PABLOMURAD.COM

~PABLO



A ZINE FROM PABLOMURAD.COM

Nº 02

NOVEMBER 2026

9 essays · tutorial · self-hosting · linux

CONTENTS

Nº 02
NOVEMBER 2026

01 Build a website with Org mode and Emacs 3
EMACS · ORG-MODE · STATIC-SITE

02 DNS blocking, seen from the terminal 15
DNS · PRIVACY · CENSORSHIP

03 Valid HTTPS for LAN-only services with acme-dns and pfSense 25
PFSENSE · DNS · LETS-ENCRYPT

04 Vim without the mystery 34
VIM · NEOVIM · TERMINAL

05 banany.lol: a free blog from a single markdown file 42
BANANY · RUNV-CLUB · BLOGGING

06 Setting up soju as an IRC bouncer on Debian 47
IRC · SOJU · DEBIAN

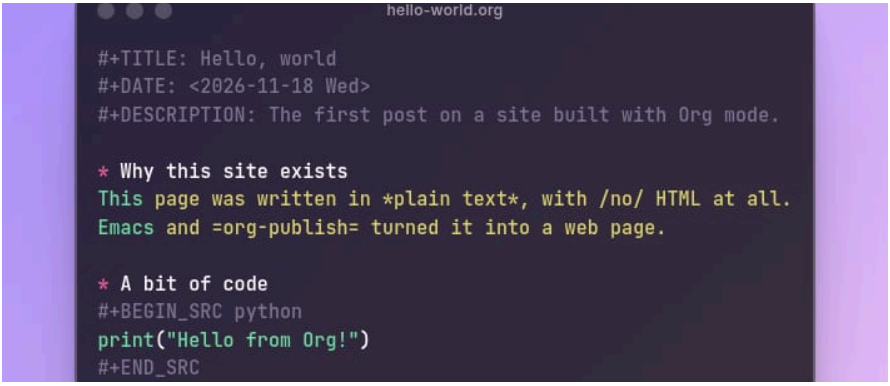
07 Org mode as a digital garden 55
EMACS · ORG-MODE · STATIC-SITE

08 Your passwords on every device, with no cloud 62
KEEPASSXC · SYNCTHING · PASSWORDS

09 TLS certificates explained: ACME, Certbot and the alternatives 72
TLS · HTTPS · LETS-ENCRYPT

Build a website with Org mode and Emacs

A complete, step-by-step guide to building a static website with Org mode and Emacs alone: writing pages in Org, a build script with ox-publish, a blog index that updates itself, styles, code highlighting, previewing, deploying and fixing the usual errors.



Most website tools ask for a framework, a template language and a build system before the first sentence gets written. Org mode asks for much less. The writing happens in plain text files in Emacs, and Emacs turns them into HTML. No Node, no Ruby, no Go, no database. The tool that does the work, `ox-publish`, has shipped with Org for years, and Org ships with Emacs.

This guide builds a small but complete site from scratch: a home page, an about page, a blog with an index that updates itself, a stylesheet, images, highlighted code, a one-command build, a local preview and a deploy. Every step explains what it does and why, so it's easy to bend to taste afterwards.



What Org mode is

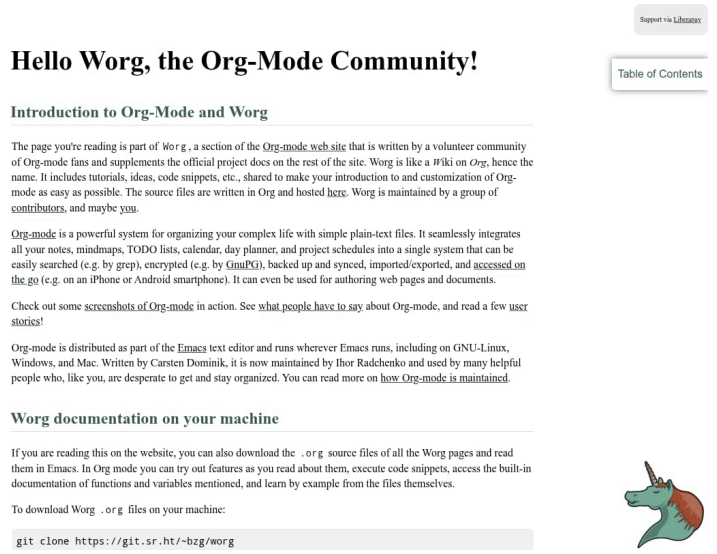
Org mode is a major mode for Emacs, created by Carsten Dominik in 2003 and part of Emacs itself since version 22. People use it for notes, to-do lists, agendas, spreadsheets, literate programming and writing. Underneath all of that, an Org file is just text with a light markup, a bit like Markdown, but with a lot more built in.

The part that matters here is the exporter. Org can turn the same file into HTML, LaTeX, PDF, ODT, Markdown and plain text. On top of the exporter sits the publishing system, `ox-publish`, which exports a whole folder of files at once, copies images and stylesheets along, and keeps track of what changed. That's a static site generator, already installed.

Real sites run on it. Worg, the community wiki of the Org project, is written in Org files and published with this same machinery.

What it takes

- Emacs, any recent version. Org comes with it.
- A terminal, to run the build.
- Python 3 or any small web server, only to preview the site locally.
- Somewhere to host the result later: a personal server, or a static host like GitHub Pages, Codeberg Pages or SourceHut pages.



No Emacs expertise required. The work is editing a few text files and running one command.

A crash course in Org markup

For anyone who has never written Org, this is enough to build a site. Headings start with stars, and more stars mean a deeper level:

```
* A top heading
** A sub-heading
*** And another level
```

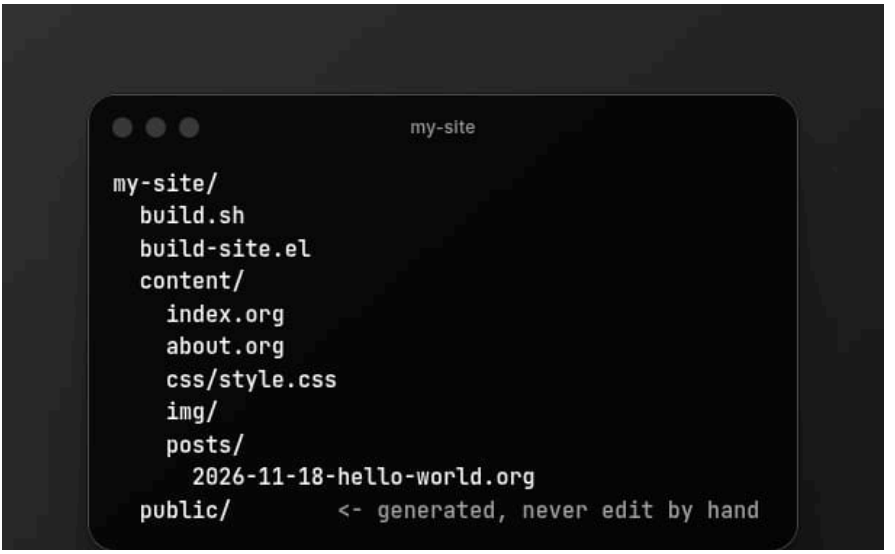
Inline formatting:

- `*bold*`, `/italic/`, `_underlined_` and `+struck through+`
- `=verbatim=` and `~code~` for monospace text

Links and images use double brackets. A link to a file that is an image becomes an image on the page:

```
[[https://orgmode.org][The Org website]]
[[file:about.org][About me]]
[[file:img/photo.jpg]]
```

Lists, tables, quotes and code blocks:




```
- an item
- another item
1. a numbered item

| Name | Language |
|------+----------|
| Org  | Elisp    |

#+BEGIN_QUOTE
Plain text lasts.
#+END_QUOTE

#+BEGIN_SRC python
print("Hello")
#+END_SRC
```

And the lines starting with `#+` at the top of a file are settings. `#+TITLE` becomes the page title, `#+DATE` the date and `#+DESCRIPTION` the meta description.

One handy detail: a link to another `.org` file, like `[[file:about.org]]`, becomes a link to `about.html` when the site is published. Links go between the source files, and Org fixes the extensions.

Step 1: the folder structure

Create this layout. The writing lives in `content/`, and the finished site appears in `public/`.

```
mkdir -p my-site/content/posts my-site/content/css my-site/content/img
cd my-site
```

Keeping sources and output apart is the most important habit here. `public/` is disposable: it can be deleted at any time and the build recreates it. Anything edited by hand in there is lost on the next build.

Step 2: write the content

`content/index.org`, the home page:

```
#+TITLE: My little corner of the web
#+DESCRIPTION: Notes, essays and experiments.

Hello! This site is written in plain text with Org mode and built by Emacs.

* Start here
- [[file:about.org][About me]]
- [[file:posts/index.org][Every post, newest first]]
```

`content/about.org`:

```
#+TITLE: About

I write about the things I build and the things I learn.
You can reach me by email at [[mailto:you@example.org][you@example.org]].
```

And the first post, `content/posts/2026-11-18-hello-world.org`:

```
#+TITLE: Hello, world
#+DATE: <2026-11-18 Wed>
#+DESCRIPTION: The first post on a site built with Org mode.

* Why this site exists
This page was written in *plain text*, with /no/ HTML at all.
Emacs and =org-publish= turned it into a web page.

* A bit of code
#+BEGIN_SRC python
print("Hello from Org!")
#+END_SRC
```

The `#+DATE` line matters for posts: the blog index uses it to sort them. Inside Emacs there's no need to type the date by hand. `C-c .` opens a calendar and inserts it in the right format.

Notice that `index.org` links to `posts/index.org`, a file nobody wrote. The build creates it, in step 4.

Step 3: a stylesheet

Org's default HTML is clean but plain. A few lines of CSS go a long way. Save this as `content/css/style.css`:

```
body { max-width: 42rem; margin: 2rem auto; padding: 0 1rem; font: 18px/1.6
Georgia, serif; color: #222; background: #fdfcf8; }
nav { margin-bottom: 2rem; }
nav a { margin-right: 1rem; }
h1, h2, h3 { line-height: 1.25; }
a { color: #8a3b12; }
pre { background: #f3f1ea; padding: 1rem; overflow-x: auto; }
img { max-width: 100%; height: auto; }
footer { margin-top: 3rem; font-size: 0.9rem; color: #666; }
```

To skip writing CSS altogether, a classless stylesheet like Simple.css or Water.css works well with Org's output, since Org produces plain, semantic HTML.

Step 4: the build script

This is the heart of the site. Create `build-site.el` at the root of the project:

```
;; build-site.el: turns content/ into a website in public/

;; 1. Keep the packages this build needs inside the project
(require 'package)
(setq package-user-dir (expand-file-name "./.packages"))
(setq package-archives '(("melpa" . "https://melpa.org/packages/")
("elpa" . "https://elpa.gnu.org/packages/")))
(package-initialize)
(unless package-archive-contents
(package-refresh-contents))
(package-install 'htmlize)

;; 2. Load the publishing system
(require 'ox-publish)

;; 3. Keep publishing timestamps inside the project too
(setq org-publish-timestamp-directory (expand-file-name "./.timestamps/"))

;; 4. HTML settings shared by every page
(setq org-html-validation-link nil
org-html-head-include-scripts nil
org-html-head-include-default-style nil
org-html-doctype "html5"
org-html-html5-fancy t
org-html-head "<link rel=\"stylesheet\" href=\"/css/style.css\">"
org-html-preamble "<nav><a href=\"/\">Home</a> <a href=\"/posts/\">Posts</a>
<a href=\"/about.html\">About</a></nav>"
org-html-postamble "<footer>Written in Org mode, built with Emacs.</
footer>")

;; 5. How each post shows up in the posts index
```

```
(defun my/sitemap-entry (entry style project)
(if (directory-name-p entry)
entry
(format "%s: [[file:%s][%s]]"
(format-time-string "%Y-%m-%d" (org-publish-find-date entry project))
entry
(org-publish-find-title entry project))))

;; 6. The projects: posts, pages and static files
(setq org-publish-project-alist
(list
(list "posts"
:base-directory "./content/posts"
:base-extension "org"
:publishing-directory "./public/posts"
:publishing-function 'org-html-publish-to-html
:section-numbers nil
:with-toc nil
:with-author nil
:time-stamp-file nil
:auto-sitemap t
:sitemap-filename "index.org"
:sitemap-title "Posts"
:sitemap-style 'list
:sitemap-sort-files 'anti-chronologically
:sitemap-format-entry 'my/sitemap-entry)
(list "pages"
:base-directory "./content"
:base-extension "org"
:publishing-directory "./public"
:publishing-function 'org-html-publish-to-html
:section-numbers nil
:with-toc nil
:with-author nil
:time-stamp-file nil)
(list "static"
:base-directory "./content"
:base-extension "css\\|js\\|png\\|jpg\\|jpeg\\|gif\\|svg\\|webp\\|ico"
:publishing-directory "./public"
:recursive t
:publishing-function 'org-publish-attachment)
(list "site" :components '("posts" "pages" "static"))))

;; 7. Build everything, ignoring what was built before
(org-publish-all t)
(message "Build complete!")
```

That's a lot at once, so here's what each part does.

Part 1 sets up Emacs' package manager inside the project folder, in `.packages`, so the build never touches the personal Emacs configuration. It installs one package, `htmlize`, which Org needs to color the code in source blocks. Without it, the build stops with "Cannot fontify source block (htmlize.el >= 1.34 required)".

Part 2 loads `ox-publish`, the publishing system.

Part 3 moves Org's record of what it already published from the home folder into the project. It's a small thing that keeps every project self-contained.

Part 4 shapes every page:

- `org-html-validation-link nil` removes the old "Validate" link Org adds at the bottom.
- The two `include` settings switch off Org's built-in CSS and JavaScript, since the site has its own stylesheet.
- `doctype "html5"` and `html5-fancy` make Org use modern tags like `<nav>`, `<section>` and `<figure>`.
- `org-html-head` adds the stylesheet to every page.
- The preamble goes at the top of each page, so it's a good place for the navigation. The postamble goes at the bottom, which suits a footer.

Part 5 is a small function that decides how each post appears in the posts index: the date from `#+DATE`, followed by the title as a link. `org-publish-find-date` and `org-publish-find-title` read those values from each file.

Part 6 defines the projects. A project is a folder Org knows how to publish:

- **posts** exports every `.org` file in `content/posts` to HTML. `:auto-sitemap t` makes Org write a list of all posts into `posts/index.org` before publishing, newest first because of `anti-chronologically`. That generated file becomes the blog index, at `/posts/`.
- **pages** exports the `.org` files at the top of `content`: the home page and the about page. It isn't recursive, so it leaves the posts alone.
- **static** copies everything that isn't Org, like CSS, scripts and images, exactly as it is. `org-publish-attachment` means "copy, don't convert".
- **site** groups the three, so a single name builds everything. The order matters: posts come first, because the home page links to `posts/index.org`, which only exists after the posts project has generated it.

The `:section-numbers`, `:with-toc`, `:with-author` and `:time-stamp-file` lines turn off things that feel bookish on a website: numbered headings, a table of contents at the top, the author line and a "created" timestamp. Turn any of them back on for a page that needs it.

Part 7 runs the whole thing. The `t` forces a full rebuild every time, which avoids confusing moments where a change doesn't show up because Org thinks the file is already up to date.

Step 5: build with one command

Create `build.sh` next to it:

```
#!/bin/sh
emacs -Q --script build-site.el
```

Make it executable and run it:

```
chmod +x build.sh
./build.sh
```

`-Q` starts Emacs without any personal configuration, so the build behaves the same on any machine, including a server or a CI job. `--script` runs the file and exits, without opening a window. The first run takes a little longer while it downloads `htmlize`. After that, `public/` holds the site:

```
public/
index.html
about.html
css/style.css
posts/index.html
posts/2026-11-18-hello-world.html
```

Step 6: preview it

The navigation uses links that start with `/`, so open the site through a small web server rather than double-clicking the files. Python has one built in:

```
cd public
python3 -m http.server 8000
```

Then visit `http://localhost:8000`. Change a file, run `./build.sh` again and reload the page.

To stay inside Emacs, the `simple-httpd` package does the same job: install it from MELPA, run `M-x httpd-serve-directory` and point it at `public/`. It serves on port 8080 by default.

Working from inside Emacs

The build script is the reliable way to publish, but Org's export menu is handy while writing. Inside any Org file:

- `C-c C-e`: open the export menu
- `C-c C-e h h`: export the current file to HTML
- `C-c C-e h o`: export it and open the result in the browser
- `C-c C-e P f`: publish the current file with its project settings
- `C-c C-e P p`: publish the current project
- `C-c C-e P a`: publish every project

- `C-c C-l`: insert a link, with completion for file paths
- `C-c .`: insert a date from a calendar

The `P` commands need `org-publish-project-alist` loaded in the running Emacs. One way is to keep the project definitions in a separate file and load it from the configuration.

Images

Put images in `content/img/` and link to them with a path relative to the Org file:

```
[[file:img/sunset.jpg]]
```

From a post inside `posts/`, go one level up:

```
[[file:../img/sunset.jpg]]
```

To add a caption, alt text or a size, put these lines right above the link:

```
#+CAPTION: The view from the window, at the end of the day
#+ATTR_HTML: :alt A sunset over the hills :width 600
[[file:../img/sunset.jpg]]
```

The static project copies the image to `public/img/`, so the same relative path works on the published site. If an image is missing from the output, check that its extension is in the `:base-extension` list of the static project.

Better code highlighting

By default, `htmlize` writes the colors straight into each code block, using whatever theme the build happens to have. With `emacs -Q`, that's the default light theme. To control the colors with CSS instead, add this to part 4 of `build-site.el`:

```
(setq org-html-htmlize-output-type 'css)
```

Now the code blocks carry CSS classes instead of colors. To get a stylesheet for them, open a normal Emacs session with the preferred theme, run `M-x org-html-htmlize-generate-css`, and save the buffer it produces as `content/css/code.css`. Then load it next to the main stylesheet:

```
org-html-head "<link rel=\"stylesheet\" href=\"/css/style.css\"><link
rel=\"stylesheet\" href=\"/css/code.css\">"
```

Per-page settings

Everything set in the build script can be changed for a single page with `#+` lines at the top of the file. For example, a long guide that deserves a table of contents and numbered sections:

```
#+TITLE: A long guide
#+OPTIONS: toc:t num:t
```

And to add something only to the `<head>` of one page:

```
#+HTML_HEAD_EXTRA: <meta name="robots" content="noindex">
```

The order of priority is simple: settings inside a file win over the project settings, which win over the global variables.

Publishing it

The finished site is just files in `public/`, so anything that serves files can host it.

To a personal server, with `rsync`:

```
./build.sh
rsync -avz --delete public/ you@your-server:/var/www/my-site/
```

The `--delete` flag removes files from the server that no longer exist locally, so deleted pages disappear online too. On the server, any web server pointing at that folder will do.

For GitHub Pages, Codeberg Pages or SourceHut pages, the idea is to run `build.sh` and upload the contents of `public/`, by hand or from a CI job that installs Emacs and runs the same script. That's the payoff of `emacs -Q --script`: the build doesn't depend on anyone's personal setup.

With the site in git, commit the sources and the build files, and leave the generated and downloaded folders out with a `.gitignore`:

```
public/
.packages/
.timestamps/
```

Common errors and what they mean

- **"Unable to resolve link"**: a `[[file:...]]` link points to an Org file that doesn't exist. Usually a typo, or a page that hasn't been written yet. Org stops the build on purpose, so broken internal links never get published.

- **"Cannot fontify source block (htmlize.el >= 1.34 required)"**: `htmlize` isn't installed. Part 1 of the build script takes care of it, so check that it ran, and that the machine could reach MELPA.
- **The navigation works online but not when the files open directly**: links starting with `/` point to the root of a web server. Use the preview server from step 6.
- **A change doesn't show up**: make sure the build ran again, and that the browser is looking at `public/`, not `content/`. The `t` in `org-publish-all t` already rules out stale caches.
- **An image or CSS file is missing**: its extension isn't in the static project's `:base-extension`.

Going further

What this guide builds covers a personal site and a simple blog. For more, there's a whole ecosystem around Org publishing:

- **RSS**: the `ox-rss` package, on MELPA, exports Org files to an RSS feed that can sit next to the posts.
- **ox-hugo**: write in Org, and let Hugo build the site. Useful for anyone who wants Hugo's themes and features but prefers Org for writing.
- **org-static-blog**: a small package focused on blogs, with an archive, tags and an RSS feed out of the box.
- **weblorg**: a static site generator for Emacs with templates, for sites that outgrow a plain `ox-publish` setup.

Worg keeps a page that compares many of these tools, listed below.

A good path is to start with plain `ox-publish`, as in this guide. It's built in, it will keep working as long as Emacs does, and every piece of it is a few lines of Emacs Lisp anyone can read and change. The bigger tools make sense once it's clear exactly what's missing. And when the site starts to grow, [Org mode as a digital garden](https://pablomurad.com/org-mode-digital-garden/) is the second part of this guide, with ideas from a site that has been tended this way for years.

Study sources

- The Org Manual: Publishing, the official reference for `ox-publish`
- The Org Manual: Options for the exporters, every project property and the variable behind it
- The Org Manual: Generating a sitemap, the options used for the posts index

- The Org Manual: Complex publishing example, a site with pages, images and stylesheets
- Worg: Publishing Org-mode files to HTML, a classic tutorial on splitting a site into components
- System Crafters: Build Your Website with Org Mode, a guide to building a site from a script with `emacs -Q`
- Worg: Blogs and wikis with Org, a list of tools for blogging with Org
- Org-mode on Wikipedia, for the history of the project

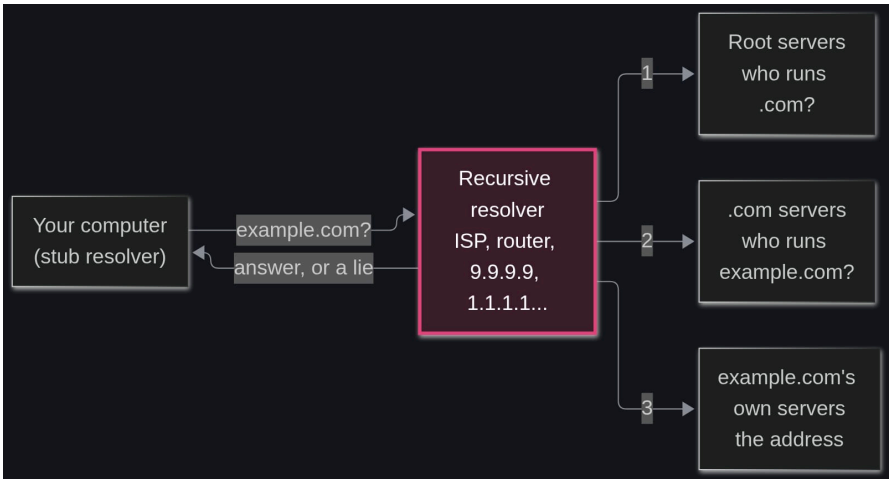
02 DNS · PRIVACY · CENSORSHIP 19 NOVEMBER 2026

DNS blocking, seen from the terminal

Quad9 leaves France on November 24 after a court fined it for refusing to block pirate sites. This is the hands-on companion: find out which resolver answers, read a blocked DNS reply with dig, encrypt the queries and run a home resolver with Unbound.



On November 24, Quad9 stops answering DNS queries from France. A Paris court fined the Swiss non-profit for refusing to block a list of pirate sports-streaming domains, about twenty similar cases are



still pending, and leaving the country was the only way out it could afford. Tech Stackups wrote the long version of this story: the law behind it, the rulings, why resolvers are a poor place to fight piracy and where the rest of Europe is heading. It's the place to start.

This post is the hands-on half. If a court can order a resolver to lie, it pays to know which resolver is answering, what a lie looks like when it comes back, and how to stop depending on somebody else's resolver at all. Everything below works on Linux, and most of it on macOS and Windows too.

Who answers when a name is typed

When a browser opens `example.com`, the computer doesn't look for the address on its own. A small part of the operating system, the stub resolver, passes the question to a recursive resolver: the ISP's, the one in the router, or a public one like 9.9.9.9 or 1.1.1.1. The recursive resolver does the actual work. It asks a root server who runs `.com`, asks the `.com` servers who runs `example.com`, asks that domain's own servers for the address, keeps the answer in its cache for a while and hands it back.

That middle box is where blocking happens. The pirate site keeps its address and its own servers keep giving it out. The recursive resolver simply decides not to pass the answer along, which is why French courts go after Google, Cloudflare and Quad9: they sit in the middle.

Step 1: find out which resolver answers

Start with what the machine is configured to use:

```
# Linux with a plain /etc/resolv.conf
cat /etc/resolv.conf

# Linux with systemd-resolved (Ubuntu and others)
resolvectl status

# macOS
scutil --dns | grep nameserver

# Windows, in PowerShell
Get-DnsClientServerAddress
```

Very often this only shows the router, something like `192.168.1.1`, and the router forwards everything to whatever it was set up to use. To find out who really answers, ask for a name whose server replies with the address of whoever asked it:

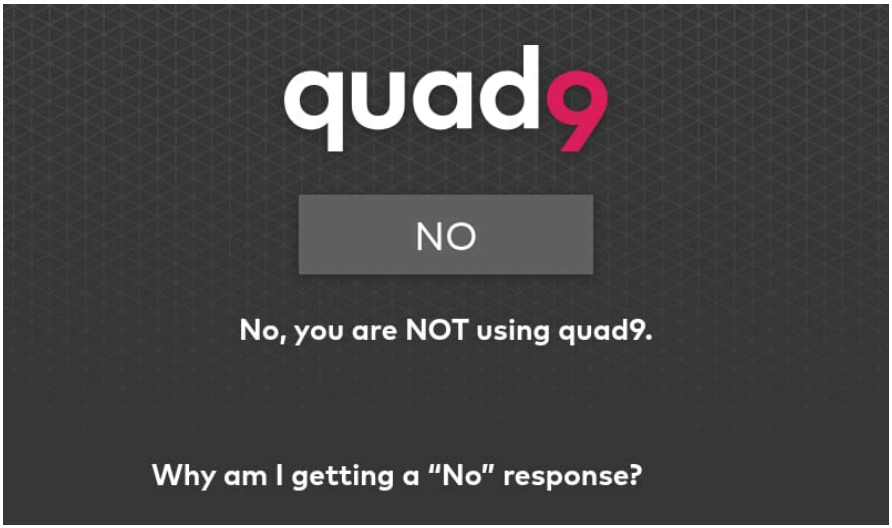
```
dig +short whoami.akamai.net
dig +short TXT o-o.myaddr.l.google.com
```

Both print the IP address of the resolver that reached them. Run `whois` on that address to see which network it belongs to. It won't always be the expected brand: big public resolvers use anycast, and many of their nodes live inside partner networks. A query sent through Quad9, for example, can come back from an address registered to a university research network in Colorado that hosts one of Quad9's nodes.

For Quad9 there's a shortcut: open on.quad9.net and it answers yes or no.

That page also lists the usual reasons for a surprise "no", and they apply to any resolver:

- Firefox and Brave: in some countries they ship with DNS over HTTPS turned on, pointing at Cloudflare, and ignore the system settings.
- iCloud Private Relay: on iOS and macOS it takes over DNS while it's on.
- VPN clients: most use their own DNS servers unless custom ones are set inside the app.
- Mixed settings: one Quad9 address next to one from another provider means some queries go to each.



Step 2: read a blocked answer

The tool here is `dig`. On Debian and Ubuntu it lives in `bind9-dnsutils` (the old package name `dnsutils` still works), macOS already has it, and on Windows `Resolve-DnsName` in PowerShell covers the basics.

```
sudo apt install bind9-dnsutils
```

Quad9 keeps a test domain, `isitblocked.org`, that its filter always blocks. The examples below use `+https`, which makes dig send the query over DNS over HTTPS (it needs BIND 9.18 or newer). Plain `dig @9.9.9.9 isitblocked.org` gets the same status. This is Quad9's filtered service, with the output trimmed:

```
$ dig +https @dns.quad9.net isitblocked.org

;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 44678
;; flags: qr rd ad; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
; EDE: 17 (Filtered)
```

Four details give the block away.

`status: NXDOMAIN` means "this name doesn't exist". The domain does exist, so this is the lie itself.

`AUTHORITY: 0` is the clearest sign. A real NXDOMAIN comes with a record from the zone that says so. Asked for a name nobody registered, the same resolver answers this:

```
$ dig +https @dns.quad9.net thisdomaindoesnotexist-pm.org

;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 31764
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 1

;; AUTHORITY SECTION:
org.  3454  IN  SOA  v0n0.pir-ns.org. hostmaster.donuts.email. 1795037395
7200 900 1209600 3600
```

That SOA record comes from the `.org` zone itself, confirming the name doesn't exist. The blocked answer has nothing like it.

The `ra` flag is missing in the blocked answer, and dig warns that recursion wasn't available. The resolver never went out to look the name up.

`EDE: 17 (Filtered)` is an Extended DNS Error, defined in RFC 8914. It's a small note attached to the answer that says why it came back the way it did.

Now ask Quad9's unfiltered service, 9.9.9.10, the same question:

```
$ dig +https @dns10.quad9.net isitblocked.org +short
74.208.236.124
```

Without the filter, the same company hands over the real address.

THE FOUR CODES THAT MATTER HERE

RFC 8914 has four codes for "I won't give you this", and the difference between them is who wanted the block:

- 15 Blocked: the resolver's operator blocked it under its own security policy.
- 16 Censored: someone other than the operator required the block. The RFC adds that how that requirement is imposed doesn't matter: "in-band DNS filtering, court order, etc."
- 17 Filtered: the client asked for filtering. Quad9 uses this one because choosing 9.9.9.9 is choosing its blocklist.
- 18 Prohibited: the resolver won't serve this client at all, for example because its address isn't on the allowed list.

The RFC also says these codes are diagnostic only and must not change how software handles the answer. Browsers generally don't show them, which is why a blocked site just looks broken. dig shows them, so dig is the way to find out.

NOT EVERY RESOLVER LIES THE SAME WAY

When the block comes from a court, each resolver answers differently:

- Google Public DNS: for legal blocks in Belgium, France, India, Italy and Portugal it answers REFUSED, optionally with EDE 16 (Censored).
- French ISPs: OONI's 2025 measurements found Free and Orange answering 127.0.0.1, the user's own machine, for blocked copyright and sports-streaming domains, while SFR and Bouygues answer NXDOMAIN. Illegal gambling sites point to a block page run by the gambling regulator, ANJ.
- Cloudflare 1.1.1.1: its transparency report says it has not blocked anything through the resolver so far. In France it complied by geoblocking domains on its CDN instead, 1,238 of them in the second half of 2025, according to TorrentFreak.

A bare NXDOMAIN or 127.0.0.1 says nothing about why. The quickest test is to ask several resolvers the same question and compare:

```
for r in 9.9.9.9 9.9.9.10 1.1.1.1 8.8.8.8; do
  printf '%s ' "$r"
  dig @"$r" isitblocked.org +noall +comments | grep -o 'status: [A-Z]*'
done
```

```
9.9.9.9 status: NXDOMAIN
9.9.9.10 status: NOERROR
1.1.1.1 status: NOERROR
8.8.8.8 status: NOERROR
```

If one resolver says the name doesn't exist and the others return an address, somebody along the way is filtering. Swap `isitblocked.org` for any site that fails to load and it becomes a small censorship test that runs anywhere.

Step 3: encrypt the trip

Plain DNS travels over UDP port 53 in clear text. Anyone on the path, the ISP or the café Wi-Fi, can read every name looked up, and some networks catch traffic to port 53 and answer it themselves no matter which server was configured. Two protocols close that gap: DNS over TLS (DoT, port 853) and DNS over HTTPS (DoH, port 443, RFC 8484).

Encryption protects the trip only. The resolver at the other end still sees every name, so encryption decides who can watch along the way and the choice of resolver decides who gets trusted at the end of it.

On Linux with systemd-resolved, create `/etc/systemd/resolved.conf.d/10-dot.conf`:

```
[Resolve]
DNS=9.9.9.9#dns.quad9.net 149.112.112.112#dns.quad9.net
DNSOverTLS=yes
```

The part after `#` is the name on the server's TLS certificate, and systemd-resolved checks it. Swap both for the chosen provider (in France, after the 24th, that can't be Quad9). Then:

```
sudo systemctl restart systemd-resolved
resolvectl status
```

The `Protocols` line should now show `+DNSOverTLS`. With `DNSOverTLS=yes`, a server that doesn't speak TLS makes every lookup fail, which is the point: no silent fallback to clear text.

Elsewhere:

- Android 9 and newer: Settings, Network, Private DNS, then type the provider's hostname, for example `dns.quad9.net`. This is DoT for the whole phone.
- Windows 11: in the DNS settings of the network adapter, turn DNS over HTTPS on. It works automatically for the providers Windows knows.
- Firefox: Settings, Privacy & Security, DNS over HTTPS. Firefox then skips the system resolver completely, which is also why it can disagree with the rest of the machine.

dig can test both protocols directly: `dig +tls @dns.quad9.net example.com` and `dig +https @dns.quad9.net example.com`.

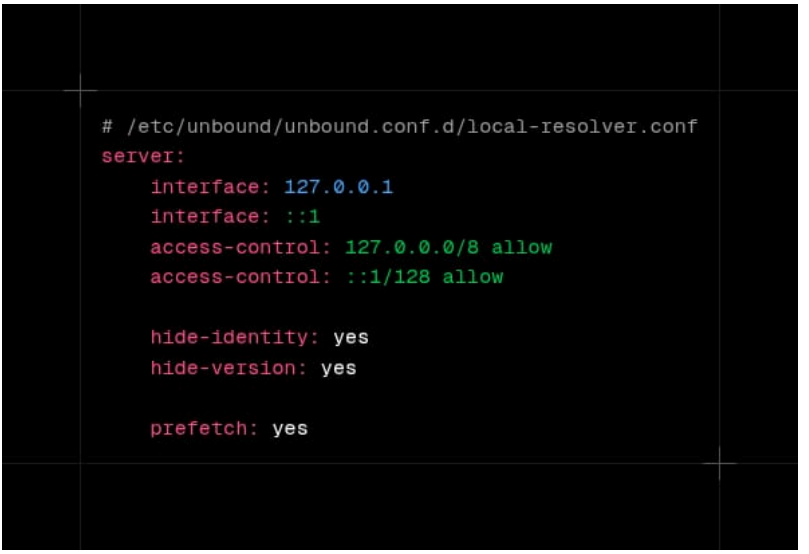
Step 4: run your own resolver

A recursive resolver at home walks from the root to the authoritative servers by itself, the same way the big public ones do. No public resolver sees the queries, and no court order aimed at one of them changes the answers. There are trade-offs, and they're worth knowing first:

- Its queries to authoritative servers leave the network unencrypted, so the ISP can still see them and still intercept port 53.
- Any filtering the public resolver did goes away, such as Quad9's malware blocklist. Pi-hole or AdGuard Home can sit in front of the resolver, with blocklists chosen by hand.
- The first lookup of each name takes a little longer. After that it comes from the cache.

Unbound is the usual choice. Debian 13 ships version 1.22:

```
sudo apt install unbound dns-root-data
```



Debian's package already turns DNSSEC validation on (that's `/etc/unbound/unbound.conf.d/root-auto-trust-anchor-file.conf`) and reads every `.conf` file in that folder. Add one more, `/etc/unbound/unbound.conf.d/local-resolver.conf`:

```
server:
    interface: 127.0.0.1
    interface: ::1
    access-control: 127.0.0.0/8 allow
    access-control: ::1/128 allow

    hide-identity: yes
    hide-version: yes

    prefetch: yes
```

Line by line:

- `interface`: listen only on this machine, over IPv4 and IPv6.
- `access-control`: who is allowed to ask. Anyone not listed gets refused.
- `hide-identity` and `hide-version`: don't tell strangers the server's name or which Unbound version it runs.
- `prefetch`: refresh popular names shortly before they expire from the cache, so they rarely make anyone wait.

QNAME minimisation (sending each server only the part of the name it needs, so the root servers see `.com` and not the full name) and the DNSSEC

hardening options are already on by default in current Unbound, so there's no need to write them.

Check the file, restart and test:

```
sudo unbound-checkconf
sudo systemctl restart unbound
dig @127.0.0.1 example.com
```

Then test DNSSEC. The first name is signed wrong on purpose and must fail, the second must resolve with `ad` (authenticated data) in the flags line:

```
dig @127.0.0.1 fail01.dnssec.works
dig @127.0.0.1 dnssec.works
```

`SERVFAIL` on the first and `flags: qr rd ra ad` on the second mean validation works.

Last, point the machine at it. On a Debian box without systemd-resolved, `/etc/resolv.conf` should contain `nameserver 127.0.0.1` (if the `resolvconf` or `openresolv` package is installed, Unbound's own `unbound-resolvconf` service writes that line by itself). With systemd-resolved, use a drop-in like the one in step 3 with `DNS=127.0.0.1` and without the TLS line.

FOR THE WHOLE HOUSE

To serve the local network from a Raspberry Pi or a home server, add a second file, say `lan.conf`, in the same folder:

```
server:
    interface: 0.0.0.0
    interface: ::0
    access-control: 192.168.1.0/24 allow
    access-control: fd00::/8 allow
```

Change `192.168.1.0/24` to the local network's range, restart Unbound and set the router's DHCP to hand out the Pi's address as the DNS server. Keep port 53 closed to the internet: open resolvers get abused to flood other people with amplified traffic.

Before November 24, in France

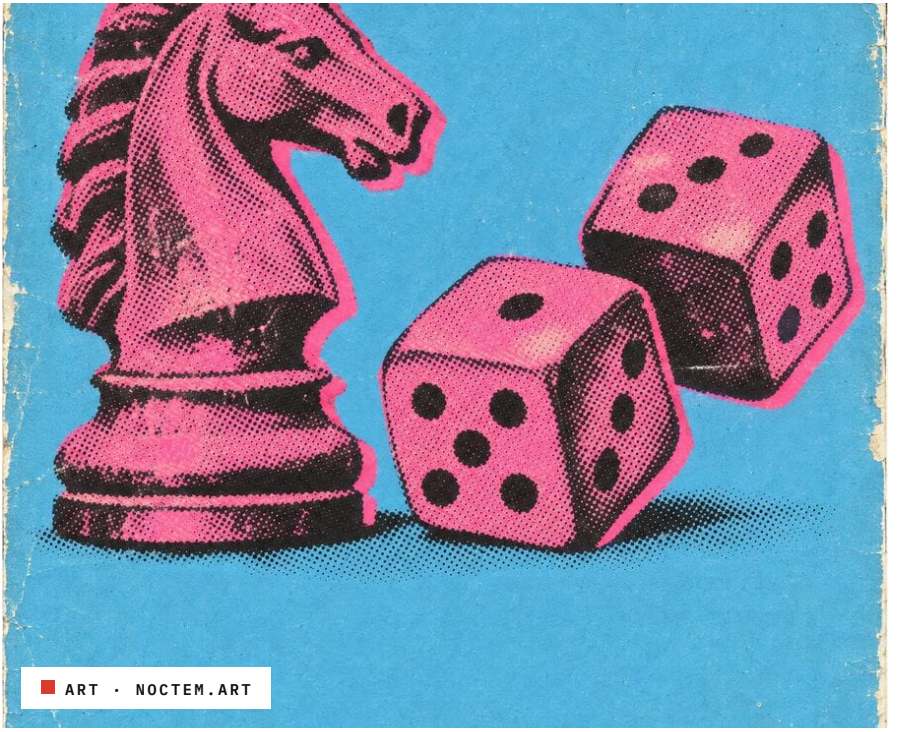
Quad9 hasn't said how the cutoff will behave on the wire, whether queries will be refused or just go unanswered. Either way, as Cloudflare put it in its post about the withdrawal, people won't see an error message: "Their Internet will simply stop working." Places where Quad9 tends to hide:

- The router: both its own DNS setting and the servers its DHCP hands out.
- `/etc/resolv.conf`, systemd-resolved drop-ins and the `FallbackDNS=` line.
- Pi-hole and AdGuard Home upstreams.
- VPN clients with custom DNS.
- Docker's `/etc/docker/daemon.json`, under `"dns"`.
- Android's Private DNS (`dns.quad9.net`) and DoH settings in Firefox or Chrome.
- Mullvad Browser on default settings, which switched to Quad9 on November 2.

On Linux machines, this finds the Quad9 addresses and hostnames in `/etc`:

```
sudo grep -rnE '9\.9\.9\.(9|10|11)|149\.112\.112\.|2620:fe:|quad9' /etc 2>/
dev/null
```

Afterwards, run the two commands from step 1 again. They're the quickest way to see who answers now. ■

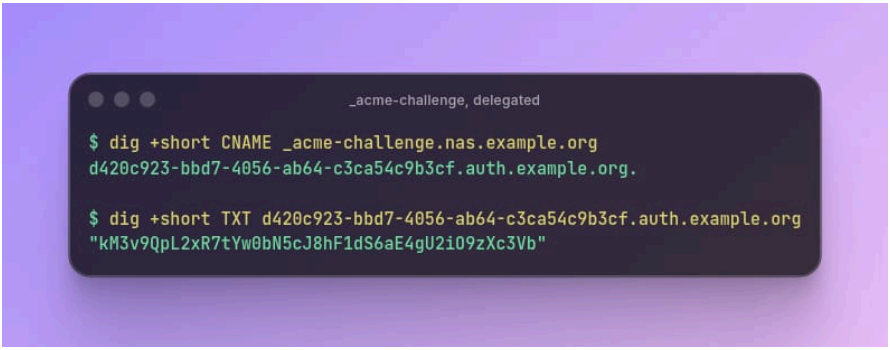


ART · NOCTEM.ART

03 PFSENSE · DNS · LETS-ENCRYPT 19 NOVEMBER 2026

Valid HTTPS for LAN-only services with acme-dns and pfSense

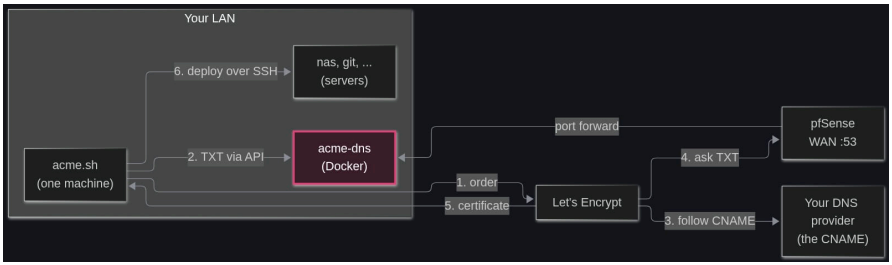
Some of my services only exist on the LAN, and I still wanted real Let's Encrypt certificates on them. A step-by-step guide to DNS-01 with acme-dns in Docker, a port forward in pfSense, and one acme.sh machine that issues every certificate and ships it to the servers.



Some of the services on my network only exist on the LAN, and I still wanted valid HTTPS on them, with certificates from Let's Encrypt.

A certificate from a public CA is trusted out of the box by browsers, phones and command-line tools, which a self-signed certificate or a private CA can't match without installing something on every device. The usual way to get one from Let's Encrypt is the HTTP-01 challenge, and it needs port 80 open to the internet on every name that needs a certificate. For services that aren't meant to be reachable from outside, that defeats the point. So I set this up with the DNS-01 challenge and a small server called acme-dns: one acme-dns container answers the challenges, pfSense forwards DNS traffic to it, and a single machine running acme.sh issues every certificate and copies it to the servers that use it.

This post walks through the whole setup. All names and addresses below are examples.



Why DNS-01 and why acme-dns

With DNS-01, control of a name is proven by publishing a TXT record at `_acme-challenge.<name>`. Let's Encrypt looks it up in public DNS and, if the value matches, issues the certificate. The server that will use the certificate never has to be reachable. As a bonus, DNS-01 is the only challenge that can issue wildcard certificates.

The catch is that the ACME client has to change DNS records, automatically, every time a certificate is issued or renewed. Many DNS providers don't have an API, and the ones that do usually hand out keys that can edit the whole zone. A leaked key like that can take over the whole domain.

acme-dns fixes this. It's a tiny authoritative DNS server with an HTTP API that can only do one thing: update a TXT record on a random subdomain it owns, such as `d420c923-bbd7-4056-ab64-c3ca54c9b3cf.auth.example.org`. A CNAME points `_acme-challenge.nas.example.org` at that subdomain, set once, and Let's Encrypt follows the CNAME during validation. If the acme-dns credentials ever leak, all someone can do is change one TXT record.

How the pieces fit

Who does what:

- acme-dns: answers DNS queries for `auth.example.org` and its subdomains on port 53, and takes TXT updates through its API. It is not a certificate authority and doesn't issue anything.
- pfSense: forwards port 53 from the WAN to the acme-dns container, so Let's Encrypt can reach it.
- acme.sh: runs on one machine on the LAN, asks Let's Encrypt for certificates, writes the challenge values into acme-dns through the API, then copies the finished certificates to each server.
- Let's Encrypt: queries public DNS, follows the CNAME to acme-dns and checks the TXT value.

The example values used throughout:

- `example.org`: the domain.
- `auth.example.org`: the name of the acme-dns server.
- `203.0.113.10`: the public WAN address.
- `192.168.10.53`: the machine running the acme-dns container.
- `192.168.10.20`: the machine running acme.sh.
- `nas.example.org`, `git.example.org`: services on the LAN that need certificates.

The order matters a little. acme-dns gets its own API certificate through DNS, so public DNS and the port forward need to be ready before the container starts for real.

Step 1: DNS records at the DNS provider

In the zone for `example.org`, at whoever hosts the DNS today, add two records:

```
auth.example.org.  A   203.0.113.10
auth.example.org.  NS  auth.example.org.
```

The NS record hands the whole `auth.example.org` subzone to acme-dns. The A record tells the world where that server is. With an IPv6 address from the ISP, an AAAA record goes in too, and pfSense forwards IPv6 as well.

If the WAN address changes from time to time, these records have to follow it, which means dynamic DNS for `auth.example.org` and a matching update to the acme-dns config.

Step 2: forward port 53 in pfSense

In pfSense, go to Firewall, NAT, Port Forward and add a rule:

- Interface: WAN
- Address Family: IPv4
- Protocol: TCP/UDP
- Destination: WAN address
- Destination port range: DNS (53)
- Redirect target IP: 192.168.10.53
- Redirect target port: DNS (53)
- Description: acme-dns (DNS-01 challenges)
- Filter rule association: Add associated filter rule

Save and apply. The associated rule is what actually lets the traffic in on the WAN.

A few things to watch here:

- Forward port 53 only. The acme-dns API stays inside the LAN and doesn't need any rule on the WAN.

```
# config/config.cfg (excerpt)
[general]
listen = "0.0.0.0:53"
protocol = "both"
domain = "auth.example.org"
nsname = "auth.example.org"
records = [
    "auth.example.org. A 203.0.113.10",
    "auth.example.org. NS auth.example.org.",
]

[api]
port = "443"
tls = "letsencrypt"
```

- Forward it to one machine. A port forward points to a single address, and one acme-dns is enough for every name on the network.
- Opening 53 doesn't create an open resolver. acme-dns only answers for its own zone and doesn't look up anything else for anyone.
- Don't geoblock it. Let's Encrypt validates from several places around the world at once, and it recommends against geoblocking the traffic it uses to validate. With pfBlockerNG country blocks on the WAN, this rule stays out of them.
- If the ISP uses carrier-grade NAT or blocks inbound port 53, this setup can't work from home, so that's the first thing to check.

One more pfSense setting makes life easier inside the LAN. Go to Services, DNS Resolver, Host Overrides and add `auth` in domain `example.org` pointing to `192.168.10.53`. Machines on the LAN then reach acme-dns directly by name, and the API's certificate matches the name they use.

Step 3: run acme-dns in Docker

On the acme-dns machine, create a folder with two subfolders, one for the config and one for the database:

```
mkdir -p ~/acme-dns/config ~/acme-dns/data
cd ~/acme-dns
```

Then `config/config.cfg`:

```
[general]
listen = "0.0.0.0:53"
protocol = "both"
domain = "auth.example.org"
nsname = "auth.example.org"
nsadmin = "admin.example.org"
records = [
    "auth.example.org. A 203.0.113.10",
    "auth.example.org. NS auth.example.org.",
]
debug = false

[database]
engine = "sqlite"
connection = "/var/lib/acme-dns/acme-dns.db"

[api]
ip = "0.0.0.0"
disable_registration = false
port = "443"
tls = "letsencrypt"
acme_cache_dir = "/var/lib/acme-dns/api-certs"
notification_email = ""
corsorigins = ["*"]
use_header = false
header_name = "X-Forwarded-For"

[logconfig]
loglevel = "info"
logtype = "stdout"
logformat = "text"
```

The parts worth explaining:

- `listen`: the project's template says `127.0.0.1:53`, which is fine for a plain install but useless inside a container. `0.0.0.0:53` listens on every interface of the container, and Docker decides what reaches it.
- `domain`, `nsname` and `records`: the zone acme-dns is responsible for, and the same two records created at the DNS provider in step 1.
- `nsadmin`: an email address written with a dot instead of the `@`, as DNS zones do. It only shows up in the zone's SOA record.
- `engine = "sqlite"`: version 2 of acme-dns renamed this from `sqlite3`. Older guides still use the old name.
- `tls = "letsencrypt"`: acme-dns gets a certificate for its own API from Let's Encrypt, using a DNS challenge against itself. That works even though the API is never exposed to the internet. While testing, `letsencryptstaging` avoids the rate limits.
- `acme_cache_dir`: where that API certificate is stored. The template uses a relative path, which would live inside the container and vanish when it's recreated, so it goes into the data folder here.

Now `docker-compose.yml`:

```
services:
  acmedns:
    image: joohoi/acme-dns:v2.0.2
    restart: unless-stopped
    ports:
      - "192.168.10.53:53:53/udp"
      - "192.168.10.53:53:53/tcp"
      - "192.168.10.53:443:443/tcp"
    volumes:
      - ./config:/etc/acme-dns:ro
      - ./data:/var/lib/acme-dns
```

Publishing the ports on the machine's LAN address, instead of on all addresses, avoids a common clash: on systems with systemd-resolved, a local DNS stub already holds port 53 on `127.0.0.53`, and Docker can't bind `0.0.0.0:53` next to it. It also keeps the API from showing up on any other interface the machine has.

Start it and watch the log:

```
docker compose up -d
docker compose logs -f
```

The log should show acme-dns starting and, shortly after, the API certificate being obtained.

CHECK IT FROM OUTSIDE

From any machine, ask a public resolver about the new zone:

```
dig +short NS auth.example.org @1.1.1.1
dig +short A auth.example.org @1.1.1.1
```

The first should return `auth.example.org.` and the second the WAN address. Then, from the LAN, check the API:

```
curl https://auth.example.org/health
```

A 200 response with no certificate error means DNS, the port forward, the host override and the API certificate all work.

Step 4: register each name

Each name that needs a certificate gets its own registration in acme-dns. From the acme.sh machine:

```
curl -s -X POST https://auth.example.org/register \
  -H 'Content-Type: application/json' \
  -d '{"allowfrom": ["192.168.10.20/32"]}'
```

The answer looks like this:

```
{
  "username": "eabcdb41-d89f-4580-826f-3e62e9755ef2",
  "password": "pbAXVjlIOE01xbut7YnAbkhMQIkcwoHO0ek2j4Q0",
  "fulldomain": "d420c923-bbd7-4056-ab64-c3ca54c9b3cf.auth.example.org",
  "subdomain": "d420c923-bbd7-4056-ab64-c3ca54c9b3cf",
  "allowfrom": ["192.168.10.20/32"]
}
```

`allowfrom` limits updates for this registration to the acme.sh machine. Even with the username and password, a request from anywhere else is refused. If acme.sh runs on the same machine as the container, Docker may show the requests as coming from its own bridge address instead; in that case leave `allowfrom` out or use the bridge network.

Store the four values somewhere safe, then go back to the DNS provider and create the CNAME, using the `fulldomain` from the answer:

```
_acme-challenge.nas.example.org.  CNAME  d420c923-bbd7-4056-ab64-
c3ca54c9b3cf.auth.example.org.
```

Repeat for every name: `git.example.org` gets its own registration and its own CNAME. It's a bit more work than one shared registration, and it means a leaked credential only exposes one name. A wildcard and its base domain (`*.example.org` and `example.org`) use the same `_acme-challenge.example.org` record, so one registration covers both; acme-dns keeps two TXT values at once for exactly this case.

This is the only time the DNS provider gets touched for a given name. Renewals happen through the API.

When every name is registered, set `disable_registration = true` in the config and restart the container, so nobody else on the LAN can create registrations. Turn it back on to add a service.

Step 5: issue certificates with acme.sh

On the acme.sh machine, install it:

```
curl https://get.acme.sh | sh -s email=you@example.org
```

The installer also adds a daily cron job that renews certificates when they get close to expiring. acme.sh uses ZeroSSL as its default CA, so Let's Encrypt takes one extra command, run once:

```
acme.sh --set-default-ca --server letsencrypt
```

Now issue the certificate for `nas.example.org`, giving acme.sh the values from its registration:

```
export ACMEDNS_BASE_URL="https://auth.example.org"
export ACMEDNS_USERNAME="eabcdb41-d89f-4580-826f-3e62e9755ef2"
export ACMEDNS_PASSWORD="pbAXVjlIOE01xbut7YnAbkhMQIkcwoHO0ek2j4Q0"
export ACMEDNS_SUBDOMAIN="d420c923-bbd7-4056-ab64-c3ca54c9b3cf"

acme.sh --issue --dns dns_acmedns -d nas.example.org --staging
```

`--staging` uses Let's Encrypt's test environment, which has generous limits and issues certificates nobody trusts. Once it works, run the same command with `--force` instead of `--staging` to get the real one.

acme.sh saves those four values in the configuration for that certificate, so renewals don't need the variables again. For the next name, export the values from its own registration and issue again. Without the variables, acme.sh registers a new account by itself and stops to ask for the CNAME, which is fine on the first run but useless in a script.

Step 6: send the certificates to the servers

With one machine issuing everything, the certificates still have to reach the servers that use them. acme.sh has deploy hooks for this, and the `ssh` hook copies files over SSH and runs a command on the other side:

```
export DEPLOY_SSH_USER="deploy"
export DEPLOY_SSH_SERVER="192.168.10.31"
export DEPLOY_SSH_KEYFILE="/etc/ssl/private/nas.example.org.key"
export DEPLOY_SSH_FULLCHAIN="/etc/ssl/certs/nas.example.org.fullchain.pem"
export DEPLOY_SSH_REMOTE_CMD="sudo systemctl reload nginx"

acme.sh --deploy -d nas.example.org --deploy-hook ssh
```

acme.sh remembers the hook and its settings, and runs it again after every renewal. For this to work unattended:

- The acme.sh machine needs an SSH key that logs into the server without a password.
- The `deploy` user needs to write to those two files, and to run the reload command. A sudoers line that allows only `systemctl reload nginx` is enough.

- Change the reload command to whatever the service needs: nginx, Caddy, a container restart.

Some devices can't take a certificate over SSH at all: appliances, NAS web panels, routers. acme.sh has specific deploy hooks for many of them, so the list is worth a look before writing a new one.

When it doesn't work

Most failures are in the DNS path. Check in this order:

1. `dig +short NS auth.example.org @1.1.1.1` returns `auth.example.org.`
2. `dig +short A auth.example.org @1.1.1.1` returns the current WAN address.
3. `dig +short CNAME _acme-challenge.nas.example.org @1.1.1.1` returns the right `fulldomain`.
4. `dig TXT d420c923-bbd7-4056-ab64-c3ca54c9b3cf.auth.example.org @1.1.1.1` answers at all. Before any issuance the TXT may be empty, but the query shouldn't time out.
5. `docker compose logs` on the acme-dns machine shows the queries arriving and the API updates being accepted.

If step 1 or 2 fail, the problem is at the DNS provider. If 4 times out, it's the port forward, the firewall rule or the ISP. If the logs show updates being refused, look at `allowfrom` and the credentials.

Test from outside the network, too. A query that works from the LAN may be answered by pfSense's host override and never touch the public path that Let's Encrypt uses.

What to keep safe

- The acme.sh machine holds every private key and every acme-dns credential. It's the most sensitive box in this setup, so treat it that way.
- Back up the acme-dns `data` folder. The database holds the registrations, and losing it means registering every name again and changing every CNAME.
- Keep the API off the WAN. Nothing outside the LAN needs it.

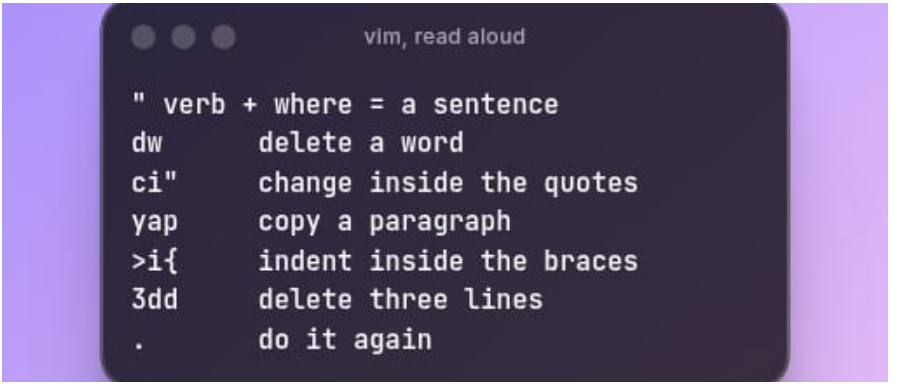
After the first round of CNAMEs, there's nothing left to do by hand. acme.sh renews each certificate before it expires, acme-dns answers the challenge, and the deploy hook puts the new files in place and reloads the service. ■

$ cd ~pablo

04 VIM · NEOVIM · TERMINAL 19 NOVEMBER 2026

Vim without the mystery

Vim feels scary until one idea clicks: modes. A friendly, step-by-step guide to getting out, moving around, editing with verbs and text objects, search and replace, a small vimrc that doesn't break the defaults, and how to practice.



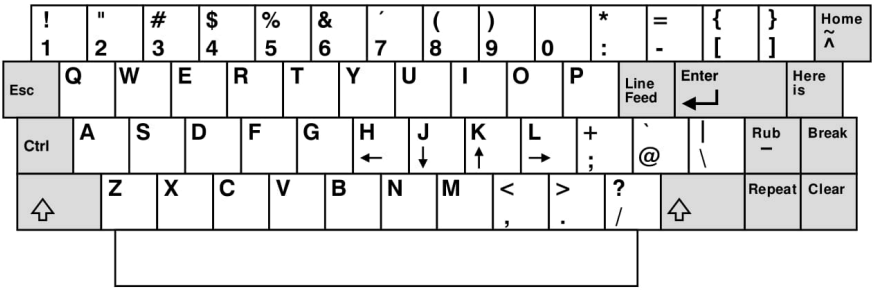
Vim has a reputation. The most famous joke about it is that nobody knows how to leave: a file opens by accident, the keyboard stops doing what keys normally do, and the only way out seems to be closing the terminal.

All of that confusion comes from one idea that nobody explains at the door. Once it clicks, Vim stops being a puzzle and starts feeling like a small language for talking to text. This post goes from "how do I get out" to editing comfortably, one step at a time. The best way to read it is with a terminal open, running `vim practice.txt`, trying things along the way. Nothing here can break.

First things first: the way out

If only one thing sticks, make it this. Press `Esc`, then type one of these and hit Enter:

- `:q`: quit. Vim refuses if something changed and wasn't saved, which is a feature.



- `:q!`: quit and throw away the changes.
- `:w`: save.
- `:wq`: save and quit. `:x` and `ZZ` do the same.

The colon opens a small command line at the bottom of the screen, and whatever is typed there runs on Enter. When everything feels lost, `Esc` a couple of times always leads back to where it started.

Why Vim works like this

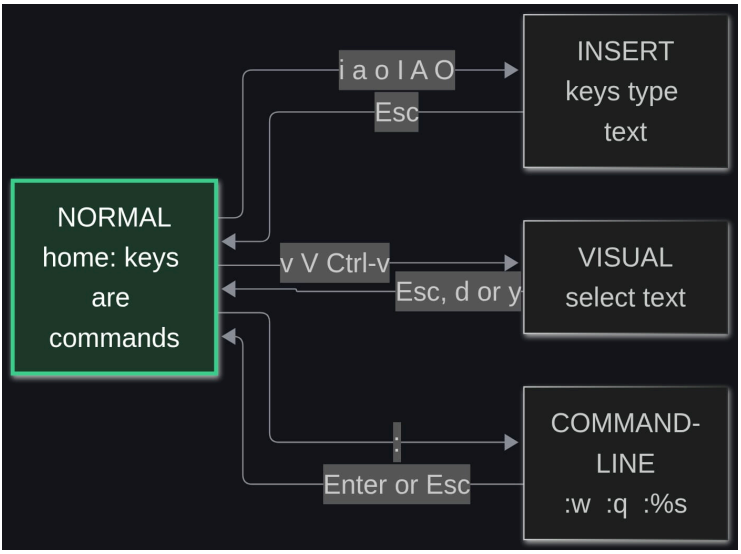
The story starts in 1976, at UC Berkeley. Bill Joy added a full-screen "visual" mode to `ex`, a line editor,[1] and that mode became `vi`. He wrote it on a Lear Siegler ADM-3A terminal, and that keyboard explains a lot. It had no separate arrow keys: the arrows were printed on `H`, `J`, `K` and `L`. And `Esc` sat where `Tab` is on today's keyboards, one finger away. Moving with `hjkl` and pressing `Esc` all the time made perfect sense on that machine, and the habit stuck.

In 1988 Bram Moolenaar, a Dutch programmer, started writing a vi clone for the Amiga. He called it Vi IMitation. It went public in 1991, and around version 2.0, in 1993, the name became Vi IMproved. Bram looked after Vim for more than thirty years, until he died in 2023.[2] A group of longtime contributors maintains it now, and Vim 9.2 came out in February 2026.

There's also Neovim, a fork started in 2014. Everything in this post works the same way in both.

1 `ex` was itself an extended version of `ed`, the original Unix editor from 1969, which shows one line at a time. Vim still has the whole `ex` inside it: every command that starts with `:` comes from there.

2 Vim has always been charityware. Its license asks users who like it to donate to ICCF Holland, a charity Bram supported that helps children in Uganda, and `:help uganda` still explains it.



The one idea: modes

In most editors, the keyboard always types. Press `d` and a “d” shows up on the screen.

Vim has modes, and the keyboard means different things in each one: - Normal mode: where Vim starts, and where most of the time goes. Keys are commands: `d` deletes, `w` jumps to the next word, `u` undoes. - Insert mode: the keyboard types, like in any other editor. The bottom of the screen says `-- INSERT --`. - Visual mode: select text first, then act on the selection. - Command-line mode: the `:` from the way out. It runs longer commands, like saving or search and replace.

The trick is to see Normal mode as home. Insert mode is where text gets typed, and `Esc` comes back. Editing in Vim is mostly moving around and changing things, and typing new text is a short trip. That’s why the “typing” mode isn’t the default.

Typing text

From Normal mode, any of these goes into Insert mode, each starting in a different spot:

- `i`: insert before the cursor.
- `a`: insert after the cursor (think “append”).
- `I`: insert at the beginning of the line.

- `A`: insert at the end of the line.
- `o`: open a new line below and start typing there.
- `O`: open a new line above.

Type the text, press `Esc`, and it's back home. `A` is worth trying on any line: it's the fastest way to add something at the end, no matter where the cursor is.

Moving around

The arrow keys work, and there's no shame in using them at first. These are the moves worth learning next, because they save the most time:

- `h j k l`: left, down, up, right. `j` looks like an arrow pointing down, if that helps.
- `w`: jump to the start of the next word.
- `b`: jump back to the start of the previous word.
- `e`: jump to the end of the word.
- `0`: go to the start of the line.
- `^`: go to the first character that isn't a space.
- `$`: go to the end of the line.
- `gg`: go to the top of the file.
- `G`: go to the bottom of the file.
- `42G`: go to line 42. `:42` does the same.
- `f(`: jump forward to the next `(` on this line. Works with any character.
- `%`: jump between matching brackets.
- `Ctrl-d` and `Ctrl-u`: scroll half a page down and up.

Most moves accept a number in front: `3w` jumps three words, `5j` goes down five lines.

The grammar: verb, then where

This is the part where Vim starts to make sense. Commands are built like short sentences: a verb, then a place.

The verbs:

- `d`: delete.
- `c`: change (delete, then switch to Insert mode).
- `y`: yank, which is Vim's word for copy.
- `>` and `<`: indent and unindent.

Any verb in front of any motion from the last section makes a command:

- `dw`: delete to the next word.
- `d$`: delete to the end of the line.
- `cw`: change to the end of the word.

- `y$`: copy to the end of the line.
- `dgg`: delete everything up to the top of the file.

A verb typed twice acts on the whole line: `dd` deletes a line, `yy` copies it, `cc` changes it, `>>` indents it. A number multiplies: `3dd` deletes three lines.

Then there are text objects, which are the real magic. Instead of "from here to there", they mean "this thing the cursor is in", so the cursor can be anywhere inside it:

- `iw`: the word under the cursor ("inner word").
- `i"`: the text inside quotes.
- `i(`: the text inside parentheses.
- `ip`: the paragraph.
- `a"`, `a(`, `ap`: the same, but including the quotes, parentheses or blank line around it.

So `ciw` reads "change inner word", `di(` is "delete inside parentheses", and `yap` is "yank a paragraph". Picture this line, with the cursor somewhere on the word `print`:

```
print("hello")
```

Type `ci"`, then `world`, then `Esc`. Vim finds the next quoted text on the line, empties it, waits for the new text, and the line becomes `print("world")`. With a handful of verbs and objects, the combinations come for free, with nothing to memorize one by one.

Two more keys complete the set:

- `.`: repeat the last change. After changing a word with `ciw`, move to another word and press `.`.
- `u`: undo. `Ctrl-r` redoes.

Copy, cut and paste

- `yy`: copy the line.
- `p`: paste after the cursor (or below, for whole lines).
- `P`: paste before the cursor (or above).
- `x`: delete the character under the cursor.

Anything deleted with `d`, `c` or `x` also goes to the same place as a copy, so deleting works like cutting: `dd` on one line, move somewhere else, `p`, and the line has moved. `ddp` swaps a line with the one below.

Vim keeps its own clipboard, separate from the system's. To reach the system clipboard, put `"+` in front: `"+yy` copies the line to it, `"+p` pastes from it. This only works when Vim was built with clipboard support. `vim --version | grep clipboard` tells: `+clipboard` means yes, `-clipboard` means no.

On Debian and Ubuntu, the `vim-gtk3` package brings a Vim that has it, even in the terminal.

Search and replace

- `/word`, then Enter: search forward for "word".
- `?word`: search backward.
- `n` and `N`: jump to the next and previous match.
- `*`: search for the word under the cursor.
- `:noh`: clear the search highlight.

Search and replace happens on the command line:

```
:%s/old/new/g
```

`%` means the whole file, `s` is substitute, and `g` replaces every match on each line, not only the first. A `c` at the end (`:%s/old/new/gc`) makes Vim ask before each replacement, which is a good habit while learning.

Selecting text

- `v`: select character by character.
- `V`: select whole lines.
- `Ctrl-v`: select a rectangle, a block of columns.

Moving grows the selection, and then a verb acts on it: `d` deletes it, `y` copies it, `>` indents it, `c` changes it.

Block selection has a trick worth showing a friend. Put the cursor on the first of a few lines, press `Ctrl-v`, move down with `j` to cover them, press `I`, type `-`, and press `Esc`. The text appears at the start of every selected line:

```
- milk
- eggs
- bread
```

Files and windows

- `:e notes.txt`: open another file.
- `:w copy.txt`: save under a new name.
- `:sp` and `:vsp`: split the window horizontally and vertically.
- `Ctrl-w w`: jump to the next window.
- `Ctrl-w h`, `Ctrl-w j`, `Ctrl-w k`, `Ctrl-w l`: jump to the window on the left, below, above or on the right.
- `:ls` and `:b 2`: list open files and switch to number 2.



A small vimrc

Vim reads its settings from `~/.vimrc`. There's one catch that trips up almost everyone: without a vimrc, Vim loads a file called `defaults.vim` with sensible settings (syntax colors, search as you type, a few lines of margin when scrolling). The moment a personal vimrc exists, that file stops loading, and Vim suddenly looks worse. The fix is two lines at the top, straight from Vim's own help.

Here's a small starting point:

```
unlet! skip_defaults_vim
source $VIMRUNTIME/defaults.vim

set number
set relativenumber
set expandtab
set shiftwidth=4
set softtabstop=4
set ignorecase
set smartcase
set hlsearch
set undofile
set undodir=~/.vim/undo//
```

What each part does:

- The first two lines bring `defaults.vim` back.

- `number` and `relativenumber`: show the current line's number and, on the others, how far away they are. With that, `5j` or `d3k` stop being guesswork.
- `expandtab`, `shiftwidth=4` and `softtabstop=4`: indent with four spaces instead of tab characters. The numbers are a matter of taste.
- `ignorecase` and `smartcase`: searches ignore case, unless the search has a capital letter.
- `hlsearch`: highlight every match of the last search.
- `undofile` and `undodir`: keep the undo history after the file closes, so `u` still works tomorrow. The folder has to exist first: `mkdir -p ~/.vim/undo`.

Settings are best added one at a time, when they're missed. A vimrc copied from someone else is full of settings nobody remembers having.

Where to practice

Vim ships with a tutorial. `vimtutor` in the terminal opens a practice file with lessons done inside Vim itself. It takes about 30 minutes, and it's the quickest way to get the basic keys into the fingers. Neovim has its own version, opened with `:Tutor`.

Inside Vim, `:help` covers everything, and it searches by topic: `:help text-objects`, `:help iw`, `:help :s`. The help opens inside Vim, so reading it is practice as well.

These keys also show up in more places than expected. `less` and `man` pages move with `j` and `k` and search with `/`, many code editors have a Vim mode, and browsers have extensions that add one. Learning them pays off outside Vim too.

Trying to learn everything at once doesn't work. Two or three new commands a week, used until the fingers know them, then a few more. After a while the keys disappear and the thinking happens in sentences: change this word, delete this paragraph, copy these three lines. ■



05 BANANY · RUNV-CLUB · BLOGGING 19 NOVEMBER 2026

banany.lol: a free blog from a single markdown file

banany.lol turns one markdown file into a blog with an RSS feed. Free, no ads, no tracking, no accounts. A step-by-step guide to creating one, writing the file, publishing from the browser or the terminal, and what the service keeps about its users.



Most ways to start a blog begin with a sign-up form: an email, a password, a confirmation link, then a dashboard full of settings. banany.lol asks for a name and a text file. That's the whole setup.

It turns one markdown file into a blog with an RSS feed. It's free, it has no ads, no tracking scripts, no accounts and no cookies. It's part of runv.club, the public Unix server I run, and this post walks through how to use it from start to finish.

What comes with it

- A blog at `banany.lol/yourname`.
- An RSS feed, so people can follow it from any feed reader.
- A light theme, a dark theme, or one that follows each reader's system.
- Nothing else to manage. No account, no dashboard, no plugins to update.



Step 1: pick a name

On the home page, type a name after `banany.lol/`, add a title if you like, and press Create.

The page answers with the handle, the blog's address and a token. The token is a long password that proves the blog is yours, and it's the only way to publish to it. It shows up once, right there, and never again. Copy it somewhere safe, like a password manager: a lost token can't be recovered or reset, and anyone who has it can replace the blog.

Step 2: write the file

The whole blog lives in one markdown file. The rules fit in three lines:

- Everything before the first post becomes the intro at the top of the blog.
- Each post starts with a small block between two `---` lines, holding its `title` and `date`.
- An optional block at the very top sets the theme: `auto`, `light` or `dark`.

Here's a complete file:

```
---
theme: auto
---

Notes from a small balcony garden.
```

```
---
title: The basil survived
date: 2026-11-12
---

Three weeks of rain, and it's still standing.

---
title: First tomato
date: 2026-11-19
---

Small, a bit sour, and completely worth it.
```

Below each post's block goes regular markdown. Images work too, as long as they're hosted on another site with an `https` address.

Step 3: publish

Back on the home page, fill in the handle and token, then drag the `.md` file onto the box that says "drop your .md file here". The page answers with how many posts it published and the blog's address.

The home page sends the file to a small API, and from the terminal `curl` can do the same:

```
curl -X POST \
  -H "x-blog-token: YOUR_TOKEN" \
  -F "file=@blog.md" \
  https://banany.lol/api/blogs/yourname/publish
```

Every upload replaces the whole blog with the new file. To add a post, add it to the file and upload it again. To fix a typo, fix it in the file and upload again. To delete a post, remove it from the file and upload again.

Why one file works well

Having the whole blog in a single text file has some nice side effects:

- The backup is the file itself. A copy on the computer holds everything.
- Any editor works, from Vim to the notes app on your phone.
- It fits in git, with every change ever made.
- Moving somewhere else is easy, since every post is already in plain markdown.

That first point matters, because banany.lol comes with no warranty: it's a free service, and the terms ask each owner to keep a copy of the file.

The limits

banany.lol is free for everyone, so it has a few limits to keep it fair:

- Each connection can create 2 blogs every 30 days.
- The file can be up to 1 MB. That's a lot of text: a long post is a few dozen KB.[1]
- A blog can have up to 300 posts.

And the usual rules apply: nothing illegal, no spam or link farms, no harassment, no impersonation, no malware. To report a blog that breaks them, write to `abuse@runv.club` with its address.

What it keeps

Very little, and the privacy page lists all of it:

- Request logs with IP address, date and time, kept for 6 months, because Brazil's Marco Civil da Internet requires it.[2]
- A keyed hash of the IP[3] when a blog is created, used only for the 2-blogs limit and deleted after 30 days.
- The token, the file and the pages made from it, for as long as the blog exists.

No accounts, no email, no analytics, no cookies. The light and dark switch saves the choice in the reader's own browser and nowhere else.

One thing to keep in mind about images: if a post shows an image from another site, the reader's browser loads it straight from there, and that site sees the reader's IP address. That's true anywhere on the web, but it's worth knowing when choosing where to host images.

And runv.club?

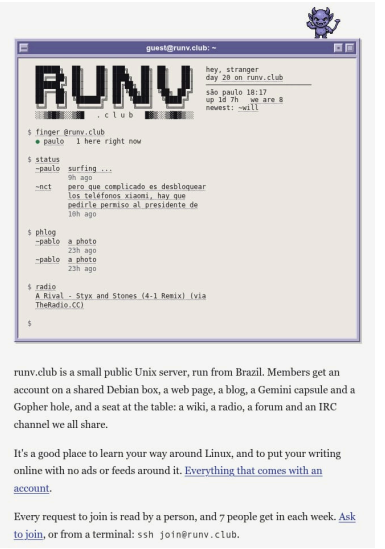
runv.club is a small public Unix server, run from Brazil. Members get an account on a shared Debian machine, a web page, a blog, a Gemini capsule and a Gopher hole,[4] plus a wiki, a radio, a forum and an IRC channel that everyone shares. A person reads every request to join, and seven people get in each week. The request can go through the website or a terminal, with `ssh join@runv.club`.

1 Plain English text takes about one byte per character, so 1 MB holds around a million characters, roughly the length of a long novel.

2 Article 15 of Law 12.965/2014 requires application providers to keep access logs for six months, under confidentiality. The privacy page cites the same article.

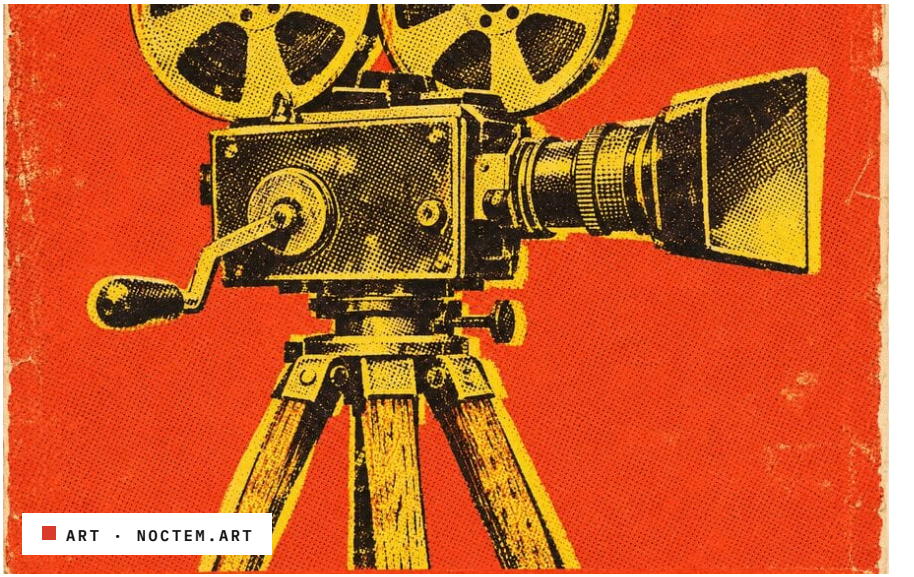
3 A keyed hash, or HMAC, mixes the address with a secret that only the server knows. The result can't be turned back into the IP, but the same IP always gives the same result, which is all a limit needs.

4 Gopher (1991) and Gemini (2019) are small protocols for text sites that live alongside the web. A Gemini site is called a capsule, and a Gopher site a hole.



banany.lol needs none of that: no account and no shell, only the file. For anyone who wants more than a blog, runv.club is the place to ask.

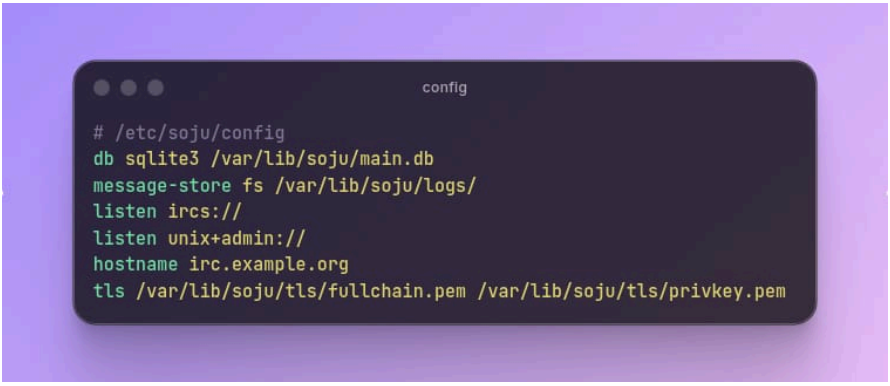
A note: this is the first post on this blog with margin notes. I may go back to some older posts and update them with more references and information. ■



06 IRC · SOJU · DEBIAN 19 NOVEMBER 2026

Setting up soju as an IRC bouncer on Debian

A step-by-step guide to running soju, a modern IRC bouncer, on Debian: installing the package, getting a real TLS certificate, creating the first user, connecting clients like goguma, senpai, motd or irssi, and managing networks and channels with BouncerServ.



Speaking of soju: in my last note I mentioned motd, an Android client that gets along well with it. This post is the other half. It walks through setting up soju on a Debian server, from the first `apt install` to a bouncer with a real certificate, its own accounts and a few networks, ready for any client.

What a bouncer does

IRC is old and simple: a nick is only in a channel while its client is connected. Close the laptop, lose the signal on the phone, and it's gone. Whatever was said in the meantime never arrives.

A bouncer fixes that. It's a small program that runs on a server, stays connected to the IRC networks around the clock, and keeps what happens while nobody is looking. The clients connect to the bouncer instead of connecting to the networks directly. When someone comes

back, the history is waiting, the nick never dropped, and every device sees the same conversations.

soju is a modern bouncer written in Go. It supports several users on the same server, many IRCv3 extensions, chat history playback, channels that can be "detached" to keep logging without cluttering the client, and an extension called `soju.im/bouncer-networks` that lets good clients see and manage every network over a single connection.

What it takes

- A Debian 13 (trixie) server that stays on, like a small VPS or a home server
- A user with `sudo`
- A domain name pointing to the server, for example `irc.example.org`, for a real TLS certificate
- Port 6697/tcp open for IRC over TLS, and port 80/tcp open while the certificate is requested

Debian stable ships soju 0.9.0, which covers everything in this guide. Testing and unstable carry newer versions, for anyone who wants the latest features.

Step 1: install the package

```
sudo apt update
sudo apt install soju
```

The package does a lot of the boring work:

- creates a system user called `soju`, with its data in `/var/lib/soju`
- installs a systemd service called `soju.service`, already enabled
- writes a default config file at `/etc/soju/config`
- installs the `soju`, `sojuctl` and `sojudb` commands, plus `soju-utils` with extra tools, like an importer for people moving from ZNC

Here's the default config Debian ships:

```
db sqlite3 /var/lib/soju/main.db
message-store fs /var/lib/soju/logs/
listen ircs://
listen unix+admin://
tls /etc/ssl/certs/ssl-cert-snakeoil.pem /etc/ssl/private/ssl-cert-
snakeoil.key
```

Line by line:

- `db sqlite3` keeps users, networks and channels in a SQLite file. PostgreSQL is also supported, but SQLite is plenty for a personal bouncer.

- `message-store fs` saves the chat logs as plain files on disk, in the same format ZNC uses. The other option, `message-store db`, keeps messages in the database with a full-text search index, which helps clients that can search the history.
- `listen ircs://` listens for IRC over TLS on the standard port, 6697.
- `listen unix+admin://` opens an admin socket at `/run/soju/admin`, which is what `sojuctl` talks to.
- `tls` points to Debian's "snakeoil" certificate,[1] a self-signed certificate created on install. It works for a first test, but clients will rightly complain that they can't verify it. The next step replaces it.

Step 2: a real certificate

Install certbot and ask Let's Encrypt for a certificate. The `--standalone` mode starts its own little web server on port 80 for a moment, so that port must be free and open:

```
sudo apt install certbot
sudo certbot certonly --standalone -d irc.example.org
```

There's one detail to solve. Certbot saves the private key where only root can read it, and soju runs as the `soju` user. The cleanest fix is a small deploy hook that copies the certificate to soju's folder with the right owner, and tells soju to reload it. Certbot runs every script in that folder after each successful renewal.

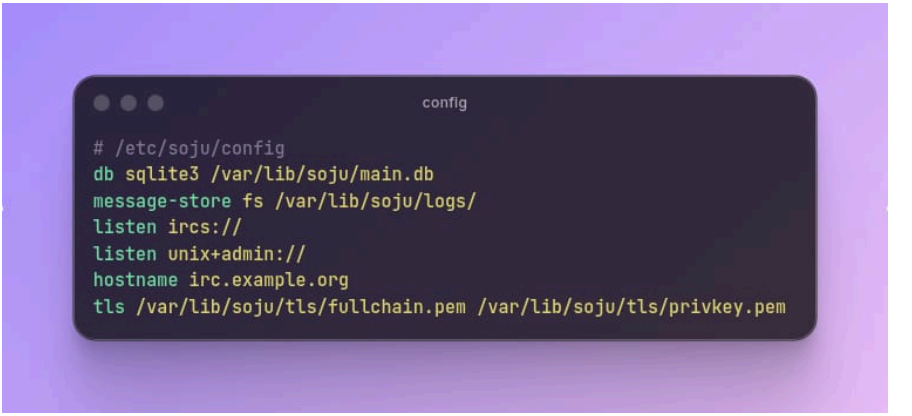
Create `/etc/letsencrypt/renewal-hooks/deploy/soju.sh`:

```
#!/bin/sh
set -e
DOMAIN=irc.example.org
install -d -m 0750 -o soju -g soju /var/lib/soju/tls
install -m 0640 -o soju -g soju /etc/letsencrypt/live/$DOMAIN/
fullchain.pem /var/lib/soju/tls/fullchain.pem
install -m 0640 -o soju -g soju /etc/letsencrypt/live/$DOMAIN/privkey.pem /
var/lib/soju/tls/privkey.pem
systemctl reload soju
```

Make it executable and run it once by hand, since the first certificate already exists:

```
sudo chmod +x /etc/letsencrypt/renewal-hooks/deploy/soju.sh
sudo /etc/letsencrypt/renewal-hooks/deploy/soju.sh
```

1 The name comes from snake oil, the fake cure-all sold from wagons in the 1800s. A snakeoil certificate promises a security that nobody can check.



Debian's certbot package comes with a systemd timer that renews certificates on its own, and from now on every renewal ends with this script. The `systemctl reload` sends soju a HUP signal, which makes it reread the config file, the certificate and the key without dropping anyone.

Now edit `/etc/soju/config` to use the new certificate and the domain:

```
plain
db sqlite3 /var/lib/soju/main.db
message-store fs /var/lib/soju/logs/
listen ircs://
listen unix+admin://
hostname irc.example.org
tls /var/lib/soju/tls/fullchain.pem /var/lib/soju/tls/privkey.pem
```

The `hostname` line tells soju its public name, which should be a full domain. Restart soju so it picks everything up:

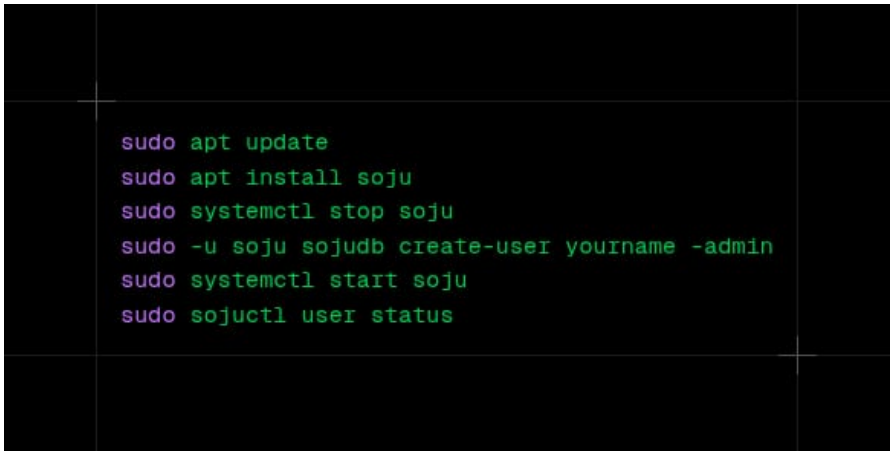
```
sudo systemctl restart soju
```

Step 3: create the first user

soju has no users after a fresh install. The simplest way to create the first one is `sojudb`, which writes straight to the database. The soju documentation asks for a restart of the daemon after changing the database this way, so stop it first:

```
sudo systemctl stop soju
sudo -u soju sojudb create-user yourname -admin
sudo systemctl start soju
```

`sojudb` asks for the password without echoing it, so it never ends up in the shell history. Running it as the `soju` user keeps the database file owned by



the right account. The `-admin` flag makes this user an administrator, which is needed later to create more users and see server statistics.

To check that everything is up:

```
sudo systemctl status soju
sudo sojuctl user status
sudo sojuctl server status
```

`sojuctl` sends commands to the running bouncer through the admin socket, which is why it needs `sudo`. If something looks wrong, the logs are one command away:

```
sudo journalctl -u soju -f
```

Step 4: open the port

With `ufw`, allowing IRC over TLS is one line:

```
sudo ufw allow 6697/tcp
```

Port 80 stays open too when certbot runs in standalone mode, since renewals need it. With another firewall, or one at the hosting provider, the idea is the same: 6697/tcp in.

Step 5: connect a client

How a client connects depends on whether it understands soju's network extension.

CLIENTS THAT UNDERSTAND SOJU

Some clients were made with soju in mind, like goguma on Android and iOS, gamja in the browser and senpai in the terminal. motd on Android also manages soju networks. With these, a single server entry is enough:

- Server: `irc.example.org`
- Port: `6697`, with TLS on
- Username: `yourname`
- Password: the soju password

The client then shows every network configured in soju, and can add new ones from its own interface.

EVERY OTHER CLIENT

Clients like irssi, WeeChat, HexChat or Emacs' ERC see the bouncer as a regular IRC server, so each network gets its own server entry. The trick is in the username, which tells soju which network is wanted:

```
yourname/irc.libera.chat
```

If that network doesn't exist in soju yet and the name is a valid hostname, soju creates it on the spot and connects. With more than one client, a name for each one at the end keeps their places in the history apart:

```
yourname/irc.libera.chat@laptop
yourname/irc.libera.chat@desktop
```

For irssi, for example:

```
/network add -user yourname/irc.libera.chat@laptop libera
/server add -auto -tls -network libera irc.example.org 6697 yourpassword
/connect libera
```

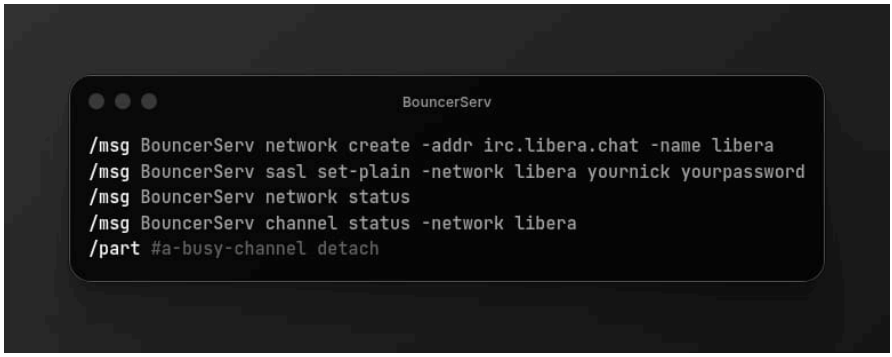
WeeChat has the soju.py script, which connects to every soju network from a single server entry.

Step 6: manage everything with BouncerServ

soju has a built-in service called BouncerServ. It answers private messages like any other user, from any connected client. `/msg BouncerServ help` lists every command.

Add a network and give it a short name:

```
/msg BouncerServ network create -addr irc.libera.chat -name libera -nick
yournick
```



Log in to the account on that network with SASL,[2] so soju identifies the nick on every reconnect:

```
/msg BouncerServ sasl set-plain -network libera yournick yourpassword
```

SASL credentials only apply after a reconnect, so update the network afterwards, which makes soju reconnect:

```
/msg BouncerServ network update libera
```

To see where things stand:

```
/msg BouncerServ network status
/msg BouncerServ channel status -network libera
```

Channels are simpler than they look. A normal `/join #channel` is enough, and soju remembers it, so it rejoins by itself on the next connection. `/part` leaves for good. To keep a busy channel logged without seeing it in the client, part it with the reason `detach`:

```
/part #a-busy-channel detach
```

The channel stays joined on the bouncer and keeps its history, and by default BouncerServ sends a notice when someone mentions the nick there. `channel update` changes that behavior, or makes a channel detach on its own after a period of silence.

An admin can also create accounts for friends:

2 SASL lets the client prove who it is during the connection itself, before joining any channel. Without it, the nick identifies later with a message to NickServ, and for a moment it shows up in channels as unidentified.

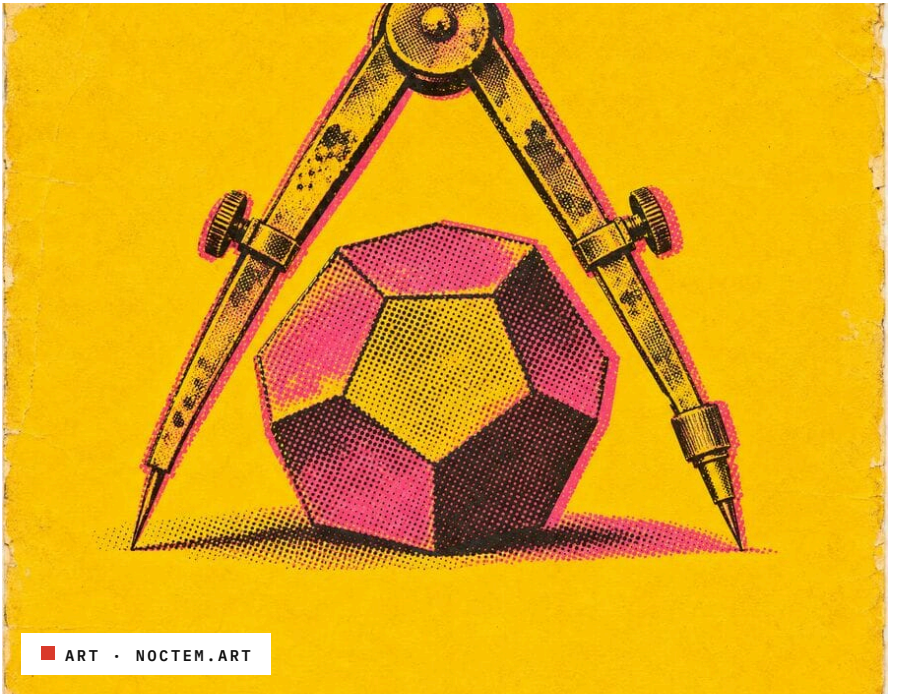
```
/msg BouncerServ user create -username friend -password a-long-password
```

Day to day

A few things worth knowing once it's running:

- **Logs** live in `/var/lib/soju/logs/`, one folder per user and network, as plain text files that can be read or grepped.
- **Backups** are easy: everything soju knows is inside `/var/lib/soju`. A copy of that folder and `/etc/soju/config` is enough to rebuild the bouncer anywhere.
- **Config changes** take effect with `sudo systemctl reload soju`, except for `listen`, `db` and the message store, which need a restart.
- **Updates** come through `apt`, like any other package.
- **Debug mode** exists (`-debug`, or `/msg BouncerServ server debug true`), but the documentation warns that it writes passwords to the logs. It's for chasing a problem, and goes off again right after.

And that's it: a bouncer that never sleeps, a real certificate that renews itself, and the networks waiting on every device. ■

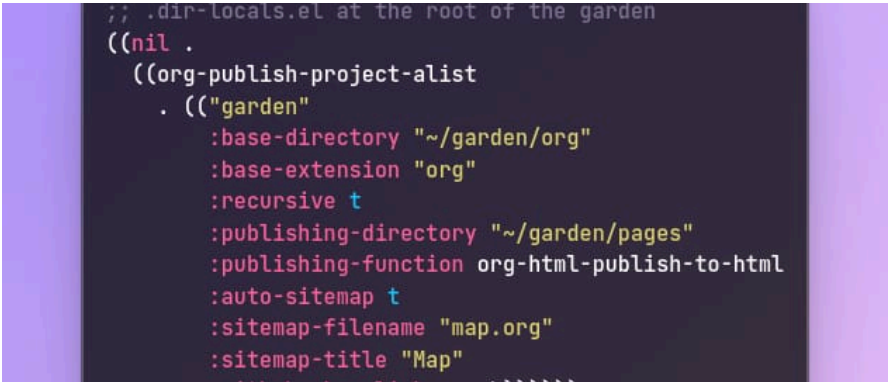


■ ART · NOCTEM.ART

07 EMACS · ORG-MODE · STATIC-SITE 20 NOVEMBER 2026

Org mode as a digital garden

Marco Bresciani runs his digital garden on Org mode and wrote up exactly how. This follow-up to my Org site guide picks up what it left out: publishing from inside Emacs, .dir-locals.el, stylesheet paths, a site map, broken links, PlantUML diagrams and more.



```
;; .dir-locals.el at the root of the garden
((nil .
  ((org-publish-project-alist
    . (("garden"
        :base-directory "~/garden/org"
        :base-extension "org"
        :recursive t
        :publishing-directory "~/garden/pages"
        :publishing-function org-html-publish-to-html
        :auto-sitemap t
        :sitemap-filename "map.org"
        :sitemap-title "Map"
```

A few days ago I published Build a website with Org mode and Emacs, a guide to turning a folder of Org files into a static site with a build script. Then I came across Marco Bresciani's page Digital Garden, where he explains how his own site is put together, and it's fantastic. His site is made with Org too, but he runs it differently, and his setup covers several things my guide left out.

This post picks up those pieces, so read it as a second part: the first one gets a site running, and this one is about living with it for a long time. Thank you, Marco, for writing it all down and publishing the configuration along with it.

Gardens and streams

A blog is a stream. Posts come out in order, the newest on top, and once something is published it mostly stays as it was. A digital garden works the other way: it's a set of notes and pages that you keep tending. You



go back, rewrite, add links between pages, plant something new next to something old. Nothing is ever really finished, and the structure comes from the links, not from the dates. Maggie Appleton's A Brief History & Ethos of the Digital Garden is a good introduction to the idea, and Marco's page opens by pointing to it.

Marco calls his site "Knowledge 枯山水". 枯山水, karesansui, is the Japanese dry landscape garden: rocks, moss and raked gravel, with no water at all.[1] His page follows the metaphor the whole way. Emacs and Org are the walls, the configuration is the gate, and the `.org` files are the garden.

Org suits this kind of site. Pages link to each other with `[[file:...]]`, a page can be reorganized in seconds by moving headings around, and nothing in the format assumes that a page has a date.

Publishing from inside Emacs

My guide builds the site from the terminal, with `emacs -Q --script build-site.el`. Marco does it from inside his running Emacs, which fits a garden where you edit a page and want to see it published right away.

His routine, in short:

1. Open the publishing configuration, `publish.el`, and run `M-x eval-buffer` to load it.
2. Open any page of the site, like `index.org`, and press `C-c C-e P p` to publish the project.
3. Find the HTML in the output folder, next to the sources.

Org only republishes the files that changed since the last run. When you want everything rebuilt, after changing the stylesheet or the export

1 The best-known karesansui is the rock garden of Ryōan-ji, in Kyoto: fifteen stones on a bed of white gravel, placed so that from any spot on the viewing veranda at least one of them stays hidden.

options for example, run `C-u M-x org-publish` and pick the project. The `C-u` prefix forces it, like the `t` in `(org-publish-all t)` in my build script.

The two ways don't compete. The script is what you want on a server or in CI, where nobody is there to press keys. The interactive way is nicer while you write.

Skipping the eval step with .dir-locals.el

Step 1 above has to happen in every new Emacs session, and it's easy to forget. Marco mentions a way around it that he plans to try: Emacs' directory variables.

A file called `.dir-locals.el` at the root of a folder sets variables for every file opened inside that folder. If one of those variables is `org-publish-project-alist`, any Org file you open in the project already knows how to publish it, and `C-c C-e P p` just works. The file holds a list of variables and values, so the projects go in as data:

```
((nil .
((org-publish-project-alist
. (("garden"
:base-directory "~/garden/org"
:base-extension "org"
:recursive t
:publishing-directory "~/garden/pages"
:publishing-function org-html-publish-to-html))))))
```

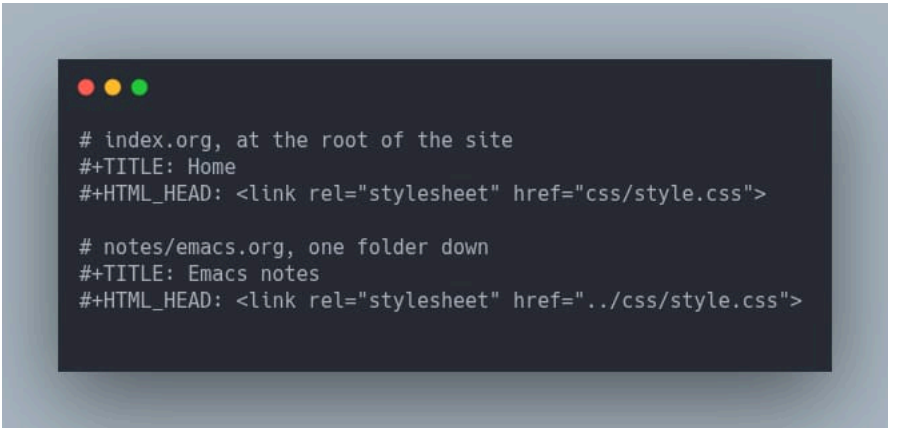
The `nil` at the start means "for every major mode". Two things to know before you rely on it. The first time you open a file there, Emacs asks whether the value is safe, because publishing settings can name functions to run.[2] Answering `!` saves your answer, so it doesn't ask again. And changes to `.dir-locals.el` only reach buffers opened after the change. For a buffer that's already open, `M-x normal-mode` reloads it.

One limit: `.dir-locals.el` sets variables, it doesn't run code. Loading a package or defining a function still belongs in your configuration or in a file you load.

The stylesheet path problem

Here my guide and Marco's setup solve the same problem in opposite ways, and the comparison is useful.

2 A project can set `:publishing-function` or `:preparation-function`, so a `.dir-locals.el` from someone else could make Emacs run code when you publish. That's why it asks before trusting the value.



A stylesheet has to be linked from every page, and the link has to point to the right place. My guide puts it in the build script, with an absolute path:

```
org-html-head "<link rel=\"stylesheet\" href=\"/css/style.css\">"
```

The leading `/` means “from the root of the site”, so the same line works on every page, at any depth. The cost is that it only works through a web server: open the HTML file straight from disk and the page has no styles. That’s why the guide has a preview step with `python3 -m http.server`.

Marco went the other way. He puts a relative link at the top of each Org file, so the path changes with the folder the file sits in:

A relative path works from disk and from a server, and it also works when you export a single file with `C-c C-e h h`. The cost is that every file carries its own line, with the right number of `../` for its depth. He notes that he’d rather set it once in the publishing configuration, but one value there can’t give each folder its own relative path, so for now the line stays in every file.

Pick the trade you prefer: one absolute path and a preview server, or one line per file and pages that open anywhere.

A map instead of an index

In my guide, the sitemap feature builds the blog index: one list of posts, newest first. Marco uses the same feature for the whole site. His project is recursive, and the sitemap is the “□□□ Map”, a page that lists every page

in the garden, grouped by folder. In a garden, that's the page a reader needs most.

The settings behind it:

```
:recursive t
:auto-sitemap t
:sitemap-filename "map.org"
:sitemap-title "Map"
:sitemap-style tree
```

`tree` is the default style. Folders become nested lists, so the map follows the way you organized the files. Two more options give every page a way back: set `:html-link-home` and `:html-link-up`, and Org adds small UP and HOME links at the top of each page.

Options worth borrowing

Marco's project has more options than mine, and a few of them deserve a place in any Org site.

`:with-broken-links mark` changes how the build treats mistakes. In my guide, a link to a missing page stops the build with "Unable to resolve link". With `mark`, Org keeps going and writes a visible `[BROKEN LINK: ...]` note where the link was. For a garden that's always half planted, that's friendlier: the rest of the site still publishes, and you find the gaps by reading.

`:exclude` keeps files out of the output. Marco's project leaves out files like `aside.org`, `footer.org` and `template.org`. The usual reason for that is `#+INCLUDE:`, which pastes one Org file into another at export time. A shared footer lives in one file, every page includes it, and `:exclude` stops the footer from being published as a page of its own.

`:with-smart-quotes t` turns straight quotes into curly ones in the HTML, and `:html-postamble-format` sets the footer text for each language, so a copyright or license line can sit at the bottom of every page without touching a single Org file.

Diagrams that live in the page

Marco's configuration also loads PlantUML, a tool that draws diagrams from text. With Org Babel, a diagram can sit in the page as a source block:

```
#+BEGIN_SRC plantuml :file img/flow.png
Alice -> Bob: request
Bob --> Alice: response
#+END_SRC
```

When the page is exported, Org runs PlantUML, saves the image and puts it in the HTML. The diagram stays editable as text, right next to the prose that explains it.

It needs two settings: `org-plantuml-jar-path`, pointing to the PlantUML `.jar` file (which needs Java), and PlantUML added to `org-babel-load-languages`. There's a catch: by default, Org asks before running any code block, and answering that on every export gets old fast. Marco's answer is a small function for `org-confirm-babel-evaluate` that skips the question for PlantUML and keeps it for every other language. Something like this:

```
(defun my/confirm-babel (lang _body)
(not (string= lang "plantuml")))
(setq org-confirm-babel-evaluate #'my/confirm-babel)
```

That's the right granularity. A diagram tool is harmless, a shell block isn't, so only the harmless one gets a pass.

Publishing the extras

Two of Marco's components publish files that aren't pages. One copies `robots.txt` to the root of the site. The other copies `publish.el` itself, so the configuration that builds the garden is part of the garden, and his page links to it for anyone who wants to see exactly how the site is made.

Both use `org-publish-attachment`, the same "copy, don't convert" function as the static project in my guide. Then one more project, with only a `:components` list, runs everything with a single name:

A component only needs a folder, an extension and a destination. This is the one for `robots.txt`:

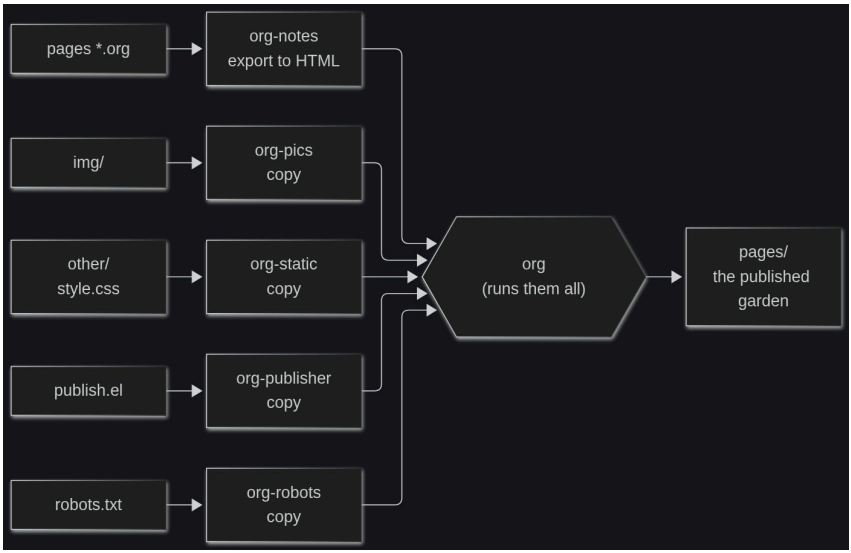
```
("robots"
:base-directory "./"
:base-extension "txt"
:publishing-directory "../pages/"
:publishing-function org-publish-attachment)
```

His `robots.txt` asks AI crawlers, scrapers and site copiers to stay out of the garden.[3]

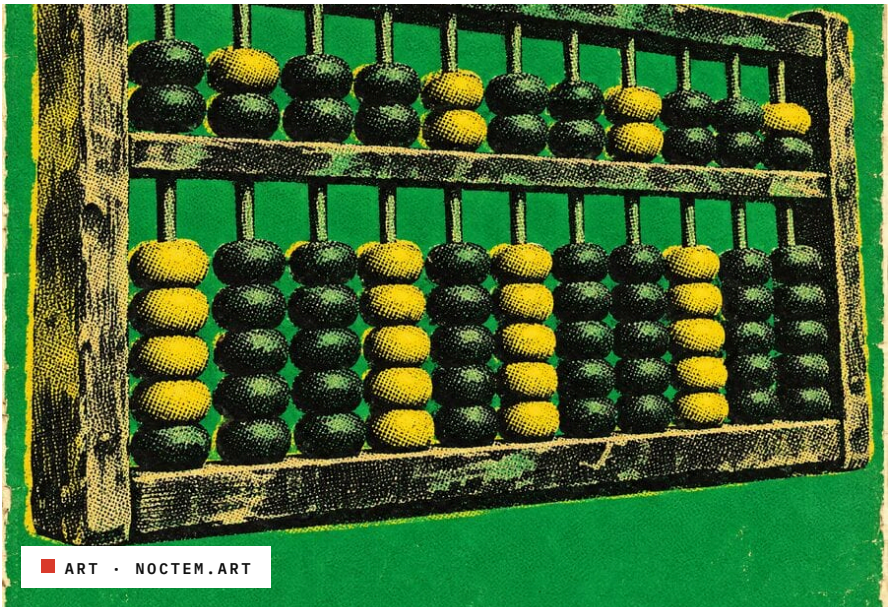
Thank you, Marco

This post exists because Marco took the time to explain his setup in public, configuration and all, under a free license. If you build anything

3 `robots.txt` is a convention from 1994 that became a standard, RFC 9309, in 2022. It's a request, and it only works with crawlers that choose to follow it.



with Org, his Digital Garden page is worth reading in full, and the rest of his site shows what the setup produces. He takes feedback through issues on Codeberg. ■



08 KEEPASSXC · SYNCTHING · PASSWORDS 20 NOVEMBER 2026

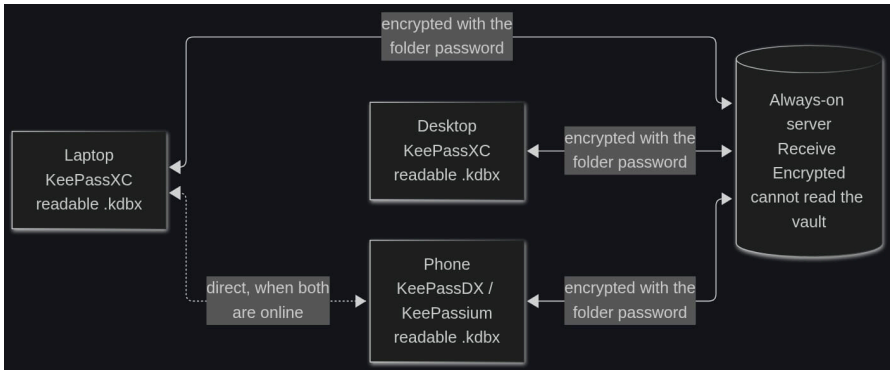
Your passwords on every device, with no cloud

Keep your passwords on every device with KeePassXC and Syncthing, no cloud involved. Add a server that stores the vault without reading it, break the setup on purpose to see how it recovers, add a YubiKey, and prepare for the day everything goes wrong.



Most password managers keep your vault on someone else's server. It's convenient, and it means trusting a company with the most sensitive file you own, and trusting it to stay online. There's another way to have your passwords on every device: keep them in a single encrypted file, and let your devices pass that file to each other directly.

This guide builds exactly that, with KeePassXC on the computer, the usual KeePass apps on the phone, and Syncthing moving the file around. Then it goes further than most guides do. You'll add a server that stores the vault without being able to read it, break the setup on purpose to see how it recovers, add a YubiKey, and write down a plan for the day everything goes wrong.



The pieces

KeePassXC is a free, open source password manager for Linux, Windows and macOS. Everything lives in one file with the `.kdbx` extension, encrypted with your master password. There's no account and no server. Open the file, type the password, and your entries are there.

The `.kdbx` format is shared by a whole family of apps, so the same file opens on the phone too:

- Android: KeePassDX or Keepass2Android
- iPhone and iPad: KeePassium or Strongbox

Syncthing keeps folders in sync between your own devices. Each device has an ID, you tell them about each other once, and from then on they exchange changes directly, encrypted with TLS. There's no account and no cloud in the middle.

Put the two together and you get a vault that exists only on machines you own, kept in sync without anyone else's server.

Step 1: create the database

In KeePassXC, go to Database → New Database. Give it a name, keep the encryption settings at their defaults and pick a strong master password. A passphrase of five or six random words works well, and KeePassXC's password generator can make one for you.

The defaults matter. A new database in KeePassXC uses the KDBX 4 format with Argon2, a key derivation function that makes every password guess slow and memory hungry. You can check it under Database → Database Settings → Security → Encryption Settings, with Advanced

Settings ticked.[1]

Save the file in a folder that will hold the vault and nothing else, for example `~/Vault/passwords.kdbx`. That folder is what Syncthing will share.

Two settings in Tools → Settings → General help a lot here:

- Automatically save after every change: keeps the file on disk always current, so Syncthing always has the latest version to send.
- Automatically reload the database when modified externally: when Syncthing replaces the file with a newer one from another device, KeePassXC reloads it on its own.

Step 2: sync the folder

Install Syncthing on each device:

- Linux: from your distribution's packages, or from syncthing.net.
- Windows and macOS: the downloads at syncthing.net.
- Android: the official app was discontinued at the end of 2024, so you'll need a community app. Syncthing-Fork is the most common one; BasicSync and Syncthing Tray are other options. The scene has been a bit turbulent, so check what's current before you choose.
- iPhone: there's no official app. Möbius Sync is the usual choice, and KeePassium or Strongbox can open the vault from its folder through the Files app.

Then connect them. Syncthing's web interface lives at `http://127.0.0.1:8384` on desktops. On each device, use Add Remote Device and paste the other device's ID (Actions → Show ID shows it as text and as a QR code, which is handy for phones). Then share the `Vault` folder with the other devices.

One more setting on every device that holds a readable copy: open the folder's File Versioning tab and pick Staggered File Versioning. When a newer file arrives from another device, Syncthing moves the old one into a hidden `.stversions` folder instead of throwing it away. You'll need this in the tests below.

At this point you have a working setup. It has one weakness, though: Syncthing only syncs when two devices are online at the same time. Change something on the phone at the bus stop, turn on the laptop at home with the phone in airplane mode, and the laptop has nothing new.

1 The command line is a different story. In the test setup, a database created with `keepassxc-cli db-create` (version 2.7.6) used the older AES-KDF instead of Argon2. If you create yours from the terminal, run `keepassxc-cli db-info` and check the KDF line.

Step 3: a server that holds the vault without reading it

The fix is a device that's always on, so every other device can drop changes there and pick them up later. Any small server or always-on machine works. The interesting part is that this server doesn't need to be trusted.

Syncthing has a feature called untrusted devices. You give the folder a password when you share it with the server, and from then on everything sent to the server is encrypted with that password: the file contents, the file names and the folder structure.[2] The server stores, sends and receives the data like any other device, but it has no way to read it. Your trusted devices encrypt on the way out and decrypt on the way in.

On the server:

1. Add your laptop and your phone as remote devices.
2. When the laptop offers the `Vault` folder, accept it and, in the folder's Advanced tab, set Folder Type to Receive Encrypted.

On the laptop and the phone:

1. Edit the server's device settings and, in the Advanced tab, tick Untrusted. From then on Syncthing refuses to share a folder with that device without a password, so a forgotten setting can't send the vault there in plain text.
2. Edit the `Vault` folder, open the Sharing tab, tick the server and type the encryption password. Use the same password on every trusted device.

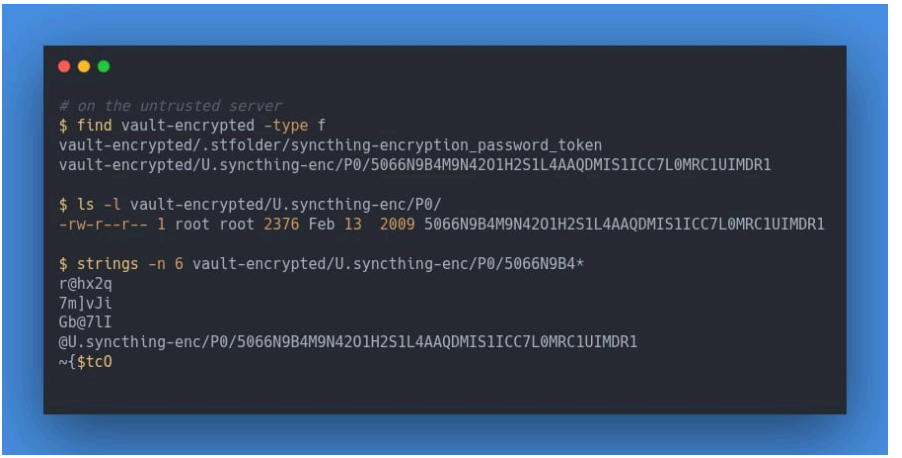
That password goes on your recovery sheet (more on that later), together with the folder ID shown in the folder settings. Syncthing builds the encryption key from both.

Here's what the server ends up holding. The vault is in there, but under a name like `5066N9B4M9N42O1H2S1L4AAQDMIS1ICC7L0MRC1UIMDR1`, its contents are random bytes, and even its date is a placeholder (13 February 2009) that says nothing about when you last changed it:

So the vault is now encrypted twice on that server: once by KeePassXC, with your master password, and once by Syncthing, with the folder password. Someone who breaks into the server gets neither.

And the phone at the bus stop is no longer a problem. In the test setup used for this post, the laptop was turned off, an entry was added on the phone, the phone was turned off too, and only then was the laptop turned

2 Syncthing's documentation still calls untrusted devices a beta feature. The server can't see names or contents, but it can see the folder ID, the folder label and the approximate size of each file.



back on. It got the new entry from the server, without the two devices ever being online at the same time.

Breaking it on purpose

Here are four bad days, run for real on a test setup: three Syncthing nodes playing a laptop, a phone and an untrusted server, with `keepassxc-cli` doing the edits.[3]

TEST 1: TWO DEVICES, TWO DIFFERENT CHANGES

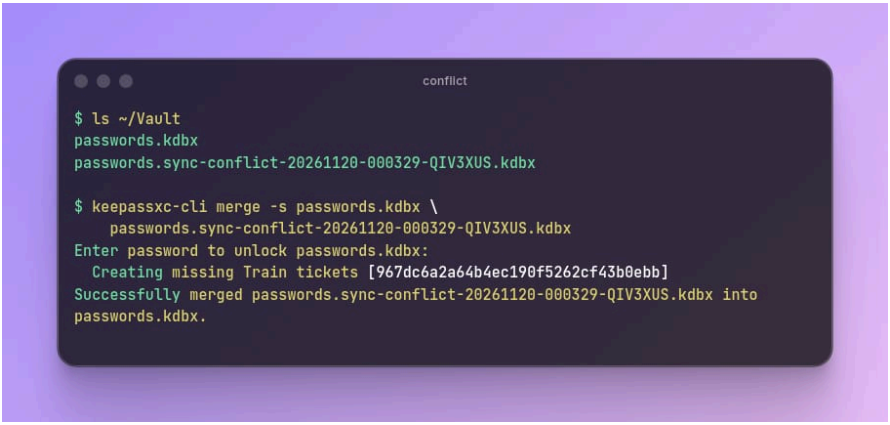
The scenario: the server is down, or you're offline. You add an entry on the laptop, then add a different entry on the phone. Later, everything reconnects.

In the test, the laptop added "Train tickets" and the phone added "Pharmacy", with the server off. When the server came back, both devices ended up with this:

```
passwords.kdbx
passwords.sync-conflict-20261120-000329-QIV3XUS.kdbx
```

Syncthing doesn't know what's inside the file, so it can't combine the two versions. It keeps the newest one as `passwords.kdbx` (the phone's, with "Pharmacy") and saves the other one beside it as a conflict copy (the

3 Syncthing 1.27 and keepassxc-cli 2.7.6, from Ubuntu's packages, all on one Linux machine. The current Syncthing is 2.1, and its documentation describes the same conflict, versioning and encryption behavior used here.



laptop's, with "Train tickets"). Nothing is lost, but the vault is now split across two files.

KeePassXC knows how to put them back together. Open `passwords.kdbx`, go to Database → Merge From Database and pick the conflict copy. KeePassXC matches entries by their internal IDs, adds what's missing and keeps the newest version of anything changed on both sides. From the terminal, it looks like this:

Then delete the conflict copy. Syncthing removes it from the other devices too, and the merged vault goes everywhere.

One warning came out of this test by accident. In an earlier run, the folders had been created by hand in Syncthing's config file, and that left a setting called Max Conflicts at 0. With 0, Syncthing doesn't create conflict copies at all: the newest file silently wins and the other change is gone. In that run, an entry really disappeared. Folders created through the web interface get the default of 10, which is what you want. To check yours, open Actions → Advanced, find the vault folder and look at Max Conflicts.

TEST 2: AN ENTRY DELETED FOR GOOD

The scenario: you delete an entry, then empty the recycle bin, then realize you needed it.

KeePassXC's recycle bin is the first safety net, so check there first. If it's already empty, the deletion has synced to every device, and the vault no longer has the entry anywhere.

Versioning is the next net, with one detail that's easy to miss: Syncthing only keeps versions of changes that arrive from other devices. In the test, "Bank" was deleted on the laptop. The laptop's `.stversions` folder

had nothing, because the change was local there. The phone's had the previous file, because for the phone, the change came from outside:

```
.stversions/passwords~20261120-000415.kdbx
```

So always look for old versions on a device other than the one where the mistake happened.

To bring the entry back, don't replace your current vault with the old file, or you'll lose everything you changed since. Merge the old version into the current one instead, the same way as in test 1. In the test, the merge reported `Creating missing Bank` and kept every newer entry as it was.[4]

TEST 3: EVERY DEVICE IS GONE

The scenario: the laptop dies, the phone is stolen, all on the same week. The only copy left is the encrypted one on the server.

Syncthing can decrypt that copy on any trusted machine with the folder password:

```
syncthing decrypt --password='your folder password' --to=restored vault-
encrypted/
```

In the test, it printed `Plaintext filename is "passwords.kdbx"`, and the restored file opened normally with the master password. With a wrong password, it stopped with an `authentication failed` error.

That's a full recovery from a machine that never could read the vault. It also shows exactly what you need to keep safe: the folder password, the folder ID and, of course, the master password.

TEST 4: THE PHONE IS STOLEN

This one is about what the thief can and can't do.

They have the phone, and on it, a copy of the `.kdbx`. Without your master password (and your YubiKey, once you add one), that copy is useless. The weak point is the phone's screen lock and the app's quick unlock: if the thief can unlock the phone and the app opens with a fingerprint or a PIN, they're in. So set the KeePass app to lock itself quickly, and don't let its quick unlock be weaker than the phone's own lock.

What to do, in order:

1. On the server and your other devices, remove the phone in Syncthing (Edit the device → Remove). It stops receiving new versions of the vault.

4 Staggered versioning keeps one version per 30 seconds for the first hour, one per hour for the first day, one per day for the first month and one per week after that.

2. If there's any chance the master password was seen, change it. Keep in mind that the stolen copy still opens with the old password, so also change the passwords of your most important accounts, starting with email.

Adding a YubiKey

So far, the vault is protected by one secret: the master password. A YubiKey adds a second ingredient that can't be copied from your head or guessed.

KeePassXC uses the YubiKey's challenge-response mode. When it opens the database, it sends a challenge to the key, the key answers with an HMAC-SHA1 of that challenge, and the answer is mixed into the master key together with your password. Without the physical key, the right password still doesn't open the file.[5]

PROGRAM TWO KEYS, NOT ONE

If your only YubiKey breaks, your vault is gone. So program two keys with the same secret, and write the secret down.

First, generate a secret of 20 random bytes and write it on paper:

```
openssl rand -hex 20
```

Then program it into slot 2 of each YubiKey, with touch required:

```
ykman otp chalresp --touch 2 <the-secret-in-hex>
```

Slot 2 is the usual choice because many YubiKeys come with slot 1 already in use. Run the same command with each key plugged in, and keep the spare somewhere other than your keychain. A YubiKey doesn't let anyone read the secret back out, so the paper copy is the only way to make a replacement later.

ADD IT TO THE DATABASE

In KeePassXC, go to Database → Database Settings → Security, choose Add additional protection, then Add Challenge-Response, and pick the key. From then on, unlocking needs the password and the key, and so does saving: KeePassXC asks for a touch every time it writes the file.

5 Strictly speaking, this isn't two-factor authentication, because there's no server checking a second factor. The YubiKey's answer becomes part of the encryption key itself, which is why the file can't be opened without it.

That last part matters in a synced setup. Every device that edits the vault needs a YubiKey there, so think about the phone before you commit:

- iPhone and iPad: KeePassium and Strongbox both support challenge-response. On the iPhone, an NFC YubiKey works inside the app, and the Lightning model works on both iPhone and iPad.
- Android: KeePassDX talks to a YubiKey through a separate driver app from the same developer, and Keepass2Android uses an app called ykDroid. NFC support depends a lot on the phone, so test it before you rely on it.

The command line works too, with `-y 2` (slot 2) on any `keepassxc-cli` command.

Who protects you from what

With every piece in place, here's what happens in each situation, and which layer does the protecting:

- Someone steals your locked laptop: the `.kdbx` encryption. They need the master password and the YubiKey.
- Someone uses your laptop while the vault is unlocked: nothing in this setup. Set KeePassXC to lock after inactivity and when the session is locked (Tools → Settings → Security).
- Someone breaks into the server: Syncthing's encryption, and behind it, the `.kdbx` encryption. They can see roughly how big the file is, and they could delete it, but your devices still have their own copies and versions.
- Someone gets a copy of the `.kdbx` file: the master password, made expensive to guess by Argon2, plus the YubiKey. An offline file can be attacked forever, which is why a long passphrase matters more here than anywhere else.
- Your phone is stolen: the phone's lock and the app's lock, then removing the device from Syncthing.
- Malware on a computer while the vault is open: nothing here can stop it. Keep the system updated and the vault locked when you're not using it.
- You lose the YubiKey: the spare key, or the secret on paper.
- You forget the master password: nobody. There's no reset, no support line, no recovery email. Read the next section.

The day everything disappears

A vault with no cloud and no company behind it has one real risk: you, locked out of it. The answer is boring and works well. Prepare three things.

A recovery sheet. One sheet of paper, kept somewhere safe at home, with:

- the master password;
- the YubiKey secret (the hex string from `openssl rand`);
- the Syncthing folder password and folder ID;
- the address of the server that holds the encrypted copy.

Keeping a password on paper sounds wrong, so compare the risks. For the sheet to hurt you, someone has to get into your home, find it and understand it. Forgetting your master password is far more likely, and it's final. A sealed envelope in a drawer you don't open often is enough.

A cold copy. Once a month, copy `passwords.kdbx` to a USB drive that stays unplugged. It's already encrypted, so the drive needs no extra protection, and it covers the case where a mistake syncs everywhere before you notice.

A yearly drill. Once a year, on a computer that has never had the vault on it, try to get back in using only the sheet and the cold copy: install KeePassXC, open the file from the USB drive with the password and the spare YubiKey, and decrypt the server copy with `syncthing decrypt`. If any step fails, you found the problem on a calm day.

Beyond passwords

The same file can do a few more jobs. Each deserves its own post, so here's just the outline:

- SSH keys: KeePassXC can hold your SSH private keys and add them to the system's SSH agent while the vault is unlocked, then remove them when it locks. The setting is in Tools → Settings → SSH Agent, and from version 2.8 it can generate new keys too.
- TOTP codes: KeePassXC can generate the six-digit codes for two-factor logins (right-click an entry → TOTP → Set up TOTP).[6]
- Scripts: `keepassxc-cli` reads the vault from the terminal. `keepassxc-cli show -s -a Password passwords.kdbx Email` prints one password, `keepassxc-cli clip passwords.kdbx Email` copies it to the clipboard for a few sec-

6 KeePassXC's own documentation warns that keeping TOTP codes in the same database as the passwords removes the benefit of two-factor authentication, because whoever opens the vault gets both. A separate database for TOTP, with its own password, keeps the two factors apart.

onds, and `keepassxc-cli show -t passwords.kdbx Email` prints the current TOTP code.

KeePassXC 2.8 also added a Remote Sync option, which downloads a copy of the vault with a command you configure (`scp`, for example), merges it into the open database and uploads the result. It only runs when you trigger it from the Database menu, so it's a different tool from Syncthing's always-on sync. For a vault that stays current on every device without you thinking about it, Syncthing is the better fit. ■

■ ■ ■

09 TLS · HTTPS · LETS-ENCRYPT 20 NOVEMBER 2026

TLS certificates explained: ACME, Certbot and the alternatives

Free TLS certificates, explained: what a certificate proves, how ACME and its three challenges work, how Certbot fits in, which alternatives fit better (acme.sh, lego, Caddy, Traefik and more) and what shorter certificate lifetimes change.



Every padlock in a browser bar sits on top of a certificate. For years, getting one meant paying a company, filling in forms, emailing files back and forth, and remembering to do it all again a year later.

Today it's free and takes one command, and the reason has a name: ACME.

This post goes through what a certificate actually proves, how ACME works, where Certbot fits, which other tools do the same job, and how to pick one. No previous knowledge needed, only a bit of patience with acronyms.

What a certificate proves

A TLS certificate (people still say "SSL certificate", the old name) is a small signed statement: "this public key belongs to `example.org`". The signature comes from a certificate authority, a CA, that the browser already trusts. When a site opens, the server shows its certificate, the browser checks the signature, and then the two sides set up an encrypted connection using that key.

Three kinds show up in every conversation about this:

- DV, domain validation: the CA checked that the requester controls the domain. That's all.
- OV, organization validation: the CA also checked that a real company is behind it.
- EV, extended validation: a stricter version of OV, with more paperwork.

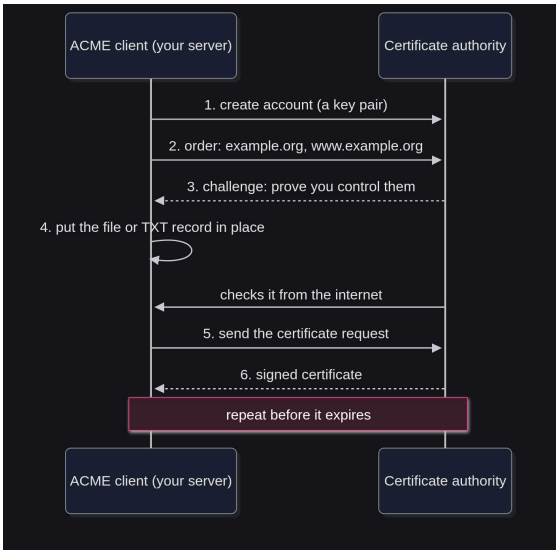
The encryption is exactly the same in all three. A free DV certificate protects the connection as well as an expensive EV one. The only difference is how much the CA checked about the owner, and browsers stopped showing EV's special green bar back in 2019, so visitors can't tell them apart at a glance. For almost every website, DV is the right choice.

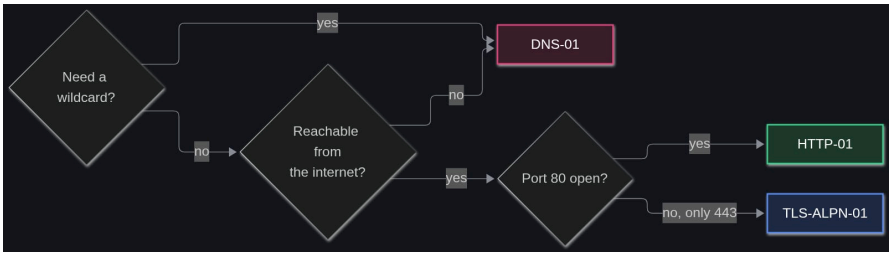
ACME in plain words

ACME stands for Automatic Certificate Management Environment. It's the protocol a program on the server uses to talk to a CA and get a certificate with no human in the loop. Let's Encrypt created it and started issuing free certificates with it at the end of 2015, and it became an internet standard, RFC 8555, in 2019.

A conversation between an ACME client and a CA goes like this:

1. The client creates an account with the CA, which is just a key pair. No username or password.
2. It asks for a certificate for one or more names, like `example.org` and `www.example.org`.
3. The CA answers with a challenge for each name: "prove you control this".





4. The client solves the challenge, and the CA checks it from the outside.
5. The client sends a certificate request with the server's public key, and the CA signs it.
6. The client downloads the certificate, installs it, and comes back to repeat everything before it expires.

Step 6 is the whole point. ACME certificates are short-lived on purpose, and the program renews them on its own. Nobody has to remember anything.

The three challenges

The challenge is how a server proves it controls a name. There are three, and picking the right one avoids most of the headaches before they start.

HTTP-01

The CA hands the client a token, the client puts it in a file at `http://example.org/.well-known/acme-challenge/<token>`, and the CA downloads it.

- Good for: an ordinary website on a server that's reachable from the internet.
- Needs: port 80 open. It has to be port 80, no exceptions.
- Can't do: wildcard certificates (`*.example.org`).

DNS-01

The client publishes a TXT record at `_acme-challenge.example.org`, and the CA looks it up in public DNS.

- Good for: wildcards, servers that aren't reachable from the internet, and setups with many servers behind one name.
- Needs: a way to change DNS records automatically, usually the DNS provider's API.
- Watch out: an API key that can edit the whole zone is a powerful secret. acme-dns limits the damage, and it's what I use at home for services that only live on my network. The full setup is in Valid HTTPS for LAN-only services with acme-dns and pfSense.

TLS-ALPN-01

The CA connects to port 443 and the server answers with a special temporary certificate.

- Good for: servers that only have 443 open, or software that handles TLS itself (Caddy and Traefik can use it).
- Can't do: wildcards. Certbot doesn't support it, so it mostly shows up inside web servers and in clients like lego and acme.sh.

Certbot

Certbot is the ACME client made by the EFF, and the one Let's Encrypt points most people to. It's written in Python, it can get a certificate and also edit the web server's config to use it, and its documentation is friendly.

The EFF recommends installing it from snap, which keeps it current on any Linux distribution:

```
sudo snap install --classic certbot
sudo ln -s /snap/bin/certbot /usr/local/bin/certbot
```

Then, for a site already running on nginx:

```
sudo certbot --nginx
```

Certbot asks which names to include, solves the HTTP-01 challenge through nginx, edits the config to turn HTTPS on and installs a timer that renews everything automatically. To keep the config hand-written and only get the certificate:

```
sudo certbot certonly --nginx -d example.org -d www.example.org
```

The files land in `/etc/letsencrypt/live/example.org/`. The two that the server needs are `fullchain.pem`, the certificate, and `privkey.pem`, the private key.

Certbot's other ways of solving the challenge, called plugins:

- `--nginx` and `--apache`: work through the web server and edit its config.
- `--webroot -w /var/www/html`: drop the challenge file in the site's folder, without touching the config.
- `--standalone`: start a tiny web server of its own on port 80, for machines with no web server running.
- DNS plugins, like `certbot-dns-cloudflare`: solve DNS-01 through the provider's API, for wildcards.

Two commands worth knowing by heart:

```
sudo certbot renew --dry-run
sudo certbot certificates
```

The first runs a fake renewal against the test server, so problems show up today instead of on the day a certificate expires. The second lists every certificate and when it expires.

With `--nginx` or `--apache`, Certbot reloads the server after each renewal by itself. With `certonly`, the server keeps using the old certificate until something tells it to reload. The simplest fix is a small script in Certbot's hooks folder, which runs after every successful renewal:

```
sudo tee /etc/letsencrypt/renewal-hooks/deploy/reload-nginx.sh <<'EOF'
#!/bin/sh
systemctl reload nginx
EOF
sudo chmod +x /etc/letsencrypt/renewal-hooks/deploy/reload-nginx.sh
```

The alternatives

Certbot is a good default, but other tools fit some setups better. They all speak the same ACME, so the certificate at the end is the same.

ACME.SH

A single shell script with no dependencies. It knows the APIs of a long list of DNS providers, which makes it a favorite for DNS-01, and it runs on routers, NAS boxes and other places where installing Python is a pain. It's also the client in my acme-dns setup. One detail: its default CA is ZeroSSL, so getting Let's Encrypt certificates takes one extra command, run once: `acme.sh --set-default-ca --server letsencrypt`.

LEGO

A single binary written in Go, also with a long list of DNS providers. Easy to drop into a container or a CI pipeline, since there's nothing to install besides the file itself.

CADDY

A web server with ACME built in. The domain name goes in its config, and Caddy gets the certificate, renews it and redirects HTTP to HTTPS by itself, using Let's Encrypt or ZeroSSL. For local names like `localhost`, it creates its own local CA. For a new site with no attachment to nginx or Apache, it's the least work of all.

TRAEFIK

A reverse proxy popular with Docker. It reads labels on the containers, works out which domains to serve and requests their certificates on its own. When the services already live in containers, it saves a lot of config.

NGINX AND APACHE, NATIVELY

Apache has had `mod_md` for years, which lets it manage certificates on its own. nginx announced its own module, `ngx_http_acme_module`, in August 2025, as a preview that started with HTTP-01 only. Both remove the need for a separate client, at the cost of being newer and less documented than Certbot.

WINDOWS AND KUBERNETES

- win-acme: a friendly command-line client for Windows and IIS.
- Posh-ACME: the same idea, as a PowerShell module.
- cert-manager: the usual way to get certificates inside Kubernetes, where certificates become resources like any other.

Other CAs that speak ACME

Let's Encrypt is the best known, but not the only one:

- ZeroSSL: free certificates through ACME, the default in acme.sh and one of Caddy's two defaults.

- Google Trust Services: free through ACME, with an extra key from a Google Cloud account to sign up.
- Paid CAs like DigiCert and Sectigo: many accept ACME too, so OV and EV certificates can be automated with the same clients.

Buypass, a Norwegian CA that used to give away free ACME certificates, stopped issuing them in October 2025. CAs come and go, which is one more reason to use a client that can switch servers with one line.

Outside ACME

Sometimes a public certificate isn't the right tool at all:

- mkcert: creates a local CA on the machine and certificates trusted by its own browsers. Made for development, never for production.
- step-ca: a private CA that also speaks ACME, so the same clients work inside a company network. Its root certificate has to be installed on every device that should trust it.
- Self-signed certificates: encrypted, but every browser complains about them. Fine for a quick test, tiring for anything used daily.
- The host or the CDN: many hosting providers and CDNs issue and renew certificates on their own. When that's the case, none of this is needed.

So which one?

- A website on a server with nginx or Apache: Certbot with `--nginx` or `--apache`.
- A brand new site, with the server still open to choice: Caddy.
- Services in Docker behind a reverse proxy: Traefik, or Caddy as the proxy.
- A wildcard, or a server that isn't reachable from the internet: DNS-01, with acme.sh, lego or Certbot plus a DNS plugin.
- A router, a NAS or a box with no Python: acme.sh.
- Windows and IIS: win-acme.
- Kubernetes: cert-manager.
- Local development: mkcert.
- A private network with many devices under the same owner: step-ca.

How well does it work, and what's changing

Automated certificates are exactly as strong as any other. What ACME changes is the risk of forgetting: a certificate that renews itself doesn't expire on a Sunday night because someone was on vacation.

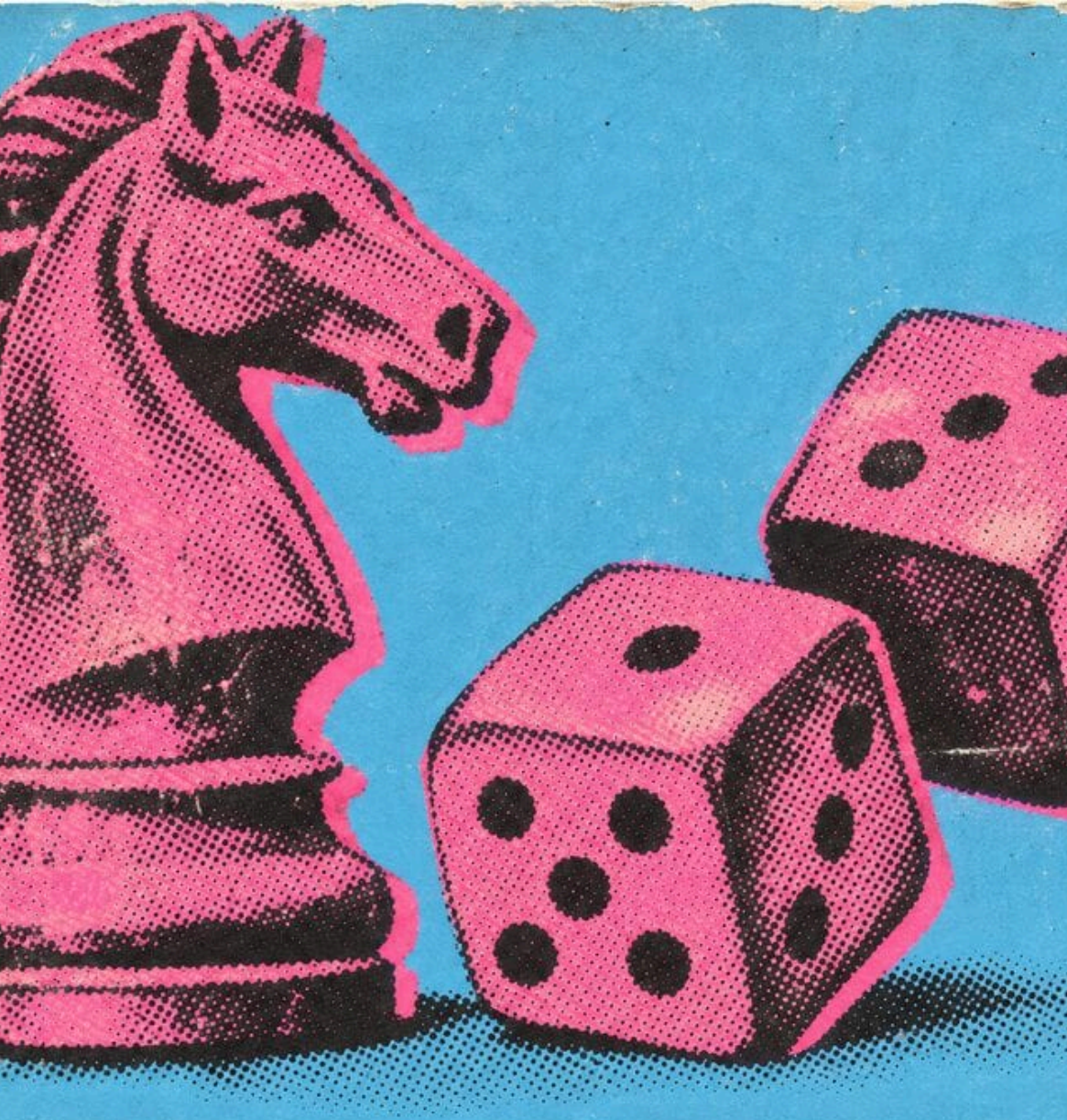
That matters more every year, because certificates are getting shorter. The industry rules set by the CA/Browser Forum cut the maximum lifetime of public certificates to 200 days in March 2026. It drops to 100 days in March 2027 and to 47 days in March 2029. Let's Encrypt has always issued 90-day certificates, and it plans to reach 45 days by February 2028. Since January 2026 it also offers optional certificates that last about six days, and certificates for IP addresses instead of names. At these lengths manual renewal stops being practical, and automation is the only way left.

A few habits keep it boring, in the good sense:

- Test against the staging server first. Let's Encrypt allows 5 certificates for the same set of names per week and 50 per registered domain, and a script that fails in a loop burns through that fast. Certbot has `--dry-run`, acme.sh has `--staging`.
- Check expiry dates independently. Let's Encrypt stopped sending expiration emails in June 2025, so a broken renewal can go unnoticed. A monitoring service, or a weekly look at `certbot certificates`, catches it in time.
- Reload the server after renewal. A new certificate on disk does nothing until the server reads it again.
- A wildcard covers one level only. `*.example.org` covers `www.example.org` but not `example.org` itself, so the order needs both.
- A CAA record on the domain lists which CAs may issue for it. If there is one, the CA in use has to be on it, or issuance fails with an error that's easy to misread.

Whatever the tool, the protocol underneath is the same, so what's learned with one carries over to the others. ■





INTERMISSION · ART COURTESY OF NOCTEM.ART


$ cat ~pablo/02

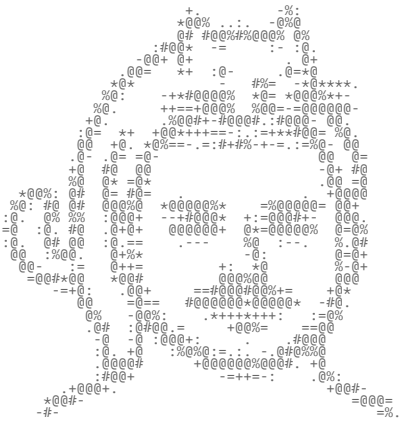


INTERMISSION · ART COURTESY OF NOCTEM.ART

~pablo

A zine made of what I wrote on pablomurad.com.

02



COLOPHON

ISSUE	Nº 02, November 2026
PUBLISHED	20 November 2026, on pablomurad.com
TYPESET	20 November 2026, A5
ESSAYS	9, each one also on the website
PLATES	Pictures between the essays are courtesy of noctem.art.
AT HOME	Print pablo-02-booklet.pdf double-sided on A4, flip on the short edge, fold the stack in half and staple the spine.

pablomurad.com/zine/02

~PABLO

Nº 02

Essays from pablomurad.com, set on paper.





