

Cultured Perl: Perl and the Amazon cloud, Part 4

Dive into the full `mod_perl` site's code base

Skill Level: Intermediate

[Teodor Zlatanov \(tzz@lifelogs.com\)](mailto:tzz@lifelogs.com)

Programmer

Gold Software Systems

14 Jun 2009

This [five-part series](#) walks you through building a simple photo-sharing Web site using Perl and Apache to access Amazon's Simple Storage Service (S3) and SimpleDB. In this installment, examine the full `mod_perl` site's code base, including how to configure the top level, what to do with the handlers, and how to set up external dependencies.

In this installment, prepare to witness a full `mod_perl` site (code only; templates are in the next part). Now our formerly relaxed pace (a saunter or a canter, if you like) will become a full gallop as we race on our `mod_perl` horse through the Plains Of Strained Metaphors.

To get the most from this series

This series requires beginner-level knowledge of HTTP and HTML, as well as intermediate-level knowledge of JavaScript and Perl (inside an Apache `mod_perl` process). Some knowledge of relational databases, disk storage, and networking will be helpful. The series gets increasingly technical, so see the [Resources](#) section if you need help with any of those topics.

I strongly encourage you to read the source code. The site is functional, but many details are not fully explained in this series because I expect you either understand them or can learn what you didn't understand. Your local or remote bookstore and search engines are your friends.

In particular, setting up a full `mod_perl` site and using the Template Toolkit are

broad topics that have been covered many times and so are not explained at all here. The best way to learn is to work through every question and obstacle until the site is running. This series presents you with the engine, wheels, body, etc.—it's up to you to get gas and get the car running.

As before, I use `share.lifelogs.com` in this series as the domain name. Remember to change it as needed for your own environment.

Top-level configuration

You need to have a working Apache server with `mod_perl` support (so set one up). Insert the following section in your Apache `httpd.conf` file, as shown in Listing 1:

Listing 1. Providing `mod_perl` support to Apache configuration file `share.httpd.conf`

```
<VirtualHost 1.2.3.4:80>
  ServerName      share.lifelogs.com
  DocumentRoot    /var/www/html
  ErrorLog        /var/log/apache/error-share.log
  PerlRequire     /home/tzz/mod_perl_require_share.pl
  <Location />
    SetHandler    perl-script
    PerlHandler   SharePerlHandler
  </Location>
  SetEnv AWS_KEY 'my-AWS-key'
  SetEnv AWS_SECRET_KEY 'my-secret-AWS-key'
</VirtualHost>
```

Everything lives under `/home/tzz`, as you can see.

Important things here:

- There is a specific error log (so you can watch errors with this site in isolation).
- You pass the Amazon developer keys in the process environment. This way, the Perl source code will not have them in case the source code leaks out somehow. (The Web server configuration is normally more secure than source code.)

Note that *everything* is handled through `SharePerlHandler`, every request on `share.lifelogs.com`! This is probably not what you want in a production environment.

The `PerlRequire` directive just set up an environment, doing nothing special. Again, everything is under `/home/tzz`.

Listing 2 shows the `mod_perl_require_share` Perl file.

Listing 2. The `mod_perl_require_share.pl` file

```
#!/usr/bin/perl -w
use strict;
use lib '/home/tzz';
use SharePerlHandler;
1;
```

The `mod_perl` handler

The `mod_perl` handler is entirely in the `SharePerlHandler.pm` file. It has several sections, broadly speaking: setup, main handler, comment and photo handlers, general utilities, and SimpleDB utilities.

The general and SimpleDB utilities could have lived in their own module, but for simplicity I left everything in one place. The comment and photo handlers and the SimpleDB utility functions are mostly from the `simple_go.pl` script (see [Downloads](#)) with some small changes.

Let's start with the setup. As I walk you through each section, I'll explain my decisions; more often than not you'll hear "simplicity" as the reason I did things in a particular way. Making a *good* Web site is hard, so take everything you find here as a rough template to be filtered through your needs and budget rather than a finished design you can pop into production. The fact that it works may, in fact, be a distraction, but I couldn't resist the temptation to get it working.

Setting up external dependencies

Listing 3 shows the general setup for the `SharePerlHandler.pm` file:

Listing 3. General setup for `SharePerlHandler.pm`

```
package SharePerlHandler;
use Apache::Constants qw(:common REDIRECT);
use strict;
use Carp qw(verbose cluck carp croak confess);
use Data::Dumper;
use Apache::Request;
use Template;
use POSIX;
use Digest::HMAC_SHA1 qw(hmac_sha1 hmac_sha1_hex);
use MIME::Base64;
use Data::UUID; # generates unique IDs
```

```
use lib '/home/tzz/amazon-simpliedb-2007-11-07-perl-library/src/';
use Amazon::SimpleDB::Client;
```

SharePerlHandler.pm depends on many modules. First of all, it uses the `strict` module, which is essential to good Perl programming. I wouldn't put something in production that didn't run under `use strict`. Also:

- The `Carp` modules provide better errors.
- `Data::Dumper` is for general debugging.
- `POSIX` is for many functions I find I need frequently.
- `Digest::HMAC_SHA1` and `MIME::Base64` are for the Amazon S3 upload policy.
- The `Template` module is the Template Toolkit, which will let us put HTML pages with some dynamic content together quickly.
- `Data::UUID` is for generating unique IDs.
- `Apache::Request` and `Apache::Constants` are for `mod_perl` interaction with the Apache server.
- Finally, `Amazon::SimpleDB::Client` is from Amazon and lets us interact with SimpleDB.

If you don't know how to install these modules from CPAN, it's done with `cpan -e 'install MODULE'` (if you don't know this, you're probably in way over your head right now...).

We don't use the `Net::Amazon::S3` modules here, although we could have (in [Part 2](#) I said we would). They were just not needed due to several architectural decisions in the interest of simplicity; more on this when I talk about uploads.

Listing 4 shows the global setup for SharePerlHandler.pm.

Listing 4. The global setup for SharePerlHandler.pm

```
use constant IMAGE_MODE    => 0;
use constant COMMENT_MODE => 1;

use constant VERBOSE => 1; # can also be done through the environment or some other way

my $template = Template->new( {
    INCLUDE_PATH => '/home/tzz/templates/share',
    RECURSION   => 1,
  }
);

my $uuid = Data::UUID->new();
```

You'll need constants to represent the image mode versus the comment mode for the basic SimpleDB operations, so define them here. The `VERBOSE` constant can be replaced by any other method to control verbosity on your server. Remember, the more dynamic the control over verbosity, the more expensive it is (because the server needs to check it every time).

Next, you get yourself a brand new `$templates` object (which will load templates from `/home/tzz/templates/share` and will recurse). Finally, you get a UUID generator you can use everywhere.

The main handler

All right, this is the big one, the place where magic happens. PAY ATTENTION! (Awake? Good!) Take a thoughtful look at Listing 5.

Listing 5. More global setup

```
sub handler
{
  my $r = shift @_ ;
  my $q = Apache::Request->new($r,
                              POST_MAX => 4096,
                              DISABLE_UPLOADS => 1);

  my $user = (rand 2 > 1) ? 'bob' : 'ted';
                              # pick a user randomly between bob and ted
                              # (50% chance each)

  handle_photo($q);
                              # always try to delete, add, or edit a URL
                              # if it's passed as a parameter

  handle_comment($q);
                              # always try to delete, add, or edit a comment
                              # if it's passed as a parameter

  my $uri = $q->uri();
  my $tfile = $uri;
  $tfile =~ s,^/,;
  $tfile =~ s,/,$,;
  $tfile =~ s,/,_,g;
                              # remove the starting "/" in the name if it exists
                              # remove the ending "/" in the name if it exists
                              # "/" in the file name becomes "_" so all the
                              # templates can be in one directory

  $tfile = 'index' unless length $tfile;
                              # make the URI "index" if it's
                              # empty (e.g. someone hit the / URI)

  if ($tfile =~ m/\.html$/)
  {
    $tfile =~ s/html$/tmpl/;
    # map ANYTHING.html to ANYTHING.tmpl
  }
  else
  {
    $tfile .= '.tmpl';
    # map ANYTHING to ANYTHING.tmpl
  }

  my $policy = '';
  my $signature = '';
}
```

```

if ($tfile eq 'upload.tmpl')
{
    $template->process('policy.tmpl',
        {
            username => $user,
        },
        \ $policy) || croak($template->error());

    my $key = $ENV{AWS_SECRET_KEY};
    $policy = encode_base64($policy);
    $policy =~ s/\n//g;
    $signature = encode_base64(hmac_shal($policy, $key));
    $signature =~ s/\n//g;
}

$q->send_http_header('text/html; charset=utf-8');
my $output = '';

$template->process($tfile,
    {
        request    => $q,
        username   => $user,
        policy     => $policy,
        signature  => $signature,
        env        => \%ENV,
        params     => \%{$q->param()},
        fimages    => sub { return list_simpledb(sprintf('SELECT * from `%s`',
            simpledb_image_domain_name())) },
        fcomments => \&get_comments,
    },
    \ $output) || croak($template->error());

print $output;
return OK;
}

```

Wow, that's a long function. It's almost too long; I would probably extract the pieces that can stand on their own ("refactor" it, as the cool kids say these days) if I had to add even a little more logic. It serves nicely to show how the main handler might look, though.

First the function gets the request object. Then it sets up a random user name ("bob" or "ted"); usually you'd have your own way to get this, perhaps through cookies, or you can let Apache handle authentication and authorization for you.

I said in [Part 1](#) that I was going to have a users table in SimpleDB, but it complicated the site too much, so I dropped it. Looking up users in SimpleDB is not simple, because I needed to provide a way to sign up and to manage user information. It made the code too big, so please forgive this omission—it's definitely possible to keep the users table in SimpleDB.

Okay, next you'll want to handle any photo or comment parameters. For example, if you see the `deletecommentid` parameter, you try to delete that comment ID. We'll dig into the photo and comment parameter handlers later.

Next you have to manage the actual request. This is done with a simple-minded mapping that transforms "any/request/here.html" into "any_request_here.tmpl" and

then requests that template. We make sure `index.tmpl` is served for the `/` request.

Any URIs that don't have a corresponding template will produce no data and in fact will throw an error. This is not a *production* technique, let's be clear, but it sets up a Web site in a few lines, so it's quite useful when the goal is brevity and simplicity in a demo.

Next, if the template file is going to be `upload.tmpl`, you know you'll need to generate an upload policy for S3, and so you do that using the `policy.tmpl` file. The user name is passed to that template, which is very similar to the one in [Part 2](#) of this series. Listing 6 shows you a policy template.

Listing 6. Sample code listing at maximum width

```
{ "expiration": "3000-01-01T00:00:00Z",
  "conditions": [
    { "bucket": "images.share.lifelogs.com" },
    { "acl": "public-read" },
    [ "starts-with", "$key", "" ],
    [ "starts-with", "$Content-Type", "" ],
    [ "starts-with", "$success_action_redirect",
      "http://share.lifelogs.com/s3uploaded?user=[% username %]" ],
    [ "content-length-range", 0, 1048576 ]
  ]
}
```

The major difference here is that instead of making the user name part of the success URL, you make it a parameter because it makes the image parameter handler much simpler. More on that to come.

Now you sign the policy and move on to sending the HTTP headers (good work there, Apache!). Next, generate the necessary output with quite a few parameters, as follows:

- `request`, the Apache request
- `username`, the random user name
- `policy`, the S3 upload policy (could be blank).
- `signature`, the S3 policy signature (could be blank)
- `env`, the process environment (don't do this in production!)
- `params`, the parameters, for example from a POST or a GET request
- `fimages`, a function to return all images
- `fcomments`, a function to return all comments, keyed by image ID

That's it for the general handler. All the other magic happens in the comment and

image parameter handlers and in the templates themselves. Let's continue our top-down journey.

The image and comment handlers are called for every request. If they find parameters that look applicable, they will do an operation: add, modify, or delete an image or a comment. The templates, which we'll study after `SharePerlHandler.pm`, contain these parameters in `POST` forms.

But don't jump ahead, there's plenty here to keep you entertained—shoddy code, dodgy architecture, and the hope you'll find a bug worth posting on Twitter ("haha @tzlatanov sux teh worst check this out map used in void context and omg b0rken templates obv not a real h4ck3r mod_perl issolastmllnm").

The comment parameter handler

Listing 7 shows off the comment parameter handler.

Listing 7. The comment parameter handler

```
sub handle_comment
{
    my $q = shift @_ ;

    my $user          = $q->param('user');
    my $imageid       = $q->param('refimageid');
    my $comment       = $q->param('comment');
    my $refcommentid  = $q->param('refcommentid');
    my $commentid     = $q->param('commentid');
    my $deleteid      = $q->param('deletecommentid');

    my $result;

    if (defined $deleteid) # delete
    {
        $result = delete_simplifiedb($deleteid, COMMENT_MODE);
    }
    elsif (defined $commentid && defined $comment) # edit
    {
        my %q = (
            comment => $comment,
        );

        put_simplifiedb($commentid, COMMENT_MODE, %q);
        $result = get_simplifiedb($commentid, COMMENT_MODE);
    }
    elsif (defined $imageid && defined $comment) # new comment
    {
        my %q = (
            image_id => $imageid,
            comment => $comment,
        );

        $q{reply_to} = $refcommentid if defined $refcommentid;
        $q{user} = $user if defined $user;

        my $id = new_uuid();
        put_simplifiedb($id, COMMENT_MODE, %q);
    }
}
```

```
$result = get_simplifiedb($id, COMMENT_MODE);  
}  
  
$q->param()->{'result'} = $result;  
}
```

The comment parameter handler has three possible modes, depending on the parameters passed to it. The modes are mutually exclusive. In every case, the `result` query parameter is set appropriately to indicate success if it's defined. Thus, the templates can later check if it's set and act accordingly. As the site grows, you would probably want to add other parameters such as `last_operation` or `error_message`.

If the `deletecommentid` parameter is set, the handler calls `delete_simplifiedb` with the appropriate values. This is the simplest, unconditional mode.

The next mode modifies a comment. It does not check that the comment exists already, so an incorrect ID here would create a new comment. Checking is easy but expensive (you have to make an extra call to SimpleDB, which is slow since it's a whole HTTP round trip plus the processing time on Amazon's side).

Note that because each comment has its own ID, editing is easy. Without individual comment records, you could have grouped comments as an image attribute (an array of strings, each one a comment), but then it would have been much harder to edit or delete an individual comment. Essentially, you would have had to implement your own record structure within a comment to represent an ID, a posting user, etc.

The editing mode is triggered by the presence of the `commentid` and `comment` query parameters, which state the edit target's UUID and the new contents of the comment, respectively. The result is the retrieval of the same UUID back from SimpleDB, so you could check here to see if the resulting comment field is the one you wanted (otherwise something went wrong). I don't check all this here in the interest of simplicity.

The final mode, creating a new comment, is triggered by the `imageid` and `comment` parameters. Optionally, it will take a reference UUID (when the comment is a reply to another one) and a user name (when the comment is not anonymous). Just like the editing mode, you just `put_simplifiedb` the attributes and then get them back without checking if the fields were modified correctly.

The image parameter handler

This handler is very similar to the comment handler, so if you slept through it all, go back and pay attention.

The image URL, if not passed, is constructed from the S3 key and bucket. This way

you have a consistent interface to handle success redirects after S3 uploads. Listing 8 shows off the image parameter handler.

Listing 8. The image parameter handler

```
sub handle_photo
{
    my $q = shift @_;

    my $user      = $q->param('user');
    my $name      = $q->param('name');
    my $bucket    = $q->param('bucket');
    my $key       = $q->param('key');
    my $url       = $q->param('url');
    my $editid    = $q->param('imageid');
    my $deleteid  = $q->param('deleteimageid');

    # set the URL from the S3 key and bucket if necessary
    if (!defined $url && defined $key && defined $bucket)
    {
        $url = sprintf("http://%s.s3.amazonaws.com/%s", $bucket, $key)
    }

    my $result;

    if (defined $deleteid) # delete
    {
        $result = delete_simplifiedb($deleteid, IMAGE_MODE);
    }
    elsif (defined $name && defined $editid) # editing an image name
    {
        my %q = (
            name => $name,
        );

        put_simplifiedb($editid, IMAGE_MODE, %q);
        $result = get_simplifiedb($editid, IMAGE_MODE);
    }
    elsif (defined $url && defined $name && defined $user) # adding a new one
    {
        my %q = (
            url => $url,
            name => $name,
            user => $user,
        );

        $q{bucket} = $bucket if defined $bucket;

        my $id = new_uuid();
        put_simplifiedb($id, IMAGE_MODE, %q);
        $result = get_simplifiedb($id, IMAGE_MODE);
    }

    $q->param()->{'result'} = $result;
}

```

Deleting the image is triggered by the `deleteimageid` parameter. Editing an image's name is done with the `name` and `imageid` parameters. Creating a new image is done with a URL, a user name, and an image name. The image bucket is optional and should only show up if this handler is called after a S3 upload success redirects to our site.

Remember from the policy, the S3 success redirect has the user name as part of the URL as a parameter. It will also have the key and bucket, so all you have to do is create an image from them.

Utility functions

Let's look at the truly miscellaneous functions.

Listing 9. Truly misc functions

```
sub qlog
{
    printf STDERR @_ ;
    print STDERR "\n";
}

sub new_uuid
{
    return $uuid->to_string($uuid->create());
}

sub simpledb_image_domain_name
{
    return simpledb_domain_name(IMAGE_MODE);
}

sub simpledb_comment_domain_name
{
    return simpledb_domain_name(COMMENT_MODE);
}

sub simpledb_domain_name
{
    return sprintf "%s.share.lifelogs.com",
        (shift == IMAGE_MODE) ? 'share_photos' : 'share_comments';
}
```

`qlog` is nice when you don't want to write `printf STDERR` every time. It also writes a line ending for you. Whether miscellaneous output should be going to the Apache error log is up to you. Also:

- `new_uuid` is for generating a new UUID.
- `simpledb_domain_name`, `simpledb_image_domain_name`, and `simpledb_comment_domain_name` use the mode parameter (which can be `IMAGE_MODE` or `COMMENT_MODE`) to provide a SimpleDB domain.

SimpleDB utility functions

The SimpleDB utility functions are almost exactly like the ones in [Part 3](#) (`simple_go.pl`). Download them from the [Downloads](#) section, below. I'll list the changes:

- `get_comments` is a new function to get all comments, keyed by image ID and then either the parent comment ID or the word `noparent`.
- `qlog` is used instead of `print`, and the `VERBOSE` constant is used instead of `$verbose`.
- The service is initialized for every request using the environment `AWS_KEY` and `AWS_SECRET_KEY` keys.
- The mode, as `$mode`, is passed around instead of being global.
- On errors, instead of using `die()`, you handle them as gracefully as possible.
- The domain name is obtained with `simpledb_domain_name` instead of a global variable.
- The functions were renamed because "get" and "put" are not good names in a namespace that does not serve a single purpose, as the old `simple_go.pl` script did.

I should mention again that this code will only do single values per attribute. If any attribute has an array of strings, only one of them will be returned. When attributes are written, only one value will remain even if there was an array of values before. The code is significantly simpler because of this, but it's also not suitable for general-purpose SimpleDB usage.

Wrapup

You've now taken a tour of the code for a full `mod_perl` site, using the bits we wrote in Parts 2 and 3 of this [series](#) (the download files for those parts are available below as well). The resulting site uses the Template Toolkit, S3, and SimpleDB to provide image uploads, browsing (threaded), editing and comment adding (anonymous or not), and deleting.

In Part 5, you'll get familiar with the template version of the site.

Downloads

Description	Name	Size	Download method
SimpleDB utility functions	simpledb_utility.zip	3KB	HTTP
Sample script (from Part 2)	s3form.zip	2KB	HTTP
Sample script (from Part 3)	simple_go.zip	4KB	HTTP

[Information about download methods](#)

Resources

Learn

- Start with "[Cultured Perl: Perl and the Amazon cloud, Part 1](#)" (developerWorks, March 2009) to understand the benefits and drawbacks of using Amazon's S3 and SimpleDB for Web site building. Then proceed to [Part 2](#) (April 2009) to upload a file into S3 from a Web page through an HTML form to minimize the load on the server, and [Part 3](#) (June 2009) to upload images via a list of URLs in a table and manage images and comments.
- Service offering details and developer resources are on Amazon.com:
 - [Amazon S3](#)
 - [Amazon SimpleDB](#)
- Learn about the amazing [JavaScript MimeType](#), a property of the navigator object, a top-level object that is the object representation of the client Internet browser or Web navigator program that is being used.
- [mod_perl](#) brings together the full power of Perl and the Apache HTTP server.
- [Prototype](#) is a JavaScript framework that makes it easier to develop dynamic Web applications via a toolkit for class-driven development and the "nicest" Ajax library on Earth. And here's an excellent article on using it: "[Developer Notes for prototype.js](#)" by Sergio Pereira.
- Yes, Virginia, it's a Web-based service and therefore can suffer [outages](#); an important consideration.
- In the [developerWorks Linux zone](#), find more resources for Linux developers, and scan our [most popular articles and tutorials](#).
- See all [Linux tips](#) and [Linux tutorials](#) on developerWorks.
- Stay current with [developerWorks technical events and Webcasts](#).

Get products and technologies

- Visit CPAN for downloads, module listings, and documentation on the [Template-Toolkit-2.20](#), a fast template-building kit. O'Reilly has a book out, [Perl Template Toolkit](#), written by core members of the technology's development team.
- At the [CPAN \(Comprehensive Perl Archive Network\)](#) site, you can find scads of modules. And module documentation.
- [S3Fox, the Amazon S3 Firefox Organizer](#), the S3 management add-on for Firefox, puts an easy-to-use front end on S3.

- With [IBM trial software](#), available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the [My developerWorks community](#); with your personal profile and custom home page, you can tailor developerWorks to your interests and interact with other developerWorks users.

About the author

Teodor Zlatanov

Teodor Zlatanov emerged with an M.S. in computer engineering from Boston University in 1999. He has worked as a programmer since 1992, using Perl, Java, C, and C++. His interests are in open source work on text parsing, database architectures, user interfaces, and UNIX system administration.