

Cultured Perl: Perl and the Amazon cloud, Part 5

You've seen the code; now scan the full `mod_perl` site's templates

Skill Level: Intermediate

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

Programmer

Gold Software Systems

23 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 templates, including one for indexing, three for uploading (general, S3 forms, and URL additions), one for image and comment browsing, and one to browse comments recursively for an image (or threading down).

In this last installment, prepare yourself to witness a full `mod_perl` site (templates this time; the code base was in [Part 4](#)). Again, 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.

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 is fairly technical, so see the [Resources](#) section if you need help with any of those topics.

In particular, setting up a full `mod_perl` site and using the Template Toolkit are broad topics, so I won't explain it all in this series. The best way to learn is to work

through every question and obstacle until the site is running. Remember, I've given you the engine, wheels, body, etc.—now you need to get gas and get the car running.

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

index.tmpl

We'll start with the templates from the top down (`policy.tmpl` was discussed in [Part 4](#)). For explanations of the Template Toolkit syntax, refer to the [Resources](#) section. I'll try to explain the tricky bits.

The `index.tmpl` is a simple HTML page. Only thing to note here is that all the URIs are relative, so this and all the other templates will work with any domain.

Listing 1. The `index.tmpl`, simple HTML

```
<html>
  <head>
<title>Share Pictures</title>
</head>
<body>
<h1>Share Pictures</h1>

You can
<a href="/upload">upload or add images</a>
or
<a href="/browse">browse images and comments</a>.

<address>
Contact <a href="mailto:tzz@bu.edu">Ted Zlatanov</a> if you have lots
of money you're trying to get out of Nigeria. The breath
is <del>baited</del>bated.
</address>
</body>
</html>
```

upload.tmpl

Now we're ready for the good stuff. JavaScript, HTML, and Template Toolkit language in one place. If this doesn't make a Web designer cry, I don't know what will.

Listing 2. The `upload.tmpl`, enough to make a Web designer shed tears of joy

```
<html>
  <head>
    <title>Upload Page For [% username %]</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
```

```

    <script src="http://ajax.googleapis.com/ajax/libs/prototype/1.6.0.3/prototype.js"
        type="text/javascript"></script>
</head>

<body>
<script language="JavaScript">
function OnSubmitForm()
{
var form = $('uploader');
var file = form['file'];
var ct    = form['Content-Type'];
var name = form['name'].value;

if (!name || name.length < 1)
{
alert("Sorry, you can't upload without a name.");
return false;
}

var filename = '+'+$F(file);
var f = filename.toLowerCase(); // always compare against the lowercase version

if (!navigator['mimeTypes'])
{
alert("Sorry, your browser can't tell us what type of file you're uploading.");
return false;
}

var type = $(navigator.mimeTypes).detect(function(m)
{
// does any of the suffixes match?
return m.type.length > 3 && m.type.match('/') &&
    $(m.suffixes.split(',')).detect(function(suffix)
    {
return f.match('\.' + suffix.toLowerCase() + '$');
});
});

if (!type || !type['type'])
{
type = { type : prompt("Enter your own MIME type, we couldn't find one through
                        the browser", "image/jpeg") };
}

if (type && type['type'])
{
ct.value = type.type;

// fix up the redirect if we're about to submit
var sar = form['success_action_redirect'];

sar.value = sar.value + escape(name);
return true;
}

alert("Sorry, we don't know the type for file " + filename);
return false;
}
</script>
<h1>Hi, [% username %]</h1>
<form id="uploader" action="https://images.share.lifelogs.com.s3.amazonaws.com/"
    method="post" enctype="multipart/form-data" onSubmit="return OnSubmitForm();">
    <input type="hidden" name="key" value="{filename}">
    <input type="hidden" name="AWSAccessKeyId" value="[% env.AWS_KEY %]">
    <input type="hidden" name="acl" value="public-read">
    <input type="hidden" name="success_action_redirect"
        value="http://share.lifelogs.com/s3uploaded?user=[% username %]&name="
    <input type="hidden" name="policy" value="[% policy %]">
    <input type="hidden" name="Content-Type" value="image/jpeg">

```

```
<input type="hidden" name="signature" value="[% signature %]">
Select File to upload to S3:
<input name="file" type="file">
<br>
Enter a Name:
<input name="name" type="text">
<br>
<input type="submit" value="Upload File to S3">
</form>
<form id="adder" action="/urluploaded" method="post" enctype="multipart/form-data">
  <input type="hidden" name="user" value="[% username %]">
  Enter a URL:
  <input name="url" type="text">
  <br>
  Enter a Name:
  <input name="name" type="text">
  <br>
  <input type="submit" value="Add URL">
</form>
</body>
</html>
```

The upload page shows two upload dialogs. They both add an image, but the second one is much simpler. In the second one, the user fills in the URL and name for the image, and that gets POSTed to `/urluploaded`, which is just `urluploaded.tmpl`. The image parameter handler will be automatically invoked when that template is displayed. The user name is obtained from the server and is a form-hidden POST parameter.

The first form is the really complicated one. Fortunately, you can read [Part 2](#) of this series, which explained all about S3 uploads, so you shouldn't be surprised by any of it.

The major changes in `s3form.pl` from Part 2 (and included again in the [Downloads](#) section):

- `success_action_redirect` passes the user name and image name as parameters. The policy is adjusted to required only a portion of this string, up to the user name but not including the name.
- The policy and signature are passed from the server.
- The AWS access and secret keys are passed in the `env` hash from the server.
- The `OnSubmitForm` function requires a name and adds it to the `success_action_redirect` form field as a parameter, escaped (note that the name is demanded *before* the MIME type is determined, but it's added to the URL only *immediately* before the form is POSTed).
- The `OnSubmitForm` function fails gracefully if the MIME type can't be found, allowing the users to specify their own.

s3uploaded.tmpl

On a S3 upload, this template is used.

Listing 3. The s3uploaded.tmpl for successful and unsuccessful uploads

```
[% success = params.result %]
<html>
  <head>
    <title>[% IF success %]Successful[% ELSE %]Unsuccessful[% END %]
      Upload Page For [% params.user %]</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  </head>
  <body>
    [% IF success %]Congratulations[% ELSE %]Sorry[% END %], [% params.user %].
    You have [% IF success %]successfully[% ELSE %]unsuccessfully[% END %]
    uploaded [% params.key %] to S3 bucket [% params.bucket %]
    named [% params.name %].<p>
    (etag is [% params.etag %] but I doubt you care.)
    <p>
  [% IF success %]
    <a href="http://[% params.bucket %].s3.amazonaws.com/[% params.key %]">
    Your new upload is probably here. Let's see if it displays already.
    
    </a>
  [% END %]
    <p>
    You can now go back to <a href="/upload">uploading</a> or
    <a href="/">the main page</a>.
  </body>
</html>
```

Through some nasty Template Toolkit IF-ELSE constructs and the `result` parameter, the page handles successful and unsuccessful uploads. The success is regarding the SimpleDB addition; the image has always been uploaded to S3 if you get to this point. Backing out the image from S3 on a SimpleDB failure is left as an exercise for the reader (nyah, nyah).

urluploaded.tmpl

On a URL addition, this template is used.

Listing 4. The urluploaded.tmpl, the dream of code reuse

```
[% success = params.result %]
<html>
  <head>
    <title>[% IF success %]Successful[% ELSE %]Unsuccessful[% END %]
      URL add Page For [% params.user %]</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  </head>
  <body>
    [% IF success %]Congratulations[% ELSE %]Sorry[% END %], [% params.user %].
    You have [% IF success %]successfully[% ELSE %]unsuccessfully[% END %]
```

```

        added [% params.url %] named [% params.name %].<p>
    <p>
[% IF success %]
    <a href="[% params.url %]">
    The URL you added is, perhaps, visible here.
    
    </a>
[% END %]
    <p>
        You can now go back to <a href="/upload">uploading</a>
        or <a href="/">the main page</a>.
    </body>
</html>

```

Besides the obviously bad HTML, this is similar to the `s3uploaded.tmpl`. Code reuse is obviously not an issue in this writer's dream world.

browse.tmpl

This template browses images and comments.

Listing 5. The `browse.tmpl`, images and comments in a single shot

```

[% SET images = fimages() %]
[% SET comments = fcomments() %]
<html>
  <head>
    <title>Browse</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  </head>
  <body>
    <ul>
      [% FOR ik IN images.keys %]
        <li>
          [% SET image = images.$ik %]
          [% image.name %]<br>
          <br>
          [% IF image.bucket %](in S3)[% END %]<br>
          uploaded by [% image.user %]<br>
          <form action="/browse" method="post" enctype="multipart/form-data">
            <input type="hidden" name="deleteimageid" value="[% ik %]">
            <input type="submit" value="Delete">
          </form>
          <form action="/browse" method="post" enctype="multipart/form-data">
            <input type="hidden" name="imageid" value="[% ik %]">
            Change Image Name:
            <input name="name" type="text" value="[% image.name|html %]">
            <input type="submit" value="Rename">
          </form>
          [% INCLUDE comments.tmpl ik=ik comments=comments %]
          <form action="/browse" method="post" enctype="multipart/form-data">
            <input type="hidden" name="user" value="[% username %]">
            <input type="hidden" name="refimageid" value="[% ik %]">
            Enter a Comment (as user [% username %]):
            <input name="comment" type="text">
            <input type="submit" value="Comment">
          </form>
          <form action="/browse" method="post" enctype="multipart/form-data">
            <input type="hidden" name="refimageid" value="[% ik %]">
            Enter Anonymous Comment:
            <input name="comment" type="text">

```

```

        <input type="submit" value="Comment" >
    </form>
</li>
[% END %]
</ul>
</body>
</html>

```

With this, you get all the images and comments in one shot. This is not efficient and will almost certainly fail horribly in a real Web site once you get a few thousand images and comments on them. You need to set up pagination on the images using SimpleDB's nice NextToken paging scheme (which the SimpleDB utility functions presented here do not use efficiently) or something of your own, and then only get the comments for the images you're about to display.

You really want to avoid getting comments unless you need them. SimpleDB requests are expensive. This is why this template passed the comments down to the `comments.tmpl` template every time.

The template uses a Template Toolkit `FOR` loop to iterate over the images, unsorted (if you need sorting, it's better to have the Perl code do it outside the Template Toolkit environment). The image key is required here to select the comments that match it. For each image, you show the image name, URL, S3 status, owner. Then you show forms to delete the image, change its name, or post a comment anonymously or as a user. It's pretty straightforward.

Finally (well, in the middle, but we're not linear thinkers, right?) the `comments.tmpl` template is `INCLUDED` with some parameters—`ik` is the image key and `comments` is the comment list—meaning that whatever that template generates will get dropped right in the middle of our image list for each image.

comments.tmpl

This template browses comments recursively for an image (threading down).

Listing 6. Threading down with the comments.tmpl

```

[% IF parent %]
  [% SET thread = comments.$ik.$parent %]
[% ELSE %]
  [% SET thread = comments.$ik.noparent %]
[% END %]

<ul>
  [% FOR ck IN thread.keys %]
    [% SET comment = thread.$ck %]
    <li>[% comment.comment %] (by [% IF comment.user %][% comment.user %]
      [% ELSE %]Anonymous[% END %])<br>
      <form action="/browse" method="post" enctype="multipart/form-data">
        <input type="hidden" name="deletecommentid" value="[% ck %]">
        <input type="submit" value="Delete">

```

```

</form>
<form action="/browse" method="post" enctype="multipart/form-data">
  <input type="hidden" name="commentid" value="[% ck %]">
  Edit Comment:
  <input name="comment" type="text" value="[% comment.comment|html %]">
  <input type="submit" value="Edit">
</form>
<form action="/browse" method="post" enctype="multipart/form-data">
  <input type="hidden" name="user" value="[% username %]">
  <input type="hidden" name="refimageid" value="[% ik %]">
  <input type="hidden" name="refcommentid" value="[% ck %]">
  Enter a Comment (as user [% username %]):
  <input name="comment" type="text">
  <input type="submit" value="Comment">
</form>
<form action="/browse" method="post" enctype="multipart/form-data">
  <input type="hidden" name="refimageid" value="[% ik %]">
  <input type="hidden" name="refcommentid" value="[% ck %]">
  Enter Anonymous Comment:
  <input name="comment" type="text">
  <input type="submit" value="Comment">
</form>
[% INCLUDE comments.tpl ik=ik comments=comments parent=ck %]
</li>
[% END %]
</ul>

```

I've saved the best, nastiest code for last.

Given an image key, this template finds all the comments for that image whose parent equals a particular comment key.

For each comment of interest (with the given parent or without a parent as the case may be), the template shows the comment itself followed by HTML forms to delete the comment, edit it, or enter a new comment (under a user name or anonymously). This is almost the same as the forms in the `browse.tpl` except they include a `refcommentid` parameter.

Now, and this is the tricky or horrible part, depending on whether you merely study Computer Science or teach it, this template includes itself again for every comment it finds, setting `parent` to the value of the comment key each time. So you have recursion in a pseudo-language (Template Toolkit) to generate recursion in a layout language (HTML) running under Perl.

Wrapup

This five-part series is complete. You've now seen the templates for a full `mod_perl` site—using the pieces we wrote in Parts 2 and 3 and the code generated from Part 4. This site uses the Template Toolkit, S3, and SimpleDB to provide image uploads, browsing, editing, deleting, as well as comment adding (anonymous or not), browsing (threaded), and deleting.

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, [Part 3](#) (June 2009) to upload images via a list of URLs in a table and manage images and comments, and [Part 4](#) to walk through the full site's code base.
- 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.
- Cloud computing with Amazon services is a hot topic. This series is about accessing the services using Perl, but for a broader overview of the offerings, read the series "[Cloud computing with Amazon Web Services](#)" (developerWorks, July 2008 - February 2009).
- 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.