Scripting Web Service Prototypes

by Christopher Vincent, IBM Systems Group

As web services become increasingly sophisticated, their practitioners will require skills spanning transaction processing, database management, middleware integration, and asynchronous messaging. IBM Lightweight Services (LWS), an experimental hosting environment, aims to support rapid prototyping of complex services while insulating developers from advanced issues in multi-threading, transactions, and resource locking. To achieve this we adapt a high-level, event-driven, single-threaded scripting environment to server-side application hosting. Developers may use this freely available environment to create robust web services that store persistent data, consume other services, and integrate with existing middleware. Lightweight services are invoked by standard HTTP SOAP clients, and may in turn invoke other web services using WSDL.

Here *prototyping* means creating fully functional web services, but prioritizing ease of development over run-time performance. LWS services are transactional, maintain state across server failures and, most importantly, interoperate with standard web service clients and servers. User-friendly development tools bridge the gap between SOAP data types and loosely typed scripts, allowing web service operations to evolve incrementally over the course of a debugging session. Script-based services are described by standard WSDL, and clients need not differentiate between LWS services or their more conventional counterparts. Once program logic, data requirements, and external interfaces have stabilized, a developer may choose to re-implement a prototype, targeting a more optimized run-time environment.

We'll use two concrete examples where script-based prototyping provides the developer a streamlined path to functional web services. These scripts are not only expressed in small amounts of source code, they may be installed, run, and re-installed very quickly due to a flexible persistent storage mechanism. We'll then look at how we've applied the "heavyweight" concepts of transactional persistence to this simplified scripting environment.

WHY SCRIPT?

We consider JavaScript a *lightweight language* because its

Creating and evolving robust applications,

ESPECIALLY THOSE WHICH
LEVERAGE LESS-RELIABLE
MIDDLEWARE SUCH AS
INSTANT MESSAGING,
PRESENTS DEVELOPERS
WITH SOME SIGNIFICANT
CHALLENGES.

relative simplicity makes it easy to learn and use. It is often associated with DHTML, where it is used as an event-driven scripting language in web pages. LWS uses JavaScript as a high-level, dynamic language for developing server-side processes with access to web services and middleware systems such as instant messaging. The hosting environment provides persistence and trans-

action management fine-tuned for scripting languages, minimizing the impact of these robustness features on script's expressive programming style. Note that while we are focusing on JavaScript as an example, similar tools could be created for other languages such as interpreted Python and Scheme.

WEB SERVICES SCENARIOS

We'll start by looking at two different types of web services scenarios. *Data-centric* web services could be defined as straightforward, request-driven wrappers for relational databases. This type of service would typically be implemented as a servlet and/or Enterprise JavaBean (EJB) in a J2EE environment. As an example, we'll prototype a simple phone book service that maps names to phone numbers, providing get and set operations for accessing the data. *Process-oriented* web services perform complex operations (e.g. network requests) that span or exist outside of individual user requests. Implementing these services in languages like Java typically requires more advanced programming techniques such as multi-threading and resource locking. To illustrate this scenario, we'll use a well-known temperature service to keep track of the average temperature across a configurable set of ZIP codes. This application dispatches multiple web service requests simultaneously and on a schedule, incorporating new results as they become available.

It won't be necessary to fully understand the following JavaScript code and programming APIs. We'll step through the (relatively succinct) service implementations with an eye to their high-level operation.

PHONE BOOK SERVICE

In a production implementation of the phone book service, we would define a database table for storing names and their corresponding phone numbers, most likely with an additional "owner" column to support multiple phone books. For the purposes of prototyping, we can skip time-intensive database administration tasks by storing the data in a JavaScript object. The LWS hosting environment automatically stores the state of a JavaScript program (including its top-level variables) in an existing, general-purpose database. This is certainly less efficient at run-time than using a special-purpose phone book database, but performance is more than adequate for prototyping and a data set of 100-200 phone book entries.

The phone book script stores all of its data in a single generic object. Because of JavaScript's *expando properties*, we can use this object like a hashtable. The numbers object is created when the script is initialized, and the programmer uses the development tools to specify the `setPhoneNumber` and `getPhoneNumber` functions as web service operations. The hosting environment creates and manages a new database transaction when one of the operations is invoked, supporting the same level of robustness as with explicit database access. If the underlying system fails in the middle of manipulating the numbers variable, the entire JavaScript program rolls back, discarding any changes made by the current operation.

```
var numbers = new Object();

function setPhoneNumber(name, number) {
    numbers[name] = number;
}

function getPhoneNumber(name) {
    return numbers[name];
}
```

Each *instance* of a service corresponds to a JavaScript program state stored in the LWS database, and represents its own web services endpoint. To support multiple phone books, e.g. for different users, the developer (or another program) would create several instances of the service. Each client would then be assigned the web services endpoint corresponding to their phone book.

MEAN TEMPERATURE SERVICE

The mean temperature service is configured with a list of ZIP codes, and then periodically fetches the corresponding temperatures from a simple lookup service. The average temperature is computed on-demand to serve incoming requests. We use this scenario to demonstrate a web services provider that also invokes another service as a client. To further complicate matters, these web service client operations are not performed in direct response to incoming requests. Rather, the service queries the temperature data on a schedule, and with no regard to incoming requests for the average value.

LWS provides an *asynchronous* web services client API, such that no JavaScript call will block on a network operation. Non-blocking APIs allow us to handle both the web service client operations and incoming requests in the same single-threaded program. The script developer specifies JavaScript functions as event-handlers for processing results as they become available. In addition to being single-threaded, incoming requests and event-handlers are *serialized*, executing in their entirety before the next event is processed. Enforcing this programming style may seem odd to accomplished users of multi-threading and synchronization, but is a familiar scenario for many script developers, especially those embedding script in web pages.

Since this script is more complicated than the first example, we'll break it up into several parts. First, consider the code for calling the temperature web service and storing the results. A top-level variable `temperatures` is initialized with a generic object; this will act as a hashtable mapping ZIP code to temperature. We then create an object representing the temperature lookup service we are going to call. Initialized with a WSDL document, this convenience object will aid in creating web service requests. We could have specified a full URL for the WSDL, but here we assume it is included as one of our project files.

```
var temperatures = new Object();
var service = WebServices.WSDL.createService(
    "TemperatureService.wsdl");
```

The asynchronous nature of the web services client API means we'll need both code to initiate a request and an event-handler to process the result. The `getTemperature` function accepts a ZIP code, creates a request using the temperature service defined above, and specifies a single parameter using the JavaScript "dot" notation for accessing object properties. After setting the event-handler to the `onGetTemperatureComplete` function, the

web services call is initiated and `getTemperature` returns immediately.

The `onGetTemperatureComplete` function starts by checking the standard response parameter for success. It then uses the response object to obtain the ZIP code used in the original request (note that several “`getTemp`” calls may be pending at any time, and that their results may arrive in any order). The `temperatures` hashtable is then updated with the most recent value for the requested location.

```
function getTemperature(zipCode) {
    var request = service.createRequest("getTemp");
    request.body.zipcode = zipCode;
    request.onComplete = onGetTemperatureComplete;
    request.call();
}

function onGetTemperatureComplete(response) {
    if (!response.failed) {
        var zipCode = response.request.body.zipcode;
        temperatures[zipCode] = response.result.value;
    }
}
```

Now we’ll implement some top-level initialization code to schedule the temperature requests. We start by parsing a comma-separated list of ZIP codes, supplied as a configuration parameter to this instance of the mean temperature service. For each specified location, we make the first `getTemperature` call immediately, then use a special timer object to schedule periodic calls to the same function at a configurable interval. Alternatively, a developer might choose to stagger these calls or adjust scheduling on-the-fly as results come in.

```
var zipCodes = Host.configuration.zipCodes.split(",");
for (var idx = 0; idx < zipCodes.length; idx++) {
    var zipCode = zipCodes[idx];
    getTemperature(zipCode);
    var timer = new Host.Timer(getTemperature, [zipCode]);
    timer.start(Host.configuration.period, true);
}
```

Finally, we’ll implement the single operation exposed by this web service, `getMeanTemperature`. This function simply accumulates any results that have been added to the `temperatures` hashtable, returning the most recent average temperature. The developer would instruct the hosting environment to tag the return value as SOAP floating point, allowing infinity to denote the “no results

available” condition.

```
function getMeanTemperature() {
    var sum = 0, count = 0;
    for (var zipCode in temperatures) {
        sum += temperatures[zipCode];
        count++;
    }
    return sum / count;
}
```

While the implementation of this service closely resembles the ephemeral, in-memory scripts found embedded in web pages, it benefits from the robustness features of a transactional database. The hosting environment not only manages the persistence and consistency of the `temperatures` hashtable, it resumes the schedule of `getTemperature` calls after a server failure. Pending SOAP HTTP requests will be discarded on server restart, but the next scheduled call will still occur.

PERSISTENCE AND TRANSACTIONS

IBM Lightweight Services is largely an experiment in minimizing what a developer must learn to benefit from a persistent, transactional programming environment. We’ve already touched on how script programs are mapped to persistent storage: the program state for each instance of a service is automatically stored in a general-purpose database. Limiting ourselves to single-threaded scripts with serialized event-handlers simplifies this persistence scheme; the hosting environment need only fetch a program’s state when an event needs processing. This arrangement, combined with asynchronous APIs, also protects developers from the resource locking issues (e.g. deadlock) common in database programming.

Since event-handlers now represent transaction boundaries, script developers must recognize that when a handler function throws an exception or otherwise fails to complete, the entire program state is rolled back. For data-centric, request-driven services, this basic awareness of transactions is enough to vastly simplify an implementation. The developer no longer needs to ensure consistency between program variables in the face of exceptions. For complex, process-oriented services, we chose to address two additional issues: managing non-transactional resources and server restarts.

While rolling back changes to JavaScript variables is relatively straight-forward, services may also, for example, make SOAP HTTP requests or send instant messages. How should the rollback of a script’s program state affect irrevocable operations on these non-transactional resources?

Our approach to simplifying this classic problem is to establish the following contract with the programmer:

1. Operations on non-transactional resources (e.g. sending messages over an instant messaging connection) are queued by the hosting environment until the current event-handler returns.
2. If the current event-handler throws an exception or the server fails, the state of the JavaScript program is rolled back and the queued operations are discarded.
3. If the current event handler returns successfully, the hosting environment attempts to perform the queued operations after the current transaction commits. Errors may be detected using additional event-handlers.

While the rules above have a relatively minor impact on most scripts, server restarts present more of a challenge. If a program stores an object representing a connection to an instant messaging server, what should the hosting environment do following a server failure? Attempt to restore the connection? Trigger an error condition? Both these approaches have drawbacks, and we formulated a compromise that aims to minimize programmer effort. By default, connections to non-transactional resources are not restored, and a script is not notified of a server restart. LWS does provide developers an optional event for handling server restarts, allowing connections to be reopened immediately when appropriate. In addition, our middleware APIs are designed such that connections can be re-established with a single line of script, automatically recalling the specifications of the previous connection.

```
// Create instant messaging session.
var session = new Sametime.Session();
...
session.connect("host.ibm.com", "user", "password");

// Handle server restarts.
Host.onResume = onResume;

function onResume() {
    session.connect();
}
```

FIG 1

Assigning SOAP types to Web service operations

In this environment, developers of simple services can be successful with only the most basic understanding of transactions and persistence. Only when more complicated operations are required does a developer need to recognize the programming contract above and explicitly handle server restarts.

DEVELOPMENT TOOLS

A vital part of streamlining developer experience is a rich integrated development environment (IDE). Services are a combination of script code and metadata, created and packaged with the LWS plug-in for Eclipse. As mentioned above, development tasks such as exporting web service operations and assigning SOAP types (JavaScript does not enforce types by default) are performed through user interfaces rather than with code.

The most important feature of our IDE is a “one-click” deploy and debug operation for interacting with remote LWS servers. With a single keystroke or mouse click, the developer may remove a previous instance of the service, repack with the latest source code, install the updated package, and create a new instance for debugging. Once a service has been instantiated, the developer has immediate access to a command-line for interacting with the script’s program state, as well as per-instance logs for viewing debugging output and error conditions. The command-line interpreter is especially useful for testing operations directly, before creating a corresponding web services client.

FUTURE WORK

We anticipate that much of the future work on lightweight services will relate to the management of persistent data. In theory, dynamic JavaScript can create arbitrary

trary data structures, and each instance of a script may have different storage requirements. In practice, we'd like our storage mechanism to exploit the similarities between multiple instances of the same service, allowing us to perform relational queries where appropriate. The ability to dynamically allocate persistent variables is a valuable development feature, but seldom occurs once a service has been deployed. Consider the phone book example, where several users have created their own instances of the service with their own hashtables. In an implementation based on a relational database, we could perform a query for all phone books containing a certain name or number. In a flexible but loosely structured lightweight services scheme, there is no way to perform the same search efficiently. A hybrid storage scheme, especially with hints from the programmer, could combine the advantages of these two worlds.

WHAT WE'VE DONE AND WHY

With a focus on developer experience, we've been able to adapt a simple scripting environment to the domain of robust, server-side applications. We tailored a new persistent storage mechanism to the event-driven scripting model popularized by the Web, shielding programmers from some of the more advanced issues in transaction processing.

We implemented a simple example in five lines of source code, with the real savings being in install time. Configuring the same phone book service in most environments would require manually creating (and managing) a relational database table. A more complex example,

the mean temperature service, would typically require multi-threading, synchronization, and an understanding of database locking issues. Leveraging a simplified hosting environment, we created a robust prototype in about twenty lines of JavaScript. We hope our approach can help script programmers and veteran software engineers alike move quickly from idea to implementation.

MORE INFORMATION

This column describes a set of tools and technologies developed by IBM. The software is available for download with a free evaluation license from IBM's alphaWorks emerging technology site. The IBM Lightweight Services site contains a number of programming examples and tutorials illustrating this style of web services development.

- IBM Lightweight Services on IBM alphaWorks:
<http://alphaworks.ibm.com/tech/lws>
- ECMAScript (standardized JavaScript):
<http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM>
- Temperature service from the example:
<http://www.xmethods.net/sd/TemperatureService.wsdl>
- The LWS development environment extends the Eclipse platform: <http://eclipse.org/>

CHRISTOPHER VINCENT (crv@us.ibm.com) is a software engineer on the Internet Technology Team, IBM Systems Group. His research interests include rapid application development, dynamic programming languages, instant messaging, and publish/subscribe infrastructure.

© Copyright is held by author/owner.

acm
queue
tomorrow's computing today

**Coming in April
Embedded Systems**