



The ultimate PHP guide to the Amazon S3 service

(Januari 07, 2009, by [Ferdy Christant](#))

In this article, I will explain how you can integrate the [Amazon S3 service](#) into your (PHP) applications. Only a small part of this article is about actual code, so you should be able to use the instructions for any platform. We'll be touching on various topics, including the basic setup, URL routing, scripted file uploads, ways to lower and control your Amazon bill and much more. I recommend you to read the **full article** to really get the most out of S3.

[Table of Contents](#)

.....	1
Introduction.....	2
Basics - setting up your account.....	2
Basics - manually uploading a file.....	3
Advanced – URL tuning.....	5
Advanced – scripted file uploads.....	7
Advanced - reducing your Amazon bill through caching.....	8
Advanced – Controlling your Amazon bill.....	9
Advanced - pushing it even further.....	12
Advanced – related services.....	13
Concluding.....	13

Introduction

Let us start with a brief overview of the Amazon Simple Storage Service (S3). It is quite simple, really. S3 is a pay-as-you-go model for storage (of any file type) in the cloud. It scales until infinity, is (pretty much) always available, and of course automatically backed up. You can see this service as an internet hard disk without the headaches of local storage:

- Local drives fail, so you will need to store data redundantly
- Local drives have limited capacity, so you soon need to manage multiple drives
- Local drives are physical: they age and are bound to a single location

The Amazon S3 service allows you to outsource the worries of local drives, and simply use their virtual storage service against an established [pay-per-use fee](#). As you can see from the list of fees, you pay for different cost components:

- Per GB of storage
- Per GB of bandwidth in (uploads)
- Per GB of bandwidth out (downloads)
- Per #requests (GET, PUT, etc)

Because of the payment model, S3 can be interesting for a broad range of users, from the individual who wishes to backup their local files to a remote service, to the application developer who wants to outsource the storage of (popular) files on the web, to large companies (such as [SmugMug](#)) who host TBs of data at S3.

Depending on your usage scenario you will interact with the S3 service in different ways. You can manually manage your files using one of the many client programs, such as [JungleDisk](#) or the Firefox add-on [S3Fox](#). If you want to dynamically interact with S3 from your application, you need to use the API, where you can choose between a SOAP and a REST API. Amazon does not officially support any platform-specific code to talk to the API, but the community has come up with great libraries for each major platform.

Basics - setting up your account

With the introduction behind us, let's get to it. The first step to using S3 is to [sign up](#). You will be asked about various personal details, including credit card details which will be used to charge you later on. Do not fear this sign up process, if you simply follow this article and play around with a few files you will not be charged more than a few cents at most.

Once signed up, you will receive an email where you can activate your account. Once you have it activated, you can go to your personal account page, where we will retrieve two things:

- Access key - public key
- Secret key - private key (share this with noone, ever!)

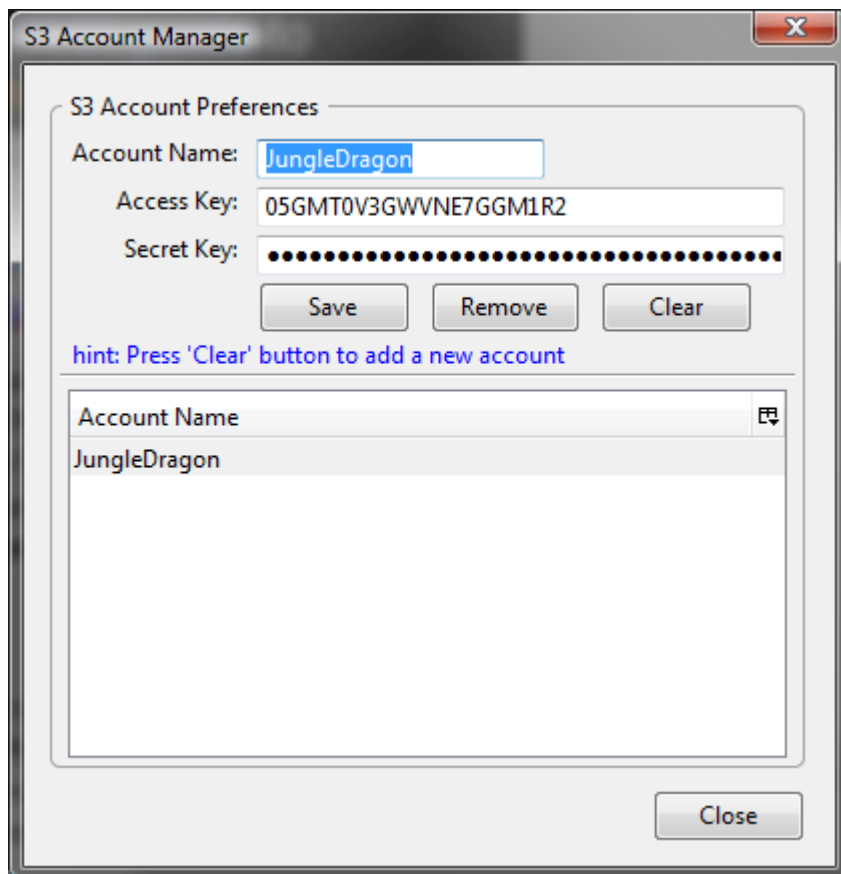
The combination of these keys will be used to sign API/access requests to the S3 service. Keep them in a safe place.

Basics - manually uploading a file

By signing up to the S3 service, your S3 storage is directly available. To test it, we are going to manually upload a file to it. The easiest way to do this is to download and install this [free Firefox plugin](#). After you restart Firefox, the bottom tray will show “S3Fox” button:

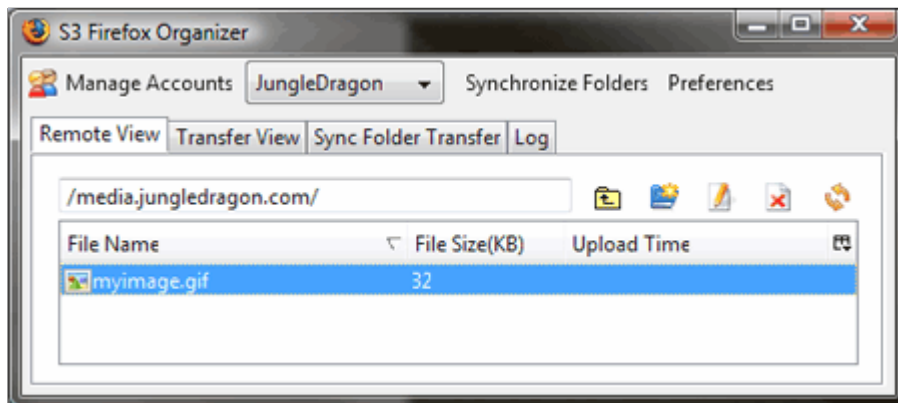


Click to open. Next, click the “Manage Accounts” button and enter a name for your account (free to chose) and your access key and secret key:



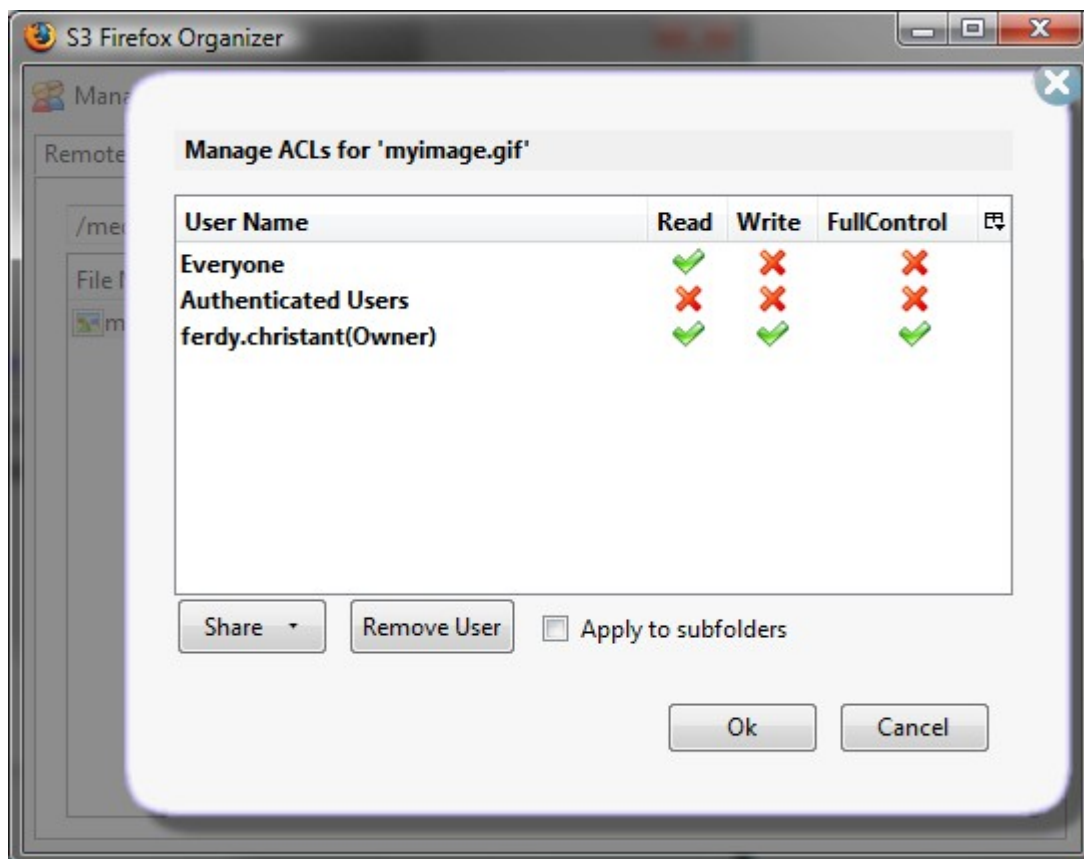
Now, in the “Remote View” tab, create a new directory. “Directory” is actually the wrong word for this. What happens is that we are creating an S3 “bucket”, which is like a virtual directory. This means that your directory/bucket name **must be unique across all accounts on S3**. For the remainder of this article I will assume the bucketname *media.jungledragon.com*, but of course you need to replace this with your unique bucket name. Note *media.jungledragon.com* is a single root level bucket, not three nested buckets.

Once you have created your bucket, you can simply drag and drop a file into it. For the purpose of testing, let us drag an image into it. Once the upload is completed, we can see our image being available in the Remote View:

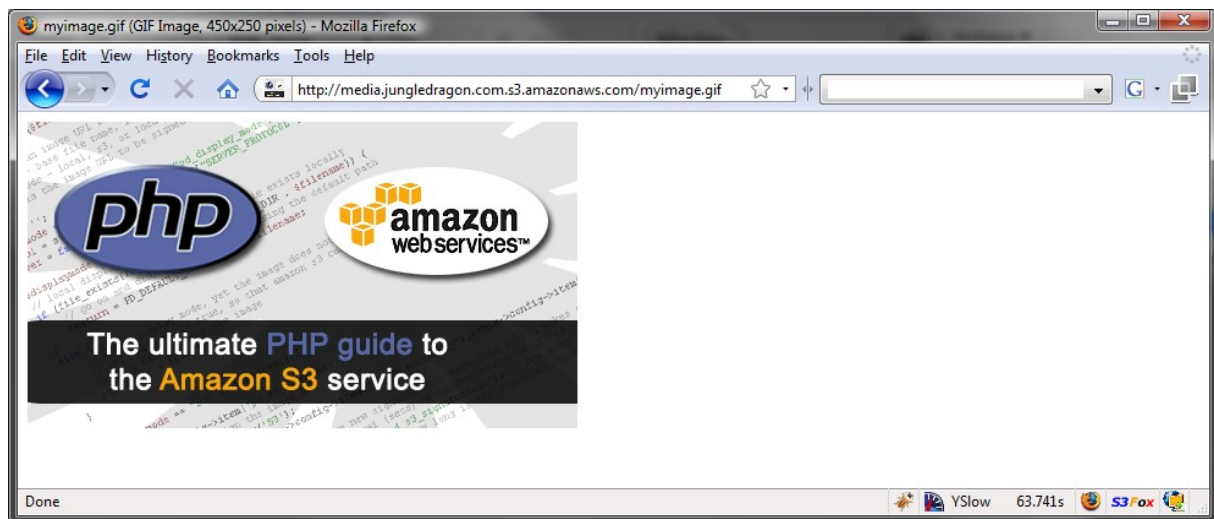


Your image is now safely stored at the S3 service, and you can access it at anytime using S3 Fox Organizer. But what if you want to access it from the web, for example by referring to the image address from your web page?

Each object (file) as well as each bucket at S3 has an ACL (Access Control List) associated with it. The ACL determines who can read, write and have full control over the object. In order to make our image public, we need to select it and click the "Edit ACL" button. Next, check the **Read** property for **Everyone**:



We can now open up the image in a web browser using the address ***media.jungledragon.s3.amazonaws.com/myimage.gif***:



Congratulations! You now have the basics working already. If you were to host some images or other media files for your blog, you can now simply refer to the Amazon S3 URLs from your pages.

Don't stop here though, in the remainder of the article there is vital information on tuning the experience, and lowering your bill.

Advanced – URL tuning

In the previous section, you saw how we used an Amazon S3 URL to access an image from our account. Now would be a great time to think about your URL strategy. There are three major ways to expose your files via URLs:

- The one demonstrated above, which includes the ***s3.amazonaws.com*** subdomain (***http://media.jungledragon.com.s3.amazonaws.com/myimage.gif***)
- A prettier URL, where we get rid of the ***s3.amazonaws.com*** subdomain (***http://media.jungledragon.com /myimage.gif***)
- A URL that points at your server which is then used as a proxy to retrieve the file from S3 (***http://media.jungledragon.com. /myimage.gif***) – yes. It's the same as the previous one.

We already saw the simple scenario, where we use the default Amazon S3 URL. It does not look pretty or professional, does it? To get rid of the ***s3.amazonaws.com*** suffix in the domain, we need to create a so-called CNAME alias in the DNS settings of our domain. It differs per domain registrar how to accomplish this, but it is typically quite simple. Here's how it works at my registrar, NameCheap:

 **Modify Domain: jungledragon.com**

HOST NAME	IP ADDRESS/ URL	RECORD TYPE ?	MX PREF
@	http://www.jungledragon	URL Redirect	n/a
www			n/a
SUB-DOMAIN SETTINGS ▼			
media	media.jungledragon.cor	CNAME (Alias)	n/a
test.media	test.media.jungledragon	CNAME (Alias)	n/a

You can see that I created two sub domains:

- media – points to media.jungledragon.com.s3.amazonaws.com
- test.media – points to test.media.jungledragon.com.s3.amazonaws.com

The reason I'm creating two sub domains is that I clearly want to separate the files from the test and production instances of my application. In my application settings I can control the sub domain to use so that I can quickly switch.

It is important to note that you should name your bucket exactly the same as the setting of the CNAME alias. Once you have confirmed your changes to your registrar, it may take between 1 and 24 hours before the alias works. Once it works, you can access the image we uploaded earlier using the URL:

media.jungledragon.com/myimage.gif

This URL clearly looks a lot better!

Both the long and short URL scenarios have a major pro: not only does it use the storage of the S3 service; S3 also takes care of the HTTP traffic for retrieving the image. If you would have a page on your web server with 40 thumbnail images (all stored at S3), your web server will only need to handle the actual page request, and not the additional 40 image requests, since they will be retrieved from S3. This takes a lot of load off your own server.

Unfortunately, there is also a downside. Since the HTTP requests for the remote files are not handled by your own server, you are limited in controlling the HTTP responses. For example, you cannot enforce an IP blacklist, prevent deeplinks, do advanced caching, control bandwidth, etc. In these cases, you may want to consider the proxy scenario, which works like this:

- You point all URLs to your own server
- Your server validates the request
- Using a script, you download the requested file from S3 in realtime (or from your local server's cache, if possible)
- You place the file in a HTTP response

Currently, I would recommend against the proxy scenario (in most cases):

- It places a lot of traffic on your server
- You pay for traffic twice: once from S3 to your server, once from your server to the user
- It can make the implementation of your application more complex

In addition, note that we do have **some** control over the HTTP traffic to S3, as we will see later on in this article.

Advanced – scripted file uploads

Finally, time for some code! We want our application to dynamically place new files on the S3 service, meaning that we will provide users an upload form which transparently transfers the file to S3. There are two ways to do this:

- Using a special HTML upload form which directly uploads to S3.
- Using the REST API to transfer the file uploaded to your server to S3.

The first scenario is not really common and quite new. It consists of a regular HTML upload form with the special addition of a policy file, which controls the upload settings. You can read the instructions [here](#).

We will focus on the second scenario, however. This allows us maximum control over the transfer process. It also allows us to keep local copies of the uploaded files, if desired. It is quite likely that the amount of uploads to your web application is far lower than the retrieval of files, so there is not much hard in doing the processing in two steps.

As mentioned before, for scripted access to S3, we need to make use of the S3 API, which comes in two flavors: REST and SOAP. Since we want to keep things light-weight, we will naturally go for the REST API. Now, in order to talk to the S3 service from PHP, we will need a client library. There are a few libraries and examples created by the community available at [Amazon's resource center](#). Beware that many are buggy, poorly documented and hardly updated. In my search for a great library, I luckily stumbled upon a high quality [Amazon S3 PHP Class](#). The library is the best available, and is stand-alone, it does not require any PEAR packages, like most others do. It does require PHP 5.2.x.

So, let's get started. [Download the class](#) and place it somewhere in your application folder structure. Now, there are many ways in PHP to process the uploading of files but in any case you will have an upload form of some kind, and a place in your code where you have a handle to the uploaded file. You may also have some validations to do for the uploaded file (for example, to check the file extension). At this point, we will call the S3 class to transfer the uploaded file to S3:

```
$s3 = new S3('accessKey', 'secretKey');
if (!$s3->putObjectFile('myimage.gif', media.jungledragon.com', myimage.gif',
S3::ACL_PUBLIC_READ)) {
    // your error handling code
}
```

Let's discuss this code shortly. The first line sets up the connection to S3. We need the access key and secret key of your account. I recommend you place these in a separate configuration file and look up the values. Never share the configuration file with anyone, it contains your secret key!

The second line calls the *putObjectFile* method of the library, which is used to upload a file to S3. The first parameter is the path to your **local** file, the second parameter is the bucket name to use, the third parameter is the path to use **at S3**, and the final parameter the ACL settings of the file. As for the path to use at S3, this does not have to be a flat filename, you can things like this:

```
'images/nature/sea/myimage.gif'
```

and S3 would automatically create those directories for you.

In the code we have wrapped the call to the method in an if statement. If it returns false, we can run our error handling code. Be aware that the S3 PHP library never throws any exceptions that you can catch. Instead, it reports warnings (which you can log using PHP's error directives) and returns true or false, depending on the success of the transfer. If something goes wrong, do log and inspect the detailed error messages.

There are many more methods available in the S3 PHP library that you can use to interact with the S3 service, which are documented [here](#).

Once again congratulations! We made a huge step in integrating the S3 service into our application. Do read on, as we will now go into ways to both lowering your Amazon bill and controlling the bill, so that it does not blow up in our face.

Advanced - reducing your Amazon bill through caching

Ok. So we now have files available at S3 and we refer to them from our web pages. As discussed before, Amazon will charge you for storage, bandwidth and the number of request you make to the file. Now, this is what happens when we would request our image 5 times, for example by refreshing the page 5 times:

```
200 OK
200 OK
200 OK
200 OK
200 OK
```

What happened? Every time we requested the page, the image was freshly retrieved from S3. Let's say this was a 1 MB image. This would total in 5MB of data transfer out, and 5 requests, for which we will be billed. This would be a much better scenario:

```
200 OK
304 NOT MODIFIED (request arrived at S3, yet image not retrieved because it is not changed)
... (no request made at all, image retrieved from user's cache)
... (no request made at all, image retrieved from user's cache)
... (no request made at all, image retrieved from user's cache)
```

This would lead to 2 requests (instead of 5), and 1 MB of transfer (instead of 5 MB). It should be clear that caching dramatically reduces our S3 bill. Luckily, it is quite easy to set caching on your files. We will need to do this during the PUT, by setting the appropriate request headers and passing it into the **putObjectFile** method:

```
$requestHeaders = array();
$requestHeaders['Cache-Control']='max-age=315360000';
$requestHeaders['Expires'] = gmdate("D, d M Y H:i:s T", strtotime("+1 year"));

$s3 = new S3('accessKey', 'secretKey');
if (!$s3->putObjectFile('myimage.gif', 'media.jungledragon.com', 'myimage.gif',
S3::ACL_PUBLIC_READ, array(), $requestHeaders)) {
    // your error handling code
}
```

In the code we're setting the image to be cached for one year. Client browsers will use this to determine whether a fresh request needs to be made or not. Do note that caching behavior differs per browser and date settings of both the client and server have a say too. Be sure to test the response codes of your images when you refresh them. I use the [Firebug](#) Net panel to monitor the HTTP response codes for the images.

Again it is wise to store your cache parameters in an external configuration file, instead of hard-coding them into inline code.

Note: Do realize that when you set an **expires** date far in the future, that the only way to get a fresh file (in case you replaced the file) from S3, is to include a version number in the file name, or to append a query string to the request.

Advanced – Controlling your Amazon bill

Up until now we have uploaded our images as publicly readable, so that we can include them in our web pages. We also know that Amazon will bill us for the requests, storage and bandwidth. Are you worried yet? What if one of the following scenarios unfolds:

- A file you host on S3 publicly becomes popular, and traffic peaks
- Some joker tries to DDOS your public file

In both cases, you will be the one paying the bill. Here's some more bad news: Amazon S3 offers no way to put a limit or cap on your bandwidth or bill, despite the community requesting this feature since the origin of the service. If you too think this needs to be changed, please support my [call for action](#).

For now, there are different ways to handle this risk:

- Ignore it – simply accept this as a risk
- Observe it – obsessively monitor your (semi-realtime) bill online. You could even automate this part and build an alert system to notify you of peak usage.

- Use Signed S3 URLs

If you can get away with scenario 1 and 2, please do so. You will have pretty URLs and a low complexity in your application. If you cannot accept the financial risk, please read on to see how signed URLs work.

In the signed S3 URL scenario, we will set all our files to private. This means that we cannot refer to these files using the URL we discussed before. However, if we append a query string with a signature to the URL, the image will be publicly visible and can be used from within your web pages.

How is this useful? The signed URL also contains an expiration date. This way you will have control over how long your generated URL is exposed to your users. If you would set the expiration to a low value, users cannot take the URL and fire a lot of requests on it, as it will soon expire. Since only your application can generate new URLs, you are in (some) control of the traffic to your images, and thus in (some) control of your bill.

Before we dive into the implementation of signing URLs, it is important to acknowledge the downsides:

- Your URLs will look ugly, as they will have a long query string appended to it
- Depending on the expiration date, your URLs may change, which is troublesome when you have valid deeplinks to your files from other sites
- Signed URLs make caching more difficult, yet not impossible
- Signed URLs put a large computing load on your server

All in all signed URLs are a compromise, as they still do not offer **total** control over your bandwidth and S3 bill

With that out of the way, let us see how to implement signed URLs. A first thing to do is to properly organize your code. Before, you may have statically inserted image/file links into your web pages. In the signed URL scenario, we will need to generate these links. So instead of doing this:

```

```

We do this:

```

```

Next, we will implement the `getImageURL` method. We will be using the S3 PHP library to generate the signed URLs:

```
function getImageURL($filename) {  
  
    // calculates an image URL for display on a view/page  
    $return = "";  
    $s3 = new S3('accessKey', 'secretKey');  
    $timestamp=3600;
```

```

$return = $s3->getAuthenticatedURL('bucket', $filename, $timestamp, true, false);
return $return;
}

```

Once again, we need to instantiate the S3 object by passing in our access key and secret key. Next, we call the `getAuthenticatedURL` method of the object, with the following parameters:

- bucket: the name of your bucket, for example 'media.jungledragon.com'
- \$filename: the S3 path to your object, which you passed to the function, for example 'myimage.gif'
- \$timestamp: this represents the amount of seconds that the URL will be valid.
- Hostbucket: determines whether to generate the long domain URL (which includes the s3.amazonaws.com sub domain) or the short URL.
- HTTPS: whether the URL is HTTPS (true) or just HTTP (false)

Reminder: Set the ACL of your S3 files to private, otherwise there is no point in using signed URLs.

When we now generate the URL for an S3 image it will return something similar to this:

```

http://media.jungledragon.com/myimage.gif?AWSAccessKeyId=05GMT0V3GWVNE7GGM1R2&Expires=1231437872&Signature=dKVn114QKkETY51VmwA9VZCCqM0%3D

```

Yes, it's long and ugly, but it is currently the only way to control access to your S3 files. This URL will be valid for the amount of time specified in the \$timestamp parameter. If you would refresh the image after this period, it will simply show an "access denied" error. Since your application will always generate a new link, the image will always work within your application, but soon expires outside of it. You may notice how the querystring "Expires" is included in the URL. Do not worry; users cannot mess with this to extend the expiration date, since the timestamp is part of the signature.

You would think that we are done now. Unfortunately, we have now introduced a problem we thought we solved earlier: the lack of caching. Earlier in this article we saw how setting the request headers during the PUT operation allowed us to control a file's cache. When we compute signed URLs like we do in the example code above, it will result in a unique URL each and every time (the first part is the same, yet the querystring changes each time). Since the URL is unique each time, browsers will not use the cache, instead retrieve a fresh copy from S3, which costs us bandwidth and requests.

The way to solve this is to compute in intervals. We need to return a constant signed URL within a configurable interval. Our new code now looks like this:

```

function getImageURL($filename) {
// calculates an image URL for display on a view/page
$return = "";
$s3 = new S3('accessKey', 'secretKey');
$expireinterval = 3600;

```

```

$gracetime = $expireinterval + 10;
$timestamp = time();
$timestamp -= ($timestamp % $expireinterval);
$timestamp += $expireinterval + $gracetime;
$timestamp = $timestamp - time();
$return = $s3->getAuthenticatedURL('bucket', $filename, $timestamp, true, false);
return $return;
}

```

The code above works exactly the same as the previous example, the only difference is that it will return the same URL for a certain interval, in our case 3600 seconds. Since during this timeframe the URL is consistent, browsers will cache the file. You can simply increase the interval to allow for longer cache intervals, but do note that this increases the length of public exposure of the URL.

A final note to make in this area concerns page caching. Up until now I have assumed that you do not cache your PHP pages. It is fine if you do cache your pages, just bear in mind that you should not cache them longer than the expiration interval of your signed URLs.

Advanced - pushing it even further

Using the information above, you should have enough information to build powerful applications that seamlessly integrate S3 storage. You have ways to lower your bill, control your bill and scripted ways to interact with the service. You can scale your storage until infinity, completely transparent for your end-users!

For some, even this is not enough. In my own project (JungleDragon), I'm using all methods described above. In fact, I can configure all the things described above without touching a line of application code. Here are some things I have control over from my configuration file:

- Storage mode. I can choose between local , s3 or dual storage mode (this includes a failover mode)
- Display mode. I can configure according to which storage mode URLs should be generated
- Cache settings. For both files and URLs
- Various URL settings. Such as the sub domain, URL mode, Signing enabled, etc
- ACL settings. Whether to expose images as public or private (using signed URLs)

All of the above makes the implementation of the code examples demonstrated in this article much more complex (too complex to cover here), yet way more powerful. It allows one to switch and reconfigure the S3 interface with minimum human intervention. A storage engine of this magnitude is only needed when you require extra control, flexibility and security (through redundancy) of your files. No matter how far you take this, it at least makes sense to make as much configurable as possible, right from the start!

Advanced – related services

I think that despite some caveats, it is in fact very simple to interact with the S3 service, which delivers “storage in the cloud”. Hereby I want to tell you about two related Amazon services that could be interesting for you:

- [CloudFront](#). CloudFront is a CDN (Content Delivery Network). Simply put, CloudFront distributes files that you provide it across a globally distributed network of servers. When your end user requests it (by visiting your web page), CloudFront will automatically use the server that is closest to the user (low latency). CloudFront will also automatically cache popular content on servers close by.

This is in fact much different from S3. At S3, at bucket creation, we specify where the bucket resides (US or Europe), and that is where it will stay forever. Of course it will be on multiple servers, but never in a different location. So, if an Asian user requests an S3 file which is stored in a US bucket, there might be considerable latency, where a US user will have a low latency.

How does CloudFront work? You assign your S3 file to a so-called “distribution”. That’s it. CloudFront integrates closely with S3, yet it comes at an additional cost via a separate bill.

- [EC2](#) (Elastic Cloud Computing). EC2 has nothing to do with storage, but is interesting nonetheless. In essence this concerns the virtual hosting of server(s). It works by defining a server (could be Linux, Windows, anything you want); next you define this as an image and start an instance. You can then access this virtual server by a URL and via SSH. You can start as many instances as you want, and you can stop them at anytime. You will be charged for the computing hours only.

EC 2 can be interesting if you foresee major scaling challenges in your database and/or web server. A special note to make is that **all bandwidth between an EC2 instance and S3 is free of charge!**

Concluding

Thank you for taking the time to read this article, I certainly enjoyed writing it. Whether this really is the ultimate PHP guide to S3 I will leave up to you. Please rate and or comment on this article [here](#).