

# Persistent Web Services with Perl

by Alex Batlin

## Table of contents

1 Overview.....	2
2 Requirements.....	2
3 Solution.....	2
4 Example.....	2

## 1. Overview

There is a growing requirement to engineer ever more sophisticated administration and monitoring tools for highly distributed ClearCase, ClearQuest and TestDirector environments. This article looks at a web services based architecture, which can be used for client-server communication.

## 2. Requirements

Increasing number of monitoring and administration tools that we engineer are client-server based, and need to permanently store state information. Systems are likely to be accessed both by people and other programs.

To be future proof, we need to base our implementations on open standards and make them as platform and language independent as possible. Last but not list, they must be easy to develop, deploy and maintain i.e. the least possible number of dependencies.

## 3. Solution

Persistent Web Services with Perl offer a sound implementation for the stated requirements. Web Services provide the communication mechanism, whilst simple XML files can be used for persistence.

Implementing systems with Web services gives the following benefits

- allow different programs to communicate with each other i.e. support client-server model
- are built on open standards, they open up the functionality of the system to both people and other programs
- are easily implemented with Perl alone, so can be easily deployed and maintained on all major platforms
- can later be re-engineered in another language and for another platform with minimum impact on clients

Persistence can be achieved through simply saving in a file data structures in an XML format

- an open standard, so if we need to move away from Perl, data can still be easily parsed
- easy to implement in Perl alone, and does not have a database store dependency

## 4. Example

The best way to demonstrate how such a system works, is by following this example.

This is a very simple application with two functional requirements:

- client must be able to add a new user to the system by passing the userid, first and last names to the service
- service must be able to return the list of users if requested by the client

This example is implemented through three scripts:

- **client.pl** - script that allows the user to either add a new user, or request all users in the system

```
# make sure that we define all variables
use strict;
# one of the standard XMLRPC web services libraries, simpler than SOAP
use XMLRPC::Lite;

# determine which method the user wants to execute
my $method = $ARGV[0];

if ($method eq "get_users") {
    # the service is running on the local host
    my $users = XMLRPC::Lite
        -> proxy('http://localhost')
        -> call('Example.get_users')
        -> result;
    # iterate through the returned list of users
    foreach my $user (@$users) {
        print $user->{userid}->[0] . " " .
            $user->{first}->[0] . " " .
            $user->{last}->[0] . "\n";
    }
}

# the service is running on the local host, user passes userid, first
# and last names
if ($method eq "add_user") {
    my $result = XMLRPC::Lite
        -> proxy('http://172.16.39.133')
        -> call('Example.add_user', $ARGV[1], $ARGV[2], $ARGV[3])
        -> result;
    # print out the result returned from the service
    print "Adding new user result : $result";
}
```

- **server.pl** - script that handles client requests and invokes the requested methods defined in Example.pm

```
# make sure that we define all variables
use strict;
# load the Example module
use Example;
# one of the standard XMLRPC web services libraries, simpler than SOAP
use XMLRPC::Transport::HTTP;

# start the stand alone service on the local host, port 80
# execute methods in the Example module
my $daemon = XMLRPC::Transport::HTTP::Daemon
```

```
-> new (LocalAddr => 'localhost', LocalPort => 80)
-> dispatch_to('Example')
-> handle;
```

- **Example.pl** - module that defines service's methods

```
package Example;

# make sure that we define all variables
use strict;
# a simple XML library allows to read in and write out XML
use XML::Simple;

# global scope the variables
our($cfg,$xs,$ref);

# execute block at load time
BEGIN {
    $cfg = "config.xml";
    $xs = new XML::Simple (
        ForceArray => 1,
        KeyAttr => [ ],
        RootName => 'config',
        OutputFile => $cfg
    );

    # try to read in an existing config file, otherwise create a new
reference
    # this will load all of the previously saved values into memory
    eval {
        $ref = $xs->XMLin($cfg);
    };
    if ($@) {
        $ref = {};
    }
}

# the add_user method, adds a new user and saves results in XML file
sub add_user {
    # get the passed parameters
    my $class = shift;
    my $userid = shift;
    my $first = shift;
    my $last = shift;

    # add the new user details into memory
    push (@{$ref->{user}}, {
        first => [$first],
        last => [$last],
        userid => [$userid]
    });

    # as soon as the new user is added, persist the file by saving
the
    # data structure as an XML file, and return the result to the
client
    return $xs->XMLout($ref);
}

# the get_users method, gets all users and returns them to the client
```

```
sub get_users {
    return $ref->{user};
}

1;
```

The service will need to be started up, before the clients can access it, here is an example:

```
C:\example>perl server.pl
```

The server.pl will start-up and HTTP daemon, and load Example.pm module. This will in turn try to load the config.xml file that contains our persisted user data, if no such file exists, then a new empty hash is created.

The user can then add a new user to the system:

```
C:\example>perl client.pl add_user batlinal Alex Batlin
Adding new user result : 1
```

This will then add the new user to the hash stored in memory of the web service, and then the whole method is written out to an XML file. This is not a very efficient mechanism, as every write will cause a re-write of the XML file, but is very safe - ideal for largely read, sometimes write systems.

Instead of writing out the file after every addition, the write can be moved to the END block - there is a risk here of a crash causing data loss, or it can be exposed as a separate method - allowing the client to manage persistence.

Note however, web services should have a higher granularity than say a database call. What this means, is that an update to the memory should where possible be a large transaction e.g. pass a whole list of users, therefore reducing the need to persist at a very granular level.

Once the user added a new entry, requesting the list of users should return the newly added entry:

```
C:\example>perl client.pl get_users
batlinal Alex Batlin
```

The XML file will something like this:

```
C:\example>type config.xml
<config>
  <user>
    <userid>batlinal</userid>
    <last>Batlin</last>
    <first>Alex</first>
  </user>
</config>
```