

# Storage made easy with S3

## Enter the cloud with Amazon's Simple Storage Service

Skill Level: Intermediate

[Andrew Glover \(ajglover@gmail.com\)](mailto:ajglover@gmail.com)  
Author and developer

07 Apr 2009

Amazon Simple Storage Service (S3) is a publicly available service that Web application developers can use for storing digital assets such as images, video, music, and documents. S3 provides a RESTful API for interacting with the service programmatically. Learn how to use the open source JetS3t library to leverage Amazon's S3 cloud service for storing and retrieving data.

### Introduction

The *cloud* is an abstract notion of a loosely connected group of computers working together to perform some task or service that appears as if it is being fulfilled by a single entity. The architecture behind the scenes is also abstract: each cloud provider is free to design its offering as it sees fit. Software as a Service (SaaS) is a related concept, in that the cloud offers some service to users. The cloud model potentially lowers users' costs because they don't need to buy software and the hardware to run it — the provider of the service has done that already.

#### Birth of a buzzword

Although the term *cloud computing* is by no means new (Amazon starting offering its cloud services in 2006), it started to gain buzzword status in 2008, as Google's and Amazon's cloud offerings garnered increasing publicity. Google's App Engine lets users build and host Web applications on Google's infrastructure. Along with S3, Amazon Web Services includes the Elastic Cloud Compute (EC2) computation Web service, which facilitates hosting applications on Amazon's infrastructure. Other companies have stated their intention to duke it out with Amazon and Google, including Microsoft® with its Azure effort and even Sun

Microsystems (whose cloud computing product hasn't been formally introduced to the market yet). IBM® recently announced that it is making [certain products](#) available for developer use in the Amazon EC2 environment too.

Take, for example, Amazon's S3 offering. As its name implies, it is a publicly available service that lets Web developers store digital assets (such as images, video, music, and documents) for use in their applications. When you use S3, it looks like a machine sitting on the Internet that has a hard drive containing your digital assets. In reality, a number of machines (spread across a geographical area) contain the digital assets (or pieces of them, perhaps). Amazon also handles all the complexity of fulfilling a service request to store your data and to retrieve it. You pay a small fee (around 15 cents per gigabyte per month) to store assets on Amazon's servers and one to transfer data to and from Amazon's servers.

Rather than reinvent the wheel, Amazon's S3 service exposes a RESTful API, which enables you to access S3 in any language that supports communicating over HTTP. The JetS3t project is an open source Java library that abstracts away the details of working with S3's RESTful API, exposing the API as normal Java methods and classes. It's always best to write less code, right? And it makes a lot of sense to borrow someone else's hard work too. As you'll see in this article, JetS3t makes working with S3 and the Java language a lot easier and ultimately a lot more efficient.

## S3 at a high level

Logically, S3 is a global storage area network (SAN), which appears as a super-big hard drive where you can store and retrieve digital assets. Technically though, Amazon's architecture is a bit different. Assets you choose to store and retrieve via S3 are called *objects*. Objects are stored in *buckets*. You can map this in your mind using the hard-drive analogy: objects are to files as buckets are to folders (or directories). And just like a hard drive, objects and buckets can be located via a Uniform Resource Identifier (URI).

For example, on my hard drive, I have a file named `whitepaper.pdf`, which is in the folder named `documents` in my home directory. Accordingly, the URI of the `.pdf` file is `/home/aglover/documents/whitepaper.pdf`. In S3's case, the URI is slightly different. First, buckets are top-level only — you can't nest them as you would folders (or directories) on a hard drive. Second, buckets must follow Internet naming rules; they can't include dashes next to periods, names shouldn't contain underscores, and so on. Lastly, because bucket names become part of a public URI within Amazon's domain (`s3.amazonaws.com`), bucket names must be unique across all of S3. (The good news is that you can only have 100 buckets per account, so it's doubtful there are squatters taking hundreds of good names.)

### DNS magic

Don't worry too much about the URLs to S3 assets. Via the magic of the Domain Name System (DNS) and CNAME (short for *canonical name*) records, for instance, you can map a more custom URL to that of an S3 one. By doing so, you can essentially hide the fact that you (or your application) is relying on S3!

Buckets serve as the root of a URI in S3. That is, a bucket's name becomes part of the URI leading to an object within S3. For example, if I have a bucket named `agdocs` and an object named `whitepaper.pdf`, the URI would be `http://agdocs.s3.amazonaws.com/whitepaper.pdf`.

S3 also offers the ability to specify owners and permissions for buckets and objects, as you can do for files and folders on a hard drive. When you define an object or a bucket in S3, you can specify an access-control policy that states who can access your S3 assets and how (for example, read and write permissions). Accordingly, you can then provide access to your objects in a number of ways; using a RESTful API is just one of them.

## Getting started with S3 and JetS3t

To begin using S3, you need an account. S3 isn't free, so when you create your account you must provide Amazon with a means of payment (such as a credit card number). Don't worry — there are no setup fees; you only pay for usage. The nominal fees for the examples in this article will cost less than \$1.

As part of the account-creation process, you also need to create some credentials: an access key and a secret key (think username and password). (You can also obtain x.509 certificates; however, they are only needed if you use Amazon's SOAP API.) As with any access information, it is imperative that you keep your secret key ... secret. Should anyone else get hold of your credentials and use them to access S3, you'll be billed. Consequently, the default behavior any time you create a bucket or an object is to make everything private; you must explicitly grant access to the outside world.

With an access key and a secret key in hand, you can [download JetS3t](#) and use it with abandon to interact with S3 via its RESTful API via.

Programmatically signing into S3 via JetS3t is a two-step process. First, you must create a `AWSCredentials` object and then pass it into a `S3Service` object. The `AWSCredentials` object is fairly straightforward. It takes your access and secret keys as `Strings`. The `S3Service` object is actually an interface type. Because S3 offers both a RESTful API and a SOAP API, the JetS3t library offers two implementation types: `RestS3Service` and `SoapS3Service`. For the purposes of this article (and indeed, most, if not all of your S3 pursuits), the RESTful API's

simplicity makes it a good choice.

Creating a connected `RestS3Service` instance is simple, as shown in Listing 1:

### Listing 1. Creating an instance of JetS3t's RestS3Service

```
def awsAccessKey = "blahblah"
def awsSecretKey = "blah-blah"
def awsCredentials = new AWSCredentials(awsAccessKey, awsSecretKey)

def s3Service = new RestS3Service(awsCredentials)
```

Now you are set to do something interesting: create a bucket, say, add a movie to it, and then obtain a special limited-time-available URL. In fact, that sounds like a business process, right? It's a business process associated with releasing a limited asset, such as a movie.

### Creating a bucket

For my imaginary movie business, I'm going to create a bucket dubbed `bc50i`. With JetS3t, the process is simple. Via the `S3Service` type, you have a few options. I prefer to use the `getOrCreateBucket` call, shown in Listing 2. As the name implies, calling this method either returns an instance of the bucket (represented by an instance of the `S3Bucket` type) or creates the bucket in S3.

### Listing 2. Creating a bucket on a S3 server

```
def bucket = s3Service.getOrCreateBucket("bc50i")
```

Don't let my simple code examples fool you. The JetS3t library is fairly extensive. For instance, you can quickly ascertain how many buckets you have by simply asking an instance of an `S3Service` via the `listAllBuckets` call. This method returns an array of `S3Bucket` instances. With any instance of a bucket, you can ask for its name and creation date. More important, you can control permissions associated with it via JetS3t's `AccessControlList` type. For instance, I can grab an instance of my `bc50i` bucket and make it publicly available for anyone to read and write to, as shown in Listing 3:

### Listing 3. Altering the access-control list for a bucket

```
def bucket.acl = AccessControlList.REST_CANNED_PUBLIC_READ_WRITE
```

Of course, via the API, you are free to remove buckets too. Amazon even allows you to specify in which geographical areas you'd like your bucket created. Amazon handles the complexity of where the actual data is stored, but you can nudge

Amazon to put your bucket (and then all objects within it) in either the United States or Europe (the currently available options).

### Adding objects to a bucket

Creating S3 objects with JetS3t's API is just as easy as bucket manipulation. The library is also smart enough to take care of some of the intricacies of dealing with content types associated with files within an S3 bucket. For instance, imagine that the movie I'd like to upload to S3 for customers to view for a limited time is `nerfwars2.mp4`. Creating an S3 object is as easy as creating a normal `java.io.File` type and associating the `S3Object` type with a bucket, as I've done in Listing 4:

#### Listing 4. Creating an S3 object

```
def s3obj = new S3Object(bucket, new File("/path/to/nerfwars2.mp4"))
```

Once you've got a `S3Object` initialized with a file and a bucket, all you need to do is upload it via the `putObject` method, as shown in Listing 5:

#### Listing 5. Uploading the movie is a piece of cake

```
s3Service.putObject(bucket, s3obj)
```

With the code in Listing 5, you're done. The movie is now on Amazon's servers, and the key for the movie is its name. You could, of course, override that name should you feel the need to call the object something else. In truth, the JetS3t API (and by relation the Amazon S3 RESTful API) exposes a bit more information for you when you create objects. As you know, you can also provide access-control lists. Any object within S3 is capable of holding additional metadata, which the API allows you to create. You can later query any object via the S3 API (and by derivation, JetS3t) for that metadata.

## Creating URLs to objects

At this point, my S3 instance has a bucket with a movie sitting in it. In fact, my movie can be found at this URI: `http://bc50i.s3.amazonaws.com/nerfwars2.mp4`. Yet, no one other than me can get to it. (And in this case, I can only access it programmatically, because the default access controls associated with everything are set to deny any noncredentialed access to it.) My goal is to provide select customers a way to view the new movie (for a limited time) until I'm ready to start charging for access (which S3 can facilitate as well).

Figure 1 shows the default access control in action. The XML document returned

(and accordingly displayed in my browser) is informing me that access is denied to the asset I was trying to reach (<http://bc50i.s3.amazonaws.com/nerfwars2.mp4>).

**Figure 1. Amazon's security in action**



### Leveraging S3 for time-sensitive downloads

S3 makes a lot of sense if your bandwidth and storage needs aren't constant. For example, imagine the business model I'm demonstrating — one in which movies are released at specific times throughout the year. In the traditional storage model, you'd need to buy a bunch of space on a rack somewhere (or provide your own hardware and pipe leading to it) and most likely see spikes of downloads followed by lulls of relatively low activity. You'd be paying, however, regardless of demand. With S3, the model is satisfied based on demand — the business pays for storage and bandwidth *only* when it's required. What's more, S3's security features let you further specify when people can download videos and even specify who can download them.

Achieving these requirements with S3 turns out to be quite easy. At a high level, creating a limited publicly available download for a movie requires four steps:

1. Sign into S3.
2. Create a bucket.
3. Add a desired video (or object) to that bucket.
4. Create a time-sensitive URL for the video.

That's it!

Creating a public URL is a handy feature exposed by S3; in fact, with S3, you can create a public URL that is only valid for a period of time (for instance, 24 hours). For

the movie I've just stored on the S3 servers, I'm going to create a URL that is valid for 48 hours. Then I'll then provide this URL to select customers so they can download the movie and watch it at will (provided they download it within two days).

To create a time-sensitive URL for an S3 object, you can use JetS3t's `createSignedGetUrl` method, which is a static method of the `S3Service` type. It takes a bucket name, a object's key (the movie's name in this case, remember?), some credentials (in the form of JetS3t's `AWSCredentials` object), and an expiration date. If you know the desired bucket name and the object's key, you can quickly obtain a URL as shown in the Groovy code in Listing 6:

### Listing 6. Creating a time-sensitive URL

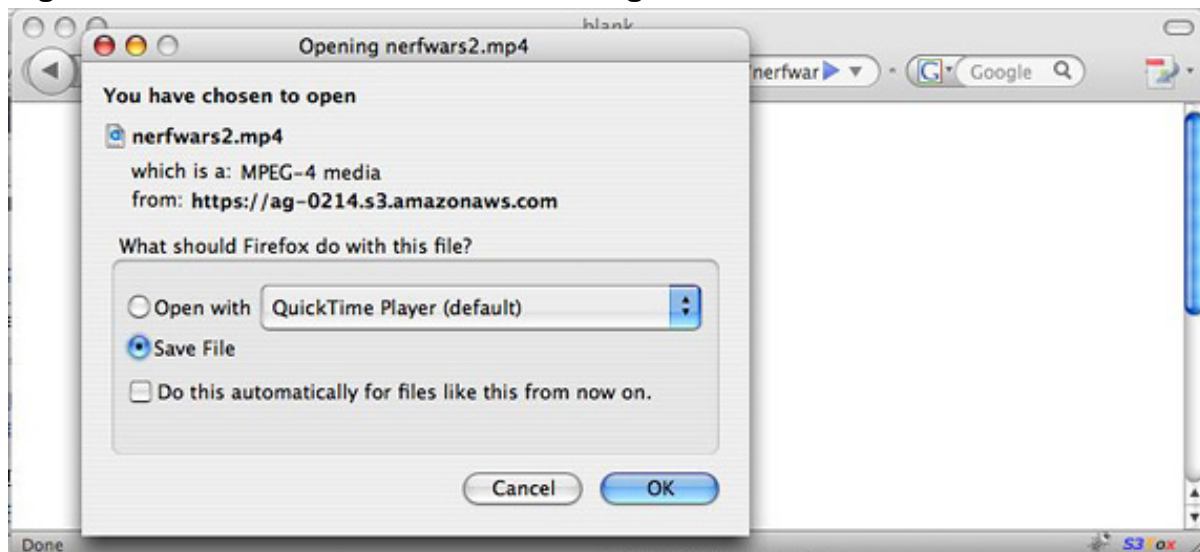
```
def now = new Date()
def url = S3Service.createSignedGetUrl(
    bucket.getName(), s3obj.key,
    awsCredentials, now + 2)
```

With Groovy, I can specify a date 48 hours in the future quite easily via the `+ 2` syntax. The resulting URL looks something like this (on a single line):

```
https://bc50i.s3.amazonaws.com/nerfwars2.mp4?AWSAccessKeyId=
1asd06A5MR2&Expires=1234738280&Signature=rZvk8Gkms%3D
```

Now, with this resultant URL, browser requests will honored, as shown in Figure 2:

**Figure 2. The URL facilitates downloading**



Wasn't this process a piece of cake? With a few lines of code, I've created a secure asset in the cloud that can only be downloaded with a special URL.

## A smart move

S3's pay-as-you-go model has some obvious advantages over the traditional storage model. For instance, to store my music collection on my own hard drive, I must buy one — say a 500GB unit for \$130 — up front. I don't have nearly 500GB of data to store, so in essence I'm paying roughly 25 cents per gigabyte for unneeded (albeit fairly inexpensive) capacity. I also must maintain my device and pay to power it. If I go the Amazon route, I don't need to fork out \$130 up front for a depreciating asset. I'll pay about 10 cents less per gigabyte and needn't pay to manage and maintain the storage hardware. Now imagine the same benefits on an enterprise scale. Twitter, for example, stores the images for its more than 1 million user accounts on S3. By paying on a per-usage basis, Twitter is spared the high expense of acquiring a hardware infrastructure to store and serve up those images, as well as ongoing labor and parts costs to configure and maintain it.

The cloud's benefits don't end there. You also gain low latency and high availability. The presumption is that the assets stored on Amazon's cloud are physically located around the globe, so content is served up faster to varying locations. What's more, because your assets are distributed to various machines, your data remains highly available should some machine (or portion of the network) go down.

In summary, the benefits of Amazon's S3 are simple: low cost, high availability, and security. Unless you're a SAN guru and enjoy maintaining hardware assets for storing digital items, Amazon probably does a better job than you. So why spend the up-front money on hardware (which loses value over time, don't forget) when you can borrow someone else's?

# Resources

## Learn

- [Amazon S3](#): Visit home base for the Amazon Simple Storage Service.
- [JetS3t](#): Learn more about the JetS3t toolkit and application suite.
- [Cloud Computing](#): Visit IBM Cloud Computing Central for a wealth of cloud resources.
- Browse the [technology bookstore](#) for books on these and other technical topics.
- [developerWorks Java technology zone](#): Find hundreds of articles about every aspect of Java programming.

## Get products and technologies

- [JetS3t](#): Download JetS3t.
- [developerWorks Cloud Computing Resource Center](#): Access IBM software products in the Amazon Elastic Compute Cloud (EC2) virtual environment.

## Discuss

- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

## About the author

Andrew Glover

Andrew Glover is a developer, author, speaker, and entrepreneur with a passion for behavior-driven development, Continuous Integration, and Agile software development. You can keep up with him at his [blog](#).

## Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.