http://www.eclipse.org/org/documents/epl-v10.php
8  *   See the LICENSE file in the project's top-level directory for details.
9  *
10 * Authors:
11 *   * Derrell Lipman (derrell)
12 */
13
14 /*
15 #asset(rpcjs/*)
16 */
17
18
19 /**
20 * Mixin to add simulator functionality to qx.io.remote.Exchange.
21 */
22 qx.Mixin.define("rpcjs.sim.remote.MRequest",
23 {
24   properties :
25   {
26     /** Whether to simulate transport using rpcjs.sim.rpc.Simulator */
27     simulate :
28     {
29       check      : "Boolean",
30       nullable   : false,
31       init       : false
32     }
33   }
34 });
```

Appendix 3

rpcjs.sim.remote.MExchange

```

1  /*
2  * qooxdoo - the new era of web development
3  *
4  * http://qooxdoo.org
5  *
6  * Copyright:
7  *   2004-2008 1&1 Internet AG, Germany, http://www.1und1.de
8  *   2006, 2011 Derrell Lipman
9  *   2006 STZ-IDA, Germany, http://www.stz-ida.de
10 *
11 * License:
12 *   LGPL: http://www.gnu.org/licenses/lgpl.html
13 *   EPL: http://www.eclipse.org/org/documents/epl-v10.php
14 *   See the LICENSE file in the project's top-level directory for details.
15 *
16 * Authors:
17 *   * Sebastian Werner (wpbasti)
18 *   * Andreas Ecker (ecker)
19 *   * Derrell Lipman (derrell)
20 *   * Andreas Junghans (lucidcake)
21 */
22
23 /*
24 #asset(rpcjs/*)
25 #require(qx.io.remote.Exchange)
26 */
27
28
29 /**
30 * Mixin to add simulator functionality to qx.io.remote.Exchange.
31 */
32 qx.Mixin.define("rpcjs.sim.remote.MExchange",
33 {
34   members :
35   {
36     /**
37      * Sends the request. Adds 'simulate' need test.
38      *
39      * @return {var|Boolean}
40      *   Returns true if the request was sent.
41      */
42     send : function()
43     {
44       var          transportImpl;
45       var          transport;
46       var          request;
47
48       // Get the current request object.

```

```

49     request = this.getRequest();
50     if (!request)
51     {
52         return this.error("Please attach a request object first");
53     }
54
55     qx.io.remote.Exchange.initTypes();
56
57     var usage = qx.io.remote.Exchange.typesOrder;
58     var supported = qx.io.remote.Exchange.typesSupported;
59
60     // Mapping settings to contenttype and needs to check later
61     // if the selected transport implementation can handle
62     // fulfill these requirements.
63     var responseType = request.getResponse();
64     var needs = {};
65
66     if (request.getAsynchronous())
67     {
68         needs.asynchronous = true;
69     }
70     else
71     {
72         needs.synchronous = true;
73     }
74
75     if (request.getCrossDomain())
76     {
77         needs.crossDomain = true;
78     }
79
80     if (request.getFileUpload())
81     {
82         needs.fileUpload = true;
83     }
84
85     // See if there are any programtic form fields requested
86     for (var field in request.getFormFields())
87     {
88         // There are.
89         needs.programaticFormFields = true;
90
91         // No need to search further
92         break;
93     }
94
95     // See if we're asked to simulate the transport
96     if (request.getSimulate())
97     {
98         needs.simulate = true;
99     }
100
101     for (var i=0, l=usage.length; i<l; i++)
102     {
103         transportImpl = supported[usage[i]];
104
105         if (transportImpl)
106         {
107             if (!qx.io.remote.Exchange.canHandle(transportImpl, needs,
108                 responseType, request)) {
109                 continue;
110             }
111
112             try
113             {
114                 if (qx.core.Environment.get("qx.debug"))
115                 {
116                     if (qx.core.Environment.get("qx.ioRemoteDebug")) {

```

```
117         this.debug("Using implementation: " + transportImpl.classname);
118     }
119 }
120
121     transport = new transportImpl;
122     this.setImplementation(transport);
123
124     transport.setUseBasicHttpAuth(request.getUseBasicHttpAuth());
125
126     transport.send();
127     return true;
128 }
129 catch(ex)
130 {
131     this.error("Request handler throws error");
132     this.error(ex);
133     return false;
134 }
135 }
136 }
137
138     this.error("There is no transport implementation available " +
139             "to handle this request: " + request);
140     return false;
141 }
142 }
143 });
```

Appendix 4

rpcjs.sim.remote.MRpcpc

```

1  /*
2  * Copyright:
3  *   2011 Derrell Lipman
4  *
5  * License:
6  *   LGPL: http://www.gnu.org/licenses/lgpl.html
7  *   EPL: http://www.eclipse.org/org/documents/epl-v10.php
8  *   See the LICENSE file in the project's top-level directory for details.
9  *
10 * Authors:
11 *   * Derrell Lipman (derrell)
12 */
13
14 /*
15 #asset(rpcjs/*)
16 */
17
18
19 /**
20 * Mixin to add simulator functionality to qx.io.remote.Rpc.
21 */
22 qx.Mixin.define("rpcjs.sim.remote.MRpc",
23 {
24   statics :
25   {
26     // Whether to request the simulator
27     SIMULATE : true
28   },
29
30   members :
31   {
32     createRequest: function()
33     {
34       // Get a remote request object
35       var request = new qx.io.remote.Request(this.getUrl(),
36                                             "POST",
37                                             "application/json");
38
39       // Have we been requested to use the simulator?
40       if (rpcjs.sim.remote.MRpc.SIMULATE)
41       {
42         // Yup. Make the request
43         request.setSimulate(true);
44       }
45
46       return request;
47     }
48   }

```

49 }) ;

Appendix 5

rpcjs.sim.Simulator

```

1  /*
2  * Copyright:
3  *   2011 Derrell Lipman
4  *
5  * License:
6  *   LGPL: http://www.gnu.org/licenses/lgpl.html
7  *   EPL: http://www.eclipse.org/org/documents/epl-v10.php
8  *   See the LICENSE file in the project's top-level directory for details.
9  *
10 * Authors:
11 *   * Derrell Lipman (derrell)
12 */
13
14 /*
15 #asset(rpcjs/*)
16 #use(rpcjs.sim.remote.Transport)
17 */
18
19
20 /**
21 * The main simulator class.
22 */
23 qx.Class.define("rpcjs.sim.Simulator",
24 {
25   statics :
26   {
27     /** TimerManager singleton */
28     __timerManager : null,
29
30     /** Place to put requests as they are posted, while awaiting processing */
31     __requestQueue : [],
32
33     /** List of handlers. Each will be tried in order of registration. */
34     __handlers      : [],
35
36     /**
37      * Whether to process requests synchronously, or to better simulate a real
38      * transport by processing requests asynchronously. The former allows
39      * easier debugging.
40      */
41     SYNCHRONOUS : true,
42
43     /**
44      * Register a handler for a particular URL.
45      *
46      * @param handler {Function}
47      *   The function which is called to process a request with the given
48      *   URL. The function will be called with the request object and a

```

```

49     * responseHeaders map which is initialized to contain a status of 200
50     * (success) and an empty statusText string. The function should return
51     * the result of the request upon success. Upon failure, it should
52     * modify the two response header fields, and return null.
53     */
54 registerHandler : function(handler)
55 {
56     rpcjs.sim.Simulator.__handlers.push(handler);
57 },
58
59
60 /**
61  * Function called by the simulation transport to enqueue a new request.
62  *
63  * @param request {Map}
64  *   A map containing the data pertaining to the request. The map will
65  *   have the following members:
66  *
67  *   url           {String}
68  *   method        {String}   GET, POST, etc.
69  *   asynchronous  {Boolean}   should always be true, for now
70  *   username      {String}
71  *   password      {String}
72  *   parameters    {Map}
73  *   formFields    {Map}
74  *   requestHeaders {Map}
75  *   data          {Any}
76  */
77 post : function(request)
78 {
79     // Get the request queue
80     var Simulator = rpcjs.sim.Simulator;
81     var requestQueue = Simulator.__requestQueue;
82
83     // Enqueue this request
84     requestQueue.push(request);
85
86     // Are we processing requests synchronously?
87     if (Simulator.SYNCHRONOUS)
88     {
89         Simulator.__processQueue();
90     }
91     else
92     {
93         // If this is the first item on the queue...
94         if (requestQueue.length == 1)
95         {
96             // ... then start a timer to process the queue immediately
97             Simulator.__timerManager.start(Simulator.__processQueue,
98                 0, window, null, 0);
99         }
100    }
101 },
102
103
104 /**
105  * Process an element from the queue. After processing it, if the queue is
106  * not empty, set a timer to be re-called immediately.
107  */
108 __processQueue : function()
109 {
110     var i;
111     var request;
112     var response;
113     var responseHeaders;
114     var transportPost;
115     var Simulator = rpcjs.sim.Simulator;
116     var requestQueue = Simulator.__requestQueue;

```

```

117
118 // Is there anything on the request queue?
119 if (requestQueue.length == 0)
120 {
121     // Nope. Nothing to do. (Should never occur.)
122     return;
123 }
124
125 // Pull the first element off of the queue
126 request = requestQueue.shift();
127
128 // Save the transport's post method, and delete it from the request. The
129 // handler should never see it.
130 transportPost = request.post;
131 delete request.post;
132
133 // Try each handler until we find one that handles this request
134 for (i = 0; i < Simulator.__handlers.length; i++)
135 {
136     // Assume that the handler will not be able to process the request
137     responseHeaders =
138     {
139         status      : 501,
140         statusText  : "Not supported by this handler"
141     };
142     // Call the handler's processRequest function.
143     response = Simulator.__handlers[i](request, responseHeaders);
144
145     // See what the handler did with this request.
146     // status=200 means it handled the request successfully
147     // status=501 means we need to try another handler;
148     // else: handled the request but an error occurred
149     if (responseHeaders.status == 200)
150     {
151         // Yes it did! Restore the transport's post method to
152         // the request object and send the response.
153         request.post = transportPost;
154         request.post(response, responseHeaders);
155         break;
156     }
157     else if (responseHeaders.status == 501)
158     {
159         // Try the next handler. Nothing to do here.
160     }
161     else
162     {
163         //
164         // The request got handled, but was not successful.
165         //
166
167         // Restore the transport's post method to the request
168         // object and send the error response.
169         request.post = transportPost;
170         request.post(null, responseHeaders);
171         break;
172     }
173 }
174
175 // Restore the transport's post method to the request object and send
176 // the error response.
177 request.post = transportPost;
178
179 // If we reached the end of the handler list...
180 if (i == Simulator.__handlers.length)
181 {
182     // ... then generate a response indicating there was no handler
183     responseHeaders =
184     {

```

```
185         status      : 404,
186         statusText : "No handler for URL " + request.url
187     };
188
189     // Send the error response.
190     request.post(null, responseHeaders);
191 }
192
193 // Is there anything left on the queue?
194 if (requestQueue.length > 0)
195 {
196     // Yup. Start a timer to process the queue again immediately
197     Simulator.__timerManager.start(Simulator.__processQueue,
198                                   0, window, null, 0);
199 }
200 }
201 },
202
203 defer : function()
204 {
205     rpcjs.sim.Simulator.__timerManager = qx.util.TimerManager.getInstance();
206 }
207 });
```

Appendix 6

rpcjs.rpc.Server

```

1  /*
2  * Copyright:
3  *   2011 Derrell Lipman
4  *
5  * License:
6  *   LGPL: http://www.gnu.org/licenses/lgpl.html
7  *   EPL: http://www.eclipse.org/org/documents/epl-v10.php
8  *   See the LICENSE file in the project's top-level directory for details.
9  *
10 * Authors:
11 *   * Derrell Lipman (derrell)
12 */
13
14 /*
15 #asset(rpcjs/*)
16 */
17
18 /**
19 * JSON-RPC server
20 */
21 qx.Class.define("rpcjs.rpc.Server",
22 {
23   extend : qx.core.Object,
24
25
26   /**
27    * Constructor for a JSON_RPC server.
28    *
29    * @param serviceFactory {Function}
30    *   A function which provides an interface to a service method. The
31    *   function will be called with a namespaced method name, and an
32    *   rpcjs.rpc.error.Error object. Under normal circumstances (success), it
33    *   should return a function reference. If the method identified by name is
34    *   not available, either because it does not exist, or for some other
35    *   reason (e.g. the function's was called in a cross-domain fashion but
36    *   the function is not permitted to be used in that fashion), the provided
37    *   error object's methods should be used to provide details of the error,
38    *   and then the error object should be returned.
39    */
40   construct : function(serviceFactory)
41   {
42     // Call the superclass constructor
43     this.base(arguments);
44
45     // The service factory is mandatory
46     if (! qx.lang.Type.isFunction(serviceFactory))
47     {
48       throw new Error("Missing service factory function");

```

```

49     }
50
51     // Save the parameters for future use
52     this.setServiceFactory(serviceFactory);
53 },
54
55 properties :
56 {
57     /**
58     * A function which provides an interface to a service method. The
59     * function will be called with a namespaced method name, and an
60     * rpcjs.rpc.error.Error object. Under normal circumstances (success), it
61     * should return a function reference. If the method identified by name is
62     * not available, either because it does not exist, or for some other
63     * reason (e.g. the function's was called in a cross-domain fashion but
64     * the function is not permitted to be used in that fashion), the provided
65     * error object's methods should be used to provide details of the error,
66     * and then the error object should be returned.
67     */
68     serviceFactory :
69     {
70         check      : "Function"
71     }
72 },
73
74 members :
75 {
76     /**
77     * Process a single remote procedure call request.
78     *
79     * @param jsonInput {String}
80     *   The input string, containing the JSON-encoded RPC request.
81     *
82     * @return {String}
83     *   The JSON response.
84     */
85     processRequest : function(jsonInput)
86     {
87         var          timer;          // timeout object
88         var          ret;            // error return object
89         var          bBatch;         // whether a batch request is received
90         var          requests;       // the parsed input request
91         var          error;          // an error object
92         var          reply;          // a textual reply in case garbage input
93         var          protocol;       // protocol version being used
94         var          fqMethod;       // fully-qualified method name
95         var          service;        // service function to call
96         var          result;         // result of calling service function
97         var          parameters;     // the parameter list for the RPC
98         var          run;            // function to run the service call
99         var          responses;      // array of responses (in case of batch)
100
101         // Assume protocol is 2.0 until we (might) discover otherwise
102         protocol = "2.0";
103
104         try
105         {
106             // Parse the JSON
107             requests = qx.lang.Json.parse(jsonInput);
108         }
109         catch(e)
110         {
111             // We couldn't parse the request.
112             // Get a new version 2.0 error object.
113             error = new rpcjs.rpc.error.Error("2.0");
114             error.setCode(qx.io.remote.RpcError.v2.error.ParseError);
115             error.setMessage("Could not parse request");
116

```

```

117     // Build the error response
118     ret =
119     {
120         jsonrpc : "2.0",
121         id      : null,
122         error   : qx.lang.Json.parse(error.stringify())
123     };
124
125     return qx.lang.Json.stringify(ret);
126 }
127
128 // Determine if this is normal or batch mode
129 if (qx.lang.Type.isArray(requests))
130 {
131     // It's batch mode.
132     bBatch = true;
133
134     // Ensure that there's at least one element in the array
135     if (requests.length === 0)
136     {
137         // Get a new version 2.0 error object.
138         error = new rpcjs.rpc.error.Error("2.0");
139         error.setCode(qx.io.remote.RpcError.v2.error.InvalidRequest);
140         error.setMessage("Empty batch array");
141
142         // Build the error response
143         ret =
144         {
145             jsonrpc : "2.0",
146             id      : null,
147             error   : qx.lang.Json.parse(error.stringify())
148         };
149
150         return qx.lang.Json.stringify(ret);
151     }
152 }
153 else if (qx.lang.Type.isObject(requests))
154 {
155     // It's normal mode
156     bBatch = false;
157
158     // Create an array as if it were batch mode
159     requests = [ requests ];
160 }
161 else
162 {
163     // Get a new version 2.0 error object.
164     error = new rpcjs.rpc.error.Error("2.0");
165     error.setCode(qx.io.remote.RpcError.v2.error.InvalidRequest);
166     error.setMessage("Unrecognized request type");
167     error.setData("Expected an array or an object");
168
169     // Build the error response
170     ret =
171     {
172         jsonrpc : "2.0",
173         id      : null,
174         error   : qx.lang.Json.parse(error.stringify())
175     };
176
177     return qx.lang.Json.stringify(ret);
178 }
179
180 // For each request in the batch (or the single non-batch request)...
181 responses = requests.map(
182     function(request)
183     {
184         var          ret;

```

```

185     var                id;
186
187     // Get the id value to use in error responses
188     id = typeof request.id == "undefined" ? null : request.id;
189
190     // Ensure that this is a valid request object
191     if (! qx.lang.Type.isObject(request))
192     {
193         // Get a new version 2.0 error object.
194         error = new rpcjs.rpc.error.Error("2.0");
195         error.setCode(qx.io.remote.RpcError.v2.error.InvalidRequest);
196         error.setMessage("Unrecognized request");
197         error.setData("Expected an object");
198
199         // Build the error response
200         ret =
201         {
202             jsonrpc : "2.0",
203             id      : id,
204             error   : qx.lang.Json.parse(error.stringify())
205         };
206
207         return ret;
208     }
209
210     // Determine which protocol to use. Is there a jsonrpc member?
211     if (typeof(request.jsonrpc) == "string")
212     {
213         // Get a new version 2.0 error object.
214         error = new rpcjs.rpc.error.Error("2.0");
215
216         // Yup. It had better be "2.0"!
217         if (request.jsonrpc != "2.0")
218         {
219             error.setCode(qx.io.remote.RpcError.v2.error.InvalidRequest);
220             error.setMessage("'jsonrpc' member must be \"2.0\".");
221             error.setData("Found value " + request.jsonrpc + "in 'jsonrpc'.");
222
223             // Build the error response
224             ret =
225             {
226                 jsonrpc : "2.0",
227                 id      : id,
228                 error   : qx.lang.Json.parse(error.stringify())
229             };
230
231             return ret;
232         }
233
234         // Validate that the method is a string
235         if (! qx.lang.Type.isString(request.method))
236         {
237             error.setCode(qx.io.remote.RpcError.v2.error.InvalidRequest);
238             error.setMessage("JSON-RPC method name is missing or " +
239                 "incorrect type");
240             error.setData("Method name must be a string.");
241
242             // Build the error response
243             ret =
244             {
245                 jsonrpc : "2.0",
246                 id      : id,
247                 error   : qx.lang.Json.parse(error.stringify())
248             };
249
250             return ret;
251         }
252     }

```

```

253 // Validate that the params member is undefined, an object, or an
254 // array.
255 if (typeof(request.params) !== "undefined" &&
256     ! qx.lang.Type.isObject(request.params) &&
257     ! qx.lang.Type.isArray(request.params))
258 {
259     error.setCode(qx.io.remote.RpcError.v2.error.InvalidRequest);
260     error.setMessage("JSON-RPC params is missing or incorrect type");
261     error.setData(
262         "params must be undefined, an object, or an array.");
263
264     // Build the error response
265     ret =
266     {
267         jsonrpc : "2.0",
268         id      : id,
269         error   : qx.lang.Json.parse(error.stringify())
270     };
271
272     return ret;
273 }
274
275 // We have what appears to be a valid version 2.0 request
276 protocol = "2.0";
277
278 else if (bBatch)
279 {
280     // We're in batch mode so we had to have been in 2.0 mode
281     error.setCode(qx.io.remote.RpcError.v2.error.InvalidRequest);
282     error.setMessage("JSON-RPC protocol version is missing.");
283     error.setData("Expected 'jsonrpc:'\"2.0\"");
284
285     // Build the error response
286     ret =
287     {
288         jsonrpc : "2.0",
289         id      : id,
290         error   : qx.lang.Json.parse(error.stringify())
291     };
292
293     return ret;
294 }
295 else
296 {
297     protocol = "qx1";
298
299     // Get a qooxdoo-modified version 1 error object
300     error = new rpcjs.rpc.error.Error("qx1");
301
302     // Ensure all of the required members are present in the request
303     if (! qx.lang.Type.isString(request.service) ||
304         ! qx.lang.Type.isString(request.method) ||
305         ! qx.lang.Type.isArray(request.params))
306     {
307         // Invalid. We still send back a 2.0 error object, however.
308         error.setCode(qx.io.remote.RpcError.qx1.error.server.Unknown);
309         error.setMessage("Missing service, method, or params");
310
311         // Build the error response
312         ret =
313         {
314             id      : id,
315             error   : qx.lang.Json.parse(error.stringify())
316         };
317
318         return ret;
319     }
320 }

```

```

321
322 // Generate the fully-qualified method name
323 fqMethod =
324   (protocol == "qx1"
325    ? request.service + "." + request.method
326    : request.method);
327
328 /*
329  * Ensure the requested method name is kosher. It should be:
330  *
331  *   First test for:
332  *   - a dot-separated sequences of strings
333  *   - first character of each string is in [a-zA-Z]
334  *   - other characters are in [_a-zA-Z0-9]
335  *
336  *   Then verify:
337  *   - no two adjacent dots
338  */
339
340 // First test for valid characters
341 if (! /^[a-zA-Z][_a-zA-Z0-9]*$/ .test(fqMethod))
342 {
343   // There's some illegal character in the service or method name
344   error.setCode(
345     {
346       "qx1" : qx.io.remote.RpcError.qx1.error.server.MethodNotFound,
347       "2.0" : qx.io.remote.RpcError.v2.error.MethodNotFound
348     }[protocol]);
349   error.setMessage("Illegal character found in service name.");
350
351   // Build the error response
352   ret =
353     {
354       id      : id,
355       error   : qx.lang.Json.parse(error.stringify())
356     };
357
358   // If this is v2, we need to add the indicator of such.
359   if (protocol == "2.0")
360   {
361     ret.jsonrpc = "2.0";
362   }
363
364   return ret;
365 }
366
367 // Next, ensure there are no double dots
368 if (fqMethod.indexOf("..") != -1)
369 {
370   error.setCode(
371     {
372       "qx1" : qx.io.remote.RpcError.qx1.error.server.MethodNotFound,
373       "2.0" : qx.io.remote.RpcError.v2.error.MethodNotFound
374     }[protocol]);
375   error.setMessage("Illegal use of two consecutive dots " +
376                   "in service name.");
377
378   // Build the error response
379   ret =
380     {
381       id      : id,
382       error   : qx.lang.Json.parse(error.stringify())
383     };
384
385   // If this is v2, we need to add the indicator of such.
386   if (protocol == "2.0")
387   {
388     ret.jsonrpc = "2.0";

```

```

389     }
390
391     return ret;
392 }
393
394 // Use the registered callback to get a service function associated
395 // with this method name.
396 service = this.getServiceFactory()(fqMethod, protocol, error);
397
398 // Was there an error?
399 if (service == null)
400 {
401     // Yup. Is this a notification?
402     if (typeof request.id == "undefined")
403     {
404         // Yes. Just return undefined so the error is ignored.
405         return undefined;
406     }
407
408 // Build error response. The error was set in the service factory.
409 ret =
410     {
411         id      : request.id,
412         error   : qx.lang.Json.parse(error.stringify())
413     };
414
415 // If this is v2, we need to add the indicator of such.
416 if (protocol == "2.0")
417     {
418         ret.jsonrpc = "2.0";
419     }
420
421     return ret;
422 }
423
424 // Were we given a parameter array, or a parameter map, or none?
425 if (qx.lang.Type.isArray(request.params))
426 {
427     // Use the provided parameter list
428     parameters = request.params;
429 }
430 else if (qx.lang.Type.isObject(request.params))
431 {
432     // Does this service allow a map of parameters?
433     if (service.parameterNames)
434     {
435         // Yup. Initialize to an empty parameter list
436         parameters = [];
437
438         // Map the arguments into the parameter array. (We are
439         // forgiving of members of request.params that are not
440         // in the formal parameter list. We just ignore them.)
441         service.parameterNames.forEach(
442             function(paramName)
443             {
444                 // Add the parameter. If it's undefined, so be it.
445                 parameters.push(request.params[paramName]);
446             }
447         );
448     }
449 }
450 else
451 {
452     // No provided parameters is equivalent to an empty list
453     parameters = [];
454 }
455
456 // Create a function that will run this one request. We do this to
457 // allow notifications to be run after we return from

```

```

457 // processRequest().
458 run = qx.lang.Function.bind(
459     function(request)
460     {
461         var          result;
462         var          params = qx.lang.Array.clone(parameters);
463         var          timer = {}; // just a reference to compare to
464
465         // Assume that any error that occurs here on out will be of
466         // Application origin. Only JavaScript errors in the script will
467         // return to being Server origin.
468         if (protocol == "qx1")
469         {
470             error.setOrigin(qx.io.remote.RpcError.qx1.origin.Application);
471         }
472
473         // Provide the error object as the last parameter.
474         params.push(error);
475
476         // We should now have a service function to call. Call it.
477         try
478         {
479             result = service.apply(window, params);
480         }
481         catch(e)
482         {
483             // The service method threw an error. Create our own error from
484             // it.
485             error.setCode(
486                 {
487                     "qx1" : qx.io.remote.RpcError.qx1.error.server.ScriptError,
488                     "2.0" : qx.io.remote.RpcError.v2.error.InternalError
489                 }[protocol]);
490
491             // Combine the message from the original error
492             error.setMessage("Method threw an error: " + e);
493
494             // This is classified as a server error, not application-origin.
495             if (protocol == "qx1")
496             {
497                 error.setOrigin(qx.io.remote.RpcError.qx1.origin.Server);
498             }
499
500             // Use this error as the result
501             result = error;
502         }
503
504         return result;
505     },
506     this);
507
508 // Is this request a notification?
509 if (typeof(request.id) == "undefined")
510 {
511     // Yup. Schedule this method to be called later, but ASAP. We
512     // don't care about the result, so we needn't wait for it to
513     // complete.
514     setTimeout(
515         function()
516         {
517             run(request);
518         },
519         0);
520
521     // Set result to the timer object, so we'll ignore it, below.
522     result = timer;
523 }
524 else

```

```

525     {
526         // It's not a notification. Run requested service method now.
527         result = run(request);
528     }
529
530     // Was this a notification?
531     if (result === timer)
532     {
533         // Yup. Return undefined so it'll be ignored.
534         return undefined;
535     }
536
537     // Was the result an error?
538     if (result instanceof rpcjs.rpc.error.Error)
539     {
540         // Yup. Stringify and return it.
541         // The error class knows how to stringify itself, but we need a map.
542         // Go both directions, to obtain the map.
543
544         // Build the error response
545         ret =
546         {
547             id      : request.id,
548             error   : qx.lang.Json.parse(result.stringify())
549         };
550
551         // If this is v2, we need to add the indicator of such.
552         if (protocol == "2.0")
553         {
554             ret.jsonrpc = "2.0";
555         }
556
557         return ret;
558     }
559
560     // We have a standard result. Stringify and return a proper response.
561     ret =
562     {
563         id      : request.id,
564         result  : result
565     };
566
567     // If this is v2, we need to add the indicator of such.
568     if (protocol == "2.0")
569     {
570         ret.jsonrpc = "2.0";
571     }
572
573     return ret;
574 },
575 this);
576
577 // Remove any responses that were for notifications (undefined ones)
578 responses = responses.filter(
579     function(response)
580     {
581         return typeof(response) !== "undefined";
582     });
583
584 // Is there a response, i.e. were there any non-notifications?
585 if (responses.length == 0)
586 {
587     // Nope. Return null to indicate that no response should be returned.
588     return null;
589 }
590
591 // Give 'em the response(s)
592 return (bBatch

```

```
593         ? qx.lang.Json.stringify(responses)
594         : qx.lang.Json.stringify(responses[0]));
595     }
596 }
597 });
```

Appendix 7

rpcjs.AbstractRpcHandler

```

1  /*
2  * Copyright:
3  *   2011 Derrell Lipman
4  *
5  * License:
6  *   LGPL: http://www.gnu.org/licenses/lgpl.html
7  *   EPL: http://www.eclipse.org/org/documents/epl-v10.php
8  *   See the LICENSE file in the project's top-level directory for details.
9  *
10 * Authors:
11 *   * Derrell Lipman (derrell)
12 */
13
14 /*
15 #asset(rpcjs/*)
16 */
17
18
19 /**
20 * Abstract handler for remote procedure calls.
21 */
22 qx.Class.define("rpcjs.AbstractRpcHandler",
23 {
24   extend : qx.core.Object,
25
26   /**
27    * Base constructor for each RPC handler.
28    *
29    * @param rpcKey {Array}
30    *   The list of prefix keys for access to the set of remote
31    *   procedure calls supported in this object's services map.
32    *
33    * Example: If the passed parameter is [ "sys", "fs" ] and
34    * one of the methods later added is "read" then the remote
35    * procedure call will be called "sys.fs.read", and the services
36    * map will contain:
37    *
38    * {
39    *   sys :
40    *     {
41    *       fs :
42    *         {
43    *           read : function()
44    *             {
45    *               // implementation of sys.fs.read()
46    *             }
47    *         }
48    *     }
49    * }

```

```

49     *   }
50     */
51     construct : function(rpcKey)
52     {
53         var             i;
54         var             services = {};
55         var             part;
56
57         // Call the superclass constructor
58         this.base(arguments);
59
60         // Initialize the services
61         rpcjs.AbstractRpcHandler._services = services;
62
63         // Add each of the RPC keys
64         for (i = 0, part = services; i < rpcKey.length; i++)
65         {
66             part = part[rpcKey[i]] = {};
67         }
68
69         // Store the final part, where registered services will go
70         rpcjs.AbstractRpcHandler._servicesByKey = part;
71
72         // Get an RPC Server instance.
73         this.__rpcServer = new rpcjs.rpc.Server(
74             rpcjs.AbstractRpcHandler._serviceFactory);
75     },
76
77     statics :
78     {
79         /** The services map */
80         _services : null,
81
82         /**
83          * Reference to the final component of the services map, where registered
84          * services are placed.
85          */
86         _servicesByKey : null,
87
88         /**
89          * Function to call for authorization to run the service method. If
90          * null, no authorization is required. Otherwise this should be a
91          * function which takes as a parameter, the fully-qualified name of the
92          * method to be called, and must return true to allow the function to be
93          * called, or false otherwise, to indicate permission denied.
94          */
95         authorizationFunction : null,
96
97         /**
98          * The service factory takes a method name and attempts to produce a
99          * service method that corresponds to that name. This implementation
100         * concatenates the method name to the name of the variable holding
101         * the service map, and looks for a corresponding method.
102         *
103         * @param fqMethodName {String}
104         *   The fully-qualified name of the method to be called.
105         *
106         * @param protocol {String}
107         *   The JSON-RPC protocol being used ("qx1", "2.0")
108         *
109         * @param error {rpcjs.rpc.error.Error}
110         *   An error object to be set if an error is encountered in
111         *   instantiating the requested service method.
112         *
113         * @return {Function}
114         *   The service method associated with the specified method name.
115         */
116         _serviceFactory : function(fqMethodName, protocol, error)

```

```

117 {
118     var          method;
119
120     // Append the fully-qualified method name to the services map and
121     // evaluate it in hopes of getting a method reference.
122     //
123     // We have a dot-separated fully-qualified method name. We want to
124     // access that entry in the map of maps in _services. Using the
125     // index notation won't work, since it's multiple levels deep
126     // (i.e. fqMethodName might be something like "a.b.c").
127     //
128     // fqMethodName was sanitized by rpcjs.rpc.Server:processRequest()
129     // so this eval is reasonably safe. (I know... Famous last words!)
130     method = eval("rpcjs.AbstractRpcHandler._services" +
131                 "." + fqMethodName);
132
133     // We might have just gotten null, which also means no such method
134     if (! method)
135     {
136         // No such method.
137         error.setCode(
138             {
139                 "qx1" : qx.io.remote.RpcError.qx1.error.server.MethodNotFound,
140                 "2.0" : qx.io.remote.RpcError.v2.error.MethodNotFound
141             }[protocol]);
142         error.setMessage("No such method");
143         return null;
144     }
145
146     // Validate allowability of calling this function
147     if (rpcjs.AbstractRpcHandler.authorizationFunction &&
148         !rpcjs.AbstractRpcHandler.authorizationFunction(fqMethodName))
149     {
150         // Permission denied
151         error.setCode(
152             {
153                 "qx1" : qx.io.remote.RpcError.qx1.error.server.PermissionDenied,
154                 "2.0" : qx.io.remote.RpcError.v2.error.PermissionDenied
155             }[protocol]);
156         error.setMessage("Permission denied.");
157         return null;
158     }
159
160     // Give 'em the reference to the method they can call.
161     return method;
162 },
163
164 /**
165  * Register a service name and function.
166  *
167  * @param serviceName {String}
168  *   The name of this service within the <[rpcKey]> namespace.
169  *
170  * @param fService {Function}
171  *   The function which implements the given service name.
172  *
173  * @param context {Object}
174  *   The context in which the service function should be called
175  *
176  * @param paramNames {Array}
177  *   The names of the formal parameters, in order.
178  */
179 registerService : function(serviceName, fService, context, paramNames)
180 {
181     var          f;
182
183     // Use this object as the context for the service
184     f = qx.lang.Function.bind(fService, context);

```

```

185
186 // Save the parameter names as a property of the function object
187 f.parameterNames = paramNames;
188
189 // Save the service
190 rpcjs.AbstractRpcHandler._servicesByKey[serviceName] = f;
191 },
192
193
194 /**
195  * Retrieve the parameter names for a registered service.
196  *
197  * @param serviceName {String}
198  *   The name of this service within the <[rpcKey]> namespace.
199  *
200  * @return {Array|null|undefined}
201  *   If the specified service exists and parameter names have been
202  *   provided for it, then an array of parameter names is returned.
203  *
204  *   If the service exists but no parameter names were provided in the
205  *   registration of the service, null is returned.
206  *
207  *   If the service does not exist, undefined is returned.
208  */
209 getServiceParamNames : function(serviceName)
210 {
211 // Get the stored service function
212 var f = rpcjs.AbstractRpcHandler._servicesByKey[serviceName];
213
214 // Did we find it?
215 if (! f)
216 {
217 // No, it is not a registered function.
218 return undefined;
219 }
220
221 // Were parameter names registered with the function?
222 if (f.parameterNames)
223 {
224 // Yup. Return a copy of the parameter name array
225 return qx.lang.Array.clone(f.parameterNames);
226 }
227
228 // The function was registered, but not its parameter names.
229 return null;
230 }
231 },
232
233 members :
234 {
235 /**
236  * Register a service name and function. This is just a convenience member
237  * method that calls the static function of the same name.
238  *
239  * @param serviceName {String}
240  *   The name of this service within the <[rpcKey]> namespace.
241  *
242  * @param fService {Function}
243  *   The function which implements the given service name.
244  *
245  * @param context {Object}
246  *   The context in which the service function should be called
247  *
248  * @param paramNames {Array}
249  *   The names of the formal parameters, in order.
250  */
251 registerService : function(serviceName, fService, context, paramNames)
252 {

```

```
253     rpcjs.AbstractRpcHandler.registerService(serviceName,
254                                             fService,
255                                             context,
256                                             paramNames);
257 },
258
259 /**
260  * Process an incoming request which is presumably a JSON-RPC request.
261  *
262  * @param jsonData {String}
263  *   The JSON-encoded RPC request to be processed
264  *
265  * @return {String}
266  *   Upon success, the JSON-encoded result of the RPC request is returned.
267  *   Otherwise, null is returned.
268  */
269 processRequest : function(jsonData)
270 {
271     // Call the RPC server to process this request
272     return this.__rpcServer.processRequest(jsonData);
273 }
274 }
275 });
```

Appendix 8

rpcjs.sim.Rpc

```

1  /*
2  * Copyright:
3  *   2011 Derrell Lipman
4  *
5  * License:
6  *   LGPL: http://www.gnu.org/licenses/lgpl.html
7  *   EPL: http://www.eclipse.org/org/documents/epl-v10.php
8  *   See the LICENSE file in the project's top-level directory for details.
9  *
10 * Authors:
11 *   * Derrell Lipman (derrell)
12 */
13
14 /*
15 #asset(rpcjs/*)
16 */
17
18
19 /**
20 * Handler for remote procedure calls.
21 */
22 qx.Class.define("rpcjs.sim.Rpc",
23 {
24     extend : rpcjs.AbstractRpcHandler,
25
26     /**
27      * Constructor for this RPC handler.
28      *
29      * @param rpcKey {Array}
30      *   The list of prefix keys for access to the set of remote
31      *   procedure calls supported in this object's services map.
32      *
33      *   Example: If the passed parameter is [ "sys", "fs" ] and
34      *   one of the methods later added is "read" then the remote
35      *   procedure call will be called "sys.fs.read", and the services
36      *   map will contain:
37      *
38      *   {
39      *     sys :
40      *     {
41      *       fs :
42      *       {
43      *         read : function()
44      *         {
45      *           // implementation of sys.fs.read()
46      *         }
47      *       }
48      *     }
49      *   }

```

```

49     *   }
50     *
51     * @param url {String}
52     *   The URL that must match for this service provider to be used
53     */
54     construct : function(rpckey, url)
55     {
56         // Call the superclass constructor
57         this.base(arguments, rpckey);
58
59         // Save the URL
60         this.setUrl(url);
61
62         // Register ourselves with the simulation transport, as a handler
63         // for the specified URL.
64         rpcjs.sim.Simulator.registerHandler(
65             qx.lang.Function.bind(this.__processRequest, this));
66     },
67
68     properties :
69     {
70         /** The URL which gains access to these RPC services */
71         url :
72         {
73             check      : "String",
74             nullable   : false
75         }
76     },
77
78     members :
79     {
80         /**
81          * Process an incoming request which is presumably a JSON-RPC request.
82          *
83          * @param request {Map}
84          *   A map of data for this request. See {@link rpcjs.sim.Simulator#post}
85          *   for details.
86          *
87          * @responseHeaders {Map}
88          *   A map containing, initially, two members: status {Number} and
89          *   statusText {String}. Upon error, this function may alter these two
90          *   values.
91          *
92          * @return {String}
93          *   Upon success, the JSON-encoded result of the RPC request is returned.
94          *   Otherwise, null is returned, and responseHeaders are updated to
95          *   indicate the source of the error.
96          */
97         __processRequest : function(request, responseHeaders)
98         {
99             var          jsonData;
100            var          result;
101
102            // Make sure we can handle this request
103            if (request.url != this.getUrl())
104            {
105                // We don't support this one. Response header status was preset.
106                return null;
107            }
108
109            // For the moment, we support only POST
110            if (request.method != "POST")
111            {
112                responseHeaders.status = 405;
113                responseHeaders.statusText =
114                    "Method " + request.method + " not allowed.";
115                return null;
116            }

```

```
117
118     // Retrieve the JSON-RPC data
119     jsonData = request.data;
120
121     // From here on out, we'll have a successful result (even if the RPC
122     // sends back an error response).
123     responseHeaders.status = 200;
124     responseHeaders.statusText = "";
125
126     // Call the RPC server to process this request
127     result = this.processRequest(jsonData);
128     return result;
129 }
130 }
131 });
```

Appendix 9

rpcjs.appengine.Rpc

```
1  /*
2  * Copyright:
3  *   2011 Derrell Lipman
4  *
5  * License:
6  *   LGPL: http://www.gnu.org/licenses/lgpl.html
7  *   EPL: http://www.eclipse.org/org/documents/epl-v10.php
8  *   See the LICENSE file in the project's top-level directory for details.
9  *
10 * Authors:
11 *   * Derrell Lipman (derrell)
12 */
13
14 /*
15 #asset(rpcjs/*)
16 */
17
18
19 /**
20 * Handler for remote procedure calls.
21 */
22 qx.Class.define("rpcjs.appengine.Rpc",
23 {
24   extend : rpcjs.AbstractRpcHandler
25
26   // Nothing special to do in this subclass.
27 });
```

Appendix 10

rpcjs.dbif.Entity

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  * Copyright (c) 2011 Reed Spool
4  *
5  * License:
6  *   LGPL: http://www.gnu.org/licenses/lgpl.html
7  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
8  */
9
10 /**
11 * The abstract base class from which all concrete entities descend.
12 */
13 qx.Class.define("rpcjs.dbif.Entity",
14 {
15   extend : qx.core.Object,
16   type   : "abstract",
17
18   /**
19    * Constructor for an Entity.
20    *
21    * @param entityType {String}
22    *   The name of this type of entity. It is used during normal operation to
23    *   retrieve the property types available for this entity.
24    *
25    * @param entityKey {String|Integer|Array}
26    *   If the entity key is a single field, this must be a string or integer
27    *   specifying that key value. If the entity key is composite, i.e., it is
28    *   made up of a series of fields, then this must be an array containing
29    *   the string or integer values of each of the fields in the key.
30    */
31   construct : function(entityType, entityKey)
32   {
33     var          i;
34     var          name;
35     var          queryResults;
36     var          keyField;
37     var          cloneObj;
38     var          properties;
39     var          entityData;
40     var          bComposite;
41     var          query;
42     var          field;
43     var          propertyType;
44     var          propertyTypes;
45     var          canonProps;
46     var          nonCanonKeyField = null;
47     var          nonCanonKey;
48

```

```

49 // Call the superclass constructor
50 this.base(arguments);
51
52 // Save the entity type
53 this.setEntityType(entityType);
54
55 // Gain easy access to the property type definition for this entity type
56 propertyTypes = rpcjs.dbif.Entity.propertyTypes[entityType];
57
58 // Get the key field name.
59 keyField = this.getEntityKeyProperty();
60
61 // Determine whether we have a composite key
62 bComposite = (qx.lang.Type.getClass(keyField) === "Array");
63
64 // Retrieve any pre-set entity data
65 entityData = this.getData();
66
67 // Was there any pre-set entity data?
68 if (!entityData)
69 {
70 // Nope. Initialize it.
71 entityData = {};
72 this.setData(entityData);
73 }
74
75 // Determine if the key field is autogenerated by canonicalization
76 canonProps = propertyTypes.canonicalize;
77 if (canonProps)
78 {
79 for (name in canonProps)
80 {
81 // Is this canonical property name our key field?
82 if (canonProps[name].prop == keyField)
83 {
84 // Yes. Save the non-canonicalized field name
85 nonCanonKeyField = name;
86
87 // Save the original (non-canonicalized) key value
88 nonCanonKey = entityKey;
89
90 // Canonicalize the entity key
91 entityKey = canonProps[name].func(entityKey);
92
93 // No need to search farther
94 break;
95 }
96 }
97 }
98
99 // If an entity key was specified...
100 if (typeof entityKey != "undefined" && entityKey != null)
101 {
102 // ... then query for the object.
103 queryResults =
104 rpcjs.dbif.Entity.query(this.constructor.classname, entityKey);
105
106 // Did we find anything there?
107 if (queryResults.length == 0)
108 {
109 // Nope. Just set this object's key. Is it composite?
110 if (bComposite)
111 {
112 // Yup. Set each of the fields
113 for (i = 0; i < keyField.length; i++)
114 {
115 entityData[keyField[i]] = entityKey[i];
116 }

```

```

117     }
118     else
119     {
120         // Not composite, so just assign the key to the single field
121         entityData[keyField] = entityKey;
122
123         // If this key field is canonicalized...
124         if (nonCanonKeyField)
125         {
126             // ... then save the original value too
127             entityData[nonCanonKeyField] = nonCanonKey;
128         }
129     }
130 }
131 else
132 {
133     // Set our object data to the data retrieved from the query
134     entityData = queryResults[0];
135     this.setData(entityData);
136
137     // It's not a brand new object
138     this.setBrandNew(false);
139 }
140 }
141
142 // Fill in any missing properties
143 for (propertyType in propertyTypes.fields)
144 {
145     // If this property type is not represented in the entity data...
146     if (typeof(entityData[propertyType]) == "undefined")
147     {
148         // ... then add it, with a null value.
149         entityData[propertyType] = null;
150     }
151 }
152
153 // If we're in debugging mode...
154 if (qx.core.Environment.get("qx.debug"))
155 {
156     // ... then ensure that there are no properties that don't belong
157     for (propertyType in entityData)
158     {
159         if (! (propertyType in
160             rpcjs.dbif.Entity.propertyTypes[entityType].fields))
161         {
162             throw new Error("Unrecognized property (" + propertyType + ") " +
163                 " in entity data for type " + entityType + ".");
164         }
165     }
166 }
167 },
168
169 properties :
170 {
171     /** A map containing the data for this entity */
172     data :
173     {
174         init : null,
175         check : "Object"
176     },
177
178     /** Flag indicating that this entity was newly created */
179     brandNew :
180     {
181         init : true,
182         check : "Boolean"
183     },
184

```

```

185  /**
186  * The property name that is to be used as the database entity key (aka
187  * primary key). If the key is composite, i.e., composed of more than one
188  * property, than this contains an array of strings, where each string is
189  * the name of a property, and the key is composed of the fields from each
190  * of these properties, in the order listed herein.
191  */
192  entityKeyProperty :
193  {
194      init : "uid",
195      check : "qx.lang.Type.isString(value) || qx.lang.Type.isArray(value)"
196  },
197
198  /** Mapping from classname to type used in the database */
199  entityType :
200  {
201      check : "String",
202      nullable : false
203  },
204
205  /**
206  * The unique id to be used as the database entity key (aka primary key),
207  * if no other property has been designated in entityKeyProperty as the
208  * primary key. The actual value is determined by the specific database
209  * interface in use, so this may be either an integer or a string.
210  */
211  uid :
212  {
213      init : null
214  }
215  },
216
217  statics :
218  {
219      /** Whether to strip the canonocalized fields from query results */
220      DEFAULT_STRIP_CANON : true,
221
222
223      /** Map from classname to entity type */
224      entityTypeMap : {},
225
226
227      /** Assignment of property types for each entity class */
228      propertyTypes : {},
229
230
231  /**
232  * Register an entity type. This is called by each subclass, immediately
233  * upon loading the subclass (typically in its defer: function), in order
234  * to register that subclass' entity type name to correspond with that
235  * subclass' class name.
236  *
237  * Each subclass of rpcjs.dbif.Entity represents a particular object type,
238  * and is identified by its class name and by its (shorter) entityType.
239  *
240  * @param classname {String}
241  *   The class name of the concrete subclass of rpcjs.dbif.Entity being
242  *   registered.
243  *
244  * @param entityType {String}
245  *   The short entity type name of the subclass being registered.
246  */
247  registerEntityType : function(classname, entityType)
248  {
249      // Save this value in the map from classname to entity type
250      rpcjs.dbif.Entity.entityTypeMap[classname] = entityType;
251  },
252

```

```

253
254 /**
255  * Register the property types for an entity class. This is called by each
256  * subclass, immediately upon loading the subclass (typically in its
257  * defer: function), in order to register the names of the properties
258  * (fields) that are stored for each object of this type.
259  *
260  * @param entityType {String}
261  *   The entity type name (as was passed to registerEntityType()), which
262  *   uniquely identifies this subclass of rpcjs.dbif.Entity.
263  *
264  * @param propertyTypes {Map}
265  *   A map containing, as its member names, the name of each of the
266  *   properties (fields) to be stored for each object of this type. The
267  *   value corresponding to each of those member names is the type of
268  *   value to be stored in that property, and may be any of: "String",
269  *   "LongString", "Date", "Key", "Integer", "Float", "KeyArray",
270  *   "StringArray", "LongStringArray", or "NumberArray".
271  *
272  * @param keyField {String|Array}
273  *   The name of the property that is to be used as the key field. If the
274  *   key is composite, i.e., composed of more than one property, than this
275  *   contains an array of strings, where each string is the name of a
276  *   property, and the key is composed of the fields from each of these
277  *   properties, in the order listed herein.
278  *
279  * @param canonicalize {Map?}
280  *   A map describing fields that are to be canonicalized, and the
281  *   canonicalization function. The map contains one or more members. Each
282  *   member name is a property name defined in the propertyTypes map. The
283  *   associated member value is itself a map, consisting of three members:
284  *   prop, which defines a new property in which the canonical value is to
285  *   be stored; type, the type of the canonical property (see the
286  *   propertyTypes parameter); and func, which is the canonicalization
287  *   function. The canonicalization function takes one parameter, the
288  *   value to be canonicalized, and returns the canonical value.
289  *
290  *   When a canonicalized field is provided, all query criteria
291  *   referencing the original field are modified to search using the
292  *   canonicalized field in its stead. The criterium is modified by
293  *   converting the search value using the canonicalization function, and
294  *   using the canonicalized property instead of the original property.
295  *
296  *   Here is an example canonicalize map. This one uses the "value"
297  *   property (previously defined in propertyTypes), and adds a
298  *   canonicalized value in a field called "value_lc". The canonicalized
299  *   value is created by converting the "value" field to lower case.
300  *   {
301  *     "value" :
302  *     {
303  *       // Property in which to store the canonicalized value. Since we
304  *       // are converting the value to lower case, we'll give the
305  *       // property a name suffix that reflects that.
306  *       prop : "value_lc",
307  *
308  *       // The canonicalized value will be a string
309  *       type : "String",
310  *
311  *       // Function to convert a value to lower case
312  *       func : function(value)
313  *       {
314  *         return value.toLowerCase();
315  *       }
316  *     }
317  *   };
318  *
319  *   Note that if the property being canonicalized is an array, the
320  *   specified function is called once for each member of the array, so

```

```

321     *   the resulting canonical value will also be an array, containing
322     *   individually-canonicalized values.
323     */
324     registerPropertyTypes : function(entityType,
325                                   propertyTypes,
326                                   keyField,
327                                   canonicalize)
328     {
329         var                pn;        // property name
330
331         // If there's no key field name specified...
332         if (! keyField)
333         {
334             // Add "uid" to the list of database properties.
335             propertyTypes["uid"] = "Key";
336             keyField = "uid";
337         }
338
339         // Add the canonicalize properties to the property list
340         if (canonicalize)
341         {
342             for (pn in canonicalize)
343             {
344                 // Add the property in which to store the canonicalized value
345                 // to the list of database properties.
346                 propertyTypes[canonicalize[pn].prop] = canonicalize[pn].type;
347             }
348         }
349
350         rpcjs.dbif.Entity.propertyTypes[entityType] =
351         {
352             keyField      : keyField,
353             fields        : propertyTypes,
354             canonicalize  : canonicalize
355         };
356     },
357
358
359
360     /**
361     * Function to query for objects.
362     *
363     * @param classname {String}
364     *   The name of the class, descended from rpcjs.dbif.Entity, of
365     *   the object type which is to be queried in the database.
366     *
367     * @param searchCriteria {Map?}
368     *   A (possibly recursive) map which contains the following members:
369     *   type {String}
370     *     "op" -- a logical operation. In this case, there must also be a
371     *     "method" member which contains the logical operation to
372     *     be performed. Currently, the only supported operation
373     *     at present is "and". There must also be a "children"
374     *     member, which is an array of the criteria to which the
375     *     specified operation is applied.
376     *
377     *     "element" -- Search by specific field in the object. The
378     *     "field" member must be provided, to specify which
379     *     field, and a "value" member must be specified, to
380     *     indicate what value must be in that field.
381     *
382     *     An optional "filterOp" member may also be
383     *     provided. If none is provided, the requested
384     *     operation is assumed to be equality. Any of the
385     *     following values may be provided for the "filterOp"
386     *     member: "<", "<=", "=", ">", ">=", "!=".
387     *
388     *   If no criteria is supplied (undefined or null), then all objects of

```

```

389     *   the specified classname will be returned.
390     *
391     * @param resultCriteria {Array?}
392     *   An array of maps. Each map contains 2 or more members: a "type" and one
393     *   or more type dependent members. The type "offset" necessitates a member
394     *   "value" with an integer value specifying how many initial objects to
395     *   skip from the result objects. The offset must be non-negative. The type
396     *   "limit" also requires a member "value" whose integer value is the total
397     *   number of result objects to return. The limit must be greater than
398     *   zero. When the type is "sort", two members must be present:
399     *   "field" and "order". The "field" is a string specifying the result
400     *   object member on which to sort. The "order" is also a string, either
401     *   "asc" or "desc".
402     *
403     *   There may be zero or one of the maps of type "limit" or "offset". Of
404     *   the maps of type "sort", there may be any number, and their order in
405     *   the array is the order the sort is applied on each field.
406     *
407     *   A final "type" is called "option", in which case there must be a
408     *   "name" field to specify which option is being set, and a "value"
409     *   field which is of the correct type for the specified name. The
410     *   possible option names and values types are:
411     *
412     *       stripCanon : boolean (default rpcjs.dbif.Entity.DEFAULT_STRIP_CANON)
413     *
414     *
415     *   An example resultCriteria value might be,
416     *   [
417     *     { type : "offset", value : 10 },
418     *     { type : "limit", value : 5 },
419     *     { type : "sort", field : "uploadTime", order : "desc" },
420     *     { type : "sort", field : "numLikes", order : "asc" }
421     *     { type : "option", name : "stripCanon", value : true }
422     *   ]
423     *
424     *
425     * @return {Array}
426     *   An array of maps, i.e. native objects (not of Entity objects!)
427     *   containing the data resulting from the query.
428     */
429     query : function(classname, searchCriteria, resultCriteria)
430     {
431         var          ret;
432         var          entityType;
433         var          canonicalize;
434         var          canonFields;
435         var          bStripCanon = rpcjs.dbif.Entity.DEFAULT_STRIP_CANON;
436
437         // Get the entity type
438         entityType = rpcjs.dbif.Entity.entityTypeMap[classname];
439         if (! entityType)
440         {
441             throw new Error("No mapped entity type for " + classname);
442         }
443
444         // Gain easy access to the canonicalize map.
445         canonicalize =
446             rpcjs.dbif.Entity.propertyTypes[entityType].canonicalize;
447         if (canonicalize)
448         {
449             // Get a list of the fields to be mapped
450             canonFields = qx.lang.Object.getKeys(canonicalize);
451         }
452
453         // Are there any canonicalization functions for this class
454         if (canonicalize && searchCriteria)
455         {
456             // Yup. Rebuild the search criteria, replacing non-canonicalized

```

```

457 // fields with their peer canonical fields. Start with a clone
458 // of the original criteria, so we don't modify the caller's map.
459 searchCriteria = qx.util.Serializer.toNativeObject(searchCriteria);
460
461 // Recursively descend through the search criteria, replacing
462 // non-canonicalized field names with their canonical peer.
463 (function replaceCanonFields(criterion)
464 {
465 // Null or undefined means retrieve all objects.
466 if (! criterion)
467 {
468 // Nothing for us to do
469 return;
470 }
471
472 // element criterion (type="element" or no type field)?
473 if (! criterion.type || criterion.type == "element")
474 {
475 // Is this field name one to be canonicalized?
476 if (qx.lang.Array.contains(canonFields, criterion.field))
477 {
478 // Replace the value with the canonical version
479 criterion.value =
480 canonicalize[criterion.field].func(criterion.value);
481
482 // Replace the field name with its canonical peer
483 criterion.field = canonicalize[criterion.field].prop;
484 }
485 }
486 else if (criterion.children)
487 {
488 // If there are children, deal with each of them.
489 criterion.children.forEach(
490 function(child)
491 {
492 replaceCanonFields(criterion[child]);
493 });
494 }
495 })(searchCriteria);
496 }
497
498 // Issue the query
499 ret = rpcjs.dbif.Entity.__query(classname,
500 searchCriteria,
501 resultCriteria);
502
503 // Match options
504 if (resultCriteria)
505 {
506 resultCriteria.forEach(
507 function(criterion)
508 {
509 if (criterion.type == "option")
510 {
511 switch(criterion.name)
512 {
513 case "stripCanon": // strip canonical fields from result
514 bStripCanon = criterion.value;
515 break;
516
517 default:
518 this.warn("Unrecognized option name: " + criterion.name);
519 break;
520 }
521 }
522 });
523 }
524

```

```

525
526 // If there is a canonicalization map...
527 if (bStripCanon && canonicalize)
528 {
529     // ... then for each entry in the canonicalization map, ...
530     canonFields.forEach(
531         function(field)
532         {
533             // ... delete the corresponding canonical field from the return map
534             ret.forEach(
535                 function(item)
536                 {
537                     delete item[canonicalize[field].prop];
538                 });
539             });
540     }
541
542     return ret;
543 },
544
545
546
547 /**
548  * Function to query for objects. The actual function that's used depends
549  * on which database driver gets installed. The database driver will
550  * register the function with us so user code can always use a common
551  * entry point to the function, here.
552  *
553  * See query() documentation for details.
554  */
555 __query : function(classname, searchCriteria, resultCriteria)
556 {
557     // This is a temporary place holder.
558     //
559     // This method is replaced by the query method of the specific database
560     // that is being used.
561     return [];
562 },
563
564
565 /**
566  * Function to put an object to the database. The actual function that's
567  * used depends on which database driver gets installed. The database
568  * driver will register the function with us so user code can always use a
569  * common entry point to the function, this.put().
570  *
571  * @param entity {rpcjs.dbif.Entity}
572  * The object whose database properties are to be written out.
573  */
574 __put : function(entity)
575 {
576     // This is a temporary place holder.
577     //
578     // This method is replaced by the put method of the specific database
579     // that is being used.
580 },
581
582
583 /**
584  * Remove an entity from the database
585  *
586  * @param entity {rpcjs.dbif.Entity}
587  * An instance of the entity to be removed.
588  */
589 __remove : function(entity)
590 {
591     // This is a temporary place holder.
592     //

```

```

593     // This method is replaced by the remove method of the specific database
594     // that is being used.
595 },
596
597
598 /**
599  * Add a blob to the database.
600  *
601  * @param blobData {LongString}
602  *   The data to be written as a blob
603  *
604  * @return {Key}
605  *   The blob ID of the just-added blob
606  *
607  * @throws {Error}
608  *   If an error occurs while writing the blob to the database, an Error
609  *   is thrown.
610  */
611 putBlob : function(blobData)
612 {
613     // This is a temporary place holder.
614     //
615     // This method is replaced by the putBlob method of the specific
616     // database that is being used.
617 },
618
619
620 /**
621  * Retrieve a blob from the database
622  *
623  * @param blobId {Key}
624  *   The blob ID of the blob to be retrieved
625  *
626  * @return {LongString}
627  *   The blob data retrieved from the database. If there is no blob with
628  *   the given ID, undefined is returned.
629  */
630 getBlob : function(entity)
631 {
632     // This is a temporary place holder.
633     //
634     // This method is replaced by the getBlob method of the specific
635     // database that is being used.
636 },
637
638
639 /**
640  * Remove a blob from the database
641  *
642  * @param blobId {Key}
643  *   The blob ID of the blob to be removed. If the specified blob id does
644  *   not exist, this request fails silently.
645  */
646 removeBlob : function(entity)
647 {
648     // This is a temporary place holder.
649     //
650     // This method is replaced by the removeBlob method of the specific
651     // database that is being used.
652 },
653
654
655
656
657 /**
658  * Register functions which are specific to a certain database interface
659  *
660  * @param query {Function}

```

```

661 * The database-specific function to be used to query the database. It
662 * must provide the signature of {@link query}.
663 *
664 * @param put {Function}
665 * The database-specific function to be used to write an entry to the
666 * database. It must provide the signature of {@link __put}.
667 *
668 * @param remove {Function}
669 * The database-specific function to be used to remove an entry from the
670 * database. It must provide the signature of {@link __remove}.
671 *
672 * @param getBlob {Function}
673 * The database-specific function to be used to retrieve a blob from the
674 * database. It must provide the signature of {@link __getBlob}.
675 *
676 * @param putBlob {Function}
677 * The database-specific function to be used to write a blob to the
678 * database. It must provide the signature of {@link __putBlob}.
679 *
680 * @param removeBlob {Function}
681 * The database-specific function to be used to remove a blob from the
682 * database. It must provide the signature of {@link __removeBlob}.
683 */
684 registerDatabaseProvider : function(query, put, remove,
685                                     getBlob, putBlob, removeBlob)
686 {
687     // Save the specified functions.
688     rpcjs.dbif.Entity.__query = query;
689     rpcjs.dbif.Entity.__put = put;
690     rpcjs.dbif.Entity.__remove = remove;
691     rpcjs.dbif.Entity.getBlob = getBlob;
692     rpcjs.dbif.Entity.putBlob = putBlob;
693     rpcjs.dbif.Entity.removeBlob = removeBlob;
694 }
695 },
696
697 members :
698 {
699     /**
700     * Put the db property data in this object to the database.
701     */
702     put : function()
703     {
704         var name;
705         var entityType = this.getEntityType();
706         var propertyTypes;
707         var canonicalize;
708         var data;
709         var newArray;
710
711         // Retrieve the property data
712         data = this.getData();
713
714         // Canonicalize each value that has canonicalization specified
715         propertyTypes = rpcjs.dbif.Entity.propertyTypes[entityType];
716         canonicalize = propertyTypes.canonicalize;
717         if (canonicalize)
718         {
719             for (name in canonicalize)
720             {
721                 // If the property is an array type, ...
722                 if (qx.lang.Array.contains(
723                     [
724                         "KeyArray",
725                         "StringArray",
726                         "LongStringArray",
727                         "NumberArray"
728                     ],

```

```

729         propertyTypes.fields[name]))
730     {
731         // ... then canonicalize each value within the array
732         newArray = [];
733         data[name].forEach(
734             function(elem)
735             {
736                 newArray.push(canonicalize[name].func(elem));
737             });
738         data[canonicalize[name].prop] = newArray;
739     }
740     else
741     {
742         data[canonicalize[name].prop] = canonicalize[name].func(data[name]);
743     }
744 }
745 }
746
747 // Write this data
748 rpcjs.dbif.Entity.__put(this);
749
750 // This entity is no longer brand new
751 this.setBrandNew(false);
752 },
753
754 /**
755  * Remove this entity from the database.
756  *
757  * NOTE: This object should no longer be used after having called this
758  *       method!
759  */
760 removeSelf : function()
761 {
762     // Remove ourselves from the database
763     rpcjs.dbif.Entity.__remove(this);
764
765     // Mark this entity as brand new again.
766     this.setBrandNew(true);
767 },
768
769 /**
770  * Provide the map of database properties and their types.
771  *
772  * @return {Map}
773  *   A map where the key is a database property, and the value is its
774  *   type. Valid types are "String", "Number", "Array", and "Key". The
775  *   latter is database implementation dependent.
776  */
777 getDatabaseProperties : function()
778 {
779     var props = rpcjs.dbif.Entity.propertyTypes[this.getEntityType()];
780     return props;
781 }
782 }
783 });

```

Appendix 11

rpcjs.appengine.Dbif

```

1  /**
2   * Copyright (c) 2011 Derrell Lipman
3   * Copyright (c) 2011 Reed Spool
4   *
5   * License:
6   *   LGPL: http://www.gnu.org/licenses/lgpl.html
7   *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
8   */
9
10 qx.Class.define("rpcjs.appengine.Dbif",
11 {
12   extend : qx.core.Object,
13   type   : "abstract",
14
15   statics :
16   {
17     /**
18      * The remote procedure code server for App Engine
19      */
20     __rpcHandler : null,
21
22     /**
23      * The next value to use for an auto-generated key for an entity
24      */
25     __nextKey : 1,
26
27     /*
28      * Build a composite key.
29      *
30      * By default, this separates the key values using ASCII value 31, which
31      * is "Unit Separator". Applications that require that value to be
32      * allowable in key component values may replace this method with one that
33      * builds the key differently.
34      */
35     _buildCompositeKey : function(keyArr)
36     {
37       return keyArr.join(String.fromCharCode(31));
38     },
39
40
41     /**
42      * Query for all entities of a given class/type, given certain criteria.
43      *
44      * @param classname {String}
45      *   The name of the class, descended from rpcjs.dbif.Entity, of
46      *   the object type which is to be queried in the database.
47      *
48      * @param criteria

```



```

117
118     dbResults = [];
119     dbResults.push(result);
120
121     // Make dbResults look like a Java array, so we can convert its
122     // contents to JavaScript types in code which is common with a Query
123     // result set.
124     dbResults.hasNext = function() { return this.length > 0; };
125     dbResults.next = function() { return this.shift(); };
126     break;
127
128 default:
129     // Create a new query
130     Query = Datastore.Query;
131     query = new Query(type);
132
133     // If they're not asking for all objects, build a criteria predicate.
134     if (searchCriteria)
135     {
136         (function(criterion)
137         {
138             var filterOp;
139
140             switch(criterion.type)
141             {
142             case "op":
143                 switch(criterion.method)
144                 {
145                 case "and":
146                     // Generate the conditions specified in the children
147                     criterion.children.forEach(arguments.callee);
148                     break;
149
150                 default:
151                     throw new Error("Unrecognized criterion method: " +
152                                     criterion.method);
153                 }
154                 break;
155
156             case "element":
157                 // Map the specified filter operator to the db's filter ops.
158                 filterOp = criterion.filterOp || "=";
159                 switch(filterOp)
160                 {
161                 case "<=":
162                     filterOp = Query.FilterOperator.LESS_THAN_OR_EQUAL;
163                     break;
164
165                 case "<":
166                     filterOp = Query.FilterOperator.LESS_THAN;
167                     break;
168
169                 case "=":
170                     filterOp = Query.FilterOperator.EQUAL;
171                     break;
172
173                 case ">":
174                     filterOp = Query.FilterOperator.GREATER_THAN;
175                     break;
176
177                 case ">=":
178                     filterOp = Query.FilterOperator.GREATER_THAN_OR_EQUAL;
179                     break;
180
181                 case "!=":
182                     filterOp = Query.FilterOperator.NOT_EQUAL;
183                     break;
184

```

```

185         default:
186             throw new Error("Unrecognized comparison operation: " +
187                 criterion.filterOp);
188         }
189
190         // Add a filter using the provided parameters
191         if (fields[criterion.field] == "Integer" ||
192             fields[criterion.field] == "Key")
193         {
194             query.addFilter(criterion.field,
195                 filterOp,
196                 java.lang.Integer(criterion.value));
197         }
198         else
199         {
200             query.addFilter(criterion.field,
201                 filterOp,
202                 criterion.value);
203         }
204         break;
205
206         default:
207             throw new Error("Unrceognized criterion type: " +
208                 criterion.type);
209     }
210     })(searchCriteria);
211 }
212
213
214 // Assume the default set of result criteria (no limits, offset=0)
215 options = Datastore.FetchOptions.Builder.withDefaults();
216
217 // If there are any result criteria specified...
218 if (resultCriteria)
219 {
220     // ... then go through the criteria list and handle each.
221     resultCriteria.forEach(
222         function(criterion)
223         {
224             switch(criterion.type)
225             {
226                 case "limit":
227                     options.limit(criterion.value);
228                     break;
229
230                 case "offset":
231                     options.offset(criterion.value);
232                     break;
233
234                 case "sort":
235                     query.addSort(criterion.field,
236                         {
237                             "asc" : Query.SortDirection.ASCENDING,
238                             "desc" : Query.SortDirection.DESCENDING
239                         }[criterion.order]);
240                     break;
241
242                 default:
243                     throw new Error("Unrecognized result criterion type: " +
244                         criterion.type);
245             }
246         }
247     });
248
249 // Prepare to issue a query
250 preparedQuery = datastore.prepare(query);
251
252 // Issue the query

```

```

253     dbResults = preparedQuery.asIterator(options);
254     break;
255 }
256
257 // Process the query results
258 while (dbResults.hasNext())
259 {
260     // Initialize a map for the result data
261     result = {};
262
263     // Get the next result
264     dbResult = dbResults.next();
265
266     // Pull all of the result properties into the entity data
267     for (fieldName in fields)
268     {
269         // Map the Java field data to appropriate JavaScript data
270         result[fieldName] =
271             (function(value, type)
272             {
273                 var          ret;
274                 var          Text;
275                 var          iterator;
276
277                 switch(type)
278                 {
279                     case "String":
280                     case "Date":
281                         return value ? String(value) : null;
282
283                     case "LongString":
284                         return value ? String(value.getValue()) : null;
285
286                     case "Key":
287                     case "Integer":
288                     case "Float":
289                         return(Number(value));
290
291                     case "KeyArray":
292                     case "StringArray":
293                     case "LongStringArray":
294                     case "NumberArray":
295                         if (value)
296                         {
297                             // Initialize the return array
298                             ret = [];
299
300                             // Determine the type of the elements
301                             var elemType = type.replace(/Array/, "");
302
303                             // Convert the elements to their proper types
304                             iterator = value.iterator();
305                             while (iterator.hasNext())
306                             {
307                                 // Call ourself with this element
308                                 ret.push(arguments.callee(iterator.next(), elemType));
309                             }
310
311                             return ret;
312                         }
313                     else
314                     {
315                         return [];
316                     }
317
318                 default:
319                     throw new Error("Unknown property type: " + type);
320             }

```

```

321         })(dbResult.getProperty(fieldName), fields[fieldName]);
322     }
323
324     // Save this result
325     results.push(result);
326 }
327
328 // Give 'em the query results!
329 return results;
330 },
331
332
333 /**
334  * Put an entity to the database. If the key field is null or undefined, a
335  * key is automatically generated for the entity.
336  *
337  * @param entity {rpcjs.dbif.Entity}
338  *   The entity to be made persistent.
339  */
340 put : function(entity)
341 {
342     var         dbKey;
343     var         dbEntity;
344     var         datastoreService;
345     var         Datastore;
346     var         entityData = entity.getData();
347     var         keyProperty = entity.getEntityKeyProperty();
348     var         type = entity.getEntityType();
349     var         propertyName;
350     var         propertyType;
351     var         fields;
352     var         fieldName;
353     var         data;
354     var         key;
355     var         keyRange;
356     var         keyFields = [];
357
358     // Gain access to the datastore service
359     Datastore = Packages.com.google.appengine.api.datastore;
360     datastoreService =
361         Datastore.DatastoreServiceFactory.getDatastoreService();
362
363
364     // Are we working with a composite key?
365     if (qx.lang.Type.getClass(keyProperty) == "Array")
366     {
367         // Yup. Build the composite key from these fields
368         keyProperty.forEach(
369             function(fieldName)
370             {
371                 keyFields.push(entityData[fieldName]);
372             }
373         );
374         key = rpcjs.appengine.Dbif._buildCompositeKey(keyFields);
375     }
376     else
377     {
378         // Retrieve the (single field) key
379         key = entityData[keyProperty];
380     }
381
382     // Ensure that there's either a real key or no key; not empty string
383     if (key == "")
384     {
385         throw new Error("Found disallowed empty key");
386     }
387
388     // Get the field names for this entity type
389     fields = entity.getDatabaseProperties().fields;

```

```

389
390 // If there's no key yet...
391 dbKey = null;
392
393 // Note: Rhino (and thus App Engine which uses Rhino-compiled code)
394 // causes "global" to returned by qx.lang.Type.getClass(key) if key is
395 // null or undefined. Without knowing whether it returns "global" in any
396 // other case, we test for those two cases explicitly.
397 switch(key === null || key === undefined
398        ? "Null"
399        : qx.lang.Type.getClass(key))
400 {
401 case "Null":
402 case "Undefined": // Never occurs due to explicit test above
403 // Generate a new key. Determine what type of key to use.
404 switch(fields[keyProperty])
405 {
406 case "Key":
407 case "Number":
408 // Obtain a unique key that no other running instances will obtain.
409 keyRange =
410     datastoreService.allocateIds(entity.getEntityType(), 1);
411
412 // Get its numeric value
413 dbKey = keyRange.getStart();
414 key = dbKey.getId();
415 break;
416
417 case "String":
418 // Obtain a unique key that no other running instances will obtain.
419 keyRange =
420     datastoreService.allocateIds(entity.getEntityType(), 1);
421
422 // Get its numeric value
423 dbKey = keyRange.getStart();
424 key = dbKey.getName();
425 break;
426
427 default:
428     throw new Error("No way to autogenerate key");
429 }
430
431 // Save this key in the key field
432 entityData[keyProperty] = key;
433 break;
434
435 case "Number":
436 // Save this key in the key field
437 entityData[entity.getEntityKeyProperty()] = key;
438 break;
439
440 case "Array":
441 // Build a composite key string from these key values
442 key = rpcjs.appengine.Dbif._buildCompositeKey(key);
443 break;
444
445 case "String":
446 // nothing special to do
447 break;
448
449 default:
450     break;
451 }
452
453 // If we didn't auto-generate one, ...
454 if (dbKey === null)
455 {
456 // ... create the database key value

```

```

457     dbKey = Datastore.KeyFactory.createKey(entity.getEntityType(), key);
458 }
459
460 // Create an App Engine entity to store in the database
461 dbEntity = new Packages.com.google.appengine.api.datastore.Entity(dbKey);
462
463 // Add each property to the database entity
464 for (fieldName in fields)
465 {
466     // Map the Java field data to appropriate JavaScript data
467     data =
468         (function(value, type)
469         {
470             var i;
471             var v;
472             var conv;
473             var jArr;
474
475             switch(type)
476             {
477                 case "String":
478                 case "Float":
479                     return value;
480
481                 case "Key":
482                 case "Integer":
483                     // Convert JavaScript Number to Java Long to avoid floating point
484                     return java.lang.Long(String(value));
485
486                 case "LongString":
487                     var Text = Packages.com.google.appengine.api.datastore.Text;
488                     return value ? new Text(value) : value;
489
490                 case "KeyArray":
491                 case "StringArray":
492                 case "LongStringArray":
493                 case "IntegerArray":
494                 case "FloatArray":
495                     if (type == "IntegerArray")
496                     {
497                         // integer Numbers must be converted to Java longs to avoid
498                         // having them saved as floating point values.
499                         conv = function(val)
500                         {
501                             return java.lang.Long(String(val));
502                         };
503                     }
504                     else
505                     {
506                         conv = function(val)
507                         {
508                             return val;
509                         };
510                     }
511
512                     jArr = new java.util.ArrayList();
513                     for (i = 0; value && i < value.length; i++)
514                     {
515                         jArr.add(arguments.callee(conv(value[i]),
516                             type.replace(/Array/, "")));
517                     }
518                     return jArr;
519
520                 default:
521                     throw new Error("Unknown property type: " + type);
522             }
523         })(entityData[fieldName], fields[fieldName]);
524

```

```

525     // Save this result
526     dbEntity.setProperty(fieldName, data);
527 }
528
529 // Save it to the database
530 datastoreService.put(dbEntity);
531 },
532
533
534 /**
535  * Remove an entity from the database
536  *
537  * @param entity {rpcjs.dbif.Entity}
538  *   An instance of the entity to be removed.
539  */
540 remove : function(entity)
541 {
542     var          entityData = entity.getData();
543     var          keyProperty = entity.getEntityKeyProperty();
544     var          type = entity.getEntityType();
545     var          propertyName;
546     var          fields;
547     var          key;
548     var          keyFields = [];
549     var          dbKey;
550     var          datastore;
551     var          Datastore;
552
553     // Are we working with a composite key?
554     if (qx.lang.Type.getClass(keyProperty) == "Array")
555     {
556         // Yup. Build the composite key from these fields
557         keyProperty.forEach(
558             function(fieldName)
559             {
560                 keyFields.push(entityData[fieldName]);
561             });
562         key = rpcjs.sim.Dbif._buildCompositeKey(keyFields);
563     }
564     else
565     {
566         // Retrieve the (single field) key
567         key = entityData[keyProperty];
568     }
569
570     // Create the database key value
571     Datastore = Packages.com.google.appengine.api.datastore;
572     dbKey = Datastore.KeyFactory.createKey(type, key);
573
574     // Remove this entity from the database
575     datastore = Datastore.DatastoreServiceFactory.getDatastoreService();
576     datastore["delete"](dbKey);
577 },
578
579 /**
580  * Add a blob to the database.
581  *
582  * @param blobData {LongString}
583  *   The data to be written as a blob
584  *
585  * @return {String}
586  *   The blob ID of the just-added blob
587  *
588  * @throws {Error}
589  *   If an error occurs while writing the blob to the database, an Error
590  *   is thrown.
591  */
592 putBlob : function(blobData)

```

```

593     {
594         var          key;
595         var          file;
596         var          fileService;
597         var          writeChannel;
598         var          printWriter;
599         var          segment;
600         var          segmentSize;
601         var          FileServiceFactory;
602         var          Channels;
603         var          PrintWriter;
604
605         FileServiceFactory =
606             Packages.com.google.appengine.api.files.FileServiceFactory;
607         Channels = java.nio.channels.Channels;
608         PrintWriter = java.io.PrintWriter;
609
610         // Get a file service
611         fileService = FileServiceFactory.getFileService();
612
613         // Create a new blob file with mime type "text/plain"
614         file = fileService.createNewBlobFile("text/plain");
615
616         // Open a write channel, with lock=true so we can finalize it
617         writeChannel = fileService.openWriteChannel(file, true);
618
619         // Get a print writer for this channel, so we can write a string
620         printWriter = new PrintWriter(Channels.newWriter(writeChannel, "UTF8"));
621
622         // Write our blob data as a series of 32k writes
623         printWriter.write(blobData);
624         printWriter.close();
625
626         // Finalize the channel
627         writeChannel.closeFinally();
628
629         // Retrieve the blob key for this file
630         key = fileService.getBlobKey(file).getKeyString();
631
632         // Give 'em the blob id
633         return String(key);
634     },
635
636     /**
637     * Retrieve a blob from the database
638     *
639     * @param blobId {Key}
640     *     The blob ID of the blob to be retrieved
641     *
642     * @return {LongString}
643     *     The blob data retrieved from the database. If there is no blob with
644     *     the given ID, undefined is returned.
645     */
646     getBlob : function(blobId)
647     {
648         var          blob;
649         var          blobstoreService;
650         var          blobKey;
651         var          blobInfoFactory;
652         var          blobInfo;
653         var          size;
654         var          maxFetchSize;
655         var          segmentSize;
656         var          startIndex;
657         var          endIndex;
658         var          BlobstoreService;
659         var          BlobstoreServiceFactory;
660         var          BlobKey;

```

```

661     var                BlobInfoFactory;
662
663     BlobstoreService =
664         Packages.com.google.appengine.api.blobstore.BlobstoreService;
665     BlobstoreServiceFactory =
666         Packages.com.google.appengine.api.blobstore.BlobstoreServiceFactory;
667     BlobKey = Packages.com.google.appengine.api.blobstore.BlobKey;
668     BlobInfoFactory = com.google.appengine.api.blobstore.BlobInfoFactory;
669
670     // Get a blobstore service
671     blobstoreService = BlobstoreServiceFactory.getBlobstoreService();
672
673     // Convert the (string) blobId to a blob key
674     blobKey = new BlobKey(blobId);
675
676     // Load the information about this blob
677     blobInfoFactory = new BlobInfoFactory();
678     blobInfo = blobInfoFactory.loadBlobInfo(blobKey);
679     size = blobInfo.getSize();
680
681     // Retrieve the blob
682     blob = [];
683     maxFetchSize = 1015808;
684     startIndex = 0;
685     while (size > 0)
686     {
687         // Determine how much to fetch. Use largest available size within limit.
688         segmentSize = Math.min(size, maxFetchSize);
689         endIndex = startIndex + segmentSize - 1;
690
691         // Fetch a blob segment and convert it to a Java string.
692         blob.push(
693             String(
694                 new java.lang.String(
695                     blobstoreService.fetchData(blobKey, startIndex, endIndex)));
696
697         // Update our start index for next time
698         startIndex += segmentSize;
699
700         // We've used up a bunch of the blob. Update remaining size.
701         size -= segmentSize;
702     }
703
704     // Join all of the parts together.
705     blob = blob.join("");
706
707     // Give 'em what they came for
708     return blob;
709 },
710
711 /**
712  * Remove a blob from the database
713  *
714  * @param blobId {Key}
715  * The blob ID of the blob to be removed. If the specified blob id does
716  * not exist, this request fails silently.
717  */
718 removeBlob : function(blobId)
719 {
720     var                blobstoreService;
721     var                blobKey;
722     var                BlobstoreServiceFactory;
723     var                BlobKey;
724
725     BlobstoreServiceFactory =
726         Packages.com.google.appengine.api.blobstore.BlobstoreServiceFactory;
727     BlobKey = Packages.com.google.appengine.api.blobstore.BlobKey;
728

```

```
729     // Get a blobstore service
730     blobstoreService = BlobstoreServiceFactory.getBlobstoreService();
731
732     // Convert the (string) blobId to a blob key
733     blobKey = new BlobKey(blobId);
734
735     // Delete the blob
736     blobstoreService["delete"](blobKey);
737 }
738 },
739
740 defer : function()
741 {
742     // Register our put, query, and remove functions
743     rpcjs.dbif.Entity.registerDatabaseProvider(
744         rpcjs.appengine.Dbif.query,
745         rpcjs.appengine.Dbif.put,
746         rpcjs.appengine.Dbif.remove);
747 }
748 });
```

Appendix 12

rpcjs.sim.Dbif

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  * Copyright (c) 2011 Reed Spool
4  *
5  * License:
6  *   LGPL: http://www.gnu.org/licenses/lgpl.html
7  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
8  */
9
10 qx.Class.define("rpcjs.sim.Dbif",
11 {
12   extend : qx.core.Object,
13   type   : "abstract",
14
15   statics :
16   {
17     /** The default database. See {@link setDb}. */
18     Database : null,
19
20     /** The database map entry to use for blobs */
21     BlobStorage : "***BLOB**",
22
23     /**
24      * The next value to use for an auto-generated key for an entity
25      */
26     __nextKey : 0,
27
28     /**
29      * Build a composite key.
30      *
31      * By default, this separates the key values using ASCII value 31, which
32      * is "Unit Separator". Applications that require that value to be
33      * allowable in key component values may replace this method with one that
34      * builds the key differently.
35      */
36     _buildCompositeKey : function(keyArr)
37     {
38       return keyArr.join(String.fromCharCode(31));
39     },
40
41
42     /**
43      * Save the user-specified database.
44      *
45      * @param db {Map}
46      *   The database map. Each top-level key in this map is an object type,
47      *   which contains a map of key/value pairs.
48      */

```

```

49  setDb : function(db)
50  {
51      rpcjs.sim.Dbif.Database = db;
52  },
53
54
55  /**
56   * Query for all entities of a given class/type, given certain criteria.
57   *
58   * @param classname {String}
59   *   The name of the class, descended from rpcjs.dbif.Entity, of
60   *   the object type which is to be queried in the database.
61   *
62   * @param searchCriteria
63   *   See {@link rpcjs.dbif.Entity#query} for details.
64   *
65   * @param resultCriteria
66   *   See {@link rpcjs.dbif.Entity#query} for details.
67   *
68   * @return {Array}
69   *   An array of maps, i.e. native objects (not of Entity objects!)
70   *   containing the data resulting from the query.
71   */
72  query : function(classname, searchCriteria, resultCriteria)
73  {
74      var          i;
75      var          qualifies;
76      var          builtCriteria;
77      var          dbObjectMap;
78      var          type;
79      var          entry;
80      var          propertyName;
81      var          result;
82      var          results;
83      var          entity;
84      var          clone;
85      var          val;
86      var          limit = Number.MAX_VALUE;
87      var          offset = 0;
88      var          sortKeys;
89      var          builtSort;
90      var          sortFunction;
91
92      // Get the entity type
93      type = rpcjs.dbif.Entity.entityTypeMap[classname];
94      if (! type)
95      {
96          throw new Error("No mapped entity type for " + classname);
97      }
98
99      // Get the database sub-section for the specified classname/type
100     dbObjectMap = rpcjs.sim.Dbif.Database[type];
101
102     if (qx.core.Environment.get("qx.debug"))
103     {
104         if (! dbObjectMap)
105         {
106             throw new Error("Type '" + type + "' " +
107                             "was not found in the simulation database.");
108         }
109     }
110
111     // Initialize our results array
112     results = [];
113
114     // If we've been given a key (single field or composite), just look up
115     // that single entity and return it.
116     switch(qx.lang.Type.getClass(searchCriteria))

```

```

117     {
118     case "Array":
119         // Join the field values using a known field separator
120         searchCriteria = rpcjs.sim.Dbif._buildCompositeKey(searchCriteria);
121
122         // fall through
123
124     case "Number":
125     case "String":
126         if (typeof dbObjectMap[searchCriteria] !== "undefined")
127         {
128             // Make a deep copy of the results
129             result =
130                 qx.util.Serializer.toNativeObject(dbObjectMap[searchCriteria]);
131             results.push(result);
132         }
133         return results;
134     }
135
136     // If they're not asking for all objects, build a criteria predicate.
137     if (searchCriteria)
138     {
139         builtCriteria =
140             (function(criterion)
141             {
142                 var i;
143                 var ret = "";
144                 var propertyTypes;
145                 var filterOp;
146
147                 // Convert or determine the filter operation
148                 filterOp =
149                     (function(filterOp)
150                     {
151                         switch(filterOp)
152                         {
153                         case "<=":
154                         case "<":
155                         case ">":
156                         case ">=":
157                             return filterOp; // Use filter operation as provided
158
159                         case "!=":
160                             return "!="; // "Not identical"
161
162                         case "=":
163                         case undefined:
164                             return "==="; // "Identical"
165
166                         default:
167                             throw new Error("Unexpected filter operation: " +
168                                 filterOp);
169                         }
170                     })(criterion.filterOp);
171
172                 switch(criterion.type)
173                 {
174                 case "op":
175                     switch(criterion.method)
176                     {
177                     case "and":
178                         // Generate the conditions
179                         ret += "(";
180                         for (i = 0; i < criterion.children.length; i++)
181                         {
182                             ret += arguments.callee(criterion.children[i]);
183                             if (i < criterion.children.length - 1)
184                                 {

```

```

185         ret += " && ";
186     }
187 }
188 ret += ")";
189 break;
190
191 default:
192     throw new Error("Unrecognized criterion method: " +
193         criterion.method);
194 }
195 break;
196
197 case "element":
198     // Determine the type of this field
199     propertyTypes = rpcjs.dbif.Entity.propertyTypes;
200     switch(propertyTypes[type].fields[criterion.field])
201     {
202     case "String":
203     case "LongString":
204     case "Date":
205         if (typeof criterion.value != "string")
206         {
207             qx.Bootstrap.warn(
208                 "Expected criterion value to be string, " +
209                 "got " + typeof(criterion.value));
210             ret += "false";
211         }
212     else
213     {
214         ret +=
215             "entry[\"" + criterion.field + "\"] " + filterOp +
216             "\"" + criterion.value + "\" ";
217     }
218     break;
219
220 case "Key":
221 case "Integer":
222 case "Float":
223     if (typeof criterion.value != "number")
224     {
225         qx.Bootstrap.warn(
226             "Expected criterion value to be number, " +
227             "got " + typeof(criterion.value));
228         ret += "false";
229     }
230     else
231     {
232         ret +=
233             "entry[\"" + criterion.field + "\"] " + filterOp +
234             criterion.value;
235     }
236     break;
237
238 case "KeyArray":
239 case "StringArray":
240 case "LongStringArray":
241     if (typeof criterion.value != "string")
242     {
243         qx.Bootstrap.warn(
244             "Expected criterion value to be string, " +
245             "got " + typeof(criterion.value));
246         ret += "false";
247     }
248     else if (criterion.filterOp)
249     {
250         qx.Bootstrap.warn(
251             "Filter operations can not be applied to array types");
252     }

```

```

253         else
254         {
255             ret +=
256                 "qx.lang.Array.contains(entry[\"" +
257                 criterion.field + "\"], " +
258                 "\"" + criterion.value + "\");";
259         }
260         break;
261
262     case "IntegerArray":
263     case "FloatArray":
264     if (typeof criterion.value != "number")
265     {
266         qx.Bootstrap.warn(
267             "Expected criterion value to be string, " +
268             "got " + typeof(criterion.value));
269         ret += "false";
270     }
271     else if (criterion.filterOp)
272     {
273         qx.Bootstrap.warn(
274             "Filter operations can not be applied to array types");
275     }
276     else
277     {
278         ret +=
279             "qx.lang.Array.contains(entry[\"" +
280             criterion.field + "\"], " + criterion.value + ")";
281     }
282     break;
283
284     default:
285         throw new Error("Unknown property type: " + type);
286     }
287     break;
288
289     default:
290         throw new Error("Unrecognized criterion type: " +
291             criterion.type);
292     }
293
294     return ret;
295 }) (searchCriteria);
296
297 // Create a function that implements the specified criteria
298 qualifies = new Function(
299     "entry",
300     "return (" + builtCriteria + ")");
301 }
302 else
303 {
304     // They want all entities of the specified type.
305     qualifies = function(entity) { return true; };
306 }
307
308 // Assume no required sort order
309 sortFunction = null;
310
311 // If there are any result criteria specified...
312 if (resultCriteria)
313 {
314     // ... then go through the criteria list and handle each.
315     resultCriteria.forEach(
316         function(criterion)
317         {
318             switch(criterion.type)
319             {
320                 case "limit":

```

```

321         limit = criterion.value;
322         if (limit <= 0)
323         {
324             throw new Error("Request for limit <= 0");
325         }
326         break;
327
328     case "offset":
329         offset = criterion.value;
330         if (offset < 0)
331         {
332             throw new Error("Request for offset < 0");
333         }
334         break;
335
336     case "sort":
337         builtSort = [ "var v1, v2;" ];
338
339         builtSort.push("v1 = a['" + criterion.field + "'];");
340         builtSort.push("v2 = b['" + criterion.field + "'];");
341
342         if ( criterion.order === "asc" )
343         {
344             builtSort.push("if (v1 < v2) return -1;");
345             builtSort.push("if (v1 > v2) return 1;");
346         }
347         else if (criterion.order === "desc" )
348         {
349             builtSort.push("if (v1 > v2) return -1;");
350             builtSort.push("if (v1 < v2) return 1;");
351         }
352         else
353         {
354             throw new Error(
355                 "Unexpected sort order for " + criterion.field + ": " +
356                 criterion.order);
357         }
358
359         builtSort.push("return 0;");
360         sortFunction = new Function("a", "b", builtSort.join("\n"));
361         break;
362
363     default:
364         throw new Error("Unrecognized result criterion type: " +
365             criterion.type);
366     }
367 });
368 }
369
370 for (entry in dbObjectMap)
371 {
372     if (qualifies(dbObjectMap[entry]))
373     {
374         // Make a deep copy of the results
375         result = qx.util.Serializer.toNativeObject(dbObjectMap[entry]);
376         results.push(result);
377     }
378 }
379
380 // Sort the results
381 if (sortFunction)
382 {
383     results.sort(sortFunction);
384 }
385
386 // Give 'em the query results, with the appropriate offset and limit.
387 return results.slice(offset, offset + limit);
388 },

```

```

389
390
391 /**
392  * Put an entity to the database. If the key field is null or undefined, a
393  * key is automatically generated for the entity.
394  *
395  * @param entity {rpcjs.dbif.Entity}
396  *   The entity to be made persistent.
397  */
398 put : function(entity)
399 {
400     var          data = {};
401     var          entityData = entity.getData();
402     var          keyProperty = entity.getEntityKeyProperty();
403     var          type = entity.getEntityType();
404     var          propertyName;
405     var          fields;
406     var          key;
407     var          keyFields = [];
408
409     // Are we working with a composite key?
410     if (qx.lang.Type.getClass(keyProperty) == "Array")
411     {
412         // Yup. Build the composite key from these fields
413         keyProperty.forEach(
414             function(fieldName)
415             {
416                 keyFields.push(entityData[fieldName]);
417             });
418         key = rpcjs.sim.Dbif._buildCompositeKey(keyFields);
419     }
420     else
421     {
422         // Retrieve the (single field) key
423         key = entityData[keyProperty];
424     }
425
426     // Get the field names for this entity type
427     fields = entity.getDatabaseProperties().fields;
428
429     // If there's no key yet...
430     switch(qx.lang.Type.getClass(key))
431     {
432     case "Undefined":
433     case "Null":
434         // Generate a new key. Determine what type of key to use.
435         switch(fields[keyProperty])
436         {
437         case "Key":
438         case "Number":
439             key = rpcjs.sim.Dbif.__nextKey++;
440             break;
441
442         case "String":
443             key = String(rpcjs.sim.Dbif.__nextKey++);
444             break;
445
446         default:
447             throw new Error("No way to autogenerate key");
448         }
449
450         // Save this key in the key field
451         entityData[entity.getEntityKeyProperty()] = key;
452         break;
453
454     case "Number":
455         // If there's no key, then generate a new key
456         if (isNaN(key))

```

```

457     {
458         key = String(rpcjs.sim.Dbif.__nextKey++);
459     }
460
461     // Save this key in the key field
462     entityData[entity.getEntityKeyProperty()] = key;
463     break;
464
465     case "Array":
466         // Build a composite key string from these key values
467         key = rpcjs.sim.Dbif._buildCompositeKey(key);
468         break;
469
470     case "String":
471         // nothing special to do
472         break;
473 }
474
475 // Create a simple map of properties and values to be put in the
476 // database
477 for (propertyName in entity.getDatabaseProperties().fields)
478 {
479     // Add this property value to the data to be saved to the database.
480     data[propertyName] = entityData[propertyName];
481 }
482
483 // Save it to the database
484 rpcjs.sim.Dbif.Database[type][key] = data;
485
486 // Write it to Web Storage
487 if (typeof window.localStorage !== "undefined")
488 {
489     qx.Bootstrap.debug("Writing DB to Web Storage");
490     localStorage.simDB = qx.lang.Json.stringify(rpcjs.sim.Dbif.Database);
491 }
492 },
493
494
495 /**
496  * Remove an entity from the database
497  *
498  * @param entity {rpcjs.dbif.Entity}
499  *   An instance of the entity to be removed.
500  */
501 remove : function(entity)
502 {
503     var          entityData = entity.getData();
504     var          keyProperty = entity.getEntityKeyProperty();
505     var          type = entity.getEntityType();
506     var          propertyName;
507     var          fields;
508     var          key;
509     var          keyFields = [];
510
511     // Are we working with a composite key?
512     if (qx.lang.Type.getClass(keyProperty) == "Array")
513     {
514         // Yup. Build the composite key from these fields
515         keyProperty.forEach(
516             function(fieldName)
517             {
518                 keyFields.push(entityData[fieldName]);
519             });
520         key = rpcjs.sim.Dbif._buildCompositeKey(keyFields);
521     }
522     else
523     {
524         // Retrieve the (single field) key

```

```

525     key = entityData[keyProperty];
526 }
527
528 delete rpcjs.sim.Dbif.Database[type][key];
529
530 // Write it to Web Storage
531 if (typeof window.localStorage !== "undefined")
532 {
533     qx.Bootstrap.debug("Writing DB to Web Storage");
534     localStorage.simDB = qx.lang.Json.stringify(rpcjs.sim.Dbif.Database);
535 }
536 },
537
538 /**
539  * Add a blob to the database.
540  *
541  * @param blobData {LongString}
542  *   The data to be written as a blob
543  *
544  * @return {String}
545  *   The blob ID of the just-added blob
546  *
547  * @throws {Error}
548  *   If an error occurs while writing the blob to the database, an Error
549  *   is thrown.
550  */
551 putBlob : function(blobData)
552 {
553     var blobStorage = rpcjs.sim.Dbif.BlobStorage;
554     var Db = rpcjs.sim.Dbif.Database;
555     var key;
556
557     // If there's no blob storage yet...
558     if (! Db[blobStorage])
559     {
560         // ... then create it.
561         Db[blobStorage] = {};
562     }
563
564     // Retrieve the next id value to use as this blob id
565     key = rpcjs.sim.Dbif.__nextKey++;
566
567     // Convert it to a string, to allow consistency with other backends
568     key = key + "";
569
570     // Store the blob
571     Db[blobStorage][key] = blobData;
572
573     // Give 'em the blob id
574     return key;
575 },
576
577 /**
578  * Retrieve a blob from the database
579  *
580  * @param blobId {Key}
581  *   The blob ID of the blob to be retrieved
582  *
583  * @return {LongString}
584  *   The blob data retrieved from the database. If there is no blob with
585  *   the given ID, undefined is returned.
586  */
587 getBlob : function(blobId)
588 {
589     var blobStorage = rpcjs.sim.Dbif.BlobStorage;
590     var Db = rpcjs.sim.Dbif.Database;
591
592     // Is there any blob storage?

```

```
593     if (! Db[blobStorage])
594     {
595         // Nope. The blob must not exist. Not found.
596         return undefined;
597     }
598
599     // Return the specified blob (or undefined, if it's not there).
600     return Db[blobStorage][blobId];
601 },
602
603 /**
604  * Remove a blob from the database
605  *
606  * @param blobId {Key}
607  *   The blob ID of the blob to be removed. If the specified blob id does
608  *   not exist, this request fails silently.
609  */
610 removeBlob : function(blobId)
611 {
612     var         blobStorage = rpcjs.sim.Dbif.BlobStorage;
613     var         Db = rpcjs.sim.Dbif.Database;
614
615     // Is there any blob storage?
616     if (! Db[blobStorage])
617     {
618         // Nope. The blob must not exist.
619         return;
620     }
621
622     // Delete the specified blob
623     delete Db[blobStorage][blobId];
624 }
625 }
626 });
```

Appendix 13

aiagallery.dbif.Constants

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  *
4  * License:
5  *   LGPL: http://www.gnu.org/licenses/lgpl.html
6  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7  */
8
9  qx.Class.define("aiagallery.dbif.Constants",
10 {
11   extend : qx.core.Object,
12
13   statics :
14   {
15     /**
16     * Value of the maximum amount of times an app or comment
17     * can be flagged
18     */
19     MAX_FLAGGED : 5,
20
21     /** Mapping of status names to values */
22     Status :
23     {
24       Banned : 0,
25       Pending : 1,
26       Active : 2
27     },
28
29     /** Reverse mapping of status: values to names */
30     StatusToName :
31     [
32       "Banned",
33       "Pending",
34       "Active"
35     ],
36
37     /** Mapping of FlagType names to values */
38     FlagType :
39     {
40       App : 0,
41       Comment : 1
42     },
43
44     /** Reverse mapping of FlagType values to names */
45     FlagTypeToName :
46     [
47       "App",
48       "Comment"

```

```

49     ],
50
51     /** Mapping of permission names to descriptions */
52     Permissions :
53     {
54         //
55         // MApps
56         //
57         "addOrEditApp" : "Add and edit applications",
58         "deleteApp"    : "Delete applications",
59         "getAppListAll" : "Get all users application list",
60
61         /* Anonymous access...
62         "getAppList"    : "Get logged in user application list",
63         "appQuery"     : "Query for applications",
64         "getAppInfo"   : "Get application detail information",
65         "intersectKeywordAndQuery" : "Get intersection of keyword search and" +
66                                     "appQuery"
67         ... */
68
69         //
70         // MComments
71         //
72         "addComment"   : "Add comments to an application",
73         "deleteComment" : "Delete comments from an application",
74
75         /* Anonymous access...
76         "getComments"  : "Retrieve comments about an application",
77         ... */
78
79         //
80         // MMobile
81         //
82         /* Anonymous access...
83         "mobileRequest" : "Mobile client requests",
84         ... */
85
86         //
87         // MTags
88         //
89         /* Anonymous access...
90         "getCategoryTags" : "Get the list of category tags",
91         ... */
92
93         //
94         // MVisitors
95         //
96         "addOrEditVisitor" : "Add and edit visitors",
97         "deleteVisitor"    : "Delete visitors",
98         "getVisitorList"   : "Retrieve list of visitors",
99
100        //
101        // MWhoAmI
102        //
103        /* Anonymous access...
104        "whoAmI" : "Identify the current user id and permissions"
105        */
106
107        //
108        // MLikes
109        //
110        "likesPlusOne"      : "Like an app"
111    }
112 }
113 });

```

Appendix 14

aiagallery.dbif.DbifAppEngine

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  *
4  * License:
5  *   LGPL: http://www.gnu.org/licenses/lgpl.html
6  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7  */
8
9  qx.Class.define("aiagallery.dbif.DbifAppEngine",
10 {
11   extend : rpcjs.appengine.Dbif,
12   type   : "singleton",
13
14   include :
15   [
16     aiagallery.dbif.MDbifCommon
17   ],
18
19   construct : function()
20   {
21     // Call the superclass constructor
22     this.base(arguments);
23
24     // Prepare for remote procedure calls to aiagallery.features.*
25     this.__rpc = new rpcjs.appengine.Rpc([ "aiagallery", "features" ], "/rpc");
26   },
27
28   members :
29   {
30     /**
31     * Register a service name and function.
32     *
33     * @param serviceName {String}
34     *   The name of this service within the <[rpcKey]> namespace.
35     *
36     * @param fService {Function}
37     *   The function which implements the given service name.
38     *
39     * @param paramNames {Array}
40     *   The names of the formal parameters, in order.
41     */
42     registerService : function(serviceName, fService, paramNames)
43     {
44       // Register with the RPC provider
45       this.__rpc.registerService(serviceName, fService, this, paramNames);
46     },
47
48     /**

```

```

49     * Process an incoming request which is presumably a JSON-RPC request.
50     *
51     * @param jsonData {String}
52     *   The data provide in a POST request
53     *
54     * @return {String}
55     *   Upon success, the JSON-encoded result of the RPC request is returned.
56     *   Otherwise, null is returned.
57     */
58     processRequest : function(jsonData)
59     {
60         return this.__rpc.processRequest(jsonData);
61     },
62
63     /**
64     * Identify the current user. Register him in the whoAmI property.
65     */
66     identify : function()
67     {
68         var            UserServiceFactory;
69         var            userService;
70         var            user;
71         var            whoami;
72         var            userId;
73         var            visitor;
74
75         // Find out who is logged in
76         UserServiceFactory =
77             Packages.com.google.appengine.api.users.UserServiceFactory;
78         userService = UserServiceFactory.getUserService();
79         user = userService.getCurrentUser();
80
81         // If no one is logged in...
82         if (! user)
83         {
84             this.setWhoAmI(
85                 {
86                     email      : "anonymous",
87                     userId     : "",
88                     isAdmin    : false,
89                     logoutUrl  : "",
90                     permissions : []
91                 });
92             return;
93         }
94
95         whoami = String(user.getEmail());
96
97         // Try to get this user's display name. Does the visitor exist?
98         visitor = rpcjs.dbif.Entity.query("aiagallery.dbif.ObjVisitors", whoami);
99         if (visitor.length > 0)
100         {
101             // Yup, he exists.
102             userId = visitor[0].displayName || String(user.getUserId());
103         }
104         else
105         {
106             // He doesn't exist. Just use the unique number.
107             userId = String(user.getUserId());
108         }
109
110         // Save the logged-in user. The whoAmI property is in MDbifCommon.
111         this.setWhoAmI(
112             {
113                 email      : whoami,
114                 userId     : userId,
115                 isAdmin    : userService.isUserAdmin(),
116                 logoutUrl  : userService.createLogoutURL("/")

```

```
117         });
118     }
119 },
120
121 defer : function()
122 {
123     // Register our put & query functions
124     rpcjs.dbif.Entity.registerDatabaseProvider(
125         rpcjs.appengine.Dbif.query,
126         rpcjs.appengine.Dbif.put,
127         rpcjs.appengine.Dbif.remove,
128         rpcjs.appengine.Dbif.getBlob,
129         rpcjs.appengine.Dbif.putBlob,
130         rpcjs.appengine.Dbif.removeBlob);
131     }
132 });
```

Appendix 15

aiagallery.dbif.DbifSim

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  *
4  * License:
5  *   LGPL: http://www.gnu.org/licenses/lgpl.html
6  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7  */
8
9  qx.Class.define("aiagallery.dbif.DbifSim",
10 {
11   extend : rpcjs.sim.Dbif,
12   type   : "singleton",
13
14   include :
15   [
16     aiagallery.dbif.MDbifCommon,
17     aiagallery.dbif.MSimData
18   ],
19
20   construct : function()
21   {
22     // Call the superclass constructor
23     this.base(arguments);
24
25     // Prepare for remote procedure calls to aiagallery.features.*
26     this.__rpc = new rpcjs.sim.Rpc([ "aiagallery", "features" ], "/rpc");
27
28     // Save the logged-in user. The whoAmI property is in MDbifCommon.
29     this.setWhoAmI(
30     {
31       email      : "jarjar@binks.org",
32       userId     : "obnoxious",
33       isAdmin    : true,
34       logoutUrl  : "javascript:aiagallery.dbif.DbifSim.changeWhoAmI();"
35     });
36   },
37
38   members :
39   {
40     __rpc : null,
41
42     /**
43     * Register a service name and function.
44     *
45     * @param serviceName {String}
46     *   The name of this service within the <[rpcKey]> namespace.
47     *
48     * @param fService {Function}

```

```

49     *   The function which implements the given service name.
50     *
51     * @param paramNames {Array}
52     *   The names of the formal parameters, in order.
53     */
54     registerService : function(serviceName, fService, paramNames)
55     {
56         // Register with the RPC provider
57         this.__rpc.registerService(serviceName, fService, this, paramNames);
58     }
59 },
60
61 statics :
62 {
63     __userNumber : 0,
64
65     changeWhoAmI : function(context)
66     {
67         var formData =
68         {
69             'username' :
70             {
71                 'type' : "ComboBox",
72                 'label' : "Login",
73                 'value' : null,
74                 'options' : [ ]
75             },
76             'isAdmin' :
77             {
78                 'type' : "SelectBox",
79                 'label' : "User type",
80                 'value' : null,
81                 'options' :
82                 [
83                     { 'label' : "Normal", 'value' : false },
84                     { 'label' : "Administrator", 'value' : true }
85                 ]
86             }
87         };
88
89         // Retrieve all of the visitor records
90         rpcjs.dbif.Entity.query("aiagallery.dbif.ObjVisitors").forEach(
91             function(visitor, i)
92             {
93                 // Add this visitor to the list
94                 formData.username.options.push(
95                     {
96                         label : visitor.id,
97                         value : visitor.id
98                     }
99                 );
100
101         dialog.Dialog.form(
102             "You have been logged out. Please log in.",
103             formData,
104             function( result )
105             {
106                 var visitor;
107                 var displayName;
108                 var guiWhoAmI;
109
110                 // Try to get this user's display name. Does the visitor exist?
111                 visitor = rpcjs.dbif.Entity.query("aiagallery.dbif.ObjVisitors",
112                     result.username);
113
114                 if (visitor.length > 0)
115                 {
116                     // Yup, he exists.
117                     displayName =

```

```

117         visitor[0].displayName ||
118         "User #" + aiagallery.dbif.DbifSim.__userNumber++;
119     }
120     else
121     {
122         // He doesn't exist. Just use the unique number.
123         displayName = "User #" + aiagallery.dbif.DbifSim.__userNumber++;
124     }
125
126     // Save the backend whoAmI information
127     aiagallery.dbif.DbifSim.getInstance().setWhoAmI(
128     {
129         email      : result.username,
130         userId     : displayName,
131         isAdmin    : true,
132         logoutUrl  : "javascript:aiagallery.dbif.DbifSim.changeWhoAmI();"
133     });
134
135     // Update the gui too
136     guiWhoAmI = aiagallery.main.Gui.getInstance().whoAmI;
137     guiWhoAmI.setIsAdmin(result.isAdmin);
138     guiWhoAmI.setEmail(result.username);
139     guiWhoAmI.setDisplayName(displayName);
140     guiWhoAmI.setPermissions("");
141     guiWhoAmI.setLogoutUrl(
142         "javascript:aiagallery.dbif.DbifSim.changeWhoAmI();");
143     }
144     );
145     }
146 },
147
148 defer : function()
149 {
150     // Retrieve the database from Web Storage, if such exists.
151     if (typeof window.localStorage !== "undefined")
152     {
153         if (typeof localStorage.simDB == "string")
154         {
155             qx.Bootstrap.debug("Reading DB from Web Storage");
156             rpcjs.sim.Dbif.setDb(qx.lang.Json.parse(localStorage.simDB));
157         }
158         else
159         {
160             // No database yet stored. Retrieve the database from the MSimData mixin
161             qx.Bootstrap.debug("No database yet. Using new SIM database.");
162             rpcjs.sim.Dbif.setDb(aiagallery.dbif.MSimData.Db);
163         }
164     }
165     else
166     {
167         // Retrieve the database from the MSimData mixin
168         qx.Bootstrap.debug("No Web Storage available. Using new SIM database.");
169
170         // Convert string appId keys to numbers
171         qx.Bootstrap.debug("Beginning appId conversion...");
172         var apps = aiagallery.dbif.MSimData.Db["apps"];
173         qx.lang.Object.getKeys(apps).forEach(
174             function(appId)
175             {
176                 qx.Bootstrap.debug("Converting " + appId);
177                 apps[parseInt(appId, 10)] = apps[appId];
178                 delete apps[appId];
179             });
180
181         rpcjs.sim.Dbif.setDb(aiagallery.dbif.MSimData.Db);
182     }
183
184     // Register our put & query functions

```

```
185     rpcjs.dbif.Entity.registerDatabaseProvider(  
186         rpcjs.sim.Dbif.query,  
187         rpcjs.sim.Dbif.put,  
188         rpcjs.sim.Dbif.remove,  
189         rpcjs.sim.Dbif.getBlob,  
190         rpcjs.sim.Dbif.putBlob,  
191         rpcjs.sim.Dbif.removeBlob);  
192     }  
193 });
```

Appendix 16

aiagallery.dbif.Decoder64

```

1  /**
2  * Copyright (c) 2011 Reed Spool
3  *
4  * License:
5  *   LGPL: http://www.gnu.org/licenses/lgpl.html
6  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7  */
8
9  qx.Class.define("aiagallery.dbif.Decoder64",
10 {
11   extend : qx.core.Object,
12
13   statics :
14   {
15     getDecodedURL : function(appId, base64field)
16     {
17       var      myObj;
18       var      fieldContent;
19       var      mimeType;
20       var      contents;
21       var      decodedContents;
22       var      blobIds;
23       var      name = null;
24       var      ret;
25
26       // Get an instance of the object whose field is requested
27       myObj = new aiagallery.dbif.ObjAppData(parseInt(appId,10));
28
29       switch(base64field)
30       {
31         case "source":
32         case "apk":
33           // Get the contents of that field, which, if it exists, is a blob id
34           blobIds = myObj.getData()[base64field];
35
36           // Was there any data in the field?
37           if (! blobIds)
38           {
39             // No, return null and let the caller decide how to handle that.
40             return null;
41           }
42
43           // Retrieve the blob data, which is the base64-encoded data. We want
44           // the most recent entry, so use index 0.
45           fieldContent = rpcjs.dbif.Entity.getBlob(blobIds[0]);
46
47           // Also specify the file name
48           name = myObj.getData()[base64field + "FileName"];

```

```

49         break;
50
51     default:
52         // Retrieve the field data, which is the base64-encoded data
53         fieldContent = myObj.getData()[base64field];
54         break;
55     }
56
57     // Was there any blob data?
58     if (! fieldContent)
59     {
60         // No, return null and let the caller decide how to handle that.
61         return null;
62     }
63
64     // Parse out the mimeType. This always starts at index 5 and ends with a
65     // semicolon
66     mimeType = fieldContent.substring(5, fieldContent.indexOf(";"));
67
68     // Parse out the actual url
69     contents = fieldContent.substring(fieldContent.indexOf(",") + 1);
70
71     // Send the url to the decoder function
72     decodedContents = aiagallery.dbif.Decoder64._decode(contents);
73
74     // Give 'em what they want
75     ret =
76     {
77         mime      : mimeType,
78         content   : decodedContents
79     };
80
81     // If there's a file name...
82     if (name)
83     {
84         // ... then add it too
85         ret.name = name;
86     }
87
88     // Give 'em what they came for
89     return ret;
90 },
91
92 /**
93  * Copyright: Dr Alexander J Turner - all rights reserved.
94  * Please feel free to use this any way you want as long as you
95  * mention I wrote it!
96  *
97  * Modification history:
98  *   - Sept. 2011, Derrell Lipman
99  *     Converted to static method in qooxdoo class
100 */
101 _decode : function(encStr)
102 {
103     var          i;
104     var          l;
105     var          c;
106     var          el;
107     var          ar2;
108     var          bits24;
109     var          decStr;
110     var          linelen;
111     var          decArray;
112     var          base64chars;
113     var          base64charToInt;
114
115     base64chars =
116     "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";

```

```

117     base64charToInt = {};
118     for (var i = 0; i < 64; i++)
119     {
120         base64charToInt[base64chars.substr(i,1)] = i;
121     }
122
123     encStr = encStr.replace(/\s+/g, "");
124     decStr = "";
125     decArray = [];
126     linelen = 0;
127     el = encStr.length;
128
129     for (i = 0; i < el; i += 4)
130     {
131         bits24 = ( base64charToInt[encStr.charAt(i)] & 0xFF ) << 18;
132         bits24 |= ( base64charToInt[encStr.charAt(i+1)] & 0xFF ) << 12;
133         bits24 |= ( base64charToInt[encStr.charAt(i+2)] & 0xFF ) << 6;
134         bits24 |= ( base64charToInt[encStr.charAt(i+3)] & 0xFF ) << 0;
135         decStr += String.fromCharCode((bits24 & 0xFF0000) >> 16);
136
137         if (encStr.charAt(i + 2) != '=') // check for padding character =
138         {
139             decStr += String.fromCharCode((bits24 & 0xFF00) >> 8);
140         }
141
142         if (encStr.charAt(i + 3) != '=') // check for padding character =
143         {
144             decStr += String.fromCharCode((bits24 & 0xFF) >> 0);
145         }
146
147         if (decStr.length>1024)
148         {
149             decArray.push(decStr);
150             decStr='';
151         }
152     }
153
154     if (decStr.length>0)
155     {
156         decArray.push(decStr);
157     }
158
159     ar2 = [];
160
161     while (decArray.length > 1)
162     {
163         l=decArray.length;
164         for(c = 0; c < l; c += 2)
165         {
166             if (c + 1 == l)
167             {
168                 ar2.push(decArray[c]);
169             }
170             else
171             {
172                 ar2.push('' +decArray[c] + decArray[c+1]);
173             }
174         }
175         decArray = ar2;
176         ar2 = [];
177     }
178
179     return decArray[0];
180 }
181 }
182 });

```

Appendix 17

aiagallery.dbif.Entity

```

1  /**
2   * Copyright (c) 2011 Derrell Lipman
3   *
4   * License:
5   *   LGPL: http://www.gnu.org/licenses/lgpl.html
6   *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7   */
8
9  /*
10 #use(aiagallery.dbif.ObjAppData)
11 #use(aiagallery.dbif.ObjComments)
12 #use(aiagallery.dbif.ObjDownloads)
13 #use(aiagallery.dbif.ObjFlags)
14 #use(aiagallery.dbif.ObjLikes)
15 #use(aiagallery.dbif.ObjTags)
16 #use(aiagallery.dbif.ObjVisitors)
17 #use(aiagallery.dbif.ObjSearch)
18 #use(aiagallery.dbif.ObjPermissionGroup)
19
20
21 #use(aiagallery.dbif.ObjTest)
22 */
23
24 qx.Class.define("aiagallery.dbif.Entity",
25 {
26   extend : rpcjs.dbif.Entity,
27
28   construct : function(entityType, entityKey)
29   {
30     // Call the superclass constructor
31     this.base(arguments, entityType, entityKey);
32   },
33
34   statics :
35   {
36     registerEntityType      : null,
37     registerPropertyTypes  : null,
38     query                   : null
39   },
40
41   defer : function()
42   {
43     // Point our own statics at our superclass' statics
44     aiagallery.dbif.Entity.registerEntityType =
45       rpcjs.dbif.Entity.registerEntityType;
46     aiagallery.dbif.Entity.registerPropertyTypes =
47       rpcjs.dbif.Entity.registerPropertyTypes;
48   }
49 }

```


Appendix 18

aiagallery.dbif.MDbifCommon

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  *
4  * License:
5  *   LGPL: http://www.gnu.org/licenses/lgpl.html
6  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7  */
8
9  qx.Mixin.define("aiagallery.dbif.MDbifCommon",
10 {
11   include :
12   [
13     aiagallery.dbif.MVisitors,
14     aiagallery.dbif.MApps,
15     aiagallery.dbif.MTags,
16     aiagallery.dbif.MMobile,
17     aiagallery.dbif.MComments,
18     aiagallery.dbif.MWhoAmI,
19     aiagallery.dbif.MSearch,
20     aiagallery.dbif.MLiking,
21     aiagallery.dbif.MFlags
22   ],
23
24   construct : function()
25   {
26     // Use our authorization function
27     rpcjs.AbstractRpcHandler.authorizationFunction =
28       aiagallery.dbif.MDbifCommon.authenticate;
29   },
30
31   properties :
32   {
33     /**
34     * Information about the currently-logged-in user. The value is a map
35     * containing the fields: email, userId, and isAdmin.
36     */
37     whoAmI :
38     {
39       nullable : true,
40       init      : null,
41       check    : "Object",
42       apply    : "_applyWhoAmI"
43     }
44   },
45
46   members :
47   {
48     _applyWhoAmI : function(value, old)

```

```

49     {
50         aiagallery.dbif.MDbifCommon.__whoami = value;
51     }
52 },
53
54 statics :
55 {
56     __whoami : null,
57     __isAdmin : null,
58     __initialized : false,
59
60     /**
61      * Standardized time stamp for all Date fields
62      *
63      * @return {Integer}
64      *   The number of milliseconds since midnight, 1 Jan 1970
65      */
66     currentTimestamp : function()
67     {
68         return new Date().getTime();
69     },
70
71     /**
72      * Function to be called for authorization to run a service
73      * method.
74      *
75      * @param fqMethod {String}
76      *   The fully-qualified name of the method to be called
77      *
78      * @return {Boolean}
79      *   true to allow the function to be called, or false to indicates
80      *   permission denied.
81      */
82     authenticate : function(fqMethod)
83     {
84         var             methodComponents;
85         var             methodName;
86         var             serviceName;
87         var             me;
88         var             meData;
89         var             meObjVisitor;
90         var             mePermissionGroups;
91         var             bAnonymous;
92
93         // Have we yet initialized the user object?
94         if (aiagallery.dbif.MDbifCommon.__whoami &&
95             ! aiagallery.dbif.MDbifCommon.__initialized)
96         {
97             // Nope. Retrieve our visitor object
98             me = new aiagallery.dbif.ObjVisitors(
99                 aiagallery.dbif.MDbifCommon.__whoami.email);
100
101             // Is it brand new, or does not contain a display name yet?
102             meData = me.getData();
103             if (me.getBrandNew() || meData.displayName === null)
104             {
105                 // True. Save it.
106                 if (! meData.displayName)
107                 {
108                     meData.displayName = aiagallery.dbif.MDbifCommon.__whoami.userId;
109                 }
110
111                 me.put();
112             }
113
114             // We're now initialized
115             aiagallery.dbif.MDbifCommon.__initialized = true;
116         }

```

```

117
118 // Split the fully-qualified method name into its constituent parts
119 methodComponents = fqMethod.split(".");
120
121 // The final component is the actual method name
122 methodName = methodComponents.pop();
123
124 // The remainder is the service path. Join it back together.
125 serviceName = methodComponents.join(".");
126
127 // Ensure that the service name is what's expected. (This should never
128 // occur, since the RPC server has already validated that the method
129 // exists.)
130 if (serviceName != "aiagallery.features")
131 {
132 // It's not. Do not allow access.
133 return false;
134 }
135
136 // If the user is an administrator, ...
137 if (aiagallery.dbif.MDbifCommon.__whoami &&
138     aiagallery.dbif.MDbifCommon.__whoami.isAdmin)
139 {
140 // ... they implicitly have access.
141 return true;
142 }
143
144 // Do per-method authorization.
145
146 // Are they logged in, or anonymous?
147 bAnonymous = (aiagallery.dbif.MDbifCommon.__whoami === null);
148
149 switch(methodName)
150 {
151
152 //
153 // MApps
154 //
155 case "getAppList":
156 case "addOrEditApp":
157     return ! bAnonymous; // Access is allowed if they're logged in
158
159 case "deleteApp":
160     return aiagallery.dbif.MDbifCommon._deepPermissionCheck(methodName);
161
162 case "getAppListAll":
163     return aiagallery.dbif.MDbifCommon._deepPermissionCheck(methodName);
164
165 case "appQuery":
166 case "intersectKeywordAndQuery":
167 case "getAppInfo":
168 case "getAppListByList":
169     return true; // Anonymous access
170
171 //
172 // MComments
173 //
174 case "addComment":
175     return ! bAnonymous; // Access is allowed if they're logged in
176
177 case "deleteComment":
178     return aiagallery.dbif.MDbifCommon._deepPermissionCheck(methodName);
179
180 case "getComments":
181     return true; // Anonymous access
182
183 //
184 // MMobile

```

```

185     //
186     case "mobileRequest":
187         return true;           // Anonymous access
188
189     //
190     // MTags
191     //
192     case "getCategoryTags":
193         return true;           // Anonymous access
194
195     //
196     // MVisitors
197     //
198     case "addOrEditVisitor":
199         return aiagallery.dbif.MDbifCommon._deepPermissionCheck(methodName);
200
201     case "deleteVisitor":
202         return aiagallery.dbif.MDbifCommon._deepPermissionCheck(methodName);
203
204     case "getVisitorList":
205         if (qx.core.Environment.get("qx.debug"))
206             {
207                 return aiagallery.dbif.MDbifCommon._deepPermissionCheck(methodName);
208             }
209         else
210             {
211                 // At present, do not allow access to visitor list on App Engine
212                 return false;
213             }
214
215     case "editProfile":
216         return ! bAnonymous;    // Access is allowed if they're logged in
217
218     //
219     // MWhoAmI
220     //
221     case "whoAmI":
222         return true;           // Anonymous access
223
224     //
225     // MSearch
226     //
227     case "keywordSearch":
228         return true;           // Anonymous access
229
230
231     //
232     // MLiking
233     //
234     case "likesPlusOne":
235         return ! bAnonymous;    // Access allowed if logged in
236
237     default:
238         // Do not allow access to unrecognized method names
239         return false;
240 }
241
242 },
243
244
245 _deepPermissionCheck : function(methodName)
246 {
247     // Find out who we are.
248     var whoami = aiagallery.dbif.MDbifCommon._whoami;
249
250     // If no one is logged in...
251     if (! whoami)
252     {

```

```

253     // ... then they do not have permission
254     return false;
255 }
256
257 var email = whoami.email;
258 var myObjData = new aiagallery.dbif.ObjVisitors(email).getData();
259 var permissionArr = myObjData["permissions"];
260 var permissionGroupArr = myObjData["permissionGroups"];
261 var permission;
262 var group;
263 var data;
264
265 // Standard check: Does my permission list contain this method?
266 if (permissionArr != null &&
267     qx.lang.Array.contains(permissionArr, methodName))
268 {
269     // Yes, allow me.
270     return true;
271 }
272
273 // Permission Groups Untested, disabling for now
274 if(false)
275 {
276     // Deeper check: Do any of my permission groups give me access to this
277     // method?
278     if (permissionGroupArr != null)
279     {
280         // For every permission group of which I am a member...
281         permissionGroupArr.forEach(
282             function (group)
283             {
284
285                 // Retrieve the list of permissions it gives me
286                 data = new aiagallery.dbif.ObjPermissonGriou(group).getData();
287                 permissionArr = data["permissions"];
288
289                 // Same as standard check: does this group contain this method?
290                 if (permissionArr != null &&
291                     qx.lang.Array.contains(permissionArr, methodName))
292                 {
293                     // Yes, allow me.
294                     return true;
295                 }
296
297                 return false;
298             });
299     }
300 }
301
302 // Did not find this permission, dissalow.
303 return false;
304 }
305 }
306 });

```

Appendix 19

aiagallery.dbif.MApps

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  * Copyright (c) 2011 Reed Spool
4  *
5  * License:
6  *   LGPL: http://www.gnu.org/licenses/lgpl.html
7  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
8  */
9
10 qx.Mixin.define("aiagallery.dbif.MApps",
11 {
12   construct : function()
13   {
14     this.registerService("addOrEditApp",
15                         this.addOrEditApp,
16                         [ "uid", "attributes" ]);
17
18     this.registerService("deleteApp",
19                         this.deleteApp,
20                         [ "uid" ]);
21
22     this.registerService("getAppList",
23                         this.getAppList,
24                         [ "bStringize", "sortCriteria", "offset", "limit" ]);
25
26     this.registerService("getAppListAll",
27                         this.getAppListAll,
28                         [ "bStringize", "sortCriteria", "offset", "limit" ]);
29
30     this.registerService("appQuery",
31                         this.appQuery,
32                         [ "criteria", "requestedFields" ]);
33
34     this.registerService("intersectKeywordAndQuery",
35                         this.intersectKeywordAndQuery,
36                         [ "queryArgs" ]);
37
38
39     this.registerService("getAppListByList",
40                         this.getAppListByList,
41                         [ "uidArr", "requestedFields" ]);
42
43
44     this.registerService("getAppInfo",
45                         this.getAppInfo,
46                         [ "uid", "bStringize", "requestedFields" ]);
47   },
48

```

```

49     statics :
50     {
51         /** The next AppId value to use */
52         nextAppId : 100,
53
54         /**
55          * Stringize the array fields of an App
56          *
57          * @param app {Object}
58          *   a reference to the App whose array fields are to be stringized.
59          */
60         _stringizeAppInfo : function(app)
61         {
62             [
63                 // FIXME: When previous Author chain is implemented, uncomment
64                 // "previousAuthors",
65                 "tags"
66             ]
67             .forEach(function(field)
68             {
69                 // ... stringize this field.
70                 app[field] = app[field].join(", ");
71             });
72
73             // Convert from numeric to string status
74             app.status = [ "Banned", "Pending", "Active" ][app.status];
75
76         },
77
78         /**
79          * Strip an App object of all but the requested fields. Also renames
80          * fields per request
81          *
82          * @param app {Object}
83          *   a reference to the App whose fields are to be deleted or renamed.
84          *
85          * @param requestedFields {Map?}
86          *   If provided, this is a map containing, as the member names, the
87          *   fields which should remain in the resultant Object. The value of each
88          *   entry in the map indicates what to name that field, in the
89          *   result. (This produces a mapping of the field names.) An example is
90          *   requestedFields map might look like this:
91          *
92          *   {
93          *     uid      : "uid",    // No change in name.
94          *     title   : "label",  // remap the title field to be called "label"
95          *     image1  : "icon",   // remap the image1 field to be called "icon"
96          *     tags    : "tags"
97          *   }
98          *
99          *   Any field which is not in this map is deleted from app
100          */
101         _requestedFields : function(app, requestedFields)
102         {
103             // Remove those members which are not requested, and rename as requested
104             var requested;
105
106             for (var field in app)
107             {
108                 // Is this field a requested field?
109                 requested = requestedFields[field];
110                 if (! requested)
111                 {
112                     // No, remove it
113                     delete app[field];
114                 }
115             }
116

```



```

185                                     appDataField]);
186         // Save the record in the DB.
187         searchObj.put();
188     });
189     break;
190
191     }
192 }
193 },
194
195 /**
196  * Ensure that there are no Search records from App with this uid
197  *
198  * @param uid {Integer}
199  * This is the app's uid whose Search records are to be wiped
200  */
201 _removeAppFromSearch : function(uid)
202 {
203     var results;
204     var resultObj;
205     var searchObj;
206
207     // Get all Search Objects with this uid then...
208     results = rpcjs.dbif.Entity.query("aiagallery.dbif.ObjSearch",
209         {
210             type : "element",
211             field: "appId",
212             value: uid
213         },
214         null);
215
216     // Remove every record found
217     results.forEach(function(obj)
218     {
219         searchObj = new aiagallery.dbif.ObjSearch([obj["word"],
220             obj["appId"],
221             obj["appField"]]);
222         searchObj.removeSelf();
223     });
224 },
225
226 members :
227 {
228     addOrEditApp : function(uid, attributes, error)
229     {
230         var i;
231         var title;
232         var description;
233         var image;
234         var previousAuthors;
235         var source;
236         var apk;
237         var tags;
238         var tagObj;
239         var tagData;
240         var oldTags;
241         var bHasCategory;
242         var categories;
243         var uploadTime;
244         var status;
245         var statusIndex;
246         var appData;
247         var appObj;
248         var bNew;
249         var whoami;
250         var missing = [];
251         var sourceData;
252         var apkData;

```

```

253     var                key;
254     var                allowableFields =
255     [
256         "uid",
257         "owner",
258         "title",
259         "description",
260         "image1",
261         "image2",
262         "image3",
263         "previousAuthors",
264         "source",
265         "sourceFileName",
266         "apk",
267         "apkFileName",
268         "tags",
269         "uploadTime",
270         "numLikes",
271         "numDownloads",
272         "numViewed",
273         "numComments",
274         "status"
275     ];
276     var                requiredFields =
277     [
278         "owner",
279         "title",
280         "description",
281         "tags",
282         "source",
283         "image1"
284     ];
285
286     // Don't let the caller override the owner
287     delete attributes["owner"];
288
289     // Determine who the logged-in user is
290     whoami = this.getWhoAmI();
291
292     // Get an AppData object. If uid is non-null, retrieve the prior data.
293     appObj = new aiagallery.dbif.ObjAppData(uid);
294
295     // Retrieve the data
296     appData = appObj.getData();
297
298     // If we were given a record identifier...
299     if (uid != null)
300     {
301         // ... it must have already existed or it's an error
302         if (appObj.getBrandNew())
303         {
304             // It didn't!
305             error.setCode(1);
306             error.setMessage("Unrecognized UID");
307             return error;
308         }
309
310         // Ensure that the logged-in user owns this application.
311         if (appData.owner != whoami.email)
312         {
313             // He doesn't. Someone's doing something nasty!
314             error.setCode(2);
315             error.setMessage("Not owner");
316             return error;
317         }
318     }
319     else
320     {

```

```

321     // Initialize the owner field
322     appData.owner = whoami.email;
323 }
324
325 // Save the existing tags list
326 oldTags = appData.tags;
327
328 // If there's no image1 value...
329 if (! attributes.image1)
330 {
331     // ... then move image3 or image2 to image1
332     if (attributes.image3)
333     {
334         attributes.image1 = attributes.image3;
335         attributes.image3 = null;
336     }
337     else
338     {
339         // image2 may not exist either, which will be caught in the
340         // code that detects missing fields.
341         attributes.image1 = attributes.image2;
342         attributes.image2 = null;
343     }
344 }
345
346 // Similarly, if there's no image2 value...
347 if (! attributes.image2)
348 {
349     // ... then move image3 to image2. (Again, it may not exist.)
350     attributes.image2 = attributes.image3;
351     attributes.image3 = null;
352 }
353
354 // Copy fields from the attributes parameter into this db record
355 allowableFields.forEach(
356     function(field)
357     {
358         // Was this field provided in the parameter attributes?
359         if (attributes[field])
360         {
361             // Handle source and apk fields specially
362             switch(field)
363             {
364                 case "source":
365                     // Save the field data
366                     sourceData = attributes.source;
367
368                     // Ensure that we have an array of keys. The most recent key is
369                     // kept at the top of the stack.
370                     if (! appData.source)
371                     {
372                         appData.source = [];
373                     }
374                     break;
375
376                 case "apk":
377                     // Save the field data
378                     apkData = attributes.apk;
379
380                     // Ensure that we have an array of keys. The most recent key is
381                     // kept at the top of the stack.
382                     if (! appData.apk)
383                     {
384                         appData.apk = [];
385                     }
386                     break;
387
388                 default:

```

```

389         // Replace what's in the db entry
390         appData[field] = attributes[field];
391         break;
392     }
393 }
394
395 // If this field is required and not available...
396 if (qx.lang.Array.contains(requiredFields, field) && ! appData[field])
397 {
398     // then mark it as missing
399     missing.push(field);
400 }
401 });
402
403 // Issue a query for all category tags
404 categories = rpcjs.dbif.Entity.query("aiagallery.dbif.ObjTags",
405     {
406         type : "element",
407         field : "type",
408         value : "category"
409     },
410     null);
411
412 // We want to look at only the value field of each category
413 categories = categories.map(
414     function(o)
415     {
416         return o.value;
417     });
418
419 // Ensure that at least one of the specified tags is a category
420 bHasCategory = false;
421 tags = appData.tags;
422 for (i = 0; i < tags.length; i++)
423 {
424     // Is this tag a category?
425     if (qx.lang.Array.contains(categories, tags[i]))
426     {
427         // Yup. Mark it.
428         bHasCategory = true;
429
430         // No need to look further.
431         break;
432     }
433 }
434
435 // Did we find at least one category tag?
436 if (! bHasCategory)
437 {
438     // Nope. Let 'em know.
439     error.setCode(3);
440     error.setMessage("At least one category is required");
441     return error;
442 }
443
444 // Were there any missing, required fields?
445 if (missing.length > 0)
446 {
447     // Yup. Let 'em know.
448     error.setCode(4);
449     error.setMessage("Missing required attributes: " + missing.join(", "));
450     return error;
451 }
452
453 // If a new source file was uploaded...
454 if (sourceData)
455 {
456     // ... then update the upload time to now

```

```

457     appData.uploadTime = aiagallery.dbif.MDbifCommon.currentTimestamp();
458 }
459
460 // FIXME: Begin a transaction here
461
462 // Add new tags to the database, and update counts of formerly-existing
463 // tags. Remove "normal" tags with a count of 0.
464 appData.tags.forEach(
465     function(tag)
466     {
467         // If the tag existed previously, ignore it.
468         if (qx.lang.Array.contains(oldTags, tag))
469         {
470             // Remove it from oldTags
471             qx.lang.Array.remove(oldTags, tag);
472             return;
473         }
474
475         // It didn't exist. Create or retrieve existing tag.
476         tagObj = new aiagallery.dbif.ObjTags(tag);
477         tagData = tagObj.getData();
478
479         // If we created it, data is initialized. Otherwise...
480         if (! tagObj.getBrandNew())
481         {
482             // ... it existed, so we need to increment its count
483             ++tagData.count;
484         }
485
486         // Save the tag object
487         tagObj.put();
488     });
489
490 // Anything left in oldTags are those which were removed.
491 oldTags.forEach(
492     function(tag)
493     {
494         tagObj = new aiagallery.dbif.ObjTags(tag);
495         tagData = tagObj.getData();
496
497         // The record has to exist already. Decrement this tag's count.
498         --tagData.count;
499
500         // Ensure it's a "normal" tag
501         if (tagData.type != "normal")
502         {
503             // It's not, so we have nothing more we need to do.
504             return;
505         }
506
507         // If the count is less than 1...
508         if (tagData.count < 1)
509         {
510             // ... then we can remove the tag
511             tagObj.removeSelf();
512         }
513     });
514
515 try
516 {
517     // Save the new source data (if there is any)
518     if (sourceData)
519     {
520         // Save the data and prepend the blob id to the key list
521         key = rpcjs.dbif.Entity.putBlob(sourceData);
522         appData.source.unshift(key);
523     }
524

```

```

525     // Similarly for apk data
526     if (apkData)
527     {
528         // Save the data and prepend the blob id to the key list
529         key = rpcjs.dbif.Entity.putBlob(apkData);
530         appData.apk.unshift(key);
531     }
532 }
533 catch(e)
534 {
535     error.setCode(5);
536     error.setMessage(e.toString());
537     // FIXME: roll back transaction here
538     return error;
539 }
540
541 // Save this record in the database
542 appObj.put();
543
544 // Add all words in text fields to word Search record
545 aiagallery.dbif.MApps._populateSearch(appObj.getData());
546
547 // FIXME: Commit the transaction here
548
549 return appObj.getData(); // This includes newly-created key (if adding)
550 },
551
552 deleteApp : function(uid, error)
553 {
554     var         appObj;
555     var         appData;
556     var         tagObj;
557     var         tagData;
558     var         whoami;
559
560     // Retrieve an instance of this application entity
561     appObj = new aiagallery.dbif.ObjAppData(uid);
562
563     // Does this application exist?
564     if (appObj.getBrandNew())
565     {
566         // It doesn't. Let 'em know.
567         return false;
568     }
569
570     // Get the object data
571     appData = appObj.getData();
572
573     // Determine who the logged-in user is
574     whoami = this.getWhoAmI();
575
576     // Ensure that the logged-in user owns this application
577     if (! whoami || appData.owner != whoami.email)
578     {
579         // He doesn't. Someone's doing something nasty!
580         error.setCode(1);
581         error.setMessage("Not owner");
582         return error;
583     }
584
585     // Decrement counts for tags used by this application.
586     appData.tags.forEach(
587         function(tag)
588         {
589             // Get this tag object
590             tagObj = new aiagallery.dbif.ObjTags(tag);
591             tagData = tagObj.getData();
592

```

```

593     // The record has to exist already. Decrement this tag's count.
594     --tagData.count;
595
596     // Ensure it's a "normal" tag
597     if (tagData.type != "normal")
598     {
599         // It's not, so we have nothing more we need to do.
600         return;
601     }
602
603     // If the count is less than 1...
604     if (tagData.count < 1)
605     {
606         // ... then we can remove the tag
607         tagObj.removeSelf();
608     }
609     });
610
611     // Remove any apk blobs associated with this app
612     if (appData.apk)
613     {
614         appData.apk.forEach(
615             function(apkBlobId)
616             {
617                 rpcjs.dbif.Entity.removeBlob(apkBlobId);
618             });
619     }
620
621     // Similarly for any source blobs
622     if (appData.source)
623     {
624         appData.source.forEach(
625             function(sourceBlobId)
626             {
627                 rpcjs.dbif.Entity.removeBlob(sourceBlobId);
628             });
629     }
630
631     // Delete the app
632     appObj.removeSelf();
633
634     aiagallery.dbif.MApps._removeAppFromSearch(uid);
635
636     // We were successful
637     return true;
638 },
639
640 /**
641  * Get a portion of the application list.
642  *
643  * @param bStringize {Boolean}
644  *   Whether the tags, previousAuthors, and status values should be
645  *   reformed into a string representation rather than being returned in
646  *   their native representation.
647  *
648  * @param sortCriteria {Array}
649  *   An array of maps. Each map contains a single key and value, with the
650  *   key being a field name on which to sort, and the value being one of
651  *   the two strings, "asc" to request an ascending sort on that field, or
652  *   "desc" to request a descending sort on that field. The order of maps
653  *   in the array determines the priority of that field in the sort. The
654  *   first map in the array indicates the primary sort key; the second map
655  *   in the array indicates the next-highest-priority sort key, etc.
656  *
657  * @param offset {Integer}
658  *   An integer value >= 0 indicating the number of records to skip, in
659  *   the specified sort order, prior to the first one returned in the
660  *   result set.

```

```

661 *
662 * @param limit {Integer}
663 *   An integer value > 0 indicating the maximum number of records to return
664 *   in the result set.
665 *
666 * @param bAll {Boolean}
667 *   Whether to return all applications (if permissions allow it) rather
668 *   than only those applications owned by the logged-in user.
669 */
670 _getAppList : function(bStringize, sortCriteria, offset, limit, bAll)
671 {
672     var          categories;
673     var          categoryNames;
674     var          applList;
675     var          whoami;
676     var          criteria;
677     var          resultCriteria = [];
678     var          owners;
679
680     // Get the current user
681     whoami = this.getWhoAmI();
682
683     // Create the criteria for a search of apps of the current user
684     if (! bAll)
685     {
686         criteria =
687         {
688             type : "element",
689             field : "owner",
690             value : whoami.email
691         };
692     }
693     else
694     {
695         // We want all objects
696         criteria = null;
697     }
698
699     // If an offset is requested...
700     if (typeof(offset) != "undefined" && offset != null)
701     {
702         // ... then specify it in the result criteria.
703         resultCriteria.push({ "offset" : offset });
704     }
705
706     // If a limit is requested...
707     if (typeof(limit) != "undefined" && limit != null)
708     {
709         // ... then specify it in the result criteria
710         resultCriteria.push({ "limit" : limit });
711     }
712
713     // If sort criteria are given...
714     if (typeof(sortCriteria) != "undefined" && sortCriteria != null)
715     {
716         // ... then add them too.
717         for (var sortField in sortCriteria)
718         {
719             resultCriteria.push(
720             {
721                 type : "sort",
722                 field: sortField,
723                 order: sortCriteria[sortField]
724             });
725         }
726     }
727
728     // Issue a query for all apps

```

```

729     appList = rpcjs.dbif.Entity.query("aiagallery.dbif.ObjAppData",
730                                     criteria,
731                                     resultCriteria);
732
733     // Manipulate each App individually, before returning
734     appList.forEach(
735         function(app)
736         {
737             // Replace the owner name with the owner's display name
738             owners = rpcjs.dbif.Entity.query("aiagallery.dbif.ObjVisitors",
739                                             app["owner"]);
740
741             // If it's not an "all" request (administrator)...
742             if (! bAll)
743             {
744                 // ... then replace his visitor id with his display name
745                 app["owner"] = owners[0].displayName;
746             }
747             else
748             {
749                 // Otherwise add the display name
750                 app["displayName"] = owners[0].displayName;
751             }
752
753             // If we were asked to stringize the values...
754             if (bStringize)
755             {
756                 // ... then send each App to the Stringizer
757                 aiagallery.dbif.MApps._.stringizeAppInfo(app);
758             }
759
760         });
761
762
763
764     // Create the criteria for a search of tags of type "category"
765     criteria =
766     {
767         type : "element",
768         field : "type",
769         value : "category"
770     };
771
772     // Issue a query for category tags
773     categories = rpcjs.dbif.Entity.query("aiagallery.dbif.ObjTags",
774                                       criteria,
775                                       [
776                                           {
777                                               type : "sort",
778                                               field : "value",
779                                               order : "asc"
780                                           }
781                                       ]);
782
783     // They want only the tag value to be returned
784     categoryNames = categories.map(function() { return arguments[0].value; });
785
786     // We've built the whole list. Return it.
787     return { apps : appList, categories : categoryNames };
788 },
789
790 /**
791  * Get a the application list of the logged-in user.
792  *
793  * @param bStringize {Boolean}
794  *   Whether the tags, previousAuthors, and status values should be
795  *   reformed into a string representation rather than being returned in
796  *   their native representation.

```

```

797 *
798 * @param sortCriteria {Array}
799 *   An array of maps. Each map contains a single key and value, with the
800 *   key being a field name on which to sort, and the value being one of
801 *   the two strings, "asc" to request an ascending sort on that field, or
802 *   "desc" to request a descending sort on that field. The order of maps
803 *   in the array determines the priority of that field in the sort. The
804 *   first map in the array indicates the primary sort key; the second map
805 *   in the array indicates the next-highest-priority sort key, etc.
806 *
807 * @param offset {Integer}
808 *   An integer value >= 0 indicating the number of records to skip, in
809 *   the specified sort order, prior to the first one returned in the
810 *   result set.
811 *
812 * @param limit {Integer}
813 *   An integer value > 0 indicating the maximum number of records to return
814 *   in the result set.
815 */
816 getAppList : function(bStringize, sortCriteria, offset, limit)
817 {
818   return this._getAppList(bStringize, sortCriteria, offset, limit, false);
819 },
820
821 /**
822 * Get a the entire application list.
823 *
824 * @param bStringize {Boolean}
825 *   Whether the tags, previousAuthors, and status values should be
826 *   reformed into a string representation rather than being returned in
827 *   their native representation.
828 *
829 * @param sortCriteria {Array}
830 *   An array of maps. Each map contains a single key and value, with the
831 *   key being a field name on which to sort, and the value being one of
832 *   the two strings, "asc" to request an ascending sort on that field, or
833 *   "desc" to request a descending sort on that field. The order of maps
834 *   in the array determines the priority of that field in the sort. The
835 *   first map in the array indicates the primary sort key; the second map
836 *   in the array indicates the next-highest-priority sort key, etc.
837 *
838 * @param offset {Integer}
839 *   An integer value >= 0 indicating the number of records to skip, in
840 *   the specified sort order, prior to the first one returned in the
841 *   result set.
842 *
843 * @param limit {Integer}
844 *   An integer value > 0 indicating the maximum number of records to return
845 *   in the result set.
846 */
847 getAppListAll : function(bStringize, sortCriteria, offset, limit)
848 {
849   return this._getAppList(bStringize, sortCriteria, offset, limit, true);
850 },
851
852 /**
853 * Issue a query for a set of applicaitons. Limit the response to
854 * particular fields.
855 *
856 * @param criteria {Map|Key}
857 *   Criteria for selection of which applications to return. This
858 *   parameter is in the format described in the 'searchCriteria'
859 *   parameter of rpcjs.dbif.Entity.query().
860 *
861 * @param requestedFields {Map?}
862 *   If provided, this is a map containing, as the member names, the
863 *   fields which should be returned in the results. The value of each
864 *   entry in the map indicates what to name that field, in the

```

```

865 *   result. (This produces a mapping of the field names.) An example is
866 *   requestedFields map might look like this:
867 *
868 *   {
869 *     uid    : "uid",
870 *     title  : "label", // remap the title field to be called "label"
871 *     image1 : "icon",  // remap the image1 field to be called "icon"
872 *     tags   : "tags"
873 *   }
874 *
875 *     return { apps : appList, categories : categoryNames };
876 * @return {Map}
877 *   The return value is a map with two members: "apps" and
878 *   "categories". The former is an array of maps, each providing
879 *   information about one application. The latter is an array of the tags
880 *   which are identified as top-level categories.
881 */
882 appQuery : function(criteria, requestedFields)
883 {
884     var          appList;
885     var          categories;
886     var          categoryNames;
887     var          owners;
888
889     appList = rpcjs.dbif.Entity.query("aiagallery.dbif.ObjAppData", criteria);
890
891     // Manipulate each App individually
892     appList.forEach(
893         function(app)
894         {
895             // Issue a query for this visitor
896             owners = rpcjs.dbif.Entity.query("aiagallery.dbif.ObjVisitors",
897                 app.owner);
898
899             // Replace the (private) owner id with his display name
900             app.owner = owners[0].displayName;
901
902             // If there were requested fields specified...
903             if (requestedFields)
904             {
905                 // Send to the requestedFields function for removal and remapping
906                 aiagallery.dbif.MApps._requestedFields(app, requestedFields);
907             }
908
909         });
910     // Create the criteria for a search of tags of type "category"
911     criteria =
912     {
913         type : "element",
914         field : "type",
915         value : "category"
916     };
917
918     // Issue a query for all categories
919     categories = rpcjs.dbif.Entity.query("aiagallery.dbif.ObjTags",
920         criteria,
921         [
922             {
923                 type : "sort",
924                 field : "value",
925                 order : "asc"
926             }
927         ]
928     );
929
930     // Tag objects contain the tag value, type, and count of uses. For this
931     // procedure, we want to return only the tag value.
932     categoryNames = categories.map(function()
933     {

```

```

933         return arguments[0].value;
934     });
935
936     return { apps : appList, categories : categoryNames };
937 },
938
939 /**
940  * Perform a keyword search on the given string, as well as an appQuery on
941  * the given criteria, and return the intersection of the results.
942  *
943  * @param queryArgs {Map}
944  *   This is a map containing member names that are the 4 unique parameters
945  *   to MApps.appQuery(criteria, requestedFields) and
946  *   keywordSearch(keywordString, queryFields, requestedFields)
947  *
948  *   The value of each of those members is the argument to be passed upon
949  *   calling that RPC
950  *
951  *   For example:
952  *
953  *   {
954  *     criteria      : {...(see MApps.appQuery() docu...)},
955  *     requestedFields : {...(see MApps.appQuery() docu...)},
956  *     keywordString  : "Some words to search on",
957  *     queryFields    : null // not implemented yet,
958  *                       // pass null for safety
959  *   }
960  *
961  * @return {Map}
962  *   The return value is an array of maps, each providing information
963  *   about one application.
964  *
965  */
966 intersectKeywordAndQuery : function(queryArgs, error)
967 {
968
969     var          keywordString;
970     var          appQueryResults;
971     var          appQueryResultArr = [];
972     var          bQueryUsed = true;
973     var          keywordSearchResultArr = [];
974     var          bKeywordUsed = true;
975     var          intersectionArr = [];
976
977     // Going to perform keyword search first
978     keywordString = queryArgs["keywordString"];
979
980     // If there was no keyword string provided
981     if (keywordString === null || typeof keywordString === "undefined" ||
982         keywordString === "")
983     {
984         // Then just use the results of appQuery
985         bKeywordUsed = false;
986     }
987     else
988     {
989         // Perform keyword search
990         keywordSearchResultArr = this.keywordSearch(keywordString,
991                                                     queryArgs["queryFields"],
992                                                     queryArgs["requestedFields"],
993                                                     error);
994
995         // If there was a problem
996         if (keywordSearchResultArr === error)
997         {
998             // Propegate the failure
999             return error;
1000         }
1001     }

```

```

1001
1002 // Was there any criteria given to perform appQuery on?
1003 if (queryArgs["criteria"]["method"] === "and" &&
1004     queryArgs["criteria"]["children"].length === 0)
1005 {
1006     // No, just use the keyword results
1007     bQueryUsed = false;
1008 }
1009 else
1010 {
1011     // Yes, use it to perform appQuery
1012     appQueryResults = this.appQuery(queryArgs["criteria"],
1013                                     queryArgs["requestedFields"],
1014                                     error);
1015
1016     // If there was a problem
1017     if (appQueryResults === error)
1018     {
1019         // Propagate the failure
1020         return error;
1021     }
1022
1023     // Unwrap the appQuery results
1024     appQueryResultArr = appQueryResults["apps"];
1025 }
1026
1027 // Was nothing given to search on?
1028 if (!bKeywordUsed && !bQueryUsed)
1029 {
1030     // This is an error
1031     error.setCode(1);
1032     error.setMessage("No keyword or search criteria given");
1033     return error;
1034 }
1035
1036 // Was just appQuery used?
1037 if (!bKeywordUsed)
1038 {
1039     // Then just return its results
1040     return appQueryResultArr;
1041 }
1042
1043 // Was just keyword search used?
1044 if (!bQueryUsed)
1045 {
1046     // Then just return its results
1047     return keywordSearchResultArr;
1048 }
1049
1050 // If we got here, then both keyword search and app query ran so...
1051
1052 // Perform intersection operation
1053 keywordSearchResultArr.forEach(function(keywordAppObj)
1054 {
1055     appQueryResultArr.forEach(function(appQueryAppObj)
1056     {
1057         if (keywordAppObj["uid"] === appQueryAppObj["uid"])
1058         {
1059             intersectionArr.push(appQueryAppObj);
1060         }
1061     });
1062 });
1063
1064 // Return the intersection between the two result sets
1065 return intersectionArr;
1066
1067 },
1068

```

```

1069  /**
1070  * Get a list of Apps from a discrete list of App UIDs
1071  *
1072  * @param uidArr {Array}
1073  * An Array containing App UIDs which are to be exchanged for actual App Data
1074  *
1075  * @param requestedFields {Map?}
1076  * If provided, this is a map containing, as the member names, the
1077  * fields which should be returned in the results. The value of each
1078  * entry in the map indicates what to name that field, in the
1079  * result. (This produces a mapping of the field names.) An example is
1080  * requestedFields map might look like this:
1081  *
1082  *   {
1083  *     uid      : "uid",
1084  *     title    : "label", // remap the title field to be called "label"
1085  *     image1   : "icon",  // remap the image1 field to be called "icon"
1086  *     tags     : "tags"
1087  *   }
1088  *
1089  * @return {Array}
1090  * An array of maps. Each map contains data about one of the Apps whose
1091  * UIDs were specified.
1092  *
1093  */
1094  getAppListByList : function( uidArr, requestedFields)
1095  {
1096     var          appList = [];
1097     var          owners;
1098
1099     uidArr.forEach(function(uid)
1100     {
1101         appList.push(rpcjs.dbif.Entity.query("aiagallery.dbif.ObjAppData",
1102                                             uid)[0]);
1103     });
1104
1105     // Manipulate each App individually
1106     appList.forEach(
1107         function(app)
1108         {
1109             // Issue a query for this visitor
1110             owners = rpcjs.dbif.Entity.query("aiagallery.dbif.ObjVisitors",
1111                                             app.owner);
1112
1113             // Replace the (private) owner id with his display name
1114             app.owner = owners[0].displayName;
1115
1116             // If there were requested fields specified...
1117             if (requestedFields)
1118             {
1119                 // Send to the requestedFields function for removal and remapping
1120                 aiagallery.dbif.MApps._requestedFields(app, requestedFields);
1121             }
1122
1123         });
1124
1125     return appList;
1126 },
1127
1128  /**
1129  * Get the details about a particular application.
1130  *
1131  * This function will also increment the number of views of the
1132  * requested app by 1.
1133  *
1134  * @param uid {Key}
1135  * The unique identifier of an application.
1136  *

```

```

1137 * @param bStringize {Boolean}
1138 *   Whether some non-string parameters should be converted to a string
1139 *   representation. For example, the "tags" and "previousAuthors" fields
1140 *   are arrays, and are returned as an array if this parameter is false,
1141 *   but are returned as a comma-separated string of the array values if
1142 *   this parameter value is true. The status value, an integer, is
1143 *   returned as a number when this parameter is false, and as the string
1144 *   representing the status value when it is true.
1145 *
1146 * @param requestedFields {Map?}
1147 *   If provided, this is a map containing, as the member names, the
1148 *   fields which should be returned in the results. The value of each
1149 *   entry in the map indicates what to name that field, in the
1150 *   result. (This produces a mapping of the field names.) An example is
1151 *   requestedFields map might look like this:
1152 *
1153 *   {
1154 *     uid    : "uid",
1155 *     title  : "label", // remap the title field to be called "label"
1156 *     image1 : "icon",  // remap the image1 field to be called "icon"
1157 *     tags   : "tags"
1158 *   }
1159 *
1160 * @param error {rpcjs.rpc.error.Error}
1161 *   All RPCs are passed, as their final argument, an error object. Most
1162 *   don't use it, but this one does. If the application being requested
1163 *   is not found (which, since the uid of the specific application is
1164 *   provided as a parameter, likely means that it was just deleted), an
1165 *   error is generated back to the client by setting the code and message
1166 *   in this object.
1167 *
1168 * @return {Map}
1169 *   All of the information about the application, with the exception that
1170 *   the owner has been converted to the owner's display name.
1171 */
1172 getAppInfo : function(uid, bStringize, requestedFields, error)
1173 {
1174     var          app;
1175     var          appObj;
1176     var          tagTable;
1177     var          whoami;
1178     var          criteria;
1179     var          owners;
1180     var          likesList;
1181
1182     whoami = this.getWhoAmI();
1183
1184     //Update the views and last viewed date
1185
1186     //Get the actual object
1187     appObj = new aiagallery.dbif.ObjAppData(uid);
1188
1189     // See if this app exists.
1190     if (appObj.getBrandNew())
1191     {
1192         // It doesn't. Let 'em know that the application has just been removed
1193         // (or there's a programmer error)
1194         error.setCode(1);
1195         error.setMessage("Application is not available. " +
1196             "It may have been removed recently.");
1197         return error;
1198     }
1199
1200     app = appObj.getData();
1201
1202     //Increment the number of views by 1.
1203     app.numViewed++;
1204

```

```

1205 //Set the "lastViewedDate" to the time this function was called
1206 app.lastViewedTime = aiagallery.dbif.MDbifCommon.currentTimestamp();
1207
1208 //Put back on the database
1209 appObj.put();
1210
1211 // If the application status is not Active, only the owner can view it.
1212 if (app.status != aiagallery.dbif.Constants.Status.Active &&
1213     (! whoami || app.owner != whoami.email))
1214 {
1215     // It doesn't. Let 'em know that the application has just been removed
1216     // (or there's a programmer error)
1217     error.setCode(2);
1218     error.setMessage("Application is not available. " +
1219                     "It may have been removed recently.");
1220     return error;
1221 }
1222
1223 // Issue a query for this visitor
1224 owners = rpcjs.dbif.Entity.query("aiagallery.dbif.ObjVisitors",
1225                                 app.owner);
1226
1227 // Replace the (private) owner id with his display name
1228 app.owner = owners[0].displayName;
1229
1230 // Determine if the current user has already liked this application
1231 // Construct query criteria for "likes of this app by current visitor"
1232 criteria =
1233 {
1234     type : "op",
1235     method : "and",
1236     children :
1237     [
1238         {
1239             type: "element",
1240             field: "app",
1241             value: uid
1242         },
1243         {
1244             type: "element",
1245             field: "visitor",
1246             value: whoami.email
1247         }
1248     ]
1249 };
1250
1251 // Query for the likes of this app by the current visitor
1252 // (an array, which should have length zero or one).
1253 likesList = rpcjs.dbif.Entity.query("aiagallery.dbif.ObjLikes",
1254                                     criteria,
1255                                     null);
1256
1257 // If there were any results, this user has already liked it.
1258 app.bAlreadyLiked = likesList.length > 0;
1259
1260 // If we were asked to stringize the values...
1261 if (bStringize)
1262 {
1263     aiagallery.dbif.MApps.__stringizeAppInfo(app);
1264 }
1265
1266 // If there were requested fields specified...
1267 if (requestedFields)
1268 {
1269     // If the "comments" field was requested
1270     if (requestedFields["comments"])
1271     {
1272

```

```
1273         // Use function from Mixin MComments to add comments to app info
1274         // object
1275         app.comments = this.getComments(uid, null, null, error);
1276     }
1277
1278     // Send it to the requestedFields function for stripping and remapping
1279     aiagallery.dbif.MApps._requestedFields(app, requestedFields);
1280 }
1281
1282 // Give 'em what they came for
1283 return app;
1284 }
1285 }
1286 });
```

Appendix 20

aiagallery.dbif.MComments

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  * Copyright (c) 2011 Reed Spool
4  *
5  * License:
6  *   LGPL: http://www.gnu.org/licenses/lgpl.html
7  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
8  */
9
10 qx.Mixin.define("aiagallery.dbif.MComments",
11 {
12   construct : function()
13   {
14     this.registerService("addComment",
15       this.addComment,
16       [ "appId", "text", "parentTreeId" ] );
17
18     this.registerService("deleteComment",
19       this.deleteComment,
20       [ "appId", "treeId" ] );
21
22     this.registerService("getComments",
23       this.getComments,
24       [ "appId", "offset", "limit" ] );
25   },
26
27   statics :
28   {
29     _base160arr :
30     [
31       48, 49, 50, 51, 52, 53, 54, 55, 56, 57, /* 0-9 */
32       58, 59, 65, 66, 67, 68, 69, 70, 71, 72, /* : ; A-H */
33       73, 74, 75, 76, 77, 78, 79, 80, 81, 82, /* I-R */
34       83, 84, 85, 86, 87, 88, 89, 90, 97, 98, /* S-Z , a-b */
35       99, 100, 101, 102, 103, 104, 105, 106, 107, 108, /* c-l */
36       109, 110, 111, 112, 113, 114, 115, 116, 117, 118, /* m-v */
37       119, 120, 121, 122, 160, 161, 162, 163, 164, 165, /* w-z, latin1 */
38       166, 167, 168, 169, 170, 171, 172, 173, 174, 175, /* latin1 */
39       176, 177, 178, 179, 180, 181, 182, 183, 184, 185, /* latin1 */
40       186, 187, 188, 189, 190, 191, 192, 193, 194, 195, /* latin1 */
41       196, 197, 198, 199, 200, 201, 202, 203, 204, 205, /* latin1 */
42       206, 207, 208, 209, 210, 211, 212, 213, 214, 215, /* latin1 */
43       216, 217, 218, 219, 220, 221, 222, 223, 224, 225, /* latin1 */
44       226, 227, 228, 229, 230, 231, 232, 233, 234, 235, /* latin1 */
45       236, 237, 238, 239, 240, 241, 242, 243, 244, 245, /* latin1 */
46       246, 247, 248, 249, 250, 251, 252, 253, 254, 255 /* latin1 */
47     ]
48   },

```

```

49
50 members :
51 {
52     /**
53      * Add a comment to the database.
54      *
55      * @param appId
56      * This is either a string or number which is the uid of the app to which
57      * this comment is associated.
58      *
59      * @param text {String}
60      * The comment text itself.
61      *
62      * @param parentTreeId {String}
63      * The parent's treeId.
64      *
65      */
66     addComment : function(appId, text, parentTreeId, error)
67     {
68         var whoami;
69         var commentObj;
70         var commentObjData;
71         var parentAppObj;
72         var parentAppData;
73         var parentCommentObj;
74         var parentCommentData;
75         var parentTreeId;
76         var myTreeId;
77         var parentList;
78         var parentNumChildren;
79
80
81         // Determine who the logged-in user is
82         whoami = this.getWhoAmI();
83
84         // Is text empty or just whitespace?
85         if ( text === null || text === "" || text.match(/\S/gi) === null)
86         {
87             // Yes, discard the trash and let the user know.
88             error.setCode(3);
89             error.setMessage("Empty Comment");
90             return error;
91         }
92
93         // Regardless, we need to have parentNumChildren and parentTreeId filled
94         // by the end of this if-else block. Where ever we got the numChildren
95         // from also needs to be incremented and updated.
96
97         // Need to get and increment the Parent App's numRootComments
98         // and numComments total
99         parentAppObj = new aiagallery.dbif.ObjAppData(appId);
100
101         parentAppData = parentAppObj.getData();
102
103         // Was the parent comment's treeId provided?
104         if (typeof(parentTreeId) === "undefined" || parentTreeId === null)
105         {
106             // No, we're going to use the root parent id, ""
107             parentTreeId = "";
108
109             // Get what we need
110             parentNumChildren = parentAppData.numRootComments || 0;
111
112             // Increment parent app's # of children
113             parentAppData.numRootComments = parentNumChildren + 1;
114         }
115         else
116         {

```

```

117     // Yes, use it to get the parent comment object.
118     parentCommentObj =
119         new aiagallery.dbif.ObjComments([appId, parentTreeId]);
120
121     parentCommentData = parentCommentObj.getData();
122
123     // Was our parentUID invalid, resulting in a new ObjComments?
124     if (parentCommentObj.getBrandNew())
125     {
126         // We can't use an invalid UID as our parent UID!
127         error.setCode(1);
128         error.setMessage("Unrecognized parent treeId");
129         return error;
130     }
131
132     // Get what we came for.
133     parentNumChildren = parentCommentData.numChildren;
134     parentTreeId = parentCommentData.treeId;
135
136     // Increment parent comment's # of children
137     parentCommentData.numChildren = parentNumChildren + 1;
138
139     // Save the new # children in the parent comment
140     parentCommentObj.put();
141 }
142
143 // Increment the total number of comments on the App
144 parentAppData.numComments++;
145
146 // Update the parent app and/or comment object.
147 // Congrats! a new baby comment!
148 parentAppObj.put();
149
150 // Append our parent's number of children, base160 encoded, to parent's
151 // treeId
152 myTreeId = parentTreeId + this._numToBase160(parentNumChildren);
153
154 // Get a new ObjComments object, with our appId and newly generated
155 // treeId.
156 commentObj = new aiagallery.dbif.ObjComments([appId, myTreeId]);
157
158 // Was a comment with this key already in the DB?
159 if (!commentObj.getBrandNew())
160 {
161     // That's an error
162     error.setCode(3);
163     error.setMessage("Attempted to overwrite existing comment");
164     return error;
165 }
166
167 // Retrieve a data object to manipulate.
168 commentObjData = commentObj.getData();
169
170 // Set up all the rest of the data
171 commentObjData.visitor = whoami.email;
172 commentObjData.text = text;
173
174 // Save this in the database
175 commentObj.put();
176
177 // Replace the visitor id with his display name.
178 commentObjData.visitor = whoami.userId;
179
180 // This includes newly-created key
181 return commentObjData;
182 },
183
184 /**

```

```

185     * Delete a specific individual comment
186     *
187     * @param appId {Number}
188     *   This is the unique identifier for the app containing the comment to
189     *   delete
190     *
191     * @param treeId {String}
192     *   This is the thread tree identifier for the comment which is to be
193     *   deleted
194     *
195     * @return {Boolean}
196     *   Returns true if deletion was successful. If false is returned, nothing
197     *   was deleted.
198     */
199 deleteComment : function(appId, treeId, error)
200 {
201     var          commentObj;
202     var          parentAppObj;
203     var          parentAppData;
204     var          parentTreeId;
205
206     // Retrieve an instance of this comment entity
207     commentObj = new aiagallery.dbif.ObjComments([appId, treeId]);
208
209     // Does this comment exist?
210     if (commentObj.getBrandNew())
211     {
212         // It doesn't. Let 'em know.
213         return false;
214     }
215
216     // Find out the App that was commented on and...
217     parentAppObj = new aiagallery.dbif.ObjAppData(appId);
218
219     parentAppData = parentAppObj.getData();
220
221     // Decrement the number of comments attached to this App.
222     parentAppData["numComments"]--;
223
224     // Save this change
225     parentAppObj.put();
226
227     // Delete the app
228     commentObj.removeSelf();
229
230     // We were successful
231     return true;
232 },
233
234 /**
235  * Get comments associated with an App
236  *
237  * @param appId {?}
238  *   The appId whose comments should be returned
239  *
240  * @param offset {Integer}
241  *   An integer value >= 0 indicating the number of records to skip, in
242  *   the specified sort order, prior to the first one returned in the
243  *   result set.
244  *
245  * @param limit {Integer}
246  *   An integer value > 0 indicating the maximum number of records to return
247  *   in the result set.
248  *
249  * @param error {rpcjs.rpc.error.Error}
250  *   All RPCs are passed, as their final argument, an error object. Most
251  *   don't use it, but this one does. If the application being requested
252  *   is not found (which, since the uid of the specific application is

```

```

253 *   provided as a parameter, likely means that it was just deleted), an
254 *   error is generated back to the client by setting the code and message
255 *   in this object.
256 *
257 * @return {Array}
258 *   An array containing all of the comments related to this app
259 *
260 */
261 getComments : function(appId, offset, limit, error)
262 {
263     var         commentList;
264     var         resultCriteria = [];
265
266
267     // If an offset is requested...
268     if (typeof(offset) !== "undefined" && offset !== null)
269     {
270         // ... then specify it in the result criteria.
271         resultCriteria.push({ "offset" : offset });
272     }
273
274     // If a limit is requested...
275     if (typeof(limit) !== "undefined" && limit !== null)
276     {
277         // ... then specify it in the result criteria
278         resultCriteria.push({ "limit" : limit });
279     }
280
281     // Issue a query for all comments, with limit and offset settings applied
282     commentList = rpcjs.dbif.Entity.query("aiagallery.dbif.ObjComments",
283         {
284             type : "element",
285             field: "app",
286             value: appId
287         },
288         resultCriteria);
289
290     try
291     {
292         commentList.forEach(function(obj)
293         {
294             // Replace this owner with his display name
295             obj.visitor =
296                 aiagallery.dbif.MVisitors._getDisplayname(obj.visitor, error);
297
298             // Did we fail to find this owner?
299             if (obj.visitor === error)
300             {
301                 // Yup. Abort the request.
302                 throw error;
303             }
304         });
305     }
306     catch(error)
307     {
308         return error;
309     }
310
311     return commentList;
312 },
313
314 /**
315 * Encode an integer as a string of base160 characters
316 *
317 * @param val {Number}
318 *   An integer to base160 encode
319 *
320 * @return {String}

```

```

321     *   A string of length 4 containing base160 digits as characters.
322     */
323     _numToBase160 : function(val)
324     {
325         var retStr = "";
326
327         for (var i = 3; i >= 0 ; i--)
328         {
329             // Take the number mod 160. Prepend the ASCII char of the result.
330             retStr = String.fromCharCode(
331                 aiagallery.dbif.MComments._base160arr[val % 160]) + retStr;
332             val = Math.floor(val / 160);
333         }
334
335         return retStr;
336     },
337     /**
338     * Increment the base160 number passed. This only augments the farthest
339     * right-most 4 characters (base160 digits).
340     *
341     * @param base160str {String}
342     *   An integer encoded as a string of base160 characters.
343     *
344     * @return {String}
345     *   An integer encoded as a string of base160 characters. This is the
346     *   argument + 1.
347     */
348     _incrementBase160 : function(base160str)
349     {
350         var len = base160str.length;
351         var i;
352         var notMyPiece = base160str.substr(0, len-4);
353         var retStr = "";
354         var ch;
355         var index;
356
357         // We only care about the rightmost 4 digits, one level in the tree.
358         for (i = len - 1; i >= len-4 ; i--)
359         {
360             // Get this digit
361             ch = base160str.charCodeAt(i);
362
363             // Find the index of this digit in the encoding array.
364             index = aiagallery.dbif.MComments._base160arr.indexOf(ch);
365
366             // Is this the last entry in the encoding array?
367             if (index === aiagallery.dbif.MComments._base160arr.length - 1)
368             {
369                 // Yup. This is a carry. This value gets base160arr[0]. We go on to
370                 // the next higher-order digit by continuing through the for-loop
371                 retStr =
372                     String.fromCharCode(aiagallery.dbif.MComments._base160arr[0]) +
373                     retStr;
374             }
375             else
376             {
377                 // No carry. Just add 1, piece everything together, and we're done.
378                 retStr =
379                     String.fromCharCode(
380                         aiagallery.dbif.MComments._base160arr[index + 1]) +
381                     retStr;
382                 retStr = base160str.substring(len - 4, i) + retStr;
383                 break;
384             }
385         }
386
387         return notMyPiece + retStr;
388     }

```

389 }
390 });

Appendix 21

aiagallery.dbif.MFlags

```

1  /**
2  * Copyright (c) 2011 Chris Adler
3  *
4  * License:
5  *   LGPL: http://www.gnu.org/licenses/lgpl.html
6  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7  */
8
9  qx.Mixin.define("aiagallery.dbif.MFlags",
10 {
11   construct : function()
12   {
13
14     this.registerService("flagIt",
15       this.flagIt,
16       [ "flagType",
17         "explanationInput",
18         "appId",
19         "commentId"
20       ]);
21   },
22
23   members :
24   {
25     /**
26     * Add one to the number of times this app has been liked
27     *
28     * @param flagType{Integer}
29     *   This is the value of the type of flag that got submitted
30     *   (App : 0, Comment : 1 )
31     *
32     * @param explanationInput{String}
33     *   This is the string the user will input as the reason the app or comment
34     *   is being flagged.
35     *
36     * @param appId {Integer}
37     *   This is either a string or number which is the uid of the app which is
38     *   being liked.
39     *
40     * @param commentId{String}
41     *   This is the string that is the treeId of the comment. If an app was
42     *   flagged input a null
43     *
44     * @return {Integer || Status}
45     *   This is the value of the status of the application
46     *   (Banned : 0, Pending : 1, Active : 2)
47     */
48

```

```

49  flagIt : function(flagType, explanationInput, appId, commentId, error)
50  {
51      var          appObj;
52      var          appDataObj;
53      var          appNum;
54      var          result;
55      var          criteria;
56      var          newFlag;
57      var          Data;
58
59
60      var          visitorId= this.whoAmI().email;
61      var          maxFlags = aiagallery.dbif.Constants.MAX_FLAGGED;
62      var          statusVals = aiagallery.dbif.Constants.Status;
63      var          flagTypeVal = aiagallery.dbif.Constants.FlagType;
64
65      // Check what type of element has been flagged.
66      switch (flagType)
67      {
68          // If an app was flagged:
69          case flagTypeVal.App:
70
71              // store the applications data
72              appObj = new aiagallery.dbif.ObjAppData(appId);
73              appDataObj = appObj.getData();
74              appNum = appDataObj.uid;
75
76              // check to ensure an already existing app was found
77              if (appObj.getBrandNew())
78              {
79                  // If not return an error
80                  error.setCode(1);
81                  error.setMessage(
82                      "Application with that ID not found. Unable to flag.");
83                  return error;
84              }
85
86              // initialize the new flag to be put on the database
87              newFlag = new aiagallery.dbif.ObjFlags();
88
89              // store the new flags data
90              var data = {
91                  type          : flagType,
92                  app           : appNum,
93                  comment       : null,
94                  visitor       : visitorId,
95                  explanation   : explanationInput
96              }
97
98              newFlag.setData(data);
99
100             // increments the apps number of flags
101             appDataObj.numCurFlags++;
102
103             // check if the number of flags is greater than or
104             // equal to the maximum allowed
105             if(appDataObj.numCurFlags >= maxFlags)
106             {
107                 // If the app is already pending do not touch the status or
108                 // send an email
109                 if(appDataObj.status != statusVals.Pending)
110                 {
111                     // otherwise set the app to pending and send an email
112                     appDataObj.status = statusVals.Pending;
113
114                     // placeholder code
115                     alert("email to be sent");
116                 }

```

```

117     }
118     // put the apps new data and the new flag on the database
119     appObj.put();
120     newFlag.put();
121
122     return appDataObj.status;
123
124     // if a comment was flagged
125     case flagTypeVal.Comment:
126
127         // store the comments data
128         var commentObj = new aiagallery.dbif.ObjComments([appId, commentId]);
129         var commentDataObj = commentObj.getData();
130         var commentNum = commentDataObj.treeId;
131
132         // check to ensure an already existing comment was found
133         if (commentObj.getBrandNew())
134         {
135             // if not return an error
136             error.setCode(1);
137             error.setMessage(
138                 "Comment not found. Unable to flag.");
139             return error;
140         }
141
142         // initialize the new flag to be put on the database
143         newFlag = new aiagallery.dbif.ObjFlags();
144
145         // store the flags data
146         var data = {
147             type      : flagType,
148             app       : appId,
149             comment   : commentNum,
150             visitor   : visitorId,
151             explanation : explanationInput
152         }
153
154         newFlag.setData(data);
155
156         // increment the number of flags on the comment
157         commentDataObj.numCurFlags++;
158
159         // check if the number of flags is greater than or
160         // equal to the maximum allowed
161         if(commentDataObj.numCurFlags >= maxFlags)
162         {
163             // If the comment is already pending do not touch the status or
164             // send an email
165             if(commentDataObj.status != statusVals.Pending)
166             {
167                 // otherwise set the comment to pending and send an email
168                 commentDataObj.status = statusVals.Pending;
169
170                 // placeholder code
171                 alert("email to be sent");
172             }
173         }
174         // put the comments new data and the new flag on the database
175         commentObj.put();
176         newFlag.put();
177
178         return commentDataObj.status;
179
180     default:
181         error.setCode(1);
182         error.setMessage(
183             "unknown flag type.");
184         return error;

```

```
185     }
186
187     // Error message should be redone
188     error.setCode(1);
189     error.setMessage(
190         "Reached an un-reachable section in the flagIt rpc.");
191     return error;
192 }
193 }
194 });
```

Appendix 22

aiagallery.dbif.MLiking

```

1  /**
2  * Copyright (c) 2011 Reed Spool
3  *
4  * License:
5  *   LGPL: http://www.gnu.org/licenses/lgpl.html
6  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7  */
8
9  qx.Mixin.define("aiagallery.dbif.MLiking",
10 {
11   construct : function()
12   {
13
14     this.registerService("likesPlusOne",
15                          this.likesPlusOne,
16                          [ "appId" ]);
17   },
18
19   statics :
20   {
21
22   },
23
24   members :
25   {
26     /**
27     * Add one to the number of times this app has been liked
28     *
29     * @param appId {Integer}
30     *   This is either a string or number which is the uid of the app which is
31     *   being liked.
32     *
33     * @return {Integer || Error}
34     *   This is the number of times this app has been liked, or an error if
35     *   the app was not found
36     *
37     */
38     likesPlusOne : function(appId, error)
39     {
40       var          appObj;
41       var          appDataObj;
42       var          myEmail;
43       var          likesList;
44       var          criteria;
45       var          likesObj;
46       var          likesDataObj;
47
48       appObj = new aiagallery.dbif.ObjAppData(appId);

```

```

49
50 // If there's no such app, return error
51 if (appObj.getBrandNew())
52 {
53     error.setCode(1);
54     error.setMessage("App with that ID not found. Unable to like.");
55     return error;
56 }
57
58 // Get the application data
59 appDataObj = appObj.getData();
60
61 // Retrieve my email address (my visitor id)
62 myEmail = this.getWhoAmI().email;
63
64 // Construct query criteria for "likes of this app by current visitor"
65 criteria =
66 {
67     type : "op",
68     method : "and",
69     children :
70     [
71         {
72             type: "element",
73             field: "app",
74             value: appId
75         },
76         {
77             type: "element",
78             field: "visitor",
79             value: myEmail
80         }
81     ]
82 };
83
84 // Query for the likes of this app by the current visitor
85 // (an array, which should have length zero or one).
86 likesList = rpcjs.dbif.Entity.query("aiagallery.dbif.ObjLikes",
87     criteria,
88     null);
89
90 // Only change things if the visitor hasn't already liked this app
91 if (likesList.length === 0)
92 {
93     // Create a new likes object to prevent future re-likes
94     likesObj = new aiagallery.dbif.ObjLikes();
95     likesDataObj = likesObj.getData();
96
97     // Put app and visitor info into it
98     likesDataObj.app = appId;
99     likesDataObj.visitor = myEmail;
100
101     // Write it back to the database
102     likesObj.put();
103
104     // And increment the like count in the DB
105     appDataObj.numLikes++;
106     appObj.put();
107 }
108
109 // Return number of likes (which may or may not have changed)
110 return appDataObj.numLikes;
111 }
112 }
113 });

```

Appendix 23

aiagallery.dbif.MMobile

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  * Copyright (c) 2011 Reed Spool
4  *
5  * License:
6  *   LGPL: http://www.gnu.org/licenses/lgpl.html
7  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
8  */
9
10 qx.Mixin.define("aiagallery.dbif.MMobile",
11 {
12   construct : function()
13   {
14     this.registerService("mobileRequest",
15                          this.mobileRequest,
16                          [ "command" ]);
17   },
18
19   members :
20   {
21     mobileRequest : function(command, error)
22     {
23       var          fields;
24       var          field;
25       var          params;
26
27       // The command is supposed to be a series of colon-separated
28       // fields. Let's split it up and see what we got.
29       fields = command.split(":");
30
31       // The first field is the command name
32       field = fields.shift();
33
34       // Add error as the last parameter to all commands
35       fields.push(error);
36
37       switch(field)
38       {
39         case "all":
40           // Retrieve a list of applications. Parameters are offset, count, and
41           // sort order.
42           return this.__getAll(fields, error);
43
44         case "search":
45           // Search for applications based on some criteria. Lone parameter is
46           // keywordString.
47           return this.__getBySearch(fields, error);
48

```

```

49     case "tag":
50         // Search by tag name. Parameters are the tag name, offset, count, and
51         // sort order.
52         return this.__getByTag(fields, error);
53
54     case "featured":
55         // Get featured apps. Parameters are the offset, count, and sort order.
56         return this.__getByFeatured(fields, error);
57
58     case "by_developer":
59         // Get apps by their owner. Parameters are the owner's display name,
60         // offset, count, and sort order.
61         return this.__getByOwner(fields, error);
62
63     case "uploads":
64         // I don't understand what this one is supposed to do. Parameters are
65         // described as userid, offset, count, and sort order. I suspect that
66         // userid is display name, but what should be returned?
67         return null;
68
69     case "getinfo":
70         // Get information about an application
71         return this.__getAppInfo(fields, error);
72
73     case "comments":
74         // Get comments made about an application
75         return this.__getComments(fields, error);
76
77     case "get_categories":
78         // Get the category list (top-level tags). There are no parameters to
79         // this request.
80         return this.__getCategories(fields, error);
81
82     default:
83         error.setCode(1);
84         error.setMessage("Unrecognized request: " + field);
85         return error;
86     }
87 },
88
89
90 __getAll : function(fields, error)
91 {
92     var requiredParams = 4;
93     for (var i = requiredParams + 1 - fields.length; i > 0; i--)
94     {
95         qx.lang.Array.insertBefore(fields, null, error);
96     }
97
98     var offset = fields.shift();
99     var count = fields.shift();
100    var order = fields.shift();
101    var field = fields.shift();
102
103    var offsetTypeCheck = offset === null || !isNaN(parseInt(offset, 10));
104    var countTypeCheck = count === null || !isNaN(parseInt(count, 10));
105    var orderTypeCheck = order === null || typeof order === "string";
106    var fieldTypeCheck = field === null || typeof field === "string";
107
108    if (!offsetTypeCheck || !countTypeCheck || !orderTypeCheck ||
109        !fieldTypeCheck)
110    {
111        error.setCode(5);
112        error.setMessage("Malformed mobile request: Incorrect parameter type.");
113        return error;
114    }
115
116    var results = rpcjs.dbif.Entity.query(

```

```

117     "aiagallery.dbif.ObjAppData",
118     // We want everything, so null search criteria
119     null,
120     // This is where resultCriteria goes
121     this.__buildResultCriteria( offset, count, order, field));
122
123     try
124     {
125         results.forEach(function(obj)
126         {
127             // Replace this owner with his display name
128             obj["owner"] =
129                 aiagallery.dbif.MVisitors._getDisplayName(obj["owner"], error);
130
131             // Did we fail to find this owner?
132             if (obj["owner"] === error)
133             {
134                 // Yup. Abort the request.
135                 throw error;
136             }
137         });
138     }
139     catch(error)
140     {
141         return error;
142     }
143
144     return this.__stripDataURLs(results);
145 },
146
147 __getBySearch : function(fields, error)
148 {
149     var results;
150     var requiredParams = 3;
151     for (var i = requiredParams + 1 - fields.length; i > 0; i--)
152     {
153         qx.lang.Array.insertBefore(fields, null, error);
154     }
155
156     var keywordString = fields.shift();
157     var offset = fields.shift();
158     var count = fields.shift();
159
160     var offsetTypeCheck = offset === null || !isNaN(parseInt(offset, 10));
161     var countTypeCheck = count === null || !isNaN(parseInt(count, 10));
162
163     if (!offsetTypeCheck || !countTypeCheck)
164     {
165         error.setCode(5);
166         error.setMessage("Malformed mobile request: Incorrect parameter type.");
167         return error;
168     }
169
170     // keyword is required.
171     if (typeof keywordString !== "string")
172     {
173         error.setCode(3);
174         error.setMessage("No search terms given");
175         return error;
176     }
177
178     // Requesting all fields except data URLs (source, apk, image1-3)
179     var requestedFields =
180     {
181         owner          : "owner",
182         title          : "title",
183         description    : "description",
184         //FIXME: Uncomment next line when previous authors are implemented

```

```

185     //previousAuthors : "previousAuthors",
186     tags              : "tags",
187     uploadTime       : "uploadTime",
188     creationTime     : "creationTime",
189     numLikes         : "numLikes",
190     numDownloads     : "numDownloads",
191     numViewed        : "numViewed",
192     numRootComments  : "numRootComments",
193     numComments      : "numComments",
194     status           : "status",
195     uid              : "uid"
196 };
197
198 // Use MSearch Mixin
199 results = this.keywordSearch(keywordString, null, requestedFields, error);
200
201 // If they have not specified a count nor an offset...
202 if (count === null && offset === null)
203 {
204     // ... then return the whole list
205     return results;
206 }
207
208 // If there's no count, return the whole list beginning at offset
209 if (count === null)
210 {
211     offset = parseInt(offset, 10);
212     return results.slice(offset);
213 }
214
215 // There's a count and an offset. Give that group.
216 offset = parseInt(offset, 10);
217 count = parseInt(count, 10);
218 return results.slice(offset, offset + count);
219 },
220
221 __getByTag : function(fields, error)
222 {
223     var requiredParams = 5;
224     for (var i = requiredParams + 1 - fields.length; i > 0; i--)
225     {
226         qx.lang.Array.insertBefore(fields, null, error);
227     }
228
229     var tagName = fields.shift();
230     var offset = fields.shift();
231     var count = fields.shift();
232     var order = fields.shift();
233     var field = fields.shift();
234
235     // tagName is required
236     if (typeof tagName !== "string")
237     {
238         error.setCode(3);
239         error.setMessage("No tag name given");
240         return error;
241     }
242     var offsetTypeCheck = offset === null || !isNaN(parseInt(offset, 10));
243     var countTypeCheck = count === null || !isNaN(parseInt(count, 10));
244     var orderTypeCheck = order === null || typeof order === "string";
245     var fieldTypeCheck = field === null || typeof field === "string";
246
247     if (!offsetTypeCheck || !countTypeCheck || !orderTypeCheck ||
248         !fieldTypeCheck)
249     {
250         error.setCode(5);
251         error.setMessage("Malformed mobile request: Incorrect parameter type.");
252         return error;

```

```

253     }
254
255     var results = rpcjs.dbif.Entity.query(
256         "aiagallery.dbif.ObjAppData",
257         {
258             type : "element",
259             field : "tags",
260             value : tagName
261         },
262         // This is where resultCriteria goes
263         this.__buildResultCriteria(offset, count, order, field));
264
265     try
266     {
267         results.forEach(function(obj)
268         {
269             // Replace this owner with his display name
270             obj["owner"] =
271                 aiagallery.dbif.MVisitors._getDisplayName(obj["owner"], error);
272
273             // Did we fail to find this owner?
274             if (obj["owner"] === error)
275             {
276                 // Yup. Abort the request.
277                 throw error;
278             }
279         });
280     }
281     catch(error)
282     {
283         return error;
284     }
285
286     return this.__stripDataURLs(results);
287 },
288
289 __getByFeatured : function(fields, error)
290 {
291     // This is the same as __getByTag so just prepend the tag name
292     fields.unshift("*Featured*");
293
294     // If the only quality of a Featured App is that it has a *Featured* tag
295     // then this works.
296     return this.__getByTag(fields, error);
297 },
298
299 __getByOwner : function(fields, error)
300 {
301     var requiredParams = 5;
302     for (var i = requiredParams + 1 - fields.length; i > 0; i--)
303     {
304         qx.lang.Array.insertBefore(fields, null, error);
305     }
306
307     var displayName = fields.shift();
308     var offset = fields.shift();
309     var count = fields.shift();
310     var order = fields.shift();
311     var field = fields.shift();
312
313     // displayName is required
314     if (typeof displayName !== "string")
315     {
316         error.setCode(3);
317         error.setMessage("No developer's name given");
318         return error;
319     }
320     var offsetTypeCheck = offset === null || !isNaN(parseInt(offset,10));

```

```

321     var countTypeCheck = count === null || !isNaN(parseInt(count,10));
322     var orderTypeCheck = order === null || typeof order === "string";
323     var fieldTypeCheck = field === null || typeof field === "string";
324
325     if (!offsetTypeCheck || !countTypeCheck || !orderTypeCheck ||
326         !fieldTypeCheck)
327     {
328         error.setCode(5);
329         error.setMessage("Malformed mobile request: Incorrect parameter type.");
330         return error;
331     }
332
333     // First I'm going to trade the displayName for the real owner Id
334     var ownerId =
335         aiagallery.dbif.MVisitors._getVisitorId(displayName, error);
336
337     // Was an error returned?
338     if (ownerId === error)
339     {
340         // Yup. We need to return it.
341         return error;
342     }
343
344     // Then use the ownerId to query for all Apps
345     var results = rpcjs.dbif.Entity.query(
346         "aiagallery.dbif.ObjAppData",
347         {
348             type : "element",
349             field : "owner",
350             value : ownerId
351         },
352         // This is where resultCriteria goes
353         this.__buildResultCriteria( offset, count, order, field));
354
355     // Then make sure the ownerId doesn't get returned
356     results.forEach(function(obj)
357     {
358         obj["owner"] = displayName;
359     });
360
361     return this.__stripDataURLs(results);
362 },
363
364 __getAppInfo : function(fields, error)
365 {
366     var requiredParams = 1;
367     for (var i = requiredParams + 1 - fields.length; i > 0; i--)
368     {
369         qx.lang.Array.insertBefore(fields, null, error);
370     }
371
372     var appId = parseInt(fields.shift(), 10);
373
374     // appId is required
375     if (isNaN(appId))
376     {
377         error.setCode(3);
378         error.setMessage("No App UID given");
379         return error;
380     }
381
382     // Using the method included by mixin MApps
383
384     // Requesting all fields except data URLs (source, apk, image1-3)
385     var requestedFields =
386     {
387         owner      : "owner",
388         title      : "title",

```

```

389     description      : "description",
390     //FIXME: Uncomment next line when previous authors are implemented
391     //previousAuthors : "previousAuthors",
392     tags             : "tags",
393     uploadTime       : "uploadTime",
394     creationTime     : "creationTime",
395     numLikes         : "numLikes",
396     numDownloads     : "numDownloads",
397     numViewed        : "numViewed",
398     numRootComments  : "numRootComments",
399     numComments      : "numComments",
400     status           : "status"
401 };
402
403 // The final parameter to each RPC when called by the RPC Server, is an
404 // error object which we can manipulate if there's an error. In this
405 // case, we're pretending to be the server when we call a different RPC,
406 // so pass its error object.
407
408 // The appId is passed in here as a string, but is a number in reality.
409 return this.getAppInfo(appId, false, requestedFields, error);
410 },
411
412 __getComments : function(fields, error)
413 {
414     var requiredParams = 1;
415     for (var i = requiredParams + 1 - fields.length; i > 0; i--)
416     {
417         qx.lang.Array.insertBefore(fields, null, error);
418     }
419
420     // Make sure appId is an integer
421     var appId = parseInt(fields.shift(), 10);
422
423     // appId is required
424     if (isNaN(appId))
425     {
426         error.setCode(3);
427         error.setMessage("No App UID given");
428         return error;
429     }
430
431     // FIXME: UNTESTED. At time of dev, no comments available to query on
432
433     // The appId is passed in here as a string, but is a number in reality.
434     return this.getComments(appId, null, null, error);
435 },
436
437 __getCategories : function(fields, error)
438 {
439     // fields is expected to be empty
440
441     // Use the method included by mixin MTags
442     return this.getCategoryTags(error);
443 },
444
445 /**
446  * Strip out dataURL fields from app object array
447  *
448  * @param appArr {Array}
449  *   Array of App-dataish objects which need to be trimmed
450  *
451  * @return {Array}
452  *   The mutated array that was passed in
453  */
454 __stripDataURLs : function(appArr)
455 {
456     appArr.forEach(

```

```

457     function(appObj)
458     {
459         delete appObj["image1"];
460         delete appObj["image2"];
461         delete appObj["image3"];
462         delete appObj["source"];
463         delete appObj["apk"];
464     });
465
466     return appArr;
467 },
468
469
470 /**
471  * Build a correctly formatted Result Criteria array for rpc queries
472  *
473  * @param offset {Number}
474  *   Specify how many results to skip
475  *
476  * @param count {Number}
477  *   Limit how many matching results are returned
478  *
479  * @param sortField {String}
480  *   The field on which to sort
481  *
482  * @param sortOrder {String}
483  *   Either "desc" or "asc" to specify the order in which results should be
484  *   returned.
485  *
486  * @return {Array}
487  *   Array contains objects specifying the result criteria
488  *
489  */
490 _buildResultCriteria : function(offset, count, sortOrder, sortField)
491 {
492     // Building the Result Criteria object based on what's given
493     var ret = [];
494
495     // Are the field on which to sort and sort order specified? Then add sort
496     // criteria.
497     if (sortField && sortOrder)
498     {
499         ret.push({ type : "sort", field : sortField, order : sortOrder});
500     }
501
502     // Did they request a certain number of results? add a limit criteria
503     if (count)
504     {
505         ret.push({ type : "limit", value : parseInt(count,10)});
506     }
507
508     // Did they want to skip a number of results? add offset criteria object
509     if (offset)
510     {
511         ret.push({ type : "offset", value : parseInt(offset,10)});
512     }
513
514     // return the whole finished Result Criteria array, or an empty array
515     return ret;
516 }
517 }
518 });

```

Appendix 24

aiagallery.dbif.MSearch

```

1  /**
2  * Copyright (c) 2011 Reed Spool
3  *
4  * License:
5  *   LGPL: http://www.gnu.org/licenses/lgpl.html
6  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7  */
8
9  qx.Mixin.define("aiagallery.dbif.MSearch",
10 {
11   construct : function()
12   {
13
14     this.registerService("keywordSearch",
15                         this.editProfile,
16                         [ "keywordString",
17                           "queryFields",
18                           "requestedFields"]);
19
20   },
21
22   statics :
23   {
24     /**
25     * Array of all searching stopwords
26     */
27     stopWordArr : [ "a", "about", "above", "above", "across", "after",
28                    "afterwards", "again", "against", "all", "almost", "alone",
29                    "along", "already", "also", "although", "always", "am",
30                    "among", "amongst", "amongst", "amount", "an", "and",
31                    "another", "any", "anyhow", "anyone", "anything", "anyway",
32                    "anywhere", "are", "around", "as", "at", "back", "be",
33                    "became", "because", "become", "becomes", "becoming", "been",
34                    "before", "beforehand", "behind", "being", "below",
35                    "beside", "besides", "between", "beyond", "bill", "both",
36                    "bottom", "but", "by", "call", "can", "cannot", "cant",
37                    "co", "con", "could", "couldnt", "cry", "de", "describe",
38                    "detail", "do", "done", "down", "due", "during", "each",
39                    "eg", "eight", "either", "eleven", "else", "elsewhere",
40                    "empty", "enough", "etc", "even", "ever", "every",
41                    "everyone", "everything", "everywhere", "except", "few",
42                    "fifteen", "fify", "fill", "find", "fire", "first", "five",
43                    "for", "former", "formerly", "forty", "found", "four",
44                    "from", "front", "full", "further", "get", "give", "go",
45                    "had", "has", "hasnt", "have", "he", "hence", "her",
46                    "here", "hereafter", "hereby", "herein", "hereupon",
47                    "hers", "herself", "him", "himself", "his", "how",
48                    "however", "hundred", "ie", "if", "in", "inc", "indeed",

```

```

49         "interest", "into", "is", "it", "its", "itself", "keep",
50         "last", "latter", "latterly", "least", "less", "ltd",
51         "made", "many", "may", "me", "meanwhile", "might", "mill",
52         "mine", "more", "moreover", "most", "mostly", "move",
53         "much", "must", "my", "myself", "name", "namely", "neither",
54         "never", "nevertheless", "next", "nine", "no", "nobody",
55         "none", "noone", "nor", "not", "nothing", "now", "nowhere",
56         "of", "off", "often", "on", "once", "one", "only", "onto",
57         "or", "other", "others", "otherwise", "our", "ours",
58         "ourselves", "out", "over", "own", "part", "per", "perhaps",
59         "please", "put", "rather", "re", "same", "see", "seem",
60         "seemed", "seeming", "seems", "serious", "several", "she",
61         "should", "show", "side", "since", "sincere", "six",
62         "sixty", "so", "some", "somehow", "someone", "something",
63         "sometime", "sometimes", "somewhere", "still", "such",
64         "system", "take", "ten", "than", "that", "the", "their",
65         "them", "themselves", "then", "thence", "there",
66         "thereafter", "thereby", "therefore", "therein",
67         "thereupon", "these", "they", "thick", "thin", "third",
68         "this", "those", "though", "three", "through", "throughout",
69         "thru", "thus", "to", "together", "too", "top", "toward",
70         "towards", "twelve", "twenty", "two", "un", "under",
71         "until", "up", "upon", "us", "very", "via", "was", "we",
72         "well", "were", "what", "whatever", "when", "whence",
73         "whenever", "where", "whereafter", "whereas", "whereby",
74         "wherein", "whereupon", "wherever", "whether", "which",
75         "while", "whither", "who", "whoever", "whole", "whom",
76         "whose", "why", "will", "with", "within", "without",
77         "would", "yet", "you", "your", "yours", "yourself",
78         "yourselves", "the"]
79
80
81     },
82
83     members :
84     {
85         /**
86          * Returns an array of App Info objects, which contain a word or words from
87          * the keyword string.
88          *
89          * NOTE: There is no ordering of the returned Apps. This is simply a
90          * keyword query.
91          *
92          * @param keywordString {String}
93          * A space delimited string of words to query on. A returned App will
94          * contain all of the words in the string, in any order.
95          *
96          * @param queryFields {Array}
97          * An array of strings of App data fields which are to be checked for the
98          * keyword(s). The array may contain none or any or all of the following:
99          *
100         * "title"
101         * "description"
102         * "tags"
103         *
104         * @param requestedFields {Map}
105         * See MApps.appQuery() documentation
106         *
107         * @return {Array}
108         * An array of objects containing info about apps which contain the word or
109         * words in the keyword string. The first results will be Apps that contain
110         * all words from the keyword string if there are any. Following that are
111         * all the results which contain one or many but not all of the words. There
112         * are no more restrictions on order.
113         */
114         keywordSearch : function(keywordString, queryFields, requestedFields, error)
115         {
116             var keywordArr;

```

```

117     var criteria;
118     var criteriaChild;
119     var searchResults;
120     var uidArr = [];
121     var queryResultsMap = {};
122     var queryResult;
123     var allWordResults = [];
124     var otherResults = [];
125     var curUID;
126
127     // Make sure there is at least 1 keyword given
128     if (keywordString === null || typeof keywordString === "undefined" ||
129         keywordString === "")
130     {
131         error.setCode(3);
132         error.setMessage("At least 1 keyword required for keyword search");
133         return error;
134     }
135
136     // Make sure all keyword searches are doing lowercase. ObjSearch stores
137     // all entries in lowercase
138     keywordString = keywordString.toLowerCase();
139
140     // The keyword string is space delimited, let's tokenize
141     keywordArr = keywordString.split(" ");
142
143     // Remove all stop words from the keyword array
144     keywordArr = qx.lang.Array.exclude(keywordArr,
145                                       aiagallery.dbif.MSearch.stopWordArr);
146
147     // Doing individual word queries
148     keywordArr.forEach(function(keyword)
149     {
150
151         criteria =
152         {
153             type : "element",
154             field : "word",
155             value : keyword
156         };
157
158         queryResult = rpcjs.dbif.Entity.query("aiagallery.dbif.ObjSearch",
159                                             criteria,
160                                             null);
161
162         // Collect all the single word queries in a map
163         queryResult.forEach(function(obj)
164         {
165             if (typeof queryResultsMap[obj.appId] ===
166                 "undefined")
167             {
168                 // First we create an array
169                 queryResultsMap[obj.appId] = [];
170             }
171             // Push this keyword into the array
172             queryResultsMap[obj.appId].push(keyword);
173         });
174     });
175
176     // All Apps which contained all keywords in the string should come first
177     for ( curUID in queryResultsMap)
178     {
179         // So separate those with all keywords and those without
180         if (queryResultsMap[curUID].length === keywordArr.length)
181         {
182             allWordResults.push(parseInt(curUID, 10));
183         }
184         else

```

```
185     {
186         otherResults.push(parseInt(curUID, 10));
187     }
188 }
189
190 // And place them in the proper order
191 uidArr = allWordResults.concat(otherResults);
192
193 // Finally, exchange array of UIDs for array of App Data objects using
194 // MApps.getAppListByList() and return that
195 return this.getAppListByList(uidArr, requestedFields);
196
197 }
198 }
199 });
```

Appendix 25

aiagallery.dbif.MTags

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  *
4  * License:
5  *   LGPL: http://www.gnu.org/licenses/lgpl.html
6  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7  */
8
9  qx.Mixin.define("aiagallery.dbif.MTags",
10 {
11   construct : function()
12   {
13     this.registerService("getCategoryTags",
14                          this.getCategoryTags,
15                          []);
16   },
17
18   members :
19   {
20     getCategoryTags : function()
21     {
22       var      categories;
23       var      criteria;
24       var      results;
25
26       // Create the criteria for a search of tags of type "category"
27       criteria =
28       {
29         type : "element",
30         field : "type",
31         value : "category"
32       };
33
34       // Issue a query for category tags
35       categories = rpcjs.dbif.Entity.query("aiagallery.dbif.ObjTags",
36                                          criteria,
37                                          [
38                                            {
39                                              type : "sort",
40                                              field : "value",
41                                              order : "asc"
42                                            }
43                                          ]
44                                          );
45
46       // They want only the tag value to be returned
47       results = categories.map(function() { return arguments[0].value; });
48
49       // Give 'em what they came for!

```

```
49     return results;
50   }
51 }
52 });
```

Appendix 26

aiagallery.dbif.MVisitors

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  *
4  * License:
5  *   LGPL: http://www.gnu.org/licenses/lgpl.html
6  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7  */
8
9  qx.Mixin.define("aiagallery.dbif.MVisitors",
10 {
11   construct : function()
12   {
13     this.registerService("addOrEditVisitor",
14       this.addOrEditVisitor,
15       [ "userId", "attributes" ]);
16
17     this.registerService("deleteVisitor",
18       this.deleteVisitor,
19       [ "userId" ]);
20
21     this.registerService("getVisitorList",
22       this.getVisitorList,
23       [ "bStringize" ]);
24
25     this.registerService("editProfile",
26       this.editProfile,
27       [ "profileParams" ]);
28   },
29
30   statics :
31   {
32     /**
33     * Exchange userId for user's displayName
34     *
35     * @param userId {String}
36     *   Visitor's userId
37     *
38     * @return {String}
39     *   Visitor's display name
40     */
41     _getDisplayName : function(userId, error)
42     {
43
44       var visitor = new aiagallery.dbif.ObjVisitors(userId);
45
46       if (qx.core.Environment.get("qx.debug"))
47       {
48         // Ensure that an error object is passed

```

```

49         qx.core.Assert.assertInstance(error,
50                                     rpcjs.rpc.error.Error,
51                                     "Need error object");
52     }
53
54     // Was our userId faulty in some way?
55     if (typeof visitor === "undefined" ||
56         visitor === null ||
57         visitor.getBrandNew())
58     {
59         // Yes, report the error
60         error.setCode(1);
61         error.setMessage("Unrecognized user ID in MVisitors");
62         return error;
63     }
64
65     // No problems, give them the display name
66     return visitor.getData().displayName;
67 },
68
69 /**
70  * Exchange user's displayName for userId
71  *
72  * @param displayName {String}
73  * Visitor's display name
74  *
75  * @return {String}
76  * Visitor's userId
77  */
78 _getVisitorId : function(displayName, error)
79 {
80
81     var owners = rpcjs.dbif.Entity.query(
82         "aiagallery.dbif.ObjVisitors",
83         {
84             type : "element",
85             field : "displayName",
86             value : displayName
87
88         },
89         // No resultCriteria. Only need a single result
90         null);
91
92     // Was there a problem with the query?
93     if (typeof owners[0] === "undefined" || owners[0] === null)
94     {
95         // Yes, report the error
96         error.setCode(2);
97         error.setMessage("Unrecognized display name: " + displayName);
98         return error;
99     }
100
101     // No problems, give them the ID
102     return owners[0].id;
103
104 }
105
106 },
107
108 members :
109 {
110     addOrEditVisitor : function(userId, attributes)
111     {
112         var         displayName;
113         var         permissions;
114         var         status;
115         var         statusIndex;
116         var         visitor;

```

```

117     var                visitorData;
118     var                ret;
119
120     displayName = attributes.displayName;
121     permissions = attributes.permissions;
122
123     // Get the status value. If the status string isn't found, we'll use
124     // "Active" when we set the database.
125     status = aiagallery.dbif.Constants.StatusToName.indexOf(status);
126
127     // Get the old visitor entry
128     visitor = new aiagallery.dbif.ObjVisitors(userId);
129     visitorData = visitor.getData();
130
131     // Remember whether it already existed.
132     ret = visitor.getBrandNew();
133
134     // Provide the new data
135     visitor.setData(
136     {
137         id            : userId,
138         displayName  : displayName || visitorData.displayName || "<>",
139         permissions : permissions || visitorData.permissions || [],
140         status       : status != -1 ? status : (visitorData.status || 2)
141     });
142
143     // Write the new data
144     visitor.put();
145
146     return ret;
147 },
148
149 deleteVisitor : function(userId)
150 {
151     var                visitor;
152
153     // Retrieve this visitor
154     visitor = new aiagallery.dbif.ObjVisitors(userId);
155
156     // See if this visitor exists.
157     if (visitor.getBrandNew())
158     {
159         // He doesn't. Let 'em know.
160         return false;
161     }
162
163     // Delete the visitor
164     visitor.removeSelf();
165
166     // We were successful
167     return true;
168 },
169
170 getVisitorList : function(bStringize)
171 {
172     var                visitor;
173     var                visitorList;
174
175     // For each visitor...
176     visitorList = rpcjs.dbif.Entity.query("aiagallery.dbif.ObjVisitors");
177
178     // If we were asked to stringize the values...
179     if (bStringize)
180     {
181         // ... then do so
182         for (visitor in visitorList)
183         {
184             var                thisGuy = visitorList[visitor];

```

```

185         thisGuy.permissions =
186             thisGuy.permissions ? thisGuy.permissions.join(", ") : "";
187         thisGuy.status =
188             [ "Banned", "Pending", "Active" ][thisGuy.status];
189     }
190 }
191
192 // We've built the whole list. Return it.
193 return visitorList;
194 },
195
196 editProfile : function(profileParams, error)
197 {
198     var         me;
199     var         meData;
200     var         whoami;
201     var         propertyTypes;
202     var         fields;
203     var         bValid = true;
204     var         validFields =
205         [
206             "displayName"
207         ];
208
209     // Find out who we are
210     whoami = this.getWhoAmI();
211
212     // Retrieve the current user's visitor object
213     me = new aiagallery.dbif.ObjVisitors(whoami.email);
214
215     // Get my object data
216     meData = me.getData();
217
218     // Get the field names for this entity type
219     propertyTypes = rpcjs.dbif.Entity.propertyTypes;
220     fields = propertyTypes["visitors"].fields;
221
222     // For each of the valid field names...
223     try
224     {
225         validFields.forEach(
226             function(fieldName)
227             {
228
229                 // Is this field being modified?
230                 if (typeof profileParams[fieldName] == "undefined")
231                 {
232                     // Nope. Nothing to do with this one.
233                     return;
234                 }
235
236                 // Ensure that the value being set is correct for the field
237                 switch(typeof profileParams[fieldName])
238                 {
239                     case "string":
240                         bValid = (fields[fieldName] == "String");
241                         break;
242
243                     case "number":
244                         bValid = (fields[fieldName] == "Number");
245                         break;
246
247                     default:
248                         bValid = false;
249                         break;
250                 }
251
252                 // Is the new profile parameter of the correct type?

```

```
253     if (! bValid)
254     {
255         // Nope.
256         error.setCode(1);
257         error.setMessage("Unexpected parameter type. " +
258             "Expected " + fields[fieldName] +
259             ", got " + typeof profileParams[fieldName]);
260         throw error;
261     }
262
263     // Assign the new value.
264     meData[fieldName] = profileParams[fieldName];
265 });
266 }
267 catch(error)
268 {
269     return error;
270 }
271
272 // Save the altered profile data
273 me.put();
274
275 // We need to return something. true is as good as anything else.
276 return true;
277 }
278 }
279 });
```

Appendix 27

aiagallery.dbif.MWhoAmI

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  *
4  * License:
5  *   LGPL: http://www.gnu.org/licenses/lgpl.html
6  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7  */
8
9  qx.Mixin.define("aiagallery.dbif.MWhoAmI",
10 {
11   construct : function()
12   {
13     this.registerService("whoAmI",
14                          this.whoAmI,
15                          []);
16   },
17
18   members :
19   {
20     /**
21     * Return the user's current login information and, optionally a logout
22     * URL.
23     */
24     whoAmI : function()
25     {
26       var          ret;
27       var          me;
28       var          whoami;
29
30       // Get the object indicating who we're logged in as
31       whoami = aiagallery.dbif.MDbifCommon.__whoami;
32
33       // Are they logged in?
34       if (! whoami)
35       {
36         // Nope.
37         return({
38           email      : "anonymous",
39           userId     : "",
40           isAdmin    : false,
41           logoutUrl  : "",
42           permissions : []
43         });
44       }
45
46       // Obtain this dude's Visitor record
47       me = new aiagallery.dbif.ObjVisitors(whoami.email);
48

```

```
49     // Create the return object, initialized to a clone of whoami. Add
50     // permissions from the database.
51     ret =
52     {
53         email      : String(whoami.email),
54         userId     : String(whoami.userId),
55         isAdmin    : whoami.isAdmin,
56         logoutUrl  : (qx.lang.Type.isArray(whoami.logoutUrl)
57                     ? whoami.logoutUrl
58                     : String(whoami.logoutUrl)),
59         permissions : me.getData().permissions.slice(0)
60     };
61
62     return ret;
63 }
64 }
65 });
```

Appendix 28

aiagallery.dbif.ObjAppData

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  *
4  * License:
5  *   LGPL: http://www.gnu.org/licenses/lgpl.html
6  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7  */
8
9  qx.Class.define("aiagallery.dbif.ObjAppData",
10 {
11   extend : aiagallery.dbif.Entity,
12
13   construct : function(uid)
14   {
15     // Pre-initialize the data
16     this.setData(
17       {
18         "tags"           : [],
19         "numLikes"      : 0,
20         "numDownloads"  : 0,
21         "numViewed"     : 0,
22         "numComments"   : 0,
23         "creationTime"  : aiagallery.dbif.MDbifCommon.currentTimestamp(),
24         "lastViewedTime" : null,
25         "numRootComments" : 0,
26         "numCurFlags"  : 0
27       });
28
29     // Call the superclass constructor
30     this.base(arguments, "apps", uid);
31   },
32
33   defer : function(clazz)
34   {
35     aiagallery.dbif.Entity.registerEntityType(clazz.classname, "apps");
36
37     var databaseProperties =
38     {
39       /** The owner of the application (id of Visitor object) */
40       "owner" : "String",
41
42       /** The application title */
43       "title" : "String",
44
45       /** Description of the application */
46       "description" : "String",
47
48       /** Image #1 (data URL) */

```

```

49     "image1" : "LongString",
50
51     /** Image #2 (data URL) */
52     "image2" : "LongString",
53
54     /** Image #3 (data URL) */
55     "image3" : "LongString",
56
57     /** Authorship chain */
58     "previousAuthors" : "StringArray",
59
60     /** Blob ids of source ZIP file (base64-encoded), newest first */
61     "source" : "StringArray",
62
63     /** Blob ids of executable APK file (base64-encoded), newest first */
64     "apk" : "StringArray",
65
66     /** File Name of Source File */
67     "sourceFileName" : "String",
68
69     /** Blob ids of executable APK file (base64-encoded), newest first */
70     "apk" : "StringArray",
71
72     /** File Name of APK File */
73     "apkFileName" : "String",
74
75     /** Tags assigned to this application */
76     "tags" : "StringArray",
77
78     /** Time the most recent Source ZIP file was uploaded */
79     "uploadTime" : "Date",
80
81     /** The date and time this App was first created */
82     "creationTime" : "Date",
83
84     /** The date this App was last viewed */
85     "lastViewedTime" : "Date",
86
87     /** Number of "likes" of this application */
88     "numLikes" : "Integer",
89
90     /** Number of downloads of this application */
91     "numDownloads" : "Integer",
92
93     /** Number of times this application was viewed */
94     "numViewed" : "Integer",
95
96     /** Number of root comments on this application */
97     "numRootComments" : "Integer",
98
99     /** Total number of comments on this application */
100    "numComments" : "Integer",
101
102    /** Status of this application (active, pending, banned) */
103    "status" : "Integer",
104
105    /** Total number of flags on this application */
106    "numCurFlags" : "Integer"
107};
108
109var canonicalize =
110{
111    "tags" :
112    {
113        // Property in which to store the canonical value. This will be a
114        // new string array.
115        prop : "tags_lc",
116

```


Appendix 29

aiagallery.dbif.ObjComments

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  * Copyright (c) 2011 Reed Spool
4  *
5  * License:
6  *   LGPL: http://www.gnu.org/licenses/lgpl.html
7  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
8  */
9
10 qx.Class.define("aiagallery.dbif.ObjComments",
11 {
12   extend : aiagallery.dbif.Entity,
13
14   // keyArr must be an array with 2 elements. The first element is
15   // an appId and the second a treeId
16   construct : function(keyArr)
17   {
18     // Pre-initialize the data
19     this.setData(
20     {
21       "timestamp"      : aiagallery.dbif.MDbifCommon.currentTimestamp(),
22       "numChildren"    : 0,
23       "app"            : keyArr[0],
24       "treeId"         : keyArr[1],
25       "status"         : aiagallery.dbif.Constants.Status.Active,
26       "numCurFlags"   : 0
27     });
28
29     // Use appId and treeId are a composite key
30     this.setEntityKeyProperty([ "app", "treeId" ]);
31
32     // Call the superclass constructor
33     this.base(arguments, "comments", keyArr);
34   },
35
36   defer : function(clazz)
37   {
38     aiagallery.dbif.Entity.registerEntityType(clazz.classname, "comments");
39
40     var databaseProperties =
41     {
42       /** UID of the AppData object which was commented upon */
43       "app" : "Key",
44
45       /**
46        * Hierarchy identifier to track comment threads.
47        *
48        * See http://www.tetilab.com/roberto/pgsql/postgres-trees.pdf for an

```


Appendix 30

aiagallery.dbif.ObjDownloads

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  *
4  * License:
5  *   LGPL: http://www.gnu.org/licenses/lgpl.html
6  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7  */
8
9  qx.Class.define("aiagallery.dbif.ObjDownloads",
10 {
11   extend : aiagallery.dbif.Entity,
12
13   construct : function(uid)
14   {
15     // Pre-initialize the data
16     this.setData(
17       {
18         "app"      : null,
19         "visitor"  : null,
20         "timestamp" : aiagallery.dbif.MDbifCommon.currentTimestamp()
21       });
22
23     // Call the superclass constructor
24     this.base(arguments, "downloads", uid);
25   },
26
27   defer : function(clazz)
28   {
29     aiagallery.dbif.Entity.registerEntityType(clazz.classname, "downloads");
30
31     var databaseProperties =
32     {
33       /** UID of the AppData object which was downloaded */
34       "app" : "Key",
35
36       /** Id of the Visitor who downloaded the application */
37       "visitor" : "String",
38
39       /** Time the download was initiated */
40       "timestamp" : "Date"
41     };
42
43     // Register our property types
44     aiagallery.dbif.Entity.registerPropertyTypes("downloads",
45                                               databaseProperties);
46   }
47 });

```

Appendix 31

aiagallery.dbif.ObjFlags

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  *
4  * License:
5  *   LGPL: http://www.gnu.org/licenses/lgpl.html
6  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7  */
8
9  qx.Class.define("aiagallery.dbif.ObjFlags",
10 {
11   extend : aiagallery.dbif.Entity,
12
13   construct : function(uid)
14   {
15     // Pre-initialize the data
16     this.setData(
17       {
18         "type"      : null,
19         "app"       : null,
20         "comment"   : null,
21         "visitor"   : null,
22         "timestamp" : aiagallery.dbif.MDbifCommon.currentTimestamp(),
23         "explanation" : null
24       });
25
26     // Call the superclass constructor
27     this.base(arguments, "flags", uid);
28   },
29
30   defer : function(clazz)
31   {
32     aiagallery.dbif.Entity.registerEntityType(clazz.classname, "flags");
33
34     var databaseProperties =
35     {
36       /** Type of flag ("App" = application, "Comment" = comment) */
37       "type" : "String",
38
39       /** UID of the AppData object which was flagged */
40       "app" : "Key",
41
42       /** Tree ID of the Comment object which was flagged */
43       "comment" : "String",
44
45       /** Id of the Visitor who flagged the application or comment */
46       "visitor" : "String",
47
48       /** Time the like occurred */

```


Appendix 32

aiagallery.dbif.ObjLikes

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  *
4  * License:
5  *   LGPL: http://www.gnu.org/licenses/lgpl.html
6  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7  */
8
9  qx.Class.define("aiagallery.dbif.ObjLikes",
10 {
11   extend : aiagallery.dbif.Entity,
12
13   construct : function(uid)
14   {
15     // Pre-initialize the data
16     this.setData(
17       {
18         "app"      : null,
19         "visitor"  : null,
20         "timestamp" : aiagallery.dbif.MDbifCommon.currentTimestamp()
21       });
22
23     // Call the superclass constructor
24     this.base(arguments, "likes", uid);
25   },
26
27   defer : function(clazz)
28   {
29     aiagallery.dbif.Entity.registerEntityType(clazz.classname, "likes");
30
31     var databaseProperties =
32     {
33       /** UID of the AppData object which was liked */
34       "app" : "Key",
35
36       /** Id of the Visitor who liked the application */
37       "visitor" : "String",
38
39       /** Time the like occurred */
40       "timestamp" : "Date"
41     };
42
43     // Register our property types
44     aiagallery.dbif.Entity.registerPropertyTypes("likes",
45                                               databaseProperties);
46   }
47 });

```

Appendix 33

aiagallery.dbif.ObjSearch

```

1  /**
2  * Copyright (c) 2011 Reed Spool
3  *
4  * License:
5  *   LGPL: http://www.gnu.org/licenses/lgpl.html
6  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7  */
8
9  qx.Class.define("aiagallery.dbif.ObjSearch",
10 {
11   extend : aiagallery.dbif.Entity,
12
13   construct : function(keyArr)
14   {
15     // All words stored in ObjSearch must be lower case. There is no case
16     // insensitive option for querieing.
17     keyArr[0] = keyArr[0].toLowerCase();
18
19     // Need all data for the key regardless, so might as well store it
20     this.setData(
21       {
22         "word"      : keyArr[0],
23         "appId"    : keyArr[1],
24         "appField" : keyArr[2]
25       });
26
27     // Use the composite of "word", "appId", and "appField" properties as
28     // the entity key
29     this.setEntityKeyProperty(["word", "appId", "appField"]);
30
31     // Call the superclass constructor
32     this.base(arguments, "search", keyArr);
33   },
34
35   defer : function(clazz)
36   {
37     aiagallery.dbif.Entity.registerEntityType(clazz.classname, "search");
38
39     var databaseProperties =
40     {
41       /** The word value */
42       "word" : "String",
43
44       /** The App in which this word appears */
45       "appId" : "Key",
46
47       /** The App data field within which this word appeared */
48       "appField" : "String"

```

```
49     };
50
51     // Register our property types
52     aiagallery.dbif.Entity.registerPropertyTypes("search",
53                                               databaseProperties,
54                                               ["word", "appId", "appField"]);
55   }
56 });
```

Appendix 34

aiagallery.dbif.ObjTags

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  *
4  * License:
5  *   LGPL: http://www.gnu.org/licenses/lgpl.html
6  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7  */
8
9  qx.Class.define("aiagallery.dbif.ObjTags",
10 {
11   extend : aiagallery.dbif.Entity,
12
13   construct : function(value)
14   {
15     // Pre-initialize the data
16     this.setData(
17       {
18         "value" : null,
19         "type"  : "normal",
20         "count" : 1
21       });
22
23     // Use the canonicalized "value" property as the entity key
24     this.setEntityKeyProperty("value_lc");
25
26     // Call the superclass constructor
27     this.base(arguments, "tags", value);
28   },
29
30   defer : function(clazz)
31   {
32     aiagallery.dbif.Entity.registerEntityType(clazz.classname, "tags");
33
34     var databaseProperties =
35     {
36       /** The tag value */
37       "value" : "String",
38
39       /** The tag type (category, invisible [e.g. "featured"], normal) */
40       "type"  : "String",
41
42       /** The number of uses of this tag value */
43       "count" : "Integer"
44     };
45
46     var canonicalize =
47     {
48       "value" :

```


Appendix 35

aiagallery.dbif.ObjVisitors

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  *
4  * License:
5  *   LGPL: http://www.gnu.org/licenses/lgpl.html
6  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7  */
8
9  qx.Class.define("aiagallery.dbif.ObjVisitors",
10 {
11   extend : aiagallery.dbif.Entity,
12
13   construct : function(id)
14   {
15     var initialData;
16
17     // Pre-initialize the data
18     this.setData(
19       {
20         "id"           : id,
21         "displayName"  : null,
22         "permissions"  : [],
23         "status"       : aiagallery.dbif.Constants.Status.Active,
24         "recentSearches" : [],
25         "recentViews"  : []
26       });
27
28     // Use the "id" property as the entity key
29     this.setEntityKeyProperty("id");
30
31     // Call the superclass constructor
32     this.base(arguments, "visitors", id);
33   },
34
35   defer : function(clazz)
36   {
37     aiagallery.dbif.Entity.registerEntityType(clazz.classname, "visitors");
38
39     var databaseProperties =
40     {
41       /** The user's email address */
42       "id" : "String",
43
44       /** How the user's name is displayed in the gallery */
45       "displayName" : "String",
46
47       /** A list of explicit permissions assigned to this user */
48       "permissions" : "StringArray",

```


Appendix 36

aiagallery.dbif.MSimData

```

1  /**
2  * Copyright (c) 2011 Derrell Lipman
3  *
4  * License:
5  *   LGPL: http://www.gnu.org/licenses/lgpl.html
6  *   EPL : http://www.eclipse.org/org/documents/epl-v10.php
7  */
8
9  /**
10 * This is the *truncated* simulation database
11 */
12
13 qx.Mixin.define("aiagallery.dbif.MSimData",
14 {
15   statics :
16   {
17     Db :
18     {
19       "visitors" :
20       {
21         "jane@uphill.org" :
22         {
23           "id" : "jane@uphill.org",
24           "displayName" : "Jane Doe",
25           "permissions" : [],
26           "status" : 2,
27           "recentSearches" : [],
28           "recentViews" : []
29         },
30         "billy@thekid.edu" :
31         {
32           "id" : "billy@thekid.edu",
33           "displayName" : "Billy The Kid",
34           "permissions" : [],
35           "status" : 2,
36           "recentSearches" : [],
37           "recentViews" : []
38         },
39         "joe@blow.com" :
40         {
41           "id" : "joe@blow.com",
42           "displayName" : "Joe Blow",
43           "permissions" : ["addOrEditApp"],
44           "status" : 2,
45           "recentSearches" : [],
46           "recentViews" : []
47         }
48       },

```

```

49     "tags" :
50     {
51         "*featured*" :
52         {
53             "value" : "*Featured*",
54             "value_lc" : "*featured*",
55             "type" : "special",
56             "count" : 8
57         },
58         "games" :
59         {
60             "value" : "Games",
61             "value_lc" : "games",
62             "type" : "category",
63             "count" : 1
64         },
65         "educational" :
66         {
67             "value" : "Educational",
68             "value_lc" : "educational",
69             "type" : "category",
70             "count" : 1
71         },
72         "development" :
73         {
74             "value" : "Development",
75             "value_lc" : "development",
76             "type" : "category",
77             "count" : 1
78         },
79         "graphics" :
80         {
81             "value" : "Graphics",
82             "value_lc" : "graphics",
83             "type" : "category",
84             "count" : 1
85         },
86         "internet" :
87         {
88             "value" : "Internet",
89             "value_lc" : "internet",
90             "type" : "category",
91             "count" : 1
92         },
93         "multimedia" :
94         {
95             "value" : "Multimedia",
96             "value_lc" : "multimedia",
97             "type" : "category",
98             "count" : 1
99         },
100        "bioscience" :
101        {
102            "value" : "Bioscience",
103            "value_lc" : "bioscience",
104            "type" : "normal",
105            "count" : 1
106        },
107        "communications" :
108        {
109            "value" : "Communications",
110            "value_lc" : "communications",
111            "type" : "normal",
112            "count" : 1
113        },
114        "computers" :
115        {
116            "value" : "Computers",

```

```

117         "value_lc" : "computers",
118         "type" : "normal",
119         "count" : 1
120     },
121     "earth science" :
122     {
123         "value" : "Earth Science",
124         "value_lc" : "earth science",
125         "type" : "normal",
126         "count" : 1
127     },
128     "energy" :
129     {
130         "value" : "Energy",
131         "value_lc" : "energy",
132         "type" : "normal",
133         "count" : 1
134     },
135     "mathematics" :
136     {
137         "value" : "Mathematics",
138         "value_lc" : "mathematics",
139         "type" : "normal",
140         "count" : 1
141     },
142     "oceanography" :
143     {
144         "value" : "Oceanography",
145         "value_lc" : "oceanography",
146         "type" : "normal",
147         "count" : 1
148     },
149     "physical sciences" :
150     {
151         "value" : "Physical Sciences",
152         "value_lc" : "physical sciences",
153         "type" : "normal",
154         "count" : 1
155     },
156     "space" :
157     {
158         "value" : "Space",
159         "value_lc" : "space",
160         "type" : "normal",
161         "count" : 1
162     },
163     "transportation" :
164     {
165         "value" : "Transportation",
166         "value_lc" : "transportation",
167         "type" : "normal",
168         "count" : 1
169     },
170     "word games" :
171     {
172         "value" : "Word Games",
173         "value_lc" : "word games",
174         "type" : "normal",
175         "count" : 1
176     },
177     "k-12" :
178     {
179         "value" : "K-12",
180         "value_lc" : "k-12",
181         "type" : "normal",
182         "count" : 1
183     }
184 },

```

```

185     "apps" :
186     {
187         "100" :
188         {
189             "uid" : 100,
190             "owner" : "jane@uphill.org",
191             "title" : "address_book-new",
192             "description" : "The description of address_book-new",
193             "image1" : "data:image/gif;base64,iVBORwOKGgoAAAANSUHE...",
194             "image2" : null,
195             "image3" : null,
196             "previousAuthors" : [],
197             "source" : "var x = 'Hello world';",
198             "apk" : null,
199             "sourceFileName" : "HelloWorld.zip",
200             "apkFileName" : null,
201             "tags" : ["Computers", "Educational", "*Featured*"],
202             "tags_lc" : ["computers", "educational", "*featured*"],
203             "uploadTime" : 1303434151234,
204             "numLikes" : 0,
205             "numDownloads" : 0,
206             "numViewed" : 0,
207             "numComments" : 0,
208             "status" : 2
209         },
210         "101" :
211         {
212             "uid" : 101,
213             "owner" : "joe@blow.com",
214             "title" : "application_exit",
215             "description" : "The description of application_exit",
216             "image1" : "data:image/gif;base64,iVBORwOKGgoAAAANSUHE...",
217             "image2" : null,
218             "image3" : null,
219             "previousAuthors" : [],
220             "source" : "var x = 'Hello world';",
221             "apk" : null,
222             "sourceFileName" : "HelloWorld.zip",
223             "apkFileName" : null,
224             "tags" : ["Oceanography", "Educational"],
225             "tags_lc" : ["oceanography", "educational"],
226             "uploadTime" : 1303434151234,
227             "numLikes" : 0,
228             "numDownloads" : 0,
229             "numViewed" : 0,
230             "numComments" : 0,
231             "status" : 2
232         }
233     },
234     "downloads" :
235     {
236
237     },
238     "comments" :
239     {
240
241     },
242     "likes" :
243     {
244
245     },
246     "flags" :
247     {
248
249     },
250     "search" :
251     {
252

```

```
253     }  
254   }  
255 }  
256 });
```
