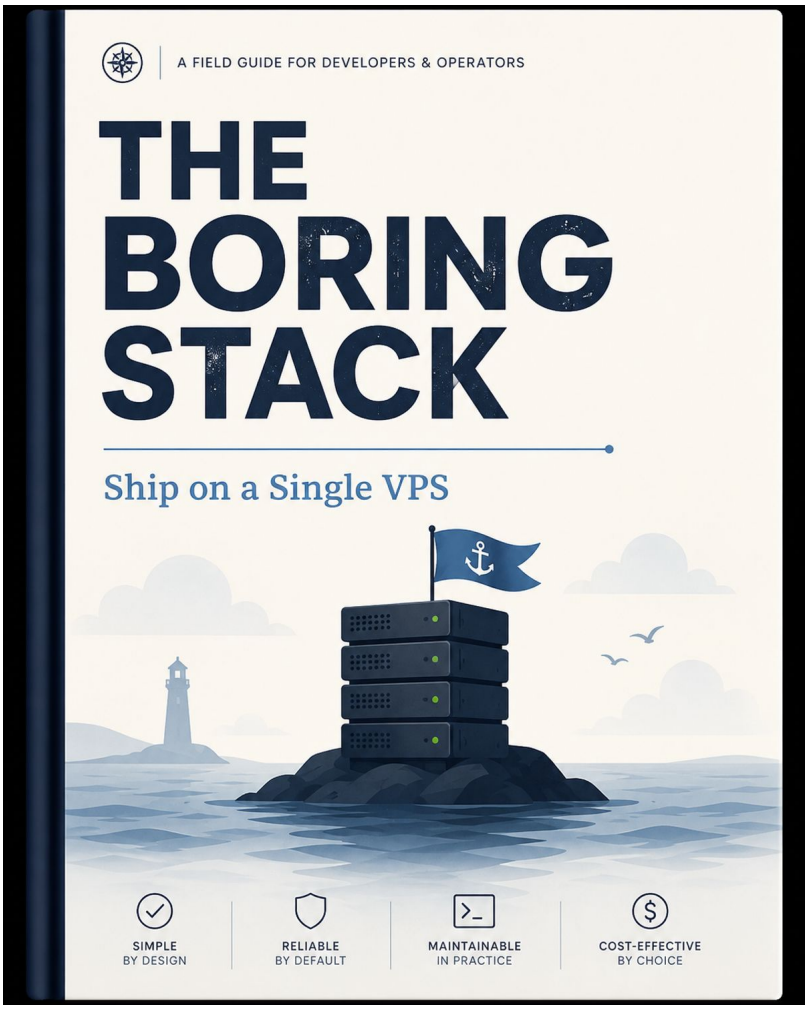

The Boring Stack: Ship on a Single VPS

Deploy and run real apps on one VPS with Coolify — no Kubernetes, no serverless. Self-hosted PaaS, databases, TLS, backups, and zero-downtime deploys on a box you own.

Roger Stringer · rogerstringer.com

June 19, 2026

Contents

Why Boring Wins

There's a default path in modern deployment, and it goes: containerize everything, orchestrate with Kubernetes, spread across a dozen managed services, wire up a CI/CD pipeline with six moving parts, and pay a platform to abstract the servers away entirely. For a company with a platform team, fine. For the rest of us — solo builders, small teams, anyone whose job is shipping the product rather than running the infrastructure — it's a tax measured in complexity you don't need and bills you don't understand.

The boring alternative: one server you actually understand. A single VPS, a tool that turns it into a Heroku-style platform, and your apps running on a box you can SSH into and reason about. No cluster, no control plane, no cold starts, no per-request pricing roulette. When something breaks you know where it is, because there's one place it can be.

The objection is reflexive: *but it won't scale.* It scales further than you think — a single modern VPS with a few cores and 8GB of RAM will happily run a real app serving real users, and we'll cover exactly when and how you outgrow it. The truth most of us don't want to hear is that the overwhelming majority of apps never need more than one good server, and the complexity we reach for "to be ready to scale" is complexity we pay for every day against a scale we never hit.

This guide builds the boring stack with **Coolify** — an open-source, self-hosted platform that gives you push-to-deploy, managed databases, automatic HTTPS, and backups on infrastructure you own. It's the missing middle between "rent a bare VPS and configure everything by hand" and "hand it all to a PaaS and pay accordingly." You get the ergonomics of Heroku on the economics and control of a box you rent for a few dollars a month.

By the end you'll have a real app deployed — with a database, TLS, backups, and zero-downtime deploys — on a server you own outright. It pairs naturally with the Mastering Directus (https://rogerstringer.com/guides/mastering-directus) and Hypermedia Stack (https://rogerstringer.com/guides/the-hypermedia-stack) guides: this is where the apps they build go to live. Let's meet the tool that makes it boring.

Meet Coolify

Coolify is, in one line, a self-hosted PaaS: you point it at a server, and it turns that server into something that deploys your apps from git, runs your databases, and handles TLS — the things Heroku and Vercel do, except on infrastructure you control and pay cloud-VPS prices for.

Under the hood it's not magic, which is the point. Coolify is a web app plus an agent that drives **Docker** on your server. When you deploy, it builds your app into a container (using Nixpacks to auto-detect your stack, or your own Dockerfile), runs it, and points a reverse proxy at it. When you add a database, it runs the official Postgres or Redis image with a persistent volume. When you add a domain, it provisions a Let's Encrypt certificate automatically. Everything it does, you *could* do by hand with Docker and Traefik — Coolify just does it for you, with a UI and sane defaults, and remembers how.

What you get out of the box:

- **Git-based deploys** — connect a repo, push, and it builds and ships; webhooks for auto-deploy on push.
- **One-click databases** — Postgres, MySQL, MongoDB, Redis and more, with persistent storage and backups.
- **Automatic HTTPS** — add a domain, get a certificate, no certbot wrangling.
- **A reverse proxy** — Traefik, configured for you, routing domains to the right container.
- **Secrets and env management** — per-app configuration kept out of your repo.
- **Backups, logs, and basic monitoring** — the operational basics, built in.

What it deliberately *isn't*: a multi-region, auto-scaling, distributed system. It's a control panel for one server (or a few), and that focus is exactly why it stays comprehensible. There's no hidden orchestration layer to debug — if you want to know what's running, you SSH in and `docker ps`, and there it is.

It's open source, which matters here for two reasons: there's no vendor sitting between you and your apps, and if Coolify vanished tomorrow your apps would keep running, because they're just Docker containers on your server. You're renting convenience, not lock-in. Now let's get a server to put it on.

Picking & Provisioning a VPS

Coolify needs a Linux server, and almost any VPS will do — the boring stack runs on boring hardware. Let's size and prep one.

Sizing. Coolify itself wants a little headroom, and then your apps need their share. A practical starting point:

- **Minimum:** 2 GB RAM, 2 vCPU, 40 GB disk — enough for Coolify plus a small app and a database.
- **Comfortable:** 4 GB RAM, 2–4 vCPU, 80 GB disk — room for several apps, a couple of databases, and build headroom (builds are the spikiest thing you'll run).
- Disk fills faster than you'd expect, mostly from Docker images and build cache, so favour more of it than feels necessary.

Provider. Hetzner, DigitalOcean, Vultr, Linode — any of them. The cheapest path is often a Hetzner box; the most familiar is DigitalOcean. Pick a region near your users and don't overthink it; you can migrate later because, again, it's just Docker. Use **Ubuntu LTS** (22.04 or 24.04) — it's what Coolify targets and what every guide assumes.

First-boot hygiene, before Coolify goes anywhere near it:

```
# As root, on the fresh box:
adduser deploy && usermod -aG sudo deploy        # a non-root user
# copy your SSH key to the new user, then lock down SSH:
sudo sed -i 's/^#\?PermitRootLogin.*/PermitRootLogin no/' /etc/ssh/sshd_config
sudo sed -i 's/^#\?PasswordAuthentication.*/PasswordAuthentication no/' /etc/ssh/
sshd_config
sudo systemctl restart ssh
sudo apt update && sudo apt upgrade -y
```

That's the security baseline the hardening chapter builds on: a non-root user, key-only SSH, no password logins, packages current. Five minutes now saves you from the internet's automated bots, which start knocking on port 22 within hours of a box coming up.

A note on where this fits: if you're also self-hosting Directus from the Mastering Directus (https://rogerstringer.com/guides/mastering-directus) guide, this same box can run it — Coolify treats Directus as just another container, which is the whole appeal. One server, several apps, one place to manage them. With the box provisioned and locked down, installing Coolify is genuinely one command.

Installing Coolify

This is the easiest chapter in the guide, because installing Coolify is one command. SSH into your server as the `deploy` user and run:

```
curl -fsSL https://cdn.coollabs.io/coolify/install.sh | bash
```

That script does the unglamorous work for you: installs Docker if it's missing, sets up Coolify's own containers, configures the proxy, and starts everything. Give it a couple of minutes. (Piping a script to bash deserves a wary eye on principle — this is Coolify's official installer from their own domain; open the URL and read it first if you want to know exactly what it does, which is good practice for anything you curl-pipe.)

When it finishes, Coolify is running on port 8000. Open `http://YOUR_SERVER_IP:8000` and you'll hit the setup screen. The first thing to do there is **create the admin account** — and because this one account controls your whole server, give it a real password and enable two-factor as soon as you're in. The first user to register becomes the owner, so do this immediately, before anyone else finds the port.

A few things to do right after first login:

- **Register your server.** Coolify installed on a server *is* a server it manages — the local server is there by default. You can add more later (Coolify can run several from one dashboard), but one is all we need.
- **Point a domain at Coolify itself.** Rather than living on a raw IP, give the dashboard a hostname (say, `coolify.yourdomain.com`) with an A record and set it in Coolify's settings — it'll grab a certificate for its own UI, so you're not administering your infrastructure over plain HTTP.
- **Look around.** Projects, servers, deploy logs, proxy status. It's a small app; ten minutes of clicking and you'll have the map.

That's the platform up. Coolify is now sitting on your server waiting to deploy something. Let's give it an app.

Your First Deploy

Time to ship something. In Coolify, apps live inside **projects** — a project is just a grouping (say, "Soapbox") that holds an app, its database, and anything else it needs. Create one, then add a resource of type **Application**.

The core of a deploy is connecting a git repo. Coolify pulls from GitHub, GitLab, or any git URL; for a private repo you connect via a GitHub App or a deploy key and Coolify handles the auth. Point it at your repo and branch, and you hit the central question of every deploy: **how does this code become a running container?** Coolify gives you two answers:

- **Nixpacks (auto-detect).** Coolify inspects your repo, recognizes it's a Node/Astro/Python/whatever app, and builds it without you writing anything. For a standard app this just works — it finds your `package.json`, runs the build, and knows how to start it. Start here.
- **Your own Dockerfile.** When you need control — a specific runtime, a build step Nixpacks doesn't guess, system dependencies — commit a `Dockerfile` and Coolify builds from that instead. It's the escape hatch, and it's the same Dockerfile you'd write anywhere.

Set the basics — the port your app listens on, the build and start commands if they're non-standard — and hit **Deploy**. Coolify clones the repo, builds the container, starts it, and streams the build log right there in the UI so you can watch it happen (and read the errors when it doesn't). The first build is the slow one; later builds reuse the cache.

Then the part that makes it feel like Heroku: **auto-deploy on push.** Flip it on and Coolify registers a webhook with your git host, so every push to the deploy branch triggers a fresh build and release. `git push` and your change is live in a couple of minutes — the whole ergonomic point of the boring stack. Your code now runs on your server, on a URL Coolify gave it.

But an app alone isn't an app — it needs somewhere to keep its data. That's next.

Databases & Persistence

Most apps need a database, and this is where Coolify earns its keep — because running a database properly (persistent storage, backups, not losing everything on a restart) is exactly the kind of fiddly operational work people get wrong by hand.

In your project, add a resource of type **Database** and pick your engine — Postgres is the default for most apps, and what the Mastering Directus (https://rogerstringer.com/guides/mastering-directus) and Hypermedia Stack (https://rogerstringer.com/guides/the-hypermedia-stack) guides assume. Coolify runs the official image, generates credentials, and — the part that matters — attaches a **persistent volume** so the data lives on the server's disk, not inside the ephemeral container. Restart or redeploy the database and the data survives, because it was never in the container to begin with.

The thing to internalize about containers and data: **containers are disposable, volumes are not.** A container can be rebuilt from your image any time — that's the whole point of them. So anything you can't afford to lose must live in a volume, never in the container's own filesystem. Coolify sets this up correctly for its managed databases, and you do the same for any app that writes files: mount a **persistent volume** for uploads, generated files, anything stateful. Forget this and the first redeploy quietly eats your users' data.

Connecting your app to the database is a matter of wiring the connection string. Coolify gives every database an **internal URL** — reachable by other containers on the same server over a private Docker network — and optionally a public one. Use the internal URL: your app talks to Postgres over the private network, the database is never exposed to the internet, and there's one less attack surface. Drop that URL into your app's environment (next chapter) and they're connected.

Redis, MySQL, and the rest work the same way — add the resource, get a managed container with a volume and an internal URL. One server now runs your app and its data side by side, privately networked, both persistent. Which raises the question of how the outside world reaches the app: domains and TLS.

Domains, TLS & Reverse Proxy

Your app is running, but it's reachable on some port on an IP address, which is no way to ship. Let's give it a real domain with real HTTPS — and this is the chapter where Coolify saves you the most genuine misery, because TLS certificates by hand are nobody's idea of fun.

The mechanism underneath is a **reverse proxy**: Coolify runs Traefik, a single process that listens on ports 80 and 443 and routes each incoming request to the right container based on its domain. Ten apps on one server, ten domains, one proxy sorting the traffic. You never configure Traefik directly; Coolify generates its config as you add domains.

To put your app on a domain:

1. **Add a DNS record** — an `A` record for `app.yourdomain.com` pointing at your server's IP. (Subdomains are easiest; a root domain works too.)
2. **Set the domain in Coolify**, on the application's settings.
3. That's it — Coolify tells Traefik to route that domain to your app's container and **automatically requests a Let's Encrypt certificate** for it. Within a minute you're on HTTPS, with auto-renewal handled forever. No certbot, no cron job, no remembering to renew before it expires and takes the site down.

A few things worth knowing:

- **HTTP redirects to HTTPS automatically** — Traefik bounces plain requests up to the secure version, so you never serve anything unencrypted.
- **Multiple domains and wildcards** are supported when you need them — an app can answer on several hostnames, and you can wildcard a subdomain if you're running many.
- **The certificate is per-domain and free**, courtesy of Let's Encrypt; Coolify just orchestrates the dance.

The relief here is real if you've ever hand-rolled nginx plus certbot for a fleet of sites. The boring stack makes "app on a custom domain with valid HTTPS" a two-field operation, and then never bothers you about it again. Your app is now properly on the internet — time to configure it safely.

Environment & Secrets

Every real app needs configuration — the database URL, API keys, feature flags — and exactly none of it belongs in your git repo. This chapter is short because the rule is simple, but it's the rule people break most, usually by committing a `.env` file "just for now" and leaking a credential.

Coolify manages configuration as **environment variables, set per application, stored on the server, never in your code.** In the app's settings there's an Environment Variables panel: add `DATABASE_URL`, `ANTHROPIC_API_KEY`, whatever your app reads from `process.env`, and Coolify injects them into the container at runtime. Your repo stays clean; the secrets live on the box, under the admin account you locked down.

The practices that keep this safe:

- **Use the internal database URL from the last chapter**, not a public one. Your app reaches Postgres over the private Docker network; the connection string never implies an internet-exposed database.
- **Mark true secrets as secret.** Coolify lets you flag a variable so its value is masked in the UI and kept out of logs. Use it for keys and passwords — there's no reason your API key should be shoulder-surfable in the dashboard.
- **Keep a `.env.example` in the repo, never a real `.env`.** The example documents what variables exist (their names, not their values) so anyone deploying knows what to set; the real values only ever live in Coolify. This is the same discipline the Hypermedia Stack (https://rogerstringer.com/guides/the-hypermedia-stack) and 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guides insist on: config from the environment, secrets out of source control, no exceptions.
- **Changing a variable triggers a redeploy.** Env vars are baked into the running container, so Coolify restarts the app to apply a change. Expect a brief blip — or use the zero-downtime deploys we're about to set up.

That's configuration handled the boring, safe way: on the server, out of the repo, masked where it matters. Now let's make deploying — which you'll do constantly — something that doesn't drop traffic.

Zero-Downtime Deploys & Rollbacks

You're going to deploy dozens of times a week, and a deploy that drops requests for ten seconds each time is a deploy you'll come to dread. The boring stack does better: it can swap a new version in without a gap, and roll back to the old one in seconds when a release goes wrong. Both fall out of how containers work.

Zero-downtime deploys work by *not* stopping the old container until the new one is ready. Coolify builds the new version, starts it, waits for it to report healthy, and only then shifts the proxy's traffic over and retires the old container. For a window of a few seconds both are running; the cutover happens once the new one is proven up. The piece that makes this trustworthy is a **health check** — you tell Coolify how to know your app is actually ready, typically an HTTP endpoint like `/health` that returns 200 once the app has booted and connected to its database:

```
Health check path:  /health
Interval / timeout: a few seconds
```

Without a health check, "ready" just means "the container started," which can be true a beat before your app can actually serve a request — so add the endpoint and point Coolify at it. Now "is it ready?" has a real answer, and the cutover waits for it.

Rollbacks are the other half of shipping fearlessly. Because each deploy is a built image, Coolify keeps the previous ones, and rolling back is selecting an earlier deployment and redeploying it — seconds, not a frantic `git revert` and rebuild while the site is down. The instinct to internalize: a bad release isn't an incident, it's a button. You ship more boldly *because* the undo is cheap — the same way the [70/30 Engineer](https://rogerstringer.com/guides/the-70-30-engineer) guide argues reversibility is what lets you move fast.

Two habits that make this real:

- **Always have a health check.** It's the difference between zero-downtime deploys and "zero-downtime" deploys that quietly serve errors during the swap.
- **Deploy small and often.** Small releases are easy to verify and trivial to roll back; a giant release that breaks is hard to diagnose and scary to revert.

Deploys are now safe and reversible. But there's one thing rollbacks can't save you from — lost data — which is why the next chapter is the most important one in the guide.

Backups & Restores

Here's the chapter that matters more than all the others, and the one people skip until the night they desperately wish they hadn't: backups. A rollback recovers a bad deploy. Nothing recovers a dropped database table except a backup you made *beforehand* — and a backup you've never tested restoring is just a hope with a filename.

Coolify has **scheduled database backups** built in. On a managed database you set a schedule (daily is the floor; more often for anything precious), and Coolify runs `pg_dump` (or the equivalent) on that cadence and keeps the dumps. Turn this on the moment you create a database, not "later" — later is the word that precedes most data-loss stories.

But a backup sitting on the *same server as the database* is barely a backup — if the server dies, disk and all, they die together. So the non-negotiable upgrade: **send backups off the box.** Coolify can push backups to **S3-compatible storage** (AWS S3, Backblaze B2, Cloudflare R2, any of them). Configure an S3 destination and your dumps land somewhere the server's death can't touch. The rule worth tattooing on: *a backup on the same machine protects against your mistakes; a backup off the machine protects against the machine.* You want both.

And the step almost everyone omits: **test a restore.** A backup you've never restored is unverified — the file might be truncated, the credentials wrong, the process broken in a way you'll only discover under maximum stress. Once, deliberately, spin up a throwaway database, restore yesterday's dump into it, and confirm the data's all there. Do it now, while it's a calm exercise, so the day you need it for real it's a procedure you've run, not one you're inventing in a panic.

A sane baseline for the boring stack:

- Daily automated database backups, minimum.
- Off-site to S3-compatible storage, always.
- Persistent-volume data (uploads, generated files) backed up too — not just the database.
- One tested restore, so you know the whole chain actually works.

Get this right and the worst day becomes a bad hour. Skip it and one mistake is permanent. Everything else in this guide is convenience; this is survival. Now, the lighter operational basics: knowing what your server is doing.

Monitoring & Logs

Once apps are live you need to know what they're doing — is the app up, why did that deploy fail, what's eating the RAM. The boring stack keeps this proportionate: you don't need a Datadog contract for one server, you need to see logs and a few numbers, and Coolify gives you both.

Logs are right there in the dashboard. Every application and database streams its container logs into the Coolify UI — live, so when a deploy fails or an app throws at 2am, you read the actual error in the browser instead of SSHing in to chase `docker logs`. This is most of debugging in practice: the app misbehaves, you open its logs, you see the stack trace. Build logs live there too, so a failed deploy explains itself.

Basic monitoring covers the vital signs. Coolify shows server-level CPU, memory, and disk usage, and whether each app and database is running. That's enough to answer the questions you'll actually ask: Is the server out of memory (the usual culprit when things get weird)? Is the disk filling up (Docker images and build cache, almost always)? Is that container up or crash-looping? For a single box, those three — RAM, disk, container status — are 90% of operational awareness.

The extras worth adding when you outgrow the basics, in rough order:

- **Uptime checks from outside.** Coolify runs *on* your server, so it can't tell you the server is down — only something external can. A free uptime monitor (UptimeRobot, Better Stack, or similar) pinging your domain tells you the box is unreachable, which the box itself never can.
- **Disk alerts.** The single most common way a boring-stack server falls over is the disk filling with old Docker images. Prune periodically (`docker image prune`) and, ideally, get warned before it's full.
- **App-level error tracking.** When you want to know about exceptions your users hit (not just crashes), an error tracker like Sentry *in the app* is the right tool — but that's a property of your app, not your infrastructure.

The philosophy stays boring: monitor proportionate to the system. One server doesn't need a distributed observability stack — it needs readable logs, three vital signs, and an outside heartbeat. Coolify gives you the first two; spend ten minutes adding the third. Which leaves the question everyone eventually asks — what happens when one server isn't enough?

Scaling the Boring Stack

Eventually someone asks the question the boring stack is supposedly bad at: what about scale? The honest, slightly deflating answer is that you scale the same boring way you did everything else — and much later than you fear.

Scale up before you scale out. The first and best move is a bigger box. Modern VPS providers will resize your server — more RAM, more cores — often with a reboot and a few dollars more a month. A single server with 8 or 16 GB of RAM and a handful of cores serves a genuinely large amount of real traffic, and vertical scaling costs you *zero* added complexity: same one server, same single place to reason about, just more of it. Exhaust this before you even consider anything distributed, because everything past here adds moving parts.

Separate the database when it's the bottleneck. The usual second step isn't more app servers — it's giving the database its own home. Move Postgres to a managed database service (or a second dedicated box) so your app server and your data server can be sized independently, and a heavy query stops starving your web app of resources. Coolify is happy to point your app at an external database URL; nothing about the deploy changes.

Add servers only when you must. Coolify can manage multiple servers from one dashboard and run more instances of an app across them behind the proxy. This is real horizontal scale — and it's also the point where the boring stack stops being quite so boring, because now you're thinking about load balancing, shared session state, and where uploads live. Most apps never reach here. The ones that do have usually earned a platform team to take it from there.

The mindset that keeps it boring: **push the growth into a bigger box and a separated database long before you push it into more boxes.** The industry's reflex is to start distributed "to be ready," and pay the complexity tax forever against a scale most apps never hit. The boring stack does the opposite — start with one server you understand, grow it the cheap way, and add complexity only when real load, not imagined load, forces your hand. By then you'll know exactly which piece is the bottleneck, because you've been able to see the whole system the entire time. Last thing: keeping that one server secure.

Gotchas & Hardening

You own this server, which is the whole appeal — and also the whole responsibility. A managed PaaS hardens the box for you; on the boring stack that's your job, and it's a short, one-time job that matters enormously. Here's the baseline, plus the gotchas that bite boring-stack runners specifically.

The security baseline (some of which you did back in provisioning):

- **No root SSH, key-only login.** Password auth and root login are how bots get in; you disabled both at provisioning — confirm they're still off.
- **A firewall.** Allow only what you need — SSH (22), HTTP (80), HTTPS (443) — and close everything else. `ufw` is the easy path: `ufw allow 22,80,443/tcp && ufw enable`. Critically, this keeps your databases' ports unreachable from the internet; they should only ever be hit over the internal Docker network.
- **Keep it patched.** `unattended-upgrades` for automatic security updates means you're not the single point of failure for a known CVE. Reboot occasionally for kernel updates.
- **Lock down Coolify itself.** It's the keys to the kingdom — strong admin password, two-factor on, dashboard behind HTTPS on a real domain rather than a raw IP over plain HTTP.

The gotchas that get boring-stack runners:

- **The disk fills with Docker images.** The number-one way these servers fall over. Every deploy leaves build cache and old images behind. Prune on a schedule (`docker image prune -af`) or you'll wake to a server that can't deploy because there's no room to build.
- **Forgetting the persistent volume.** We said it in the database chapter; it's worth repeating because the failure is silent and total: any app data not in a volume is gone on the next redeploy. If your app writes files, mount a volume, or lose them.
- **One server is one basket.** The flip side of simplicity: if the box dies, everything on it is down. That's an acceptable trade for most apps — *provided* your backups are off-site (the backups chapter) so "the server died" means "restore to a new one," not "start over." The off-site backup is what makes single-server honest.
- **Treating the server as a pet you SSH in and tweak.** The temptation is to hand-edit things on the box until it's a unique snowflake nobody can reproduce. Resist it. Let Coolify own the configuration; keep your apps reproducible from git and your data in backups, so the server itself stays disposable. A boring stack you could rebuild from scratch in an hour is a boring stack that can't really hurt you.

That's the whole boring stack: one server you understand, a tool that gives it Heroku's ergonomics, apps deployed from git with HTTPS and zero-downtime releases, databases with off-site backups, and a hardened box you could rebuild if you had to. No cluster, no per-request billing, no platform between you and your software — just infrastructure you own and can reason about, which for the vast majority of what we build is not the compromise it's made out to be. It's the right amount of complexity.

This is where the apps from the Hypermedia Stack (https://rogerstringer.com/guides/the-hypermedia-stack) and Mastering Directus (https://rogerstringer.com/guides/mastering-directus) guides go to live. Point a domain at your box and ship something.

About Roger

I'm Roger Stringer — I build things, break them, and write up what I learned so you don't have to learn it the hard way. These Field Guides come straight out of that work.

Working on something bigger? I take on a handful of fractional CTO (https://rogerstringer.com/guides/the-fractional-cto-field-guide) engagements — helping founders and teams set technical direction, build AI-powered workflows, and actually ship the hard parts. If you're wrestling with the kind of problem this guide covers and want someone in your corner who's done it before, that's exactly what I help with. Drop me a line. (mailto:roger.stringer@hey.com)

And if a guide helped, got something wrong, or you just want to compare notes, I'd love to hear from you:

- **Email** — roger.stringer@hey.com (mailto:roger.stringer@hey.com)
- **X** — @freekrai (https://x.com/freekrai)
- **GitHub** — github.com/freekrai (https://github.com/freekrai)
- **LinkedIn** — linkedin.com/in/rogerstringer (https://www.linkedin.com/in/rogerstringer)

New guides go up as I hit problems worth documenting — follow along wherever suits you.