Mastering Directus

Directus has quietly become the first thing I reach for on almost every project — and not just as a headless CMS. Point it at a Postgres database and you get an instant REST and GraphQL API, a genuinely nice admin app your non-technical teammates can actually use, granular permissions, file handling, automation, and realtime — without writing a line of boilerplate. I rarely start an app these days without standing up Directus at some point.

This is the guide I'd hand a competent developer who's never run Directus in anger: how to self-host it properly with Docker and Railway, how to model your data, how to lock it down, and how to ship real apps on top of it.

This is a living document and will be updated as Directus updates

Roger Stringer · rogerstringer.com

June 23, 2026

Contents

What Directus Actually Is

Most people meet Directus as "a headless CMS," and that framing undersells it badly. Here's the more accurate version: Directus wraps any SQL database in an instant REST and GraphQL API, a slick Vue-based admin app, authentication, file handling, automation, and realtime — without forcing you into a proprietary data model.

That last part is the whole trick. Directus doesn't own your data. You give it a Postgres (or MySQL, or SQLite) database, and it introspects your tables and exposes them. Your `articles` table is just a table. You can query it with raw SQL, point another service at it, or migrate away, and nothing about it is locked inside Directus's head. It adds its own `directus_*` system tables alongside yours for users, permissions, and config, but your application data stays plain.

This is why I use it as a *backend*, not just a CMS. On a typical project I'll model my schema in Directus, get a typed API for free, hand the admin app to a non-technical client so they can manage content, wire up a couple of automations, and spend my actual time on the frontend instead of writing yet another CRUD API and yet another admin panel. The boring 60% of most apps is just... done.

A few things worth knowing up front:

- It's **database-first or Directus-first**, your choice. Point it at an existing schema, or model in the Studio and let it write the DDL for you. Both work.
- The admin app is **Vue**, not React. If you're going to build custom interfaces or panels, that's the framework you're in. For most projects you'll never touch it.
- It's **source-available and free to self-host** for typical projects. The exact licensing terms have shifted over time, so check the current licensing page rather than trusting any blog post (including this one) on the specifics — but for an indie dev or small team self-hosting their own app, you're fine.

The rest of this guide is about running it well: getting it up on Docker and Railway, modeling data properly, locking it down, and the patterns and gotchas I've collected from using it across a lot of projects.

Self-Hosting with Docker

The official `directus/directus` image is the way to run Directus. It bundles every database driver and storage adapter, so you almost never need a custom image. You bring a database, a few environment variables, and somewhere to put uploads.

The smallest possible thing that works for local tinkering is SQLite with no external database at all — but I'm going to skip straight to the shape you actually want, because the gap between "it runs" and "it survives a restart" is where people get burned.

Here's a production-shaped `docker-compose.yml` with Postgres:

```
services:
  database:
    image: postgres:16-alpine
    volumes:
      - ./data/database:/var/lib/postgresql/data
    environment:
      POSTGRES_USER: directus
      POSTGRES_PASSWORD: a-strong-db-password
      POSTGRES_DB: directus
    healthcheck:
      test: ["CMD", "pg_isready", "--username=directus"]
      interval: 10s
      timeout: 5s
      retries: 5
      start_period: 30s
    restart: unless-stopped

  directus:
    image: directus/directus:11.17.2   # pin a real tag, never :latest in prod
    ports:
      - "8055:8055"
    volumes:
      - ./uploads:/directus/uploads
      - ./extensions:/directus/extensions
    depends_on:
      database:
        condition: service_healthy
    environment:
      SECRET: replace-me-with-a-long-random-string
      PUBLIC_URL: https://directus.example.com
      DB_CLIENT: pg
      DB_HOST: database
      DB_PORT: 5432
      DB_DATABASE: directus
      DB_USER: directus
      DB_PASSWORD: a-strong-db-password
      ADMIN_EMAIL: admin@example.com
      ADMIN_PASSWORD: change-me-on-first-login
    restart: unless-stopped
```

The environment variables that actually matter:

- **`SECRET`** — signs your access tokens. If you don't set it, Directus generates a random one on boot, which means every restart invalidates everyone's session, and a scaled-out deployment with multiple replicas will hand out tokens the other replicas reject. Set it explicitly. `openssl rand -base64 32` and move on.
- **`DB_*`** — passed straight through to Knex, so any connection-pool option you know from Knex works here too.
- **`PUBLIC_URL`** — the exact public URL Directus lives at. Get this wrong and your asset links, OAuth redirects, and resumable uploads break in confusing ways. Set it to the real HTTPS URL.
- **`ADMIN_EMAIL`** / **`ADMIN_PASSWORD`** — only used on the very first boot to create your initial admin user. You can also just fill in the onboarding screen the first time you load the Studio. Once your admin exists, drop these from the environment.

Pin your image tag. I know `:latest` is tempting. Don't. Directus ships roughly monthly and does *not*

follow semantic versioning — a minor-looking bump can contain breaking changes. Pin a specific version, read the changelog before you move it, and upgrade on purpose.

Upgrading, once you're pinned, is just: back up the database, bump the tag, `docker compose pull && docker compose up -d`. All schema migrations run automatically on container start. The back-up-first step is not optional.

Databases: Postgres is what I run in production every time — it's solid, handles JSON well, and has real full-text search. SQLite is great for local dev and tiny projects. MySQL/MariaDB and the others are supported if that's what you've got.

Storage, Caching, and Email

The bare compose file from the last chapter works, but three pieces turn it from "runs on my machine" into "runs in production": where files live, caching, and email.

File storage

By default, uploads land on the local disk at `/directus/uploads`, which is why that volume mount matters. The moment you deploy somewhere with an ephemeral filesystem — or you want to scale past one container — you move uploads to object storage.

Directus has adapters for S3 and anything S3-compatible (MinIO, Cloudflare R2, Backblaze B2), plus GCS, Azure, Supabase, and Cloudinary. S3 looks like this:

```
environment:
  STORAGE_LOCATIONS: s3
  STORAGE_S3_DRIVER: s3
  STORAGE_S3_KEY: your-access-key
  STORAGE_S3_SECRET: your-secret
  STORAGE_S3_BUCKET: your-bucket
  STORAGE_S3_REGION: us-east-1
  STORAGE_S3_ENDPOINT: s3.us-east-1.amazonaws.com
```

You can define multiple locations and pick per file; the active one is recorded on each file row. For R2 or MinIO you just point `STORAGE_S3_ENDPOINT` at their endpoint.

Caching with Redis

Caching is off by default. For anything with real traffic, turn it on — and if you ever run more than one Directus replica, Redis isn't optional, because it's also what coordinates the rate limiter and WebSocket fan-out across instances.

```
  cache:
    image: redis:7-alpine
    restart: unless-stopped

  directus:
    environment:
      CACHE_ENABLED: "true"
      CACHE_STORE: redis
      REDIS: redis://cache:6379
```

Email

Without SMTP configured, password resets, user invites, and the "Send Email" step in Flows all quietly do nothing. Wire it up:

```
environment:
  EMAIL_FROM: no-reply@example.com
  EMAIL_TRANSPORT: smtp
  EMAIL_SMTP_HOST: smtp.youremailprovider.com
  EMAIL_SMTP_PORT: 587
  EMAIL_SMTP_USER: your-smtp-user
  EMAIL_SMTP_PASSWORD: your-smtp-password
```

One gotcha worth knowing: if you configure an email transport and it's unhealthy, the deep health endpoint (`/server/health`) will start returning 503. That's a feature — it means your monitoring catches a broken mailer — but it surprises people who point a strict uptime check at it before their SMTP creds are right.

Deploying to Railway

Railway is my favorite low-friction way to get a Directus instance live, and there's an official template that provisions the whole stack — Directus, Postgres, and optionally Redis and an S3-compatible bucket — wired together over Railway's private network so your database isn't exposed to the internet.

The one-click template gets you most of the way. But there's a single gotcha that *will* bite you if you don't know it, so let's lead with it.

Railway's filesystem is ephemeral

This is the big one. Railway containers do not have persistent local disk by default. If you let Directus write uploads to `/directus/uploads` on the container's filesystem, **every redeploy wipes them.** You'll upload images, everything will look fine, and then a week later you'll push a change and all your media is gone.

Two ways to fix it:

1. **Mount a Railway Volume** at `/directus/uploads`. Simple, keeps the local-storage adapter.
2. **Use object storage** (the bucket service the template can include, or external S3/R2). This is what I'd reach for if you might ever scale past one instance.

Pick one before you put anything real in. The templates that include a bucket handle this for you, which is why I usually start from one of those rather than a blank Directus service.

The rest of the setup

- **Postgres** comes as a managed Railway service. On deploy you'll typically be prompted for your admin email and password.
- **Connecting externally** (say, to import a database dump): enable the **TCP Proxy** on the Postgres service to get a public port, do your work, then turn it off again to re-close it. Don't leave it open.
- **Custom domain**: add it under Settings !Networking and Railway provisions SSL automatically. Then — and this is the step people forget — set `PUBLIC_URL` to that domain, or your asset and auth URLs will point at the wrong host.
- **Healthchecks**: point them at `/server/ping`, which just confirms the HTTP server is up. Save the heavier `/server/health` (which checks the database, cache, and email) for deeper monitoring.

Railway vs. your own Docker host comes down to how much ops you want to own. Railway trades a little money and control for not having to think about the box. For most client projects and side projects, that's a trade I happily make.

Hardening for Production

Directus is reasonable out of the box, but "reasonable" and "exposed to the internet" are different standards. Here's the checklist I run before anything goes public.

Set `SECRET`. Covered earlier, but it belongs on every hardening list. A random per-boot secret means broken sessions and broken scaling.

Lock down CORS. Don't leave it permissive. Set an explicit origin:

```
CORS_ENABLED: "true"
CORS_ORIGIN: https://yourfrontend.com
CORS_CREDENTIALS: "true"   # only if you use cookie auth
```

Terminate HTTPS at a reverse proxy. Put Caddy, nginx, or Traefik in front. Caddy is the path of least resistance because it does automatic HTTPS with basically no config. Whatever you use, set `PUBLIC_URL` to the resulting HTTPS address.

Mind `IP_TRUST_PROXY`. Recent versions default this to `false` to prevent IP spoofing. If you're behind a trusted proxy and you rely on the real client IP — for per-IP rate limiting, say — set it to `true`. If you're not, leave it off.

Turn on rate limiting. `RATE_LIMITER_ENABLED` for per-IP limits, `RATE_LIMITER_GLOBAL_ENABLED` for an instance-wide ceiling. Back it with Redis (`RATE_LIMITER_STORE=redis`) if you run more than one replica, or each replica will count separately.

Keep secrets out of your repo. Inject env vars through your platform's secret manager, not a committed `.env`.

Guard the system collections. This is the one people miss. Never give non-admin roles create/update/delete access to the `directus_*` collections. In particular, lock down `directus_flows` and `directus_operations` — a Flow can run arbitrary code with elevated privileges, so write access to flows is effectively write access to your server. And prefer shipping extensions through your `extensions/` directory under code review over installing them from the Studio on a whim.

Know your two health endpoints. `/server/ping` returns `pong` if the HTTP server is alive — cheap, good for load-balancer checks. `/server/health` does a deep check of the database, cache, and email and returns 503 if a dependency is unhealthy — heavier, good for real monitoring. Don't point an aggressive uptime check at `/server/health` or you'll get false alarms whenever a non-critical dependency hiccups.

Data Modeling

This is where Directus earns its keep, so it's worth getting the mental model right. Collections are tables, fields are columns, items are rows. The interesting part is the relationships, because how you model them determines what your API looks like.

The relationship types

- **Many-to-One (M2O)** — the "many" side stores an actual foreign-key column. An article has one author; the `articles` table gets an `author` column. This is the only relationship that creates a real column.
- **One-to-Many (O2M)** — the inverse view of an M2O. It's a *virtual* field; it creates no column. An author's list of articles is just the M2O read from the other direction.
- **Many-to-Many (M2M)** — creates a junction collection holding both primary keys. Articles and tags, where each can have many of the other. It can also self-reference, which is how you do "related articles" or a friends list.
- **Many-to-Any (M2A)** — the polymorphic one. A single field relates to items across *multiple* collections. This is the secret to page builders: a `page` has a list of blocks, and each block can be a hero, a rich-text section, a gallery, or an image — different collections, one ordered relationship. (Heads up: filtering an M2A in queries needs collection-scoped syntax, which is fiddlier than the others.)
- **Translations** — a specialized O2M backed by a junction and a `languages` collection, for multilingual content.

There's no dedicated one-to-one; you model it as a constrained M2O when you need it.

If you've looked at this Directus instance's own schema, you've seen all of this in action — the `page_blocks` / `block_*` collections are a classic M2A page builder.

Database-first or Directus-first

You can model entirely in the Studio and let Directus write the DDL, or point Directus at an existing database and have it introspect what's there. Both are first-class. I tend to model in the Studio for greenfield apps because the relationship UI is genuinely good, and introspect when I'm adding Directus on top of something that already exists.

The one best practice people skip

Index your foreign keys. Directus does not automatically add database indexes on the foreign-key columns it creates. On small data you'll never notice. On a table with tens of thousands of rows and a few relationships, the difference between an indexed and unindexed foreign key is the difference between a millisecond query and a multi-second one. Add the indexes at the database level. This is the single most common cause of "Directus is slow" that isn't actually Directus's fault.

The API and the SDK

Every collection you make is instantly available over REST (`/items/<collection>`) and GraphQL (`/graphql`). No code, no route definitions. That's the headline feature and it never gets old.

For talking to Directus from a TypeScript app, the official `@directus/sdk` is the move. It's dependency-free and composable — you build a client out of only the pieces you need:

```
import {
  createDirectus,
  rest,
  authentication,
  readItems,
} from '@directus/sdk';

interface Schema {
  articles: Article[];
}

const directus = createDirectus<Schema>('https://directus.example.com')
  .with(rest())
  .with(authentication());

const articles = await directus.request(
  readItems('articles', {
    fields: ['id', 'title', { author: ['name'] }],
    filter: { status: { _eq: 'published' } },
    sort: ['-published_date'],
    limit: 10,
  })
);
```

Because you typed the client with your `Schema`, that whole query is type-checked end to end.

The query parameters worth knowing

- **`fields`** — dot-notation into relations, `*` wildcards if you're lazy. In production, *name your fields.* `fields: ['*']` on a collection with deep relations will happily fetch half your database.
- **`filter`** — `_eq`, `_in`, `_contains`, `_gt`/`_lt`, `_and`/`_or`, relational operators like `_some` and `_none`, and function filters like `year(date_published)`.
- **`sort`** — prefix with `-` for descending.
- **`limit`** / **`offset`** / **`page`** — pagination.
- **`search`** — full-text-ish search across the collection.
- **`aggregate`** / **`groupBy`** — counts, sums, averages without pulling rows.
- **`deep`** — apply filters/sorts/limits to *nested* relational data, with the params prefixed by `_`. This is how you say "give me these authors, but only their 3 most recent published articles."

Authentication modes

Three to know:

1. **Static tokens** — create one under Settings !Access Tokens, attach it to a user, and send it as a bearer token. Great for server-to-server and for a public frontend reading published content. `.with(staticToken('...'))`.
2. **Temporary tokens** — `client.login({ email, password })` gets you a short-lived access token plus a refresh token. For real user sessions.
3. **Session cookies** — cookie mode for browser apps, with `credentials: 'include'` on both `authentication` and `rest`.

For a typical Astro setup I use the SDK server-side with a static token scoped to a read-only role, request exactly the fields I need, and let Directus be the boring, reliable data layer behind a fast static frontend.

Access Control: Roles and Policies

This is the part most likely to trip you up if you learned Directus from an older tutorial, because the model changed. Permissions no longer live directly on roles. They live in **policies**.

Here's the current shape:

- A **policy** is a reusable bundle of permission rules — "can read published articles," "can manage own profile," that kind of thing.
- Policies attach to **roles and/or users**. A role gets a baseline set of policies; individual users can have extra ones layered on.
- **Roles** are organizational and can **nest**, inheriting policies down the tree.

Three properties of this system are worth burning into memory:

1. It's additive. A user receives the *union* of every permission from every policy that applies to them. There's no "deny" that overrides an "allow." If any policy grants access, they have it. You compose access by adding small policies, not by carving exceptions out of a big one.

2. It starts at zero. A brand-new role or policy grants *nothing*. You explicitly switch on each collection and each action — create, read, update, delete, share — that you want. This is the right default (deny by default), but it surprises people who expect a new role to at least be able to read things. It can't. You grant it.

3. There are two roles you can't really change. Public is the unauthenticated role — whatever you enable here is what the anonymous internet can do, so guard it carefully; it's off by default for everything. **Administrator** has unrestricted access that cannot be limited, and you must always have at least one admin user.

Going granular

Policies aren't just collection-level. You can scope them down to:

- **Item-level rules** — filter conditions that decide *which rows* a policy applies to. "Users can only update articles where `author` equals themselves."
- **Field-level access** — which columns a policy can read or write.
- **Presets** — values auto-set on create or update, like stamping `user_created` with the current user.

The magic ingredients here are the dynamic variables **`$CURRENT_USER`** and **`$CURRENT_ROLE`**. They're what let you write "owner can edit their own records" or build multi-tenant-style isolation where each customer only sees their own data, all in declarative filter rules with no custom code.

A practical recipe I reach for constantly: a "public read published" policy on the Public role (read-only, filtered to `status = published`), plus an authenticated policy that lets logged-in users manage rows where they're the owner. Two small policies, composed, and you've got a working content app's permission model.

Flows: Automation Inside Directus

Flows are Directus's built-in automation engine. A Flow is one **trigger**, then a sequence of **operations**, passing a data chain along as it goes. If you've used n8n or Zapier, the shape is familiar — but Flows live *inside* Directus, with access to your data and your events.

Triggers

- **Event hook** — fires on data events (item created, updated, deleted). Two flavors: a **blocking filter** that runs *inside* the database transaction so it can validate, transform, or outright cancel the operation; and a **non-blocking action** that runs after the fact.
- **Schedule** — cron. Nightly cleanups, periodic syncs.
- **Webhook** — an inbound URL, GET or POST, sync or async.
- **Manual** — a button in the Studio, optionally with a confirmation dialog that collects input. Great for giving editors a "publish to production" button.
- **Another Flow** — compose flows together.

Operations

The useful ones: **Run Script** (vanilla JS/TS in an isolated sandbox — no filesystem, no network, just data transformation), **Condition** (branch on a filter rule), **Webhook / Request** (call external APIs), **Send Email**, **Send Notification**, **Create / Read / Update / Delete Data**, **Transform** (build a custom payload), **Trigger Flow** (including running a child flow once per item in an array), **Throw Error** (cancel a blocking transaction), and **Sleep**.

Recipes I actually use

- Auto-generate a slug from a title on create.
- Email an editor when an article's status flips to "needs review."
- Nightly cron that archives stale records.
- A webhook that kicks off a frontend rebuild when content changes.

Where Flows stop and n8n begins

Flows are excellent for in-app, data-event-driven automation that you want logged and auditable next to your content. But the condition and data-chain syntax is genuinely fiddly — referencing the output of a previous step, conditions on arrays, the `$last` versus named-step business — and there's no rich visual branching or big library of pre-built third-party integration nodes.

So my rule of thumb: if the automation is *about my Directus data and triggered by my Directus events*, it's a Flow. If it's complex multi-service orchestration, long-running, or needs a hundred pre-built integrations, I run n8n alongside Directus and let a Flow's Request operation hand off to it. They're complementary, not competing.

Extensions

When Directus doesn't do something out of the box, you write an extension. The ecosystem covers two broad families.

App extensions (the admin UI, in Vue):

- **Interfaces** — custom field input widgets.
- **Displays** — how a field's value renders in lists and detail views.
- **Layouts** — alternative ways to view a collection (a calendar, a kanban board).
- **Panels** — tiles for Insights dashboards.
- **Modules** — whole custom pages in the admin app.
- **Themes** — restyle the Studio.

API extensions (the backend, in JS/TS):

- **Hooks** — react to events or filter them.
- **Endpoints** — add custom REST routes.
- **Operations** — custom steps for Flows.

And **bundles**, which group several of the above into one installable package.

Scaffolding one

```
npx create-directus-extension@latest
```

It asks what type you want and generates the project. You build to a `dist/` folder, and install locally by dropping the built extension into the `extensions/` directory you mounted in your compose file (`./extensions:/directus/extensions`). Restart, and it's loaded.

The Marketplace and a security note

The Studio has a Marketplace you can install extensions from directly. By default it only surfaces client-side extensions and **sandboxed** server-side ones. Sandboxed API extensions have to declare exactly what they're allowed to do — which scopes they need, which URLs they can call — and they can't import arbitrary third-party modules. That's a deliberate safety boundary.

If you want to install non-sandboxed server extensions from the Marketplace, you have to set `MARKETPLACE_TRUST=all`, which is exactly as much of a "are you sure?" as it sounds. In production, keep it at the default and vet anything you install by hand, because a non-sandboxed API extension runs with full server privileges. For client work I lean toward writing my own small extensions and shipping them through the `extensions/` directory under version control, so everything that runs on the server has been through code review.

Beyond CRUD: Files, Realtime, and Versioning

A few capabilities that push Directus past "API over a database" and into "actual app backend."

Files and on-the-fly image transforms

Upload a file and Directus gives you a URL you can transform with query parameters:

```
/assets/<file-id>?width=600&height=400&fit=cover&quality=80&format=webp
```

Resize, crop, re-encode, all on request, with generated derivatives cached after the first hit. You can save named presets in settings so your frontend just asks for `?key=card-thumb` instead of repeating dimensions everywhere, and you can set focal points so smart crops don't decapitate people.

One real gotcha: **transforms run inside Directus.** If your files live in a bucket in a different region from your server, every transform round-trips the original image server-ward and back, which is slow and burns bandwidth twice. The fix is to put a CDN in front of the assets endpoint, or serve genuinely static assets straight from the bucket. (Also: resumable uploads need `PUBLIC_URL` set correctly — another reason that variable matters.)

Realtime

Directus speaks WebSockets and GraphQL subscriptions, but when you self-host it's **off by default.** Turn it on:

```
WEBSOCKETS_ENABLED: "true"
WEBSOCKETS_HEARTBEAT_ENABLED: "true"
```

Then you can subscribe to create/update/delete events on a collection, with field selection, and the SDK gives you `connect()`, `subscribe()`, and reconnect handling. Everything still persists to the database — realtime is a live view over your data, not a separate ephemeral channel. Good for dashboards, presence, live content updates.

Versioning and revisions

Directus keeps a revision history of changes in `directus_revisions`, so you can see what changed and roll back. On top of that it has content versioning — the ability to create named draft versions of an item, work on them, and publish when ready, querying a specific version with a `?version=` parameter. If you're building anything with an editorial workflow — content that gets drafted, reviewed, and published — this is the machinery you want rather than rolling your own `status` gymnastics.

Insights

Last one: Insights are dashboards built from panels — metrics, charts, lists — over your own data, inside the Studio. Handy for giving a client a numbers view without standing up a separate BI tool.

Real-World Patterns and Gotchas

Everything I wish someone had told me before my first few Directus projects, in one place.

Using it as a frontend backend

For Astro or Next, use the SDK **server-side** and request only the fields you need. In Astro, fetch in `getStaticPaths()` or page frontmatter for static builds, or in a server endpoint for dynamic data. In Next's App Router, call the SDK from server components; if you're getting stale data, add a no-store cache option to your `rest()` config.

The M2A page-builder pattern maps beautifully to component-based frontends: each block type in the relationship corresponds to a frontend component, and you render the ordered list by switching on the block's collection. Model the page builder once in Directus and your editors get a real visual page builder for free.

One rough edge: authenticated *Next.js* flows are awkward, because server functions can't easily juggle the Directus session cookie. Pair it with Auth.js / NextAuth to handle the token dance rather than fighting it directly.

Performance

The big three, in order of how often they're the actual problem:

1. **Index your foreign keys.** Said it before, saying it again, because it's the number one cause of mystery slowness. Directus won't do it for you.
2. **Don't over-fetch.** Name your `fields`, paginate, and resist `*` on relational collections.
3. **Turn on Redis caching** once you have traffic.

Without these, a deeply relational query over tens of thousands of rows can take *seconds* through the API where the raw SQL would be milliseconds. With them, it's fine. The overhead is real but it's almost always tunable.

The common mistakes

- Leaving `SECRET` unset (broken sessions, broken scaling).
- Leaving CORS permissive.
- Forgetting `PUBLIC_URL` (broken asset URLs, OAuth, uploads).
- Trusting an ephemeral filesystem for uploads (looking at you, Railway).
- Running `:latest` and getting surprise breaking changes.
- Expecting a new role to have any permissions. It has none. You grant them.
- Granting write access to `directus_*` system collections.

When Directus is the right tool — and when it isn't

Reach for it when you want instant APIs over a real SQL database you control, a polished admin for non-technical teammates, granular permissions, and freedom from vendor lock-in. That's a huge swath of the apps people actually build.

Look elsewhere when you need a pure document/NoSQL store, extreme high-throughput low-latency

reads over massive relational datasets without wanting to tune anything, a React-based custom admin (the Studio is Vue), or a deep mature third-party plugin marketplace on day one. Know the edges and it'll serve you well for years — which is roughly how long it's been my default.

Schema Migrations: Dev to Prod

You modeled your schema in a dev environment. Now you need that same schema in production without clicking through the Studio a second time and hoping you matched it exactly. Directus has a proper answer.

Snapshot and apply

The core workflow is two commands. On your source (dev) instance:

```
npx directus schema snapshot ./snapshot.yaml
```

That writes your entire data model — collections, fields, relationships — to a YAML file. Commit it to version control. Then on the target (prod) instance:

```
npx directus schema apply ./snapshot.yaml
```

Directus diffs the snapshot against the target's current schema and applies the difference. There are matching API endpoints too — `/schema/snapshot`, `/schema/diff`, `/schema/apply` — if you'd rather drive it over HTTP from a deploy script.

One safety detail: the diff is guarded by a version-and-vendor hash. Directus will refuse to apply a snapshot taken from a different Directus version or a different database vendor unless you explicitly pass `force=true`. That guard is there for a good reason; if you find yourself wanting to force it, stop and make sure you understand why the versions differ first.

The discipline that makes this work

The rule I hold to: **only ever change the data model in dev.** Never edit collections or fields directly in production through the Studio. Model in dev, snapshot, commit, apply on deploy. The moment prod's schema drifts from your version-controlled snapshot because someone "just added one quick field" in the live admin, you've lost the thread, and the next `apply` is going to surprise you.

For code-based migrations — data backfills, transformations that go beyond schema shape — there's a `migrations/` directory you apply with `directus database migrate:latest`, which runs on container start anyway.

Treat your schema like code, because with the snapshot file, that's exactly what it is. That's the note to end on: Directus rewards being treated like a real part of your codebase rather than a magic admin panel off to the side. Model deliberately, version your schema, lock it down, and it'll quietly run the boring 60% of your app while you build the parts that matter.

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.