./blog.sh

What it does, and why it doesn't do the rest.

A blog engine you run from a terminal. Posts are files, the site is a build, and the comments live on the Fediverse.

Version 1.1 · MIT licensed
blogsh.app · github.com/DanielSnor/blog.sh

Twenty-one chapters, assembled from the articles published on blogsh.app.

Contents

Part I — The idea

01 ./blog.sh — a minimalist blogging system
02 Why I built this (38 seconds)
03 Built for one blog on purpose
04 What ./blog.sh doesn't do

Part II — Writing

05 No database. One JSON file per post.
06 Some posts are conversations
07 Some posts are downloads
08 $ ./blog.sh add
09 The menu got shorter

Part III — Publishing

10 Three drafts, three mornings
11 The post that stays
12 An old link still knows the way
13 Comments I don't host
14 Why not X, and why Threads waits

Part IV — Media and migration

15 Fourteen years, and every image came home
16 The photo your iPhone won't show anyone

Part V — Build, deploy, appearance

17 No gems (one asterisk)
18 Seven keys
19 A deploy that refuses to nuke your site
20 The guard that switched itself off
21 Add your language. It's one file.

Part I

The idea

What this engine is, who it is for, and the long list of things it deliberately refuses to do.

01 · start-here

./blog.sh — a minimalist blogging system

./blog.sh is a blog engine you run from a terminal. Posts are files, the site is a build, and the comments live on the Fediverse — there is no database anywhere in that sentence.

Three things make it different from the other five hundred static site generators:

- No database. One post is one JSON file. Backup is `tar` , history is `git` , leaving is a `for` loop.
- No dependencies. Ruby stdlib and bash. No gems, no npm, no lockfile. Clone and run.
- No comment system. Every post is announced on Mastodon or Bluesky; replies to that announcement are the comments, loaded by your browser from the public API.

This is what it looks like:

Light mode

Dark mode

Try it in four lines:

```
git clone https://github.com/DanielSnor/blog.sh.git
cd blog.sh && cp config/site.yml.example config/site.yml
./blog.sh add
./blog.sh preview
```

Everything else on this site is a chapter, not a feature list: authoring · markdown · deploy · comments · appearance · migration · i18n · philosophy. The reference manual is the README, and the whole Markdown dialect fits on one page — generated by the parser itself, so it can't lie.

MIT licensed. Source, issues and the occasional opinion: github.com/DanielSnor/blog.sh

02 · philosophy

Why I built this (38 seconds)

Audio. Why I built this — 38 seconds, one take, recorded on a phone — plays on blogsh.app

Thirty-eight seconds, recorded on a phone, no edit. The written version is below if you'd rather read.

There are hundreds of static site generators. I wrote another one anyway, and the reasons fit in less than a minute.

My posts lived on big platforms for fourteen years. Some of those platforms changed the rules. Some just disappeared. I spend my whole day in a terminal, so I wanted to write my blog there too. And a small personal blog does not need a database — files are enough.

So: posts are files, the site is a build, and comments live on the Fediverse. That's the whole idea.

Built for one blog on purpose

This is not a general-purpose engine, and pretending otherwise would waste your afternoon.

./blog.sh was built around exactly one deployment: one author, one archive, a terminal, and comments on the Fediverse. That's usually the part a project hides on the "about" page. I'd rather lead with it, because it's also the explanation for everything opinionated about this tool. A general-purpose generator solves the general case and then makes you configure your way back to your specific one. This engine started at the specific case and never left.

So here's the honest table:

	./blog.sh	Hugo	Jekyll	Ghost	WordPress
Database	none	none	none	required	required
Dependencies	Ruby stdlib*	one binary	gem ecosystem	Node + MySQL	PHP + MySQL
Comments	Fediverse, built in	bring your own	bring your own	native + members	native + plugins
Deploy	6 backends, guarded	bring your own	bring your own	it's a server	it's a server
Multiple authors	no	yes	yes	yes	yes
Plugins / ecosystem	no, on purpose	huge	huge	good	unbeatable
Community size	population: 1	enormous	large	large	a third of the web
10,000-post build	slower	seconds, wins easily	slow	n/a	n/a
Newsletter	no	no	no	yes, native	via plugins

(the asterisk is documented — one optional widget needs `rexml` on some distros)

Read the losing rows first; that's what they're there for. Hugo on ten thousand posts is dramatically faster, and if raw build speed on a huge archive is your constraint, take Hugo and my blessing. Ghost has a real newsletter business built in. WordPress has an ecosystem nobody catches up to, including a plugin for whatever you just thought of. A comparison table where the home team wins every row isn't a comparison — it's an ad with borders.

What the table can't show is the shape of the wins. "No database" isn't one row — it's why backup is `tar`, why hosting is any static server, why nothing needs a security patch on a Tuesday. "Comments built in" means the announcement toot, the reply thread and the unpublish cleanup are one feature, not three services on three bills.

Who this is for: one writer with a terminal, an archive they want to own outright, and an account on the Fediverse or Bluesky.

Who this isn't for: teams, clients, editorial workflows, anyone who needs a web admin, anyone whose readers must comment without leaving the page.

Built for exactly one blog — which is why it's this opinionated, and why it fits the blogs shaped like it unusually well.

04 · philosophy

What ./blog.sh doesn't do

A feature list tells you what a tool wants to be. This is the other list.

Features that aren't there — and aren't coming

- [ ] Multiple authors — the engine is built around one person's workflow, and every multi-author feature taxes the single author with roles, permissions and attribution UI
- [ ] Plugins — a plugin API is a promise to keep internals stable forever; a tool this small should be forked, not extended
- [ ] A WYSIWYG editor — your `$EDITOR` took you years to configure; I'm not going to compete with it in a textarea
- [ ] Its own comment system — storage, moderation, spam, GDPR, and a database, all to rebuild what the Fediverse already does in public
- [ ] Themes as downloadable packages — seven colour keys and your own banner get you further than a theme marketplace, at zero marketplace

Unchecked checkboxes, as rendered by the engine's own task lists. They'll stay that way.

What the parser refuses

The Markdown dialect turns things down too — each considered, each rejected, reasons attached:

- Underscore italics (`_like this_`)
  - underscores live inside ordinary text: `file_names` , `snake_case`
  - the cost of supporting them lands on everyone who never uses them, as surprise italics mid-identifier
- Code blocks indented with spaces
  - collides head-on with nested-list indentation
  - the backtick fence says what it means
- Headings underlined with `===`
  - a line of dashes already means a horizontal rule and the frontmatter delimiter
  - one more meaning for the same characters is one too many
- Nested quotes (`>>`), reference links (`[text][id]`), footnotes
  - each adds parser states whose main output, in practice, is surprising rendering of innocent text

Notice the pattern: it's the same reason every time. The cost of a feature lands on the people who don't use it — as complexity, as surprise, as one more way for plain text to mean something you didn't intend. A feature has to clear that bar, not just be useful to someone, somewhere, occasionally.

The upside of no

Every "no" above is load-bearing. No plugins is why a fresh clone runs. No WYSIWYG is why the whole thing works over SSH. No own comments is why there's no database, and no database is

why backup is `tar` and migration is a `for` loop.

This list isn't an apology. It's most of the design.

Part II

Writing

One file per post, a small Markdown dialect, and an editor you already have.

05 · content, build

No database. One JSON file per post.

Here are three lines of Markdown, and here is what the file on disk looks like after you save them.

What you type:

```
Posts are **files**. The build is a loop over a folder.

There is no step three.
```

What lands on disk, one JSON file per post:

one post on disk

```
{
  "slug": "no-step-three",
  "title": "No step three",
  "date": "2026-07-31T09:00:00+02:00",
  "state": "published",
  "tags": ["content"],
  "content": [
    {
      "type": "text",
      "text": "Posts are files. The build is a loop over a folder.",
      "formatting": [
        { "type": "bold", "start": 10, "end": 15 }
      ]
    },
    {
      "type": "text",
      "text": "There is no step three."
    }
  ],
  "source": { "platform": "manual" }
}
```

(Trimmed for the post — a real file carries a couple more housekeeping fields — but this is the real shape.)

Look at where the bold went

The text is plain text. The bold is a pair of offsets pointing into it — not `<strong>` tags baked into a string, not a Markdown blob waiting to be parsed again. That one decision quietly pays for half the engine:

- Search indexes the text as-is. No stripping tags, no "why does searching for strong match every post".
- The build never parses Markdown. Parsing happens once, at save time; the build just renders blocks it already understands.
- Importers from seven platforms all target this same schema — which is why a new import source is an adapter with three methods, not a fork of the parser.

Round trips, with a seatbelt

`edit` converts the JSON back to Markdown, you edit, it parses again on save. And when a post contains something Markdown can't express — an embed imported from Bluesky, a link card — the editor warns you before saving would flatten it, and asks. Nothing gets silently thrown away in the round trip.

The boring finale

One post is one file, and I mean that as infrastructure, not poetry:

- backup is `tar`
- history is `git`
- migrating away from ./blog.sh is a `for` loop over a folder of self-describing JSON

And one consequence worth saying out loud: the deployed site is just a build artifact. The working copy — content, media, config — lives on your machine, entirely under your control, no matter where the web part runs or what happens to it. Host disappears, server catches fire, provider triples its prices? Nothing of value was there. You point the deploy at a different backend and the site exists again, byte for byte. The server was never the blog. Your folder is.

An engine should be easiest to leave the same way it was easiest to join. This is that, in one file per post.

06 · markdown, authoring

Some posts are conversations

a chat block, as the engine renders it

```
Petr: I need a dialogue in a blog post.
./blog.sh: Write it in a chat fence. Name, colon, line.
Petr: That's it?
./blog.sh: A line without a colon continues the one above it.
Petr: And Markdown inside?
./blog.sh: Works. Bold, links, code — the usual.
Petr: Every other engine wants a plugin for this.
./blog.sh: Every other engine wants a plugin for tables, too.
```

The source of that exchange is exactly what you just read — speaker's name, a colon, the line:

```
Petr: I need a dialogue in a blog post.
./blog.sh: Write it in a chat fence. Name, colon, line.
Petr: That's it?
```

Wrap those lines in a code fence tagged `chat` and you're done.

You just read the feature. That's the whole documentation.

07 · content, markdown, authoring

Some posts are downloads

A photo in a post is a bare filename in an image line. It always has been:

```
![A view from the window](window.jpg)
```

So a file attached to a post is the same line without the exclamation mark:

```
[Reading notes, 2025](reading-notes.pdf)
```

A whole line that is nothing but a link, pointing at a bare filename with a known extension, is an attachment. It gets staged through `incoming/` like a photo, stored next to the post like a photo, and it never leaves your site.

A line with a URL in it stays what it has always been: a link. The difference is the bare filename — the same rule photos already follow, so there's nothing new to learn.

Rendered as a card, not a link

An attachment renders as a card carrying the label, the extension and the size. The size is there because a download deserves to say what it costs before you commit to it — a 40 MB PDF on a phone on mobile data is a different proposition from a 200 kB one, and the reader should get to make that call.

Over about 50 MB you get a warning when you save. Not a refusal — sometimes a big file is exactly the point — just a note that you're about to ask a lot of somebody.

The extensions that count: `.pdf` , `.zip` , `.tgz` , `.epub` , `.txt` , `.md` , `.ics` , `.gpx` , `.csv` . A whitelist, because the engine should only publish files it can honestly describe.

And a new kind of post

A short line plus a file makes the post a document — the same way a short caption plus a photo makes an image post. Write more than a caption and it's an article that happens to have an attachment.

The first document you publish makes DOCUMENTS appear in the navigation. Nothing to configure: the menu has shown only the types a site actually has since 1.0, so the section arrives with its first tenant and would leave with its last.

Publish and forget. That was the whole brief: a PDF, a GPX track, a calendar file — things worth putting somewhere permanent without writing an essay around them.

08 · authoring

$ ./blog.sh add

Video. From an empty terminal to a shareable draft, uncut. — plays on blogsh.app

One take, fifty-seven seconds, no commentary and no cuts. A phone with a folding keyboard, and `./blog.sh` running over SSH on the screen behind it.

What you're watching:

1. `./blog.sh` with no arguments — a menu, arrow keys, one-key answers. 2. `add` opens the editor. The post is Markdown with a tiny frontmatter on top. 3. Saving builds the draft and deploys it to a hidden preview URL — which is where the video ends: the same draft open in a browser on the desktop, and then on a tablet.

That ending is the point. The draft is already a real page on the real site, behind an unguessable URL — reviewable from any device that can open a link, shareable with anyone you want feedback from. The one step the video doesn't show is a single menu choice — publish — after which the announcement toot goes out on its own.

No admin panel logged into, no step you couldn't do over SSH from a train.

Two footnotes for the sceptics. The real-time pace is the argument — that's why there are no cuts. And the whole wizard degrades gracefully: run it in a pipe and it falls back to plain line prompts with no escape sequences, so everything you just watched can also be scripted.

09 · authoring

The menu got shorter

The wizard used to list ten things you could do. It lists five now, and the engine can do more than it could before.

That's not a paradox, it's an admission: eight of those ten were not activities. They were operations on a post, wearing the costume of a menu item.

Everything about a post, in one place

`./blog.sh props <slug>` — or, in the wizard, pick a post and press `v` — shows what there is to know about it. State, type, tags, whether it's pinned, whether it's been announced and where. If it's scheduled, the whole publishing queue.

And the actions live there too, next to the facts they act on. A draft offers publish and schedule. A published post offers unpublish, re-announce, pin or unpin, rename, delete.

You no longer go to a menu, choose "unpublish", and then hunt for the post. You go to the post and see that unpublishing is one of the things it can do right now.

What stayed where it was

Type and tags are still edited in the frontmatter of `edit`, prefilled with their current values, because they're text you write. The pin is a switch rather than a value, so it toggles right in the dialog with `c` — and the header line keeps working, for anyone who'd rather type it.

The five that remain are the ones that really are activities: write, edit, import, list, rebuild.

One guard worth mentioning

The dialog acts on the post it read when it opened. If the scheduled-publish cron fired in between — you opened the properties of a draft, went for coffee, came back at 09:31 — an action taken on that stale copy could have reverted a post the cron had just published, dropping its announcement URL along the way.

`edit` already guarded against exactly that. Now the dialog's actions do too. It's the kind of bug you only find by asking "what if something else touched this file while I was looking at it?", which is a question worth asking of every screen that holds state open.

If you script

The wizard's numbering changed with the shorter menu, so a scripted `printf "4\n" | ./blog.sh` picks something different than it did in 1.0. The CLI commands are the stable interface — none of them changed, and none of them will.

Part III

Publishing

When a post goes out, where it sits afterwards, and who gets to talk about it.

10 · authoring, content

Three drafts, three mornings

Writing doesn't arrive evenly. You get an evening where three posts want out, then nothing for a week. Publishing them the way they were written means three posts in one night and silence afterwards — which is a worse blog than the same three posts spread across three mornings.

So a site can now say when it usually publishes:

```
publishing:
  slots:
    - "mon 09:30"
    - "wed 09:30"
    - "fri 09:30"
```

Or, if you write more than that, a single `"daily 09:00"` .

With that in the config, scheduling a draft offers you the next slot no other scheduled post already occupies. Three drafts written in one evening are offered Monday, Wednesday and Friday, in that order, without you counting days.

It only ever offers

This is the part I care about most, because a queue that takes decisions away from you is a queue you fight.

Typing a date overrides the offer, always. A post you hand-schedule for 14:17 on a Tuesday blocks nobody — it isn't in a slot, so the queue routes around it. And nothing ever moves a post that already has a time. The slots suggest; you decide.

Without the key in your config, the prompt is exactly the one that was there in 1.0.

The offer explains itself

An offer of Sunday, on a site with a Saturday slot, reads like a queue that skips Saturdays. So the offer names the slots it walked past and who is holding them:

```
next free slot: Sun 09:30
  Sat 09:30 taken by "the-post-that-stays"
```

A scheduled draft's properties print the whole queue for the same reason. A queue you can't see is a queue you don't trust.

The unglamorous half

Two drafts landing in the same slot would be worse than no queue at all, so the slot search skips anything already claimed — and it does that across a daylight-saving change too, which is where the first version double-booked. Clocks that move an hour twice a year are the reason scheduling code is never as short as it looks.

11 · content, authoring

The post that stays

A blog's front page is the only page that's a statement. Everything else is chronology doing its job.

So `pinned: true` in a published post's header now holds a copy of it at the top of the first listing page. That's the whole feature, and the interesting part is everywhere it deliberately doesn't apply.

Only the front page

Type listings, tag listings, the RSS feed and the sitemap stay strictly chronological. A pin is a claim about what a visitor should read first, not a rewrite of when things were written. Someone subscribed to the feed gets posts in the order they happened, and a pinned post doesn't jump their queue every time it's toggled.

What happens as it ages

While the post is still recent enough to sit on page 1 by date, it appears exactly once — pinning it doesn't print it twice on the same screen. Once it has aged onto page 2, it appears in both places: at the top of page 1, and in its own chronological spot on page 2.

That second half is the part I'd have got wrong if I hadn't thought about the archive. A pinned post that vanished from its own date would leave a hole in the record, and the record is the point.

What it costs

One or two files in a deploy. Pagination here is anchored — page 2 doesn't renumber itself because page 1 changed — so toggling a pin doesn't reshuffle the archive behind it.

Visually it costs even less: the pinned copy carries a small mark in the corner of its date badge, drawn in the neutral colour pair the badge already inverts to on hover. No new colour entered the palette. Seven keys are still seven keys.

Turning it on

Put `pinned: true` in the post's header, or press `c` in the properties dialog. Every list and picker marks it `[PINNED]` afterwards, so you can't forget which post is currently doing the talking.

12 · content, build

An old link still knows the way

The first post on this site is about a photo that disappeared from a 2012 blog entry while the link to it kept working. Link rot is the reason this engine exists, so an engine that quietly broke its own URLs would be a bad joke.

Renaming a post changes its address. That's unavoidable — the slug is the URL. What's avoidable is the old address turning into a 404.

What renaming does now

Rename a post — `r` in the properties dialog — and the post records its old address inside itself. The build then keeps a one-page redirect standing at every address the post has ever had.

The link in a two-year-old toot keeps resolving. The link somebody put in their own blog post keeps resolving. You get to fix a slug you regret without the fix costing you every reference to it that already exists in the world.

The redirects belong to the post

They live and die with it. Unpublish the post and its redirects come off the site with it — a redirect pointing into nothing is just a slower 404. Publish it again and they come back, because the old links didn't stop existing while the post was away.

That's the same rule the announcement toot follows, and for the same reason: everything a post owns should leave and return with it, or the site accumulates debris nobody remembers creating.

The small print

The redirect list is the post's own history, so it never contains the post's current address — a post redirecting to itself is a loop, and the build says so if a date edit across a year boundary ever creates one.

And slugs have a length limit now, with a plain message when you exceed it. The first version handed you a filesystem error instead, which is a technically accurate way of saying nothing useful at all.

13 · comments

Comments I don't host

The star, boost and reply counters sitting right under this post's title belong to a toot, not to this site. And the comments at the bottom of this page are that toot's replies. That's the whole system — here's how it works.

When a post is published, the engine announces it on Mastodon or Bluesky — one network per site, never both; the build refuses a config with two, because comments live in exactly one place. Replies to that announcement are the comments: your browser loads the thread straight from the network's public API, no login, no key, no middleman. The counters under the title are the same numbers, read from the same toot.

What falls out of that

No comment database — nothing to migrate, back up, or moderate at 2am. No spam filter, because the network already has one, and it's better funded than mine. The comment count is just the reply count. And moderation is your instance's moderation: block, mute and report all work exactly where the conversation actually happens.

The comments also belong to the people who wrote them, in their own timeline, under their own name. A reply here is a real post by a real account — not a row in my database wearing a nickname.

What it costs

Honesty first: if you're not on the Fediverse or Bluesky, you don't get to comment. That's a real cost and I won't pretend otherwise. You can still read the thread — it's public — and the bar for joining is an email address. For a personal blog, I'll take that trade over hosting a login system every day of the week.

Two details that earn their keep

`unpublish` deletes the announcement too. A toot pointing at a dead URL helps nobody, so taking a post down takes its thread anchor down with it.

Imported posts don't announce. The auto-announcement has a 24-hour window around "now" — so migrating a thousand posts from your old blog doesn't turn into a thousand-toot night for your followers. (You can still announce any post manually, whenever it deserves it.)

Try it on this very post

This is the part I can't fake, which is exactly why it's the demo. Reply to this post's toot — what's missing here, what would make you use this engine, or what convinced you not to — and your reply appears below, hosted by no one, moderated by your instance, owned by you.

14 · philosophy, comments

Why not X, and why Threads waits

> Post read: $0.005 per resource. Post creation: $0.015 per request — $0.200 when it carries a URL. Free public read access: none. — X API pay-per-use pricing, docs.x.com, checked July 2026

Two networks are in. Two are not — and here's the whole reasoning, because "we decided against it" is not a reason.

./blog.sh announces every published post on Mastodon or Bluesky, and the replies to that announcement become the post's comments. For that to work, the engine needs two things from a network: a way to post, and a way for a browser to read a public thread. Cheap, ideally free, forever — because a personal blog is a decades-long project with a hobby budget.

X: no

Look at that price list again. Announcing a post with a link costs real money. Loading the reply thread — which happens every time anyone scrolls to the comments — costs real money, per read, forever. On a personal blog, that's a subscription to your own comment section, billed by your readers' curiosity.

There's no engineering answer to a pricing decision. So: no.

Threads: not yet

Threads is genuinely feasible, and the design is sitting in a drawer. But look at what "feasible" means here:

> Publishing and reading replies to your own posts works — server-side only, through a registered Meta developer app, with OAuth and 60-day tokens. — the Threads API docs, summarized

In practice that's a cron job refreshing tokens so they never silently expire, and comments cached into JSON with roughly a half-hour delay — instead of the live thread Mastodon and Bluesky hand any browser for free. It's all buildable. It's just a lot of standing machinery for a network nobody has asked me about yet. The day someone does, the drawer opens.

Scraping Threads: no, and firmly

The tempting shortcut — skip the API, scrape the web app — fails three ways at once. The app's internals change constantly and no community project maintains a stable interface to them. Meta blocks datacenter IPs and forbids automated collection in its terms, so shipping a scraper in a community engine means handing every user a feature that breaks without warning and can put their account at risk. And the paid scraping services fail the same test with extra steps: per-request pricing, your data flowing through a third party, and a dependency on someone else's ongoing legal cat-and-mouse.

The scoreboard

Mastodon and Bluesky are in because they pass the boring test: free to post, free for a browser to read a public thread, no tokens with expiry dates in the critical path. Any network that starts passing it is welcome. Any network that stops — well, now you know the exit criteria too.

Part IV

Media and migration

Bringing fourteen years of somebody else's platform home, and keeping the photos readable once they arrive.

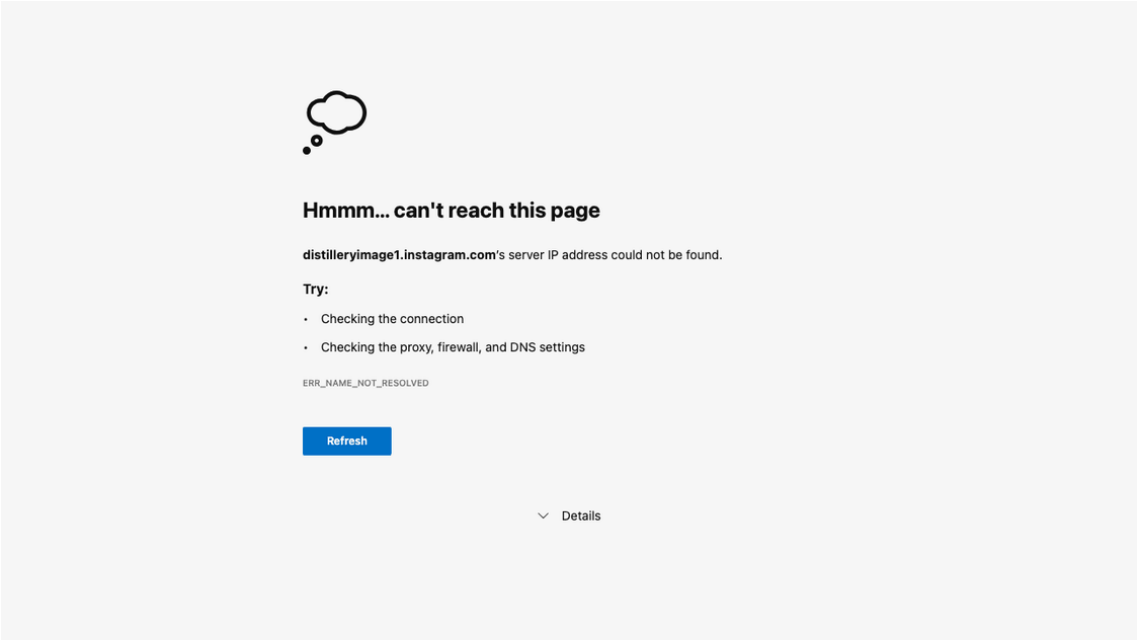
15 · migration, philosophy

Fourteen years, and every image came home

In 2012 I posted a photo to Instagram. The link still works. The photo doesn't.

If you kept a blog anywhere between 2010 and now, you know the drill. The platform got acquired, or pivoted, or "sunset some legacy infrastructure", and the images you embedded from their CDN quietly turned into grey rectangles. My personal archive goes back fourteen years across Tumblr, Twitter, Mastodon and a couple of other places — and a depressing number of its images now live at addresses that answer with nothing at all.

Here's what I mean, from my own archive. On February 22, 2012, I published a blog post called #53: Messengers of Spring? and embedded its photo straight from Instagram's CDN, the way everyone did back then. This is that link today:



distilleryimage1.instagram.com, fourteen years later

Notice it's not even a polite 404. `distilleryimage1.instagram.com` doesn't resolve at all — the photo is gone along with the entire piece of infrastructure that used to serve it. The post itself is still alive, imported into this engine, minus the one photo nobody can download anymore.

Here's the uncomfortable part: the post was mine. The photo was mine. The URL never was.

What the importer does about it

./blog.sh ships importers for eight sources, and they all follow one stubborn rule: media comes home. Every image, video and audio file gets downloaded next to its post, into `media.nosync/<year>/<slug>/`, and the post references the local copy from then on. No hotlinks, no third-party CDN, no "this content is no longer available".

A few details I'm fond of:

- Downloads are measured on arrival. The build refuses to render an image without known dimensions — so the importer simply never writes one.
- Failed downloads get retried. And when a file genuinely can't be fetched anymore, because the source deleted it years ago, you lose that one image — not the post.
- The origin becomes a tag. Import an old Tumblr blog and every post lands tagged `tumblr` . Your old blog turns into a browsable archive instead of a zip file in a drawer.

The numbers

From a real migration of a real fourteen-year archive:

Source	Items in the archive	Notes
Mastodon	6,591	2,984 replies and 1,059 boosts skipped on purpose
Pixelfed	333	two accounts, 571 media attachments
Tumblr	1,099	four separate blogs, importing 1,247 media files that came home with them
Twitter	5,388	2008–2022, with 1,134 replies and 417 retweets in the pile — and 610 media files

Here's a handful from that archive — photos that spent years on one platform or another, and now live as files next to this post:

The Astronomical Clock — Hipstamatic, 2019

Prague Castle — Hipstamatic, 2019

Emauzy — Hipstamatic, 2019

Spholio, a sculpture at night — 2022

Shot on a Palm phone — 2019, because the archive keeps the weird experiments too

The point

None of this is clever engineering. Downloading a file and putting it in a folder is about as advanced as computing gets. The clever part was done by every platform that convinced us it wasn't necessary.

An archive is only yours if the files are yours. Everything else is a lease — and fourteen years is long enough to watch a few landlords disappear.

16 · content, authoring

The photo your iPhone won't show anyone

An iPhone shoots HEIC by default. Safari renders it. Chrome doesn't. Firefox doesn't. So a photo straight off the phone, attached to a post and published, is a photo most of your readers will never see.

The old behaviour was the worst possible one: the file went in, the build couldn't measure it, and the image quietly vanished from the page — taking its caption with it. No error. Just a post with a hole where a photo was.

Refused, with the command you need

Attaching a HEIC now stops the save and prints the exact conversion command for the machine you're standing at:

```
sips -s format jpeg photo.heic --out photo.jpg      # macOS
heif-convert photo.heic photo.jpg                    # Linux
```

Your file stays exactly where it was, in `incoming/`. Nothing is deleted, nothing is guessed at. You convert it and carry on.

Or converted, if you ask

Set `media.convert_heic: true` and the engine converts it for you, using whichever tool it finds — `sips`, `heif-convert`, `magick`, `vips`. If it finds none, it falls back to the refusal above rather than pretending.

That's off by default on purpose. Converting means the file on your site isn't the file you handed over, and that should be a decision you made, not a thing that happened to you.

Detection is by content

Renaming `photo.heic` to `photo.jpg` doesn't get it past the check — the engine reads what the file actually is, not what it's called. Phones and export tools mislabel files often enough that trusting the extension would just move the silent failure somewhere less obvious.

The rule underneath

An image must never disappear quietly. If the engine can't handle a file, it says so, names the file, and leaves it where it is. A build that drops a photo and finishes with a cheerful summary has lied to you, and you'll find out months later when someone mentions the empty space.

(If you'd rather sidestep all of this: Settings -> Camera -> Formats -> Most Compatible makes the phone shoot JPEG.)

Part V

Build, deploy, appearance

What it costs to run, what it looks like, and what stops a bad build from erasing a good site.

17 · philosophy, build

No gems (one asterisk)

./blog.sh needs no gems. Here's the one asterisk on that sentence, because a claim without its exception isn't worth much.

The claim first: zero gems, zero npm packages, no Bundler, no lockfile, no `node_modules` folder quietly gaining weight in the dark. Ruby 2.7 or newer and bash. Clone it and it runs.

The asterisk, up front

Two optional sidebar widgets — Pixelfed and generic RSS — parse XML with `rexml` . That's a default gem: it ships bundled with every normal Ruby installation, so for most people the claim holds as written. But some Linux distributions split Ruby into packages and put the default gems in a separate one.

So: if you enable those two widgets on a distro-packaged Ruby and the build complains about `rexml` , you're the asterisk. Two ways out — install your distro's `ruby-rexml` -ish package, or `gem install rexml` . Either way it's one package, once, and only if you use those widgets at all.

That's the entire dependency story. I'd rather write it out than round it down to zero and wait for someone's build to disagree.

What a real zero buys you

No network dependencies at build time means the entire site builds offline. On a plane, on a train, on a server that firewalled itself into a corner — `ruby build/build_blog.rb` neither knows nor cares.

And the output is as boring as the input. Measured on this site, today, fourteen posts:

Measured on this site	Value
Full build, cold	1.9 s
Full build, warm (only changed files written)	0.9 s
Homepage HTML	54 kB
All CSS	24 kB
All JavaScript	23 kB — 555 lines across 8 files
Requests to third-party domains	1*

That JavaScript is the complete inventory: theme toggle, lightbox, client-side search, the comments loader, sidebar widgets, mobile nav, scroll-to-top, and a shared utility file. No framework is hiding under any of them.

And yes, a second asterisk snuck into the table, same policy as the first: the one third-party request is my own self-hosted Umami analytics, on another domain of mine, added through the config. The engine itself ships exactly zero — no fonts CDN, no tracker, no embed phoning home. Turn the analytics line off in `site.yml` and the number is a structural 0.

Why bother

Because every dependency is a small standing appointment with the future. Some Tuesday it needs updating, audits, replacing, or it deprecates the one function you used. A stdlib-only tool skips those appointments — the price is writing a bit more code yourself, and for a blog engine that price turned out to be surprisingly low.

A zero with its exception written down is more useful than a zero without one. If you trust the asterisk, you can trust the rest of the table.

18 · appearance

Seven keys

There is not a single colour value in this site's stylesheet.

Every colour you're looking at right now — background, text, links, the navigation, the little tag pills — comes from seven keys in `config/site.yml`. The build compiles them into a `colors.css` file, and `site.css` just uses the variables. Change seven lines, rebuild, and the site is someone else entirely.

Here's a complete light-mode palette:

```
config/site.yml
colors:
  light:
    bg: "#f5f8fa"
    text: "#444a5a"
    meta_text: "#657784"
    accent: "#1da1f2"
    nav_bg: "#eaf5fd"
    border: "#e1e8ed"
    pill_bg: "#d6ecfc"
```

And here's the same homepage in five palettes — nothing changed between these shots except seven lines of YAML and a rebuild:

Default blue — the palette this site ships with

Warm — cream, brown, terracotta accent

Monochrome — the accent is just a darker grey

High contrast — black on white, hypertext blue

Deliberately hideous — lime, purple, magenta, orange

The hideous one is the proof, by the way: if the config can make it that ugly, the config really is in charge.

What you don't configure

Everything else is derived: card backgrounds, link hover states, the search field, code block backgrounds. Those aren't extra config keys, because in practice they never varied independently — every time I tried a palette, the derived values followed the base ones anyway. Seven keys per mode is the honest number of decisions involved.

Dark mode is the same seven keys again, so light and dark are two palettes, not one palette and a filter.

Two exceptions, on purpose

The banner can optionally override its title and claim colours — `banner_title` and `banner_claim` — because a banner is an image, and text sitting on an image plays by different rules than text on a flat background.

And one small thing I like too much not to mention: the scrim that keeps banner text readable only darkens the corner the text actually sits in. Turn the overlays off and your banner stays exactly as

you drew it, corner to corner.

19 · deploy

A deploy that refuses to nuke your site

```
a chat block, as the engine renders it

Petr: I ran --prune after a broken build.
Pavel: So your site is empty.
Petr: No. The deploy refused.
Pavel: …I want that.
```

That's the whole feature: the deploy assumes a large change is a bug until you say otherwise.

How it works

Every deploy keeps a manifest — SHA-256, size and mtime for each uploaded file. The next deploy diffs against it and only touches what actually changed. Fast, boring, correct.

The interesting part is what happens before anything uploads. If the file count dropped by more than 20% since the last deploy, the whole thing stops. This is real output from this very site — I parked half the build's folders out of sight for a minute and asked for a deploy:

```
× Stopped: public.nosync/ has 28 files, but 62 were uploaded last time.
   That's a 55% drop -- looks like a broken build.
   Check the build output. If the drop is expected (you deleted a lot of posts), run again
with --force.
```

A drop like that almost never means "I deliberately deleted half my blog." It means a typo in a path, an empty content directory, a build that crashed halfway. Without the guard, `--prune` would cheerfully delete the rest of your live site and upload the wreckage.

And it's symmetric: a sharp increase trips the same wire, because a site that suddenly doubled usually means a duplicated tree or a badly merged import, not a very productive afternoon.

The escape hatches

- `--dry-run` shows exactly what would upload, change or be deleted — and touches nothing. Here's today's, after editing two posts:

```
Deploy web -> Surfer: https://blogsh.app  [DRY-RUN]
  62 file(s) total, 10 new/changed, 52 unchanged (skipped)
  [dry] index.html (54990 B)
  [dry] posts/2026/no-gems-one-asterisk/index.html (14852 B)
  [dry] posts/2026/why-not-x/index.html (14584 B)
  [dry] rss.xml (38156 B)
  [dry] search-index.json (29072 B)
  ...
```

Two posts changed, so ten files upload: the posts, the pages that list them, the feed, the search index. Everything else is a checksum match and stays home.

- `--force` is the "yes, I meant it" switch, for the day you really do delete a hundred posts.

- `--prune` is the only destructive flag in the toolbox, and it's opt-in. (On the `git` backend every deploy is a snapshot anyway, so pruning is implicit — and reversible.)

One honest side effect

Do a bulk import — or publish a fourteen-post series in one evening, as this site just did — and the growth guard trips too. Again, real output, from this site's launch night:

```
× Stopped: public.nosync/ has 63 files, only 30 were uploaded last time.
   That's a 110% increase -- looks like a duplicated or broken build.
   Check the build output. If the increase is expected (a bulk import/migration), run again
 with --force.
```

That's not a bug; that's the guard doing precisely its job on the one occasion you'd forgive it for staying quiet. Check the numbers, feel briefly smug that something checked them at all, and run it again with `--force` .

20 · deploy, philosophy

The guard that switched itself off

This site's deploy has two guards. If a build suddenly has far fewer files than what's live, or far fewer bytes, the deploy stops. They exist because a broken build looks exactly like a deliberate one to `rsync --delete`, and a static site is only ever one confident sync away from being erased.

In 1.0 they could turn themselves off. Permanently. Silently. Here is how, because the shape of this mistake is more useful than the fix.

The reference was the wrong thing

The guards compared the new build against the deploy manifest — a record of what's on the target. Reasonable, until an upload fails. A failed upload leaves the manifest out of true, and a guard measuring against a record it knows is wrong would fire on every subsequent run.

So there was a marker: after a failed run, stand the guards down until a clean run comes along and restores the reference.

You can see it already. When the failure is permanent — a file the host keeps refusing, expired credentials, a target that no longer exists — no clean run ever comes. The marker never lifts. The guards are off, and nothing says so. A build collapsing from 7,500 files to a handful would have been mirrored faithfully, `--prune` included.

The fix was to stop measuring the target

They now compare the build against the last build that was accepted, recorded before the first byte moves. That number doesn't care whether the upload then succeeded, failed, or died halfway — so there's no longer anything to stand down, and no marker to get stuck.

The reference stopped being "what's out there" and became "what I last agreed to". Nothing else had to change.

Three more things came out of the same review

They also fired when they shouldn't. Twenty per cent of a 32-file build is six files, so publishing two posts at once could abort a deploy — inside a flow `./blog.sh` runs for you, which has no way to pass `--force`. The percentages carry absolute floors now.

Bytes are guarded in both directions. The same file count with every page nearly empty used to be invisible. A byte drop stops the deploy; a byte increase only mentions itself, because attaching media is authoring, not a fault.

An empty build is refused outright. With an empty manifest, it used to sail through every check that existed.

Why write this up

Because "we found a bug" is worth less than "here is the reasoning that produced it". The bug wasn't sloppiness. It was a patch that solved the problem in front of it and created a worse one behind it, and it survived because the failure mode was silence.

The lesson I'm keeping: a safety mechanism that can be disabled by the thing it's protecting against isn't a safety mechanism. It's a suggestion.

21 · i18n

Add your language. It's one file.

The one way to contribute to ./blog.sh without writing a line of Ruby is to translate it — and it's a single YAML file.

Everything the reader sees — navigation, dates, archive headings, the search box, the comments prompt — comes from a locale file. The whole guide lives here:

docs/localization.md — adding a language to ./blog.sh

Three things make it genuinely easy:

- A language is data, not code. You copy `en.yml` , rename it, translate values. No templates touched, no Ruby read.
- Fallback runs per key. A half-finished translation works from day one — anything you haven't translated yet simply shows up in English until you get to it.
- Only `en.yml` must be complete. Yours can grow at whatever pace your weekends allow.

English, Czech and German are already in. Yours would make four.

One design decision worth a paragraph, because it surprises people: there is no plural system, on purpose. Instead, every count phrase is written to fit any number — the way Czech, with its three plural forms, has been quietly working around other people's plural systems forever. It's the difference between shipping a translation tonight and reading a CLDR spec first.

The reward: a locale reviewed by a native speaker gets marked as shipped in the README, with your name on it. Community drafts are welcome too — they're labeled as such until a native speaker signs off.

One honest limit: no RTL yet. The templates don't mirror the layout, and pretending otherwise would waste an Arabic or Hebrew translator's evening. It's written down in the same doc.

Where this came from

Every chapter in this document is a post published on blogsh.app, which runs on the engine it describes. The site is the working demo; this file is the same material in one place, for reading offline or on paper.

The engine is MIT licensed. Source, issues and the changelog: github.com/DanielSnor/blog.sh

Set in Open Sans and JetBrains Mono — the two typefaces the engine bundles — and coloured with its seven default palette keys.