;login:

THE MAGAZINE OF USENIX & SAGE
April 2004 • volume 29 • number 2

inside:

USENIX
The Advanced Computing Systems Association

practical perl

by Adam Turoff

Adam is a consultant who specializes in using Perl to manage big data. He is a long-time Perl Monger, a technical editor for *The Perl Review*, and a frequent presenter at Perl conferences.

ziggy@panix.com

An Introduction to Web Services

Web services are easily the most hyped topic in software development since the big "Push Technology" craze of the mid-1990s. Proponents believe that Web services represent the future of software development. Detractors believe that Web services are nothing more than an ever-expanding orb of complexity.

In fact, the concept of Web services is a very broad topic with ill-defined edges. There's a grain of truth in each of these perspectives.

At the low end, Web services are just a new form of remote procedure calls (RPC). They generally use HTTP as a transport layer, coupled with some form of XML to encode messages and data. This enables simple CGI programs, mod_perl handlers, and stand-alone Web servers to implement a service. By using HTTP and XML, Web services make it easier to focus on writing an application other than on the low-level details of handling sockets, implementing a protocol, or debugging client-server transactions.

At the high end, Web services are a loosely federated series of specifications that describe business processes. These specifications are expressed in XML-based languages that describe how to execute a business process, such as processing a purchase order or provisioning new hardware. Other activities that can be described under the Web services rubric include transactional reliability, process coordination, and security.

With these two different perspectives on Web services, it is easy to get confused. On the one hand, Web services are just a new way to build software using ubiquitous standards such as HTTP and XML. On the other hand, Web services are a very high-level description of how to automate business, where the implementation details are mostly irrelevant. In this article, I focus on the first meaning of "Web service" – the low-level reinvention of RPC using HTTP and XML.

Building Services

Let's consider a practical example. Suppose you wanted to work with your friends to create a dictionary. You need to collect definitions and look up terms you have defined already. You could start with a server that supports the dict protocol, except that that protocol does not allow you to create new definitions.

Where do you begin? You could start by creating a new protocol, "newdict," that supports the features you need. Then you would need to create (and test!) new software for your client and server programs. Frankly, that path leads to a lot of work and not a lot of value. Your mission here is to create a new dictionary, not rediscover the arcana of dealing with sockets and writing protocols.

Another alternative is to use a preexisting RPC library. Instead of developing a protocol, using RPC allows you to create one service that responds to multiple procedure calls, such as add_definition and get_definition.

This is where Web services become interesting. Low-level RPC libraries require packing and unpacking data (bytes) to send messages over the network. On the other hand, Web services transfer data between clients and servers using XML. This helps, because Perl's support for low-level RPC libraries has always been weak, but its support for XML and text has always been strong. Additionally, Web services are a language-neutral RPC mechanism. It is easy to write a connect for Perl clients to Java and .Net Web services, or vice versa.

There are three main ways to write a Web service: REST, XML-RPC, and SOAP. REST is not a Web service technology, but a description of how the Web works, and a set of conventions for how to write well-behaved programs that respect those principles. Chances are good that if you've ever written a CGI script, you've written a REST service.

XML-RPC is a very simple protocol for creating Web services. It is an XML description of how to invoke a remote procedure, and an XML description of the corresponding result. XML-RPC is a simplification of SOAP, a richer Web services protocol that can be used for simple RPC or for more complex interactions involving message-based communications.

Perl supports all three kinds of Web services. To use REST Web services, the only thing that is absolutely required is a Web client library, like LWP. The RPC::XML and XMLRPC::Lite modules both support XML-RPC, and the SOAP::Lite module supports the SOAP protocol. (XMLRPC::Lite can be found in the SOAP::Lite distribution.)

Publishing a Dictionary Authoring Service

One of the easiest ways to start with Web services is to add an XML-RPC interface onto an existing program. Here are the guts of a newdict program to look up definitions and define new terms. It has two main actions, add_definition and get_definition, which insert data into a database and query it:

```
#!/usr/bin/perl

use strict;
use warnings;
use DBI;

my $dbh = DBI->connect("dbi:SQLite:dbname=dictionary.db");

my $insert_stmt = $dbh->prepare("INSERT INTO dictionary (term, definition)  VALUES (?, ?)");

my $select_stmt = $dbh->prepare("SELECT definition FROM dictionary WHERE term=?");

sub add_definition {
      my $term = shift;
      my $definition = shift;

      return $insert_stmt->execute($term, $definition);
}

sub get_definition {
      my $term = shift;

      $select_stmt->execute($term);

      my $rows = $select_stmt->fetchall_arrayref();

      ## Transform a list of lists of strings into a list of strings
      my @definitions = map {$_->[0]} @$rows;

      ## Return a structure containing the term, and all definitions found
      return {term => $term,
          definitions => \@definitions};
}
```

That’s it! Like many services (DNS, dict, whois), this newdict service is merely an interface to a data store. In this case, the data store is a simple SQLite database. The newdict service could grow into something more complex, but this is enough to get started.

The rest of the server side of this application – responding to requests, and running as a daemon – is handled through a Web service library. To publish this small program as an XML-RPC Web service, all that’s necessary is to add some glue code using the RPC::XML module from CPAN:

```
use RPC::XML::Server;

my $server = new RPC::XML::Server(port => 10000);

## Add a method to add new definitions
$server->add_method({
          name => "newdict.add_definition",
          code => \&add_definition,
          signature=>['int string string'],
      });

## Add a method to add search for definitions
$server->add_method({
          name => "newdict.get_definition",
              code => \&get_definition,
              signature=>['struct string'],
      });
```

```
## Launch the service
## (This statement never returns)
$server->server_loop();
```

The first step is to create a RPC::XML::Server object and specify which server port to use (10000 in this example). Next, each operation that this XML-RPC server will support must be added into the server object with a call to add_method. Each method is defined with a name (like newdict.add_definition), a reference to the subroutine to call, and a description of the inputs and outputs to that method.

The third part of the XML-RPC method definition, the "signature," is a textual description of data types for the result and input values for a method. XML-RPC defines eight basic datatypes that can be used in XML-RPC messages: integers, floating point numbers, Boolean values, date/time values, strings, and also binary structures, arrays, and hashes (called struct in XML-RPC).

With XML-RPC, methods return a single value, and the first datatype in a method signature is the result. The other values in the signature describe the types of the input parameters. In the example above, the add_definition method returns an integer value and takes two strings, a term to be defined and its definition. The get_definition takes a single string as input (the term to be defined) and returns a hash containing both the term and its definitions.

Using RPC::XML::Server, the final bit of glue is the call to $server->server_loop(). This starts a stand-alone Web server (using HTTP::Daemon) and waits for requests to arrive. Each XML-RPC request will be an HTTP request containing a bit of XML that describes what method to call and the parameters to pass to that method. For each message received, the RPC::XML::Server object will parse the XML message, identify what method is requested, pass the arguments, and receive a result. The server object will then encode the result in XML and send back to the client program, where its XML-RPC library will perform the same deserialization.

We're almost done. The RPC::XML library calls RPC methods with an initial parameter containing information about the XML-RPC request received from the client. This parameter needs to be captured in each of the published methods:

```
sub add_definition {
    my $xmlrpc = shift;
    my $term = shift;
    my $definition = shift;

    ##...
}

sub get_definition {
    my $xmlrpc = shift;
    my $term = shift;

    ##....
}
```

And now we're done. By adding a few lines of glue code and making two small changes to the original newdict program, we now have a newdict Web service.

Using the Dictionary Authoring Service

Now that we have a newdict service, anyone with an XML-RPC library can communicate with it. All that is necessary is a URL to find this XML-RPC server (*http://localhost:10000/* in this example), the names of the methods to call, and an understanding of the signatures for those methods.

To connect to this service, we can use the RPC::XML::Client module that also comes with RPC::XML distribution:

```
#!/usr/bin/perl

use strict;
use warnings;

use RPC::XML::Client;
```

```
my $client = new RPC::XML::Client("http://localhost:10000");

$client->simple_request('newdict.add_definition',
       'XML-RPC',
       'An RPC Protocol');

$client->simple_request('newdict.add_definition',
       'XML-RPC',
       'A Web Services Protocol');

my $result = $client->simple_request('newdict.get_definition', 'XML-RPC');

my $definitions = $result->{definitions};
print join("\n\t", "$result->{term}:", @$definitions), "\n";
```

This new dict-client program starts by creating an RPC::XML::Client object that will act as a proxy for the XML-RPC server found at *http://localhost:10000*. Each call to $client->simple_request() encodes an XML message to send to the XML-RPC server, where the request is processed and a result returned. The client object then translates the XML result into native Perl data structures, which are then returned from the simple_request method.

And that's it! All of the necessary glue code to talk to the server is handled in the RPC::XML::Client module. Communicating with an XML-RPC server is as simple as writing straightforward Perl code.

Conclusion

Web services are many things to many people. At the heart of it all are simple RPC mechanisms that use HTTP and XML. While Web services may not provide the highest-performance solutions, they greatly simplify the work required to create client and server programs.

Using XML-RPC is an easy way to get started using Web services with Perl. The RPC::XML module makes it easy to glue an XML-RPC server interface to an existing piece of code, and easy to write clients to talk to XML-RPC servers.