BEN HOYT

benhoyt.com benhoyt@gmail.com

Switching from S3 to Tigris on Fly.io

February 2024

A year ago I switched my GiftyWeddings.com side project from being hosted on Amazon EC2 to using Fly.io. I had fun doing it and saved a few bucks a month.

Last week someone from Tigris contacted me, asking if I'd like to try out a private beta of the S3-compatible storage service they've built on top of Fly.io infrastructure and have integrated with the `fly` CLI. (Tigris paid me a small amount to try their beta and write about it, but they didn't have a say in the content.)

I hadn't actually heard of Tigris, but this piqued my interest – I've had a good experience with Fly.io, and I think it's great that more small companies are competing with the AWS Bezomoth.

I already use Fly.io to run the Gifty server (written in Go), but before a couple of days ago I still used Amazon S3 for file storage: user-uploaded images and SQLite backups.

It was straight-forward to switch: copy the files over, update the server's config to point to the Tigris endpoint instead of S3, and `fly deploy`. I had to do a minor code change due to Tigris handling authentication and caching differently, and I ran into a couple of quirks, but overall it was very smooth.

Creating a bucket

The first step was to create a test bucket for user-uploaded images. I went straight in with a `fly storage create` command:

```
$ cd gifty  # change to app directory
$ fly storage create --public
? Choose a name, use the default, or leave blank to generate one:
    test-gifty-registry-images
```

```
Your Tigris project (test-gifty-registry-images) is ready. See details an
next steps with: https://fly.io/docs/reference/tigris/

Setting the following secrets on gifty:
BUCKET_NAME
AWS_ENDPOINT_URL_S3
AWS_ACCESS_KEY_ID
AWS_SECRET_ACCESS_KEY
AWS_REGION

Updating existing machines in 'gifty' with rolling strategy

-------
 ✔ Machine 4d89601c6e9487 [app] update succeeded
-------
Checking DNS configuration for gifty.fly.dev
```

It surprised me that just creating a bucket automatically set a bunch of environment variables and re-deployed my app. It did this because I was in the app directory where the app's `fly.toml` config file lives, however, this behaviour seems unexpected and too magical – after all, I was creating a test bucket for local development.

This is documented, so it's kind of my fault for not reading that bit – though they probably should add "and re-deploy":

> Running the following command in a Fly.io app context – inside an app directory or specifying `-a yourapp` – will automatically set secrets on your app.

This feature could potentially be downtime-inducing, so I think it should be opt-in, for example an `--update-app` or `--add-secrets` command line option.

There's also no easy way to access the value of those automatically-set secrets. This is for security and by design, but it meant I had to create a new test bucket (outside the app directory) to get the credential for local testing:

```
$ cd ..  # change out of app directory
$ fly storage create --public --org ben-hoyt --name test-gifty-registry-i
Your Tigris project (test-gifty-registry-images2) is ready. See details a
next steps with: https://fly.io/docs/reference/tigris/
```

```
Set one or more of the following secrets on your target app.
AWS_REGION: auto
BUCKET_NAME: test-gifty-registry-images2
AWS_ENDPOINT_URL_S3: https://fly.storage.tigris.dev
AWS_ACCESS_KEY_ID: ***hidden***
AWS_SECRET_ACCESS_KEY: ***hidden***
```

Okay, that's better. Then I updated my local app config with that access key and secret, and it worked … mostly. I first had to iron out a few wrinkles, described below.

One other quirk when creating buckets – in case the `fly` CLI devs are reading this – is that when you run `fly storage create`, it prints out the environment variables in an undefined order. You can see that above: in the first example, `BUCKET_NAME` is printed first; in the second, `AWS_REGION` comes first. It's not a problem, but it is a bit weird, and I did a double-take to determine it was printing out the same keys both times.

Endpoint change

It's probably obvious that I needed to start using the new Tigris API endpoint rather than the default AWS one. When creating the S3 client, I do the following:

```
cfg := aws.NewConfig().WithCredentials(creds).WithRegion(region)
```

To use the correct endpoint, I added these lines:

```
endpoint := os.Getenv("GIFTY_S3_BACKUP_ENDPOINT")
if endpoint != "" {
        cfg = cfg.WithEndpoint(endpoint)
}
```

Note that I'm still using v1 of the AWS SDK for Go. This works a bit differently in v2.

Caching differences

When I used Amazon S3, and a user uploaded a new image for their wedding registry, I found I could get away with just uploading to the same filename (key). When the page reloaded and the image was re-fetched, the new image would appear.

I'm not sure this was a good idea even using S3, because originally S3 was only eventually consistent. However, in 2020 they switched to strong read-after-write consistency, so all GETs to a file after a PUT would give you the updated content right away.

When I starting using Tigris, and re-uploaded an image, I had to do a Ctrl-Shift-R (reload, overriding cache) for the new image to appear. Tigris seems to do more aggressive caching: looking at the response headers from an image uploaded to Tigris, it sends a `Cache-Control: max-age=3600` header by default, even though I'm not setting the header.

I'm not sure this is a good default, but because they're trying to be a CDN as well as a storage system, I can see why they chose that. It forces people to use caching techniques like content-based filenames, which are a good thing.

Update: After reading this article, Tigris added a nice page about caching that documents this behaviour and their intention on this point.

In any case, it was a chance for me to improve Gifty's handling of uploads. I changed the code to add a short content-based hash to the filename: instead of using `{registryID}.jpg` as the key, I use `{registryID}-{contentHash}.jpg`, and it fixes the caching problem:

```
// Add content-based hash to filename to break caching (12 hex digits).
hash := sha1.New()
_, _ = hash.Write(data)
key := fmt.Sprintf("%d-%x.jpg", registry.ID, hash.Sum(nil)[:6])
```

I also changed the full image URL from hard-coding `amazonaws.com` and made the URL format configurable:

```
// Old version
path := fmt.Sprintf("https://%s.s3-%s.amazonaws.com/%s",
    h.config.S3ImageBucket, h.config.S3ImageRegion, key)

// New version; under Tigris, I set S3ImageURLFormat to
// "https://fly.storage.tigris.dev/gifty-registry-images/%[1]s"
path := fmt.Sprintf(h.config.S3ImageURLFormat, key)
```

Authentication differences

When using S3, I have two buckets – one for registry images, and one for backups – and they both use the same AWS key and secret for authentication. This is probably not great (I probably should have used a different credential for backups), but in my experience it's also fairly typical for small sites to have a single key.

When using Tigris, you get a new key and secret for each bucket you create. So I had to make a minor code change to add two new S3-related environment variables. A few lines of code later, backups were working.

Copying files, and "shadow buckets"

Now that everything was working in my test environment, I wanted to switch the production site over.

For migrating from S3, Tigris has this cool feature called shadow buckets, which allows you to point your Tigris bucket at the original S3 bucket, and it will seamlessly pull from the "shadow bucket" if the file is missing from the Tigris bucket (and copy it there so it's available from Tigris for future requests).

I probably should have tried this feature, but I decided not to, as I only have a few hundred files in my largest bucket, and I wanted to do a clean switchover and avoid the dependency on S3 entirely when I was done.

So I used the `aws s3 sync` command to copy the files from S3 to my local drive, and then copy them from there to the new Tigris bucket. My site is very small scale, so the likelihood of missing any during the few minutes of switchover was extremely low.

I created two `aws` CLI profiles with the credentials for the new Tigris buckets, called `gifty-registry-images` and `gifty-backup`. The default profile still points to Amazon S3.

Here are the commands I used to copy from S3 to Tigris (I used the handy `--dryrun` option first):

```
# Copy the user-uploaded images
$ mkdir gifty-registry-images
$ cd gifty-registry-images
$ aws s3 sync s3://gifty-registry-images .
...
$ aws s3 sync . s3://gifty-registry-images --profile gifty-registry-image
...
```

```
# Copy the backup files
$ cd ..
$ mkdir gifty-backup
$ cd gifty-backup
$ aws s3 sync s3://gifty-backup .
...
$ aws s3 sync . s3://gifty-backup --profile gifty-backup
...
```

Database migration

As long as I keep paying AWS, the existing S3 URLs will keep working fine. However, as mentioned, I wanted to remove the dependency on S3.

I store the full image URL in the site's SQLite database. So after copying the files, I updated the image URLs in the database using a simple SQLite query:

```
-- First check how many S3 image URLs there are
sqlite> SELECT count(*) FROM registry
        WHERE image_url LIKE 'https://gifty-registry-images.s3%';
389

-- Perform the update
sqlite> UPDATE registry SET image_url = replace(image_url,
                'https://gifty-registry-images.s3-us-west-1.amazonaws.com
                'https://fly.storage.tigris.dev/gifty-registry-images/')
        WHERE image_url LIKE 'https://gifty-registry-images.s3%';

-- Ensure there are no more S3 URLs
sqlite> SELECT count(*) FROM registry
        WHERE image_url LIKE 'https://gifty-registry-images.s3%';
0
```

After that, the site was fully on Tigris and Fly.io, and 100% AWS-free!

Final thoughts

Performance

I did a quick, unscientific test comparing the time taken to fetch an image from the old AWS bucket versus the new Tigris one. Tigris buckets are “global by default”, so they can use the nearest Fly.io region (Sydney), which is much closer to New Zealand than the `us-west-1` region the AWS bucket was in.

As expected, I saw significantly faster download times from Tigris: for a small image, about 1s on AWS compared to 300ms on Tigris. For medium-sized images, I was seeing more like 1.3s on AWS compared to 600ms on Tigris.

Pricing

My AWS bill for Gifty – which was solely S3 usage at this point – is a few cents per month. I’m sure I cost them more in admin overhead than what I actually paid them. So pricing was not a concern for me, but still, Tigris pricing is similar to S3’s:

- Tigris: storage is $0.02 per GB per month, GET requests are $0.0005 per 1000.
- S3: storage is $0.023 per GB per month, GET requests are $0.0004 per 1000.

Either way, Tigris gave me $150 free credit as part of the beta sign-up process. That’s 7500 GB-months or 300 million GET requests – almost certainly more than my small website will use in my lifetime.

Disclaimer and durability

One thing you might want to know: when I signed up for their **early access beta**, Tigris sent me an email saying, “Please avoid using this service for production workloads during the beta!”

I asked if this was just a disclaimer to cover their backsides, or if it was more like “files will go missing randomly, and we’ll delete all data nightly during the beta”. They said it’s more of a disclaimer, and that both Fly.io and Tigris are treating this as a production platform. Obviously there’s a risk there, but I was happy with that, and they’ll be out of beta soon.

I also wondered about Tigris’s durability compared to S3, who advertise their 99.999999999% durability of objects. Tigris certainly seems to have thought about their architecture a lot, but I wonder if it’s possible to measure their durability and compare to S3’s insane numbers? Then again, is S3’s eleven 9s number actually meaningful?

The end

All that said, I had a very good experience doing the switchover, and haven’t yet had any problems – but I’m sure time will tell! I hope both Tigris and Fly.io continue to do well, and introduce some much-needed competition for the big cloud companies.