READING TIME • 16 MIN

How We Built Fly Postgres

Annie Ruygt

Like many public cloud platforms, Fly.io has a database offering. Where AWS has RDS, and Heroku has Heroku Postgres, Fly.io has Fly Postgres. You can spin up a Postgres database, or a whole cluster, with just a couple of commands. Sign up for Fly.io and launch a full-stack app in minutes!

Fly.io is an ambivalent database provider—one might even use the word "reluctant". The reasons for that are interesting, as is the way Fly Postgres works. When we relate this in conversations online, people are often surprised. So we thought we'd take a few minutes to lay out where we're coming from with databases.

We started Fly.io without durable storage. We were a platform for "edge apps", which is the very 2019 notion of carving slices off of big applications, leaving the bulk running

in Northern Virginia, and running the slices on small machines all around the world. In an "edge app" world, not having durable storage makes some sense: the real data store is in `us-east-1`, and the slices are chosen carefully to speed the whole app up (by caching, running an ML model, caching, serving images, and caching).

Of course, people asked for databases from day one. But, on days one through three hundred thirty-one, we held the line.

Somewhere around day fifteen, we grew out of the idea of building a platform exclusively for edge apps, and started looking for ways to get whole big crazy things running on Fly.io. We flirted with the idea of investing in a platform built-in database. We rolled out an (ultimately cursed) shared Redis. We even toyed with the idea of offering a managed CockroachDB; like us, Cockroach is designed to run globally distributed.

And then we snapped out of it. Databases! Feh!

Here's our 2020 reasoning, for posterity: just because we didn't offer durable storage on the platform didn't mean that apps running on Fly.io needed to be stateless. Rather, they just needed to use off-platform database services, like RDS, CrunchyData, or PlanetScale. Hooking globally distributed applications up to RDS was (and remains) something ordinary teams do all the time. What did we want to spend our time building? Another RDS, or the best platform ever for you to run stuff close to your users?

By day two hundred and ninety or so, the appeal of articulating and re-articulating the logic of a stateless global platform for stateful global apps began to wear off. RDS! Feh! Somewhere around then, Jerome and Steve figured out LVM2, gave all our apps attached disk storage, and killed off the stateless platform talking point.

Now, disk storage is just one of the puzzle pieces for giving apps a reliable backing store. Storage capabilities or not, we still didn't want to be in the business of replicating all of RDS. So we devised a cunning plan: Build the platform out so it can run a database app, build a friendly database app for customers to deploy on it, and add some convenience commands to deploy and manage the app.

We wouldn't have a managed database.

No, we have an automated database.

Postgres is a good database for this. It's familiar and just works with the migration tools baked into full-stack frameworks.

In January 2021, we soft-launched a `fly pg create` command that would deploy an automagically configured two-node Postgres cluster complete with metrics, health checks, and alerts. (The alerts were as cursed as our shared Redis.) This was a big-deal effort for us. Back in 2020, we were really small. Almost everyone here had a hand in it.

When Shaun arrived at Fly.io later that year, he took over the job of making Fly Postgres more reliable and more convenient to manage—still in hard mode: developing and shipping features that make the platform better for apps *like* Fly Postgres, and making Fly Postgres plug into those.

This post is mostly ancient history! Shaun's no longer a team of one, and lots has happened since this post should have been written and shipped. Everything still holds; it's just more and better now.

Postgres Is Really Cool All by Itself

Here's a way you can run Postgres on Fly.io: point `fly launch` at the latest official Postgres Docker image. Remove

the default services in `fly.toml`, since this isn't a public app. Provision and mount a volume. Store `POSTGRES_PASSWORD` as a Fly Secret. Deploy.

(Then `fly ssh` in and create a database and user for your app.)

If you'll only ever want this one instance, this is pretty good. If anything happens to your lonely node, though, your Postgres service—and so, your app—is down (and you may have lost data).

Here's a better setup: one primary, or leader, instance that deals with all the requests, and one replica instance nearby (but preferably on different hardware!) that stays quietly up to date with the latest transactions. And if the leader goes down, you want that replica to take over automatically. Then you have what you can call a high-availability (HA) cluster.

Postgres has a lot of levers and buttons built right in. You can deploy two Postgres VMs configured so one's a writable leader and the other is a standby replica staying up to date by asynchronous streaming replication.

What Postgres itself doesn't have is a way to adapt cluster configuration on the fly. It can't notify a replica that the primary has failed and it should take over, and it certainly can't independently elect a new leader if there's more than one eligible replica that could take over. Something else has to manipulate the Postgres controls to get HA clustering behaviour.

That's where Stolon comes in.

Postgres, WAL, and Streaming Replication

Write-Ahead Logging (WAL): Before a transaction is applied to tables and indexes on the primary (or only) instance, it's written to nonvolatile storage, in the Write-Ahead Log. This means you can afford not to write changes to every affected data file on disk after every single transaction; if data pages in memory are lost, they can be reconstructed by replaying transactions from the WAL.

Postgres streaming replication sends each WAL record along to the replica right after the transaction is committed on the leader. As the record is received, it's replayed to bring the replica up to date.

We have some heartier, SQLite-flavoured WAL content around here somewhere.

Clustering With Stolon

Stolon is a Golang Postgres manager. We chose it for a few reasons: it's open source, it's easy to build and embed in a Docker image, and it can use Consul as its backend KV store (we're good at Consul).

We spun up a Consul cluster for Fly Postgres to use, and since it was there, we also made it available for any Fly app that wanted a locking service.

Stolon comes with three components that run alongside Postgres in each instance's VM: a sentinel, a keeper, and a proxy.

- The lead sentinel keeps an eye on the cluster state as recorded by keepers in the Consul store, and decides if leadership needs to change.
- Keepers each manage their local Postgres instance, making sure it behaves as a writable leader or a read-only

replica (as dictated by the leader sentinel), and update Consul with their latest state.
- Proxies are there to route all incoming client connections to the current leader, as recorded in the store (and only if it's healthy).

If the leader instance fails, the proxies start dropping all connections and Stolon elects a new leader, using Consul to lock the database in the meantime. If both (all) your instances fail, the database is unavailable until one or the other recovers. New connections go to the new leader as soon as it's ready, without rebooting clients or changing their config.

If you've ever received a late-night email from Heroku saying your DB was replaced, you know why this is awesome.

Stolon + Consul Intensifies

Stolon is chatty as hell with Consul, and this can be a problem.

Keepers, sentinels, and proxies do all their communication via the Consul leader. If a Stolon component can't reach Consul, it repeats its request until it can. A single flapping Stolon cluster, early on, could saturate our Consul connections.

Meanwhile, if a Stolon proxy can't reach Consul, it throws its hands in the air and drops all client connections until it can. We had several Postgres outages traceable to either Consul falling over or faraway Postgres nodes not being able to connect to it.

The more Postgres clusters people spun up, the more of a problem this was.

Less Consul With HAProxy

The Stolon proxy relies on Consul to know which instance to route connections to.

But Consul isn't the intrinsic authority on who the leader is: Postgres on every instance knows its own role. If we can replace the Stolon proxy with one that can just ask the nodes who's leader, that's less load on our shared Consul cluster, and if there's trouble with Consul there's one component fewer to freak out about it.

It's not exactly supported, but it's possible to use HAProxy with Stolon, and we did.

Here's how we've got HAProxy set up:

- HAProxy listens on port 5432 on all Fly Postgres instances for read or write requests.
- When you create a Fly Postgres cluster using `fly postgres create`, it's configured with a `PRIMARY_REGION` environment variable. HAProxy gets the list of candidates from our internal DNS server using `$PRIMARY_REGION.$FLY_APP_NAME.internal`.
- Then, every two seconds, it asks the HTTP health check server on each of these nodes for its role.
- HAProxy marks the replicas as unhealthy and removes them from its list; it won't pass any requests to them.
- If the incumbent leader fails its role check by returning "replica" or "offline", or not responding at all, HAProxy drains connections from it while Stolon sorts out a new leader.
- If there's a healthy leader, HAProxy routes all requests to it, on port 5433 (where the keeper has told actual Postgres to listen).

We also added Consul clusters in a couple more regions. This spreads the burden on Consul, but crucially, it puts Consul clusters close to people's primary Postgres VMs. Network flakiness between Stolon and Consul breaks Stolon. The

internet is flaky. The less internet we can span, the happier Stolon is.

Stolon and Consul are still intense: we've been adding new Consul clusters ever since to keep up.

Here's the Fly Postgres App

We're running a few things on each Fly Postgres VM:

- Stolon keeper
- Stolon sentinel
- HAProxy
- Postgres
- a cornucopia of internal commands and health checks
- HTTP server to serve the command and health check endpoints
- Golang supervisor code

This is a pretty deluxe Postgres cluster app. You can shell into a running instance and add a database, restart the PG process, trigger a failover, run stolonctl commands directly, and more.

Our Golang supervisor, flypg, glues the other processes together and does nice things like try to recover from individual process crashes before giving up and letting the whole VM get rescheduled.

All the parts are open source; you can fork it and add PgBouncer or whatever.

Sidenote: You can enable extensions for WAL-G, TimescaleDB, and PostGIS yourself, without forking.

So that's the Fly Postgres app. You can deploy it with `fly launch` like any Fly app, straight from a clone of the postgres-ha repo. It is faster to deploy the built image straight from Docker Hub, and the image has version metadata you can use to upgrade later.

The following will create a 2-instance HA cluster that apps on your org's internal WireGuard network can connect to:

1. Copy fly.toml from the postgres-ha repo
2. Edit fly.toml to set the `PRIMARY_REGION` environment variable to match the region you're about to deploy to
3. `fly apps create` a new app
4. Create a volume
5. Set passwords as secrets on the newly-created app: `SU_PASSWORD`, `REPL_PASSWORD`, and `OPERATOR_PASSWORD`
6. `fly deploy --image=flyio/postgres:14`
7. Create a second volume
8. Add a replica by scaling up to 2 instances

Then, to let an app use this Postgres:

1. Use aforementioned in-VM commands on the Postgres leader to create a new user and database for the consuming app (you find the leader by running `fly ssh console -C "pg-role" -s` on each instance until you hit the one with the `"leader"` role)
2. Then set a connection string, containing the new user and password, as a `DATABASE_URL` secret on the consuming app.

Now I don't know if I made that look complicated or simple!

It's simple for what you get. Every instance of your postgres-ha app is a magical cluster building block! Add an instance and it automatically becomes a member of the cluster and starts replicating from the leader. If it's in the `PRIMARY_REGION`, it's eligible to participate in leader

elections. You can add nodes in other regions, too; they can't become leader, but you can read from them directly on port 5433. It's all inside the app. Get a bit fancier with the Fly-Replay header in your consuming app, and you can do your reads from the closest instance and send your writes to the primary region.

But yeah, this isn't *quite* the Fly Postgres experience. Since we expect lots of people to deploy this exact app, it was reasonable to bundle up that mild cluster-creation rigamarole into a `fly pg create` command, which is much like `fly launch` with one of our more mature framework launchers. There are similar nuggets of flyctl convenience for managing your `fly pg create`d database cluster.

An Observation

We've mentioned that continual reliance on Consul is something of an Achilles' heel for Stolon-managed clusters. It's not unique to Stolon and Consul, but a matter of needing a separate backend store for cluster state: in return for high availability and Borg-like assimilation of new instances, we accept an additional failure mode.

If you're running a single node, and you're never going to add another one to make a cluster, there's no upside to this high-availability machinery. A lone node is more reliable without any of it.

Sidenote: We did briefly deploy a leaner, standalone Postgres app for the "Development" `fly pg create` configuration. This created a poor experience for users wanting to scale up to a HA cluster—the plumbing wasn't there to do it.

But quite a lot of people do run Fly Postgres on a single instance (just for development, right??). It's still automated, and you still get the knowledge that you're in good company and deploying a maintained app.

The great thing is: if you really want the simpler setup, you can just deploy your own Postgres app. It's all apps on Fly.io!

Snapshots and Restores

You can, and should, make your own backups of data that's important to you. That being said, a restore-your-database feature is guaranteed to make people's lives easier.

If you're shipping Postgres as a Service and don't care about the underlying infrastructure, you'll do Postgres native backups, copy data files and the WAL to object storage somewhere, then restore from those. Stolon will manage this for you.

But if you're building infrastructure that can run databases, this doesn't move you forward: every database has its own mechanism for backing up individual files. Some require data dumps using specific tools, some let you copy files out of the file system, etc.

Volumes, which hold users' persistent data—for Postgres, SQLite, or whatever—are logical volumes on SSDs physically installed in our servers. We have low-level block device powers and the ability to take consistent, block-level snapshots of a disk.

So that's how we back up a Postgres database: by periodically grabbing a point-in-time version of the raw block device it's on. You recover a database by restoring this to an entirely new block device and deploying a Postgres instance to use it.

Conveniently, that approach works for pretty much anything that writes to a file system, solving backups for anything you want to run on Fly.io.

Once we got user-facing snapshot restores working for Postgres apps, we could generalize that to Volumes at large. Which is good, because people run every database you can think of on Fly.io.

This is a good example of "Postgres" work that was actually platform work with an elephant face taped on. Like persistent storage itself, shared Consul, our crap health-check alerts, image version updates, and countless "how should flyctl and the platform behave" minutiae.

Back to Fly Postgres vs. Managed Databases

So Fly Postgres is an app, not a database service. This is not a bummer: it's fascinating, I tell you! Working on this one app helps us work through what we want the platform to offer to apps and how to implement that. It's an intrinsic part of the process of building a platform you could run *your* fully managed database service on.

Meanwhile, we don't blame you if you'd actually prefer a boring managed database over our fascinating app. We love boring! Boring can be the best experience! We think the best solution to this is to partner with service providers to do integrations that really nail the Postgres, or MySQL, or Redis(!), or whatever, UX on Fly.io. After all, there's no single best database for everyone.

And for all that, heading for 2023, Fly Postgres is doing the job for lots of apps! Automated Postgres turned out more useful than we'd have predicted.

LAST UPDATED • NOV 29, 2022

Chris Nicoll
@beepcat

Shaun Davis
@davissp14

Previous post ↓
Real-Time Collaboration with Replicache and Fly-Replay