

# Cultured Perl: Perl and the Amazon cloud, Part 2

## Securely upload data to S3 via HTML form

Skill Level: Intermediate

[Teodor Zlatanov \(tzz@lifelogs.com\)](mailto:tzz@lifelogs.com)  
Programmer  
Gold Software Systems

08 Apr 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, learn how to upload a file into S3 from a Web page through an HTML form to minimize the load on the server, while maintaining a tight security policy.

You can upload a file into Amazon's Simple Storage Service (S3) from a Web page in several ways:

- From the command line using the appropriate modules from CPAN
- From the command line using the appropriate modules from Amazon
- Directly from an HTML form

### 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.

This article shows you upload a file directly from a HTML form, thus minimizing the load on your server. A successful upload will go to a custom URL that we'll use in

later parts of the series to set up the other pieces of the photo-sharing site we're building. We'll use `share.lifelogs.com` in the series as the domain name.

## Uploading to S3

You can upload data into S3 through a form `POST`. (The `PUT` HTTP method and the SOAP `PutObject` call can also be used.) For this article, we'll use the form `POST`, because it's simple and does not use server resources (disk, CPU, or bandwidth).

One big problem with uploads to S3 is that the metadata cannot be changed. That may be due to the distributed nature of S3 or to Amazon's wish to keep it simple. On the S3 forums, Amazon has indicated this may change in the future.

In any case, this means that the `Content-Type` must be set when the content is uploaded, or you'll get the rather unpleasant content type `binary/octet-stream` set. The other metadata is not too important because Amazon's SimpleDB will be used to track the uploads, and you can store your metadata there. We offer a mostly satisfactory [JavaScript solution](#) to the `Content-Type` problem.

The user name will be part of the successful URL. You could also put it in the metadata for the upload, so it's permanently associated with the file, but then a user can't change his or her name without making you redo all of their S3 uploads some way in order to store new metadata. You could keep user IDs that are independent of the user name and associate the user ID with the S3 object, but that gets needlessly complicated for our purpose—to build a simple photo-sharing site.

We (meaning the Web site operators) have at this point set up a bucket called `images.share.lifelogs.com` on S3. The bucket has the right access controls to allow public reading. If you have trouble doing this from the Amazon documentation, use a tool like S3Fox, JungleDisk, or any of the many other S3 interfaces to set the bucket up. If you're following along, you also have AWS access and secret keys.

You'll use a policy to control uploads; the policy is a set of rules expressed in the JSON data format. You will sign the policy with your secret Amazon key.

Signing the policy is done with the `Digest::HMAC_SHA1` Perl module. You must first convert the policy to Base64, remove all newlines, sign the resulting data, Base64 encode the signature, then turn around three times, touch your toes, pull a hair and burn it, and finally send \$1.28 in pennies to Ms. Elle Cowalsky and wait for her to mail you back the signature you should use. Just kidding! Burning the hair is optional. Try this instead:

### Listing 1. Signing the upload policy

```
my $aws_secret_access_key = 'get it from
```

```
http://aws-portal.amazon.com/gp/aws/developer/account/index.html?action=access-key';  
my $policy = 'entire policy here';  
$policy = encode_base64($policy);  
$policy =~ s/\n//g;  
  
my $signature = encode_base64(hmac_shal($policy, $aws_secret_access_key));  
$signature =~ s/\n//g;
```

## The S3 upload policy

You'll start with the policy before constructing the form; this means you will decide your security and usability goals before writing HTML, always a good thing.

The policy document is fairly simple:

### Listing 2. Upload policy document

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

It's all very well explained in the S3 developer documentation (see [Resources](#) for a link). The bucket must be as named, the ACL of the bucket must match, the key can start with anything, and on success you'll go to a particular URL. The size of the uploaded document is limited between 0 bytes and 1 megabyte. Note the expiration date (more on that to follow).

The policy's contents will be signed and publicly viewable, so your worst enemy will find it hard to forge its contents. This attribute makes the policy one piece of your site's security—it ensures that uploads to S3 are allowed based on particular criteria and not allowed otherwise. Remember, you pay for the S3 usage, so this is important.

The expiration date is set to 3000 (yes, the year 3000). The point is that this policy, for all practical purposes, does not expire. Instead, you could set the expiration date to 10 minutes in the future, and you'll be assured that the policy can't be used by deleted users more than 10 minutes after their last legitimate access. But then you might get complaints from users who take more than 10 minutes to upload a file and get rejected. So think about what would be a good period to set the expiration date to instead of arbitrarily setting it to some value.

The policy *must* have conditions for all of the fields specified in the form. This prevents forgery and encourages thorough policy documents.

Now that we've established a policy, let's set up the upload form.

## The S3 upload form

Remember we discussed the `Content-Type` metadata associated with S3 objects and how it has to be set before the object is uploaded? Unfortunately, this does not work well with image uploads because you don't know in advance what the user will upload. JPEG images and PNG images, for instance, have different content types (they are really called MIME types, and there are literally hundreds of common ones).

The solution is some intermediate-level JavaScript. Because you're putting everything inside one Perl script, you have to escape every `\` and `$` character (thus the slightly confusing contents). *Look at the generated HTML for the actual JavaScript*, you will use the Template Toolkit later in this series to do it right, but the idea was to make the script self-contained and simple. It's straightforward, but unfortunately readability suffered a bit.

### Listing 3. s3form.pl

```
#!/usr/bin/perl

use warnings;
use strict;
use Data::Dumper;
use Digest::HMAC_SHA1 qw(hmac_sha1 hmac_sha1_hex);
use MIME::Base64;

my $aws_access_key_id      = 'get it from
  http://aws-portal.amazon.com/gp/aws/developer/account/index.html?action=access-key';
my $aws_secret_access_key = 'get it from
  http://aws-portal.amazon.com/gp/aws/developer/account/index.html?action=access-key';

my $user = 'username'; # this is the user name for the upload

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

$policy = encode_base64($policy);
$policy =~ s/\n//g;

my $signature = encode_base64(hmac_sha1($policy, $aws_secret_access_key));

$signature =~ s/\n//g;
```

```

print <<EOHIPPIUS;
<html>
  <head>
    <title>S3 POST Form</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 submitUploaderForm()
{
  var form = $('uploader'); // note the escapes we do from Perl
  var file = form['file'];
  var ct = form['Content-Type'];

  var filename = '+' + \${F(file)}; // note the escapes we do from Perl
  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 = \${A(navigator.mimeTypes).detect(function(m)
  {
    // does any of the suffixes match?
    // note the escapes we do from Perl
    return m.type.length > 3 && m.type.match('/') &&
           \${A(m.suffixes.split(',')).detect(function(suffix)
           {
             return f.match('\.\.' + suffix.toLowerCase() + '\.$');
           // note the escapes we do from Perl
           });
    });
  });

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

  alert("Sorry, we don't know the type for file " + filename);
  return false;
}
</script>
  <form id="uploader" action="https://images.share.lifelogs.com.s3.amazonaws.com/"
        method="post" enctype="multipart/form-data"
        onSubmit="return submitUploaderForm();">
    <input type="hidden" name="key" value="\${filename}">
    <input type="hidden" name="AWSAccessKeyId" value="\${aws_access_key_id}">
    <input type="hidden" name="acl" value="public-read">
    <input type="hidden" name="success_action_redirect"
          value="http://share.lifelogs.com/s3uploaded/\${user}">
    <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>
    <input type="submit" value="Upload File to S3">
  </form>
</body>
</html>
EOHIPPIUS

```

Sorry to anyone offended by the inelegant JavaScript, but it should work in most modern browsers. Basically, we intercept the submit button and return true only if we know the type of the uploaded file.

**Editor's note:** You can [download this script](#). The two lines in bold in Listing 3 shouldn't actually break this way, but our display width is limited. If you copy and paste the script from the article, restore each to one line. They are correctly laid out in the script you can download.

## The upload form: JavaScript and reasoning

You load Prototype from the Google API site (see [Resources](#)). You can host it yourself if you're paranoid, obsessive, or a control freak ("just because you're paranoid doesn't mean they are not after you").

Grab the file name from the form using Prototype utilities, then look at the file name (in lower case). For each MIME type known to the browser, you use the Prototype `detect()` array method to find the first match to the following conditions:

- The type must be longer than three characters.
- It must contain a `/` character.
- Any of the suffixes for the MIME type must match the file name.

The three-character check and the `/` character check are due to the fact that in Firefox, at least, there's a `"*"` MIME type that will match anything. Because that's not useful, you want to be able to skip it and (we hope!) any other MIME types that are useless for our purposes.

You iterate through the suffixes using the JavaScript `split()` string method, which produces an array of pieces. So if the suffixes are `jpg, jpeg`, then you'll iterate over those two split on the comma. Again, you use the Prototype `detect()` array method to find the first match to the file name among those suffixes. Compare the lower-case version of the suffix with the lower-case version of the file.

If you're confused, read up on Prototype and JavaScript in general. For now you can just assume this works for most common cases. It may break for older or uncommon Web browsers, and, of course, it won't work if the user has disabled JavaScript. That's life. We're just looking for something that works for the majority of visitors.

If the type detection fails, so do we. Although it'll show a message to the user, this could be done better. You could try some guesses, for example, and maybe just fall back to `image/jpeg` if you don't know the type. Refining this is up to you. The function returns false if this happens, which will abort the upload.

Note that with a successful upload, you are redirecting to a URL that contains the user name. Refer to the section "[Uploading to S3](#)" for the reasons why.

Finally, this script has Perl, JavaScript, and HTML mixed into one *interesting* bunch (imagine a sports car with sails and a rudder, playing the saxophone). Examples written for the express purpose of showing a particular technique should not be your guideline to style and architecture. I dearly hope you don't copy and paste the included script without at least thinking about breaking it into template pieces and refactoring it. Later in this series, I promise to show you how that will work in the context of a whole `mod_perl` Web site.

## Wrapup

You've now seen how to set up an HTML upload form to upload files directly to S3. Perl, JavaScript, and HTML were employed. A single script using the Prototype JavaScript library, the Perl `MIME::Base64` and `Digest::HMAC_SHA1` modules, and inline JavaScript and HTML, is presented with some caveats. The script will upload a single file to S3 for a given user, redirecting to a particular successful URL on the `share.lifelogs.com` site in the end.

Part 3 will show how the successful URL will create a SimpleDB record for the uploaded file. You'll also discover how to create, edit, and delete comments as SimpleDB records on a photo for a particular user. Parts 4 and 5 will put it all together in a `mod_perl` Web site for you. Stay tuned.

## Downloads

Description	Name	Size	Download method
Sample script for this article	s3form.zip	2KB	<a href="#">HTTP</a>

[Information about download methods](#)

# Resources

## Learn

- Read "[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.
- 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.
- [Outages](#) are an important consideration because S3 and SimpleDB are Web-based services and can suffer outages.
- 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

- At the [CPAN \(Comprehensive Perl Archive Network\)](#) site you can find modules—scads and 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.
- [Order the SEK for Linux](#), a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

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

## Discuss

- Get involved in the [developerWorks community](#) through blogs, forums, podcasts, and spaces.

## 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.

## Trademarks

IBM, the IBM logo, ibm.com, DB2, developerWorks, Lotus, Rational, Tivoli, and WebSphere are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. See the current list of [IBM trademarks](#).

Linux is a trademark of Linus Torvalds in the United States, other countries, or both. UNIX is a registered trademark of The Open Group in the United States and other countries.