Caddy Course

Flavio Copes

Updated August 3, 2026

Contents

Learn to serve websites and proxy applications with Caddy, automatic HTTPS, safe routing, health checks, logs, metrics, and recovery practices.

What you'll learn: Configure, secure, and operate a Caddy server that serves files, proxies applications, manages HTTPS, reloads safely, and recovers predictably.

Caddy foundations

Install Caddy, run disposable servers, write a Caddyfile, and validate the effective configuration.

Install and inspect Caddy

Install an official Caddy build, record its version and modules, and understand why the exact binary matters.

Caddy is a web server written in Go. It ships as a single self-contained binary with no dependencies, and it manages HTTPS certificates for you out of the box.

Before any of that matters, you need the right binary on your machine. Caddy is modular: features like DNS providers are compiled into the binary itself, so two Caddy installs with the same version number can behave differently. Knowing exactly what you installed saves you hours of confusion later.

On macOS, install it with Homebrew:

```
brew install caddy
```

On Debian or Ubuntu, add the official repository and install the package:

```
sudo apt install -y debian-keyring debian-archive-keyring apt-transport-https curl
curl -1sLf 'https://dl.cloudsmith.io/public/caddy/stable/gpg.key' | sudo gpg --dearmor -o /us
curl -1sLf 'https://dl.cloudsmith.io/public/caddy/stable/debian.deb.txt' | sudo tee /etc/apt/
sudo apt update
sudo apt install caddy
```

The package does more than copy a binary. It creates a dedicated `caddy` system user, installs a systemd service, and drops a default configuration at `/etc/caddy/Caddyfile`. We'll use all of that later in the course.

Now inspect what you got:

```
caddy version
caddy build-info
caddy list-modules --packages
```

`caddy version` prints something like `v2.8.4`. `caddy build-info` shows the Go version and every dependency compiled into the binary. `caddy list-modules --packages` lists the modules this build contains, with non-standard ones flagged at the end of the output.

Write down the version and any extra modules. When something behaves oddly in six months, you'll want evidence tied to one exact build, not a guess about what was probably installed.

One warning before we move on. Only install Caddy from the official repositories or the official downloads page. A web server binary sees every request and holds your TLS private keys, so a binary copied from a random location can do anything with them. If you need extra modules, build your own binary with `xcaddy` instead of trusting someone else's.

Run disposable servers

Use Caddy's command shortcuts to serve one response and one directory before creating persistent configuration.

You don't need a configuration file to try Caddy. The CLI includes shortcuts that start a working

server from a single command. They're perfect for quick experiments, for checking whether a port works, or for sharing a folder on your machine for five minutes.

`caddy respond` starts a server that returns a fixed response:

```
caddy respond --listen 127.0.0.1:8080 --body 'hello from Caddy'
```

In another terminal, prove it works:

```
curl http://127.0.0.1:8080/
```

You get `hello from Caddy` back. That one line proved the binary runs, the port is free, and HTTP flows end to end. When you're debugging a networking problem, a server this dumb is exactly what you want, because nothing else can go wrong.

Stop it with Ctrl-C, then try the second shortcut. `caddy file-server` serves a directory:

```
mkdir -p public && echo '<h1>Hi</h1>' > public/index.html
caddy file-server --listen 127.0.0.1:8080 --root ./public --browse
```

Request `http://127.0.0.1:8080/` and you get your HTML back. The `--browse` flag adds a directory listing for folders that have no index file.

Notice that both commands listen on `127.0.0.1`. That keeps the server private to your machine. Bind to `:8080` instead and every device on your network can reach it — and a directory can contain `.env` files, source code, or anything else you forgot was in there. My advice is to start on loopback and widen access deliberately, never by accident.

A failure you'll hit sooner or later: the port is taken. Caddy exits immediately with `bind: address already in use`. Find the culprit with `lsof -i :8080` on macOS or `ss -lntp` on Linux, then stop it or pick another port.

Write your first Caddyfile

Create a site block, understand its address, and run the configuration in the foreground.

Shortcuts are fine for experiments. For anything you want to keep, you write a **Caddyfile**, a small text file that describes your sites.

The Caddyfile exists because Caddy's native configuration format is JSON, and nobody wants to write that by hand. The Caddyfile is a human-friendly layer that Caddy converts to JSON for you.

Create a file named `Caddyfile` in an empty directory:

```
:8080 {
  respond "hello from a Caddyfile"
}
```

The first line is the **site address**. It tells Caddy what to listen for. `:8080` means: any hostname, port 8080, plain HTTP. Everything inside the braces defines how that site responds.

Run it in the foreground:

```
caddy run --config Caddyfile
```

Then test it from another terminal:

```
curl http://127.0.0.1:8080/
```

While you're learning, always run Caddy in the foreground with `caddy run`. You see startup errors

and log lines the moment they happen, and Ctrl-C stops the process cleanly instead of leaving a mystery server running in the background.

Why did `:8080` give us plain HTTP? Because Caddy has no name to issue a certificate for. Change the site address to a hostname like `blog.example.com` and Caddy switches to automatic HTTPS for that site. That behavior gets its own module later in this course.

Now the mistake everyone makes on day one: you edit the Caddyfile and nothing changes. A running Caddy does not watch the file. You have to stop and start it, or better, tell the running server to pick up the change with `caddy reload`. We'll build a safe reload habit in the operations module — for now, remember that saving the file does nothing by itself.

Format, adapt, and validate

Format a Caddyfile, inspect the native JSON it produces, and run the stronger validation check before loading it.

Before you trust a Caddyfile, Caddy gives you three commands to check it. Each one goes deeper than the last.

First, formatting. `caddy fmt` rewrites your Caddyfile with consistent indentation:

```
caddy fmt --diff Caddyfile
```

The `--diff` flag shows what would change without touching the file. Add `--overwrite` when you want to apply it. Formatting sounds cosmetic, but it keeps diffs in version control readable, and Caddy logs a warning at startup when the file isn't formatted.

Second, adaptation. This converts the Caddyfile to Caddy's native JSON and shows you the result:

```
caddy adapt --config Caddyfile --pretty
```

Read that JSON once, even if it looks verbose. You'll see the routes Caddy actually built, in the order it will try them, plus configuration that was implicit in the Caddyfile. When routing behaves strangely, this output is where you find out why.

Third, validation:

```
caddy validate --config Caddyfile
```

`caddy validate` goes further than adaptation. It loads the configuration and provisions every module, as if the server were starting for real — it just doesn't open any sockets. A Caddyfile can adapt fine and still fail validation, for example when a `tls` directive references a certificate file that doesn't exist on disk.

On success it prints that the configuration is valid. Make the full sequence a habit: format, adapt, validate, and only then load the configuration on a real server. It catches typos on your laptop instead of in production.

One thing validation cannot catch: runtime problems. It won't tell you that port 443 is already taken or that DNS doesn't point at your machine. It checks the configuration, not the environment. Keep that boundary in mind when a valid config still fails to start.

Check your understanding: caddy foundations

Check the commands, configuration choices, trust boundaries, and failure cases from caddy foundations.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Serve and route requests

Serve static files, match requests, control path handling, and add deliberate response behavior.

Serve a static site

Set a document root, serve files, and verify that filesystem ownership matches the Caddy process.

Serving files is the oldest job a web server has, and in Caddy it takes two directives.

`root` sets the directory that request paths are resolved against. **`file_server`** does the actual work: it takes the request path, finds the matching file under the root, and sends it back.

```
:8080 {
  root * ./public
  file_server
}
```

The `*` after `root` is a matcher meaning every request. Create some content and start the server:

```
mkdir -p public
echo '<h1>My site</h1>' > public/index.html
caddy run --config Caddyfile
```

Verify both the happy path and a miss:

```
curl http://127.0.0.1:8080/
curl -i http://127.0.0.1:8080/missing.html
```

The first request returns your HTML — `file_server` serves `index.html` for directory requests automatically. The second returns `404 Not Found`, which is what you want: no hints about what else exists on disk.

Why two separate directives? Because other handlers use the root too. `try_files`, `php_fastcgi`, and rewrites all resolve paths against it. Setting it once keeps every handler pointed at the same tree.

Caddy protects you from path traversal, so a request for `/../../etc/passwd` cannot escape the root. But it follows symbolic links. A symlink inside `./public` that points at `/etc` will happily be served. Audit links and permissions before you serve a real directory.

The classic production failure is permissions. The packaged service runs as the `caddy` user, and home directories are often mode `700`. Your files exist, your config is valid, and every request still fails. Check the runtime log with `journalctl -u caddy`, then give the service user read access to the site tree. Don't make the tree world-writable out of frustration — grant the smallest access that fixes the error.

Match and handle requests

Use named matchers and mutually exclusive handle blocks to make routing decisions visible.

Real sites don't give the same answer to every request. You need routing, and in the Caddyfile routing is built from **matchers** and `handle` blocks.

A matcher selects requests by path, method, host, header, query, and other facts about the request. A **named matcher** starts with `@`: you define it once, then reference it from directives.

```
:8080 {
  @health path /health

  handle @health {
    respond "ok" 200
  }

  handle {
    respond "application"
  }
}
```

`@health` matches requests whose path is exactly `/health`. The first `handle` block runs only for those. The second `handle` has no matcher, so it acts as the fallback for everything else.

Test both routes:

```
curl http://127.0.0.1:8080/health
curl http://127.0.0.1:8080/anything-else
```

The key property: sibling `handle` blocks are **mutually exclusive**. Exactly one of them runs per request, like the branches of a switch statement. Caddy orders them so blocks with more specific path matchers are tried first, and a matcher-less `handle` catches whatever remains. That makes routing decisions something you can read straight off the file.

To see what Caddy built from this, adapt the configuration:

```
caddy adapt --config Caddyfile --pretty
```

Find the two routes in the JSON. Each `handle` became a subroute, and the health route carries the path matcher you wrote. When a request takes a branch you didn't expect, this output settles the argument.

Always keep a final fallback `handle` when every request needs an answer. Without it, an unmatched request falls through to whatever comes after your routing — often an empty 200 response that looks like success in your monitoring while users see a blank page. That bug is far easier to prevent than to notice.

Strip a path prefix

Choose handlepath when an upstream or file tree should receive a path without its matched prefix.

Sometimes you mount something under a path prefix — docs under `/docs/`, an API under `/api/` — but the thing being served doesn't know about that prefix. A directory of HTML files has no `docs/` folder inside it. An upstream app expects `/users`, not `/api/users`.

`handle_path` solves exactly this. It works like `handle`, with one addition: it strips the matched prefix from the path before running its handlers.

```
:8080 {
```

```
  handle_path /docs/* {
    root * ./manual
    file_server
  }
}
```

Create a file and test it:

```
mkdir -p manual
echo '<h1>Getting started</h1>' > manual/start.html
curl http://127.0.0.1:8080/docs/start.html
```

The request path is `/docs/start.html`. Inside the block, the path becomes `/start.html`, so `file_server` looks for `manual/start.html`. Without the stripping it would look for `manual/docs/start.html` and return a 404 even though your file is right there.

The same logic applies to proxying:

```
:8080 {
  handle_path /api/* {
    reverse_proxy 127.0.0.1:3000
  }
}
```

The upstream receives `/users`, not `/api/users`. The prefix stays your routing concern; the app never learns it exists.

Choose deliberately between the two directives. Use `handle` when the downstream expects the full original path. Use `handle_path` when the prefix is purely a mounting point. Getting this wrong produces 404s that are confusing to debug, because the path Caddy looks up is not the path in your browser. When in doubt, check what actually arrived: for a file server read the runtime log, for a proxy log the request path inside your application.

And resist the temptation to pile up `rewrite` rules until a request happens to work. `handle_path` states your intent in one word. A stack of rewrites hides it.

Shape the response

Add compression, security headers, redirects, and rewrites while keeping internal and client-visible changes distinct.

A response is more than a body. Headers, compression, and redirects all shape what the client receives, and Caddy gives each one a small directive.

Get the vocabulary right first, because two similar-sounding things behave very differently. A **redirect** is visible to the client: Caddy answers with a 3xx status and a `Location` header, and the client makes a second request. A **rewrite** is internal: Caddy changes the URI it processes, and the client never finds out.

```
:8080 {
  encode zstd gzip
  header X-Content-Type-Options nosniff
  redir /old /new 308
  respond /new "new location"
}
```

`encode` compresses responses when the client supports it. `header` sets a response header on every response. `redir` sends the client from `/old` to `/new` with a permanent 308 — pick 308 over 301 when the request method must survive the redirect, because clients may turn a 301 POST into a GET.

Watch the redirect happen:

```
curl -I http://127.0.0.1:8080/old
curl -L http://127.0.0.1:8080/old
```

The first command shows the raw `308 Permanent Redirect` with `Location: /new`. The second follows it and prints the final body. Two requests happened — you can see both in the log.

Now inspect the headers on the destination:

```
curl -s -D - -H "Accept-Encoding: gzip" http://127.0.0.1:8080/new -o /dev/null
```

You'll see `X-Content-Type-Options: nosniff` on the response. Don't worry if `Content-Encoding` is missing here: `encode` skips tiny bodies, where compression costs more than it saves. Serve a real HTML file and it kicks in.

A word of caution on security headers. `nosniff` is safe nearly everywhere. But don't paste a full Content-Security-Policy from a blog post. CSP describes what your site actually loads, so a copied one either breaks the site or allows so much that it protects nothing. Derive it from your application's real behavior, one directive at a time, and test after each addition.

Check your understanding: serve and route requests

Check the commands, configuration choices, trust boundaries, and failure cases from serve and route requests.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Automatic HTTPS

Use local and public HTTPS, understand certificate requirements, and protect advanced certificate automation.

Activate automatic HTTPS

Understand how a hostname activates certificate management and HTTP-to-HTTPS redirects.

Here's the feature Caddy is famous for. Give it a domain name and it obtains a TLS certificate, renews it forever, and redirects HTTP to HTTPS. That whole feature set is called **automatic HTTPS**, and you enable it by writing a hostname.

```
blog.example.com {
  respond "hello over HTTPS"
}
```

That's the entire configuration. When Caddy sees a qualifying hostname as the site address, it asks an ACME certificate authority — Let's Encrypt or ZeroSSL — for a certificate, answers the

validation challenge, installs the certificate, and schedules renewal. It also serves a redirect from port 80 to port 443 for that name.

The trigger is the site address, nothing else. Compare these two sites:

```
http://app.example.test {
  respond "HTTP only"
}

blog.example.com {
  respond "automatic HTTPS"
}
```

The explicit `http://` scheme opts the first site out. The bare hostname opts the second one in. Addresses like `:8080` don't trigger public issuance either, and `localhost` gets a locally-trusted certificate instead — that's the next lesson.

You can study the effect without owning a domain. Adapt the configuration and read the JSON:

```
caddy adapt --config Caddyfile --pretty
```

For issuance to work in production, two things must be true: DNS for the name points at your server, and ports 80 and 443 reach Caddy. That's it. No renewal cron job, no certbot, nothing to remember in three months.

Now the mistake that bites beginners: testing against the real Let's Encrypt endpoint over and over. Production CAs enforce rate limits, and if you restart a broken setup twenty times you can get locked out of issuance for days. While you're testing, point Caddy at the staging CA:

```
blog.example.com {
  tls {
    ca https://acme-staging-v02.api.letsencrypt.org/directory
  }
  respond "testing issuance"
}
```

Staging certificates aren't trusted by browsers, but issuance succeeds or fails for the same reasons as production. Once the staging flow works, remove the `ca` line and let Caddy get the real certificate once.

Use local HTTPS

Issue a local certificate, trust Caddy's local root narrowly, and test clients that do and do not trust it.

You should develop against HTTPS, not HTTP. Secure cookies, service workers, and mixed-content rules all behave differently over TLS, and you want to discover that on your laptop, not in production.

Caddy makes this painless. For `localhost` and other internal names it doesn't contact a public CA. It creates its own **local certificate authority** and issues certificates from it.

```
localhost {
  respond "local HTTPS works"
}
```

Run it:

```
caddy run --config Caddyfile
```

On first run, Caddy generates the local CA and issues a certificate for `localhost`. It also tries to install the CA root into your system trust store, which may prompt for your password.

Test it:

```
curl https://localhost/
```

If you get the response body, the trust installation worked. If curl complains about an untrusted certificate, install the root explicitly:

```
caddy trust
```

`caddy trust` asks the running Caddy instance for its root certificate and installs it into your local trust stores. Retry the curl, then look at who signed the certificate you're trusting:

```
curl -v https://localhost/ 2>&1 | grep issuer
```

The issuer names the Caddy Local Authority. You're not talking to Let's Encrypt here — you're trusting a CA that lives in Caddy's data directory on this machine.

Now the part that trips people up. Trust lives per client, per machine. Your browser works, but a Docker container hitting your host fails, your phone on the same Wi-Fi fails, and a colleague's laptop fails. Each has its own trust store that has never heard of your Caddy root. The server cannot push trust to clients — that's the entire point of a trust store. Export the root from Caddy's data directory, under `pki/authorities/local`, and install it on each device you administer. Only on devices you administer.

Keep the local CA local. Never use it for a public site, and never install its root on machines you don't control.

Prepare public certificate issuance

Verify DNS, ports, permissions, and persistent storage before asking Caddy to obtain a public certificate.

Automatic HTTPS is one line of configuration and four external dependencies. When issuance fails, the Caddyfile is almost never the problem — the environment is. This lesson is the preflight checklist I run before pointing a real domain at a server.

First, DNS. The name must resolve to this machine, in both address families:

```
dig +short A app.example.com
dig +short AAAA app.example.com
```

Compare both answers against the server's actual addresses. The sneaky one is a stale AAAA record: IPv4 points at the new server while IPv6 still points at the old one. The CA may validate over IPv6, fail, and leave you staring at a config that looks perfect.

Second, ports. ACME challenges arrive on port 80 for the HTTP challenge and port 443 for the TLS-ALPN challenge. Check that nothing else holds them:

```
sudo ss -lntp | grep -E ":80 |:443 "
```

If nginx or Apache is squatting on those ports, Caddy either fails to bind or never sees the challenge traffic. Your firewall and any cloud security group must allow both ports too.

Third, storage. Caddy keeps certificates, private keys, and its ACME account in a data directory. For the packaged Linux service that resolves to `/var/lib/caddy/.local/share/caddy`; you can confirm the environment Caddy sees with:

```
caddy environ
```

Two requirements: the `caddy` user can write there, and the directory survives deployments. In containers, that means a volume.

Why persistence matters: wipe the data directory on every deploy and Caddy requests fresh certificates on every deploy. That works until you hit the CA's duplicate-certificate rate limit, and then issuance stops for days. The data directory is state, not cache. Never delete it casually.

Once all three checks pass, load the configuration and watch it happen:

```
journalctl -u caddy -f
```

You'll see Caddy obtain the certificate, with log lines naming the challenge and the issuer. When something fails, the error names the failing step — DNS lookup, connection, or authorization — which tells you exactly which check to redo.

Choose advanced certificate automation

Understand when DNS challenges, wildcard certificates, and on-demand TLS are justified and what new risks they add.

The default challenges cover most sites. Before reaching for the advanced options, make sure you actually need them, because each one adds authority that can be abused.

The **DNS challenge** proves domain control by creating a DNS TXT record instead of answering a request on your server. Caddy needs a DNS provider module compiled in — this is the main reason custom `xcaddy` builds exist — plus an API credential for your DNS host:

```
*.example.com {
  tls {
    dns cloudflare {env.CLOUDFLARE_API_TOKEN}
  }
  respond "wildcard site"
}
```

Two situations justify it. **Wildcard certificates**, which ACME only issues through the DNS challenge. And origins that cannot accept traffic on ports 80 and 443, like a server behind a strict firewall.

The cost is that API token. Anyone holding it can change your DNS, which is close to owning your domain. Scope the token to the one zone it needs, keep it out of the Caddyfile itself — the `{env.…}` placeholder reads it from the environment — and rotate it if you ever suspect a leak.

On-demand TLS goes further: Caddy obtains a certificate during the TLS handshake, the first time it ever sees a hostname. It exists for SaaS platforms serving customer domains that are unknown at deploy time.

```
{
  on_demand_tls {
    ask http://127.0.0.1:5555/check
  }
```

```
}

https:// {
  tls {
    on_demand
  }
  respond "customer site"
}
```

The `ask` endpoint is the safety control. Before issuing, Caddy calls it with the hostname and only proceeds on a 200 response — your app answers the question "is this a real customer?". Without that permission check, anyone who points a DNS name at your server can make you request certificates, burning CA rate limits and disk on garbage names. Never enable on-demand issuance without it.

My advice: write the decision down before touching configuration. Something like this:

```
need wildcard: no
ports 80/443 reachable: yes
hostnames known at deploy time: yes
choice: default automatic HTTPS
```

If your answers match those, the default setup is right and you're done. If not, add the smallest advanced feature that changes the blocking answer, and test it against the staging CA first, exactly like the previous lesson.

Check your understanding: automatic https

Check the commands, configuration choices, trust boundaries, and failure cases from automatic https.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Proxy applications

Proxy applications, preserve trustworthy client identity, secure upstream TLS, and handle backend failure.

Proxy a local application

Place Caddy in front of one loopback application and verify the client and upstream as separate connections.

Most Caddy deployments do one thing: sit in front of an application. Your Node or Go or Python app listens on a local port, Caddy owns ports 80 and 443, and the piece connecting them is a **reverse proxy**.

Why not expose the app directly? Because Caddy brings HTTPS, compression, access logs, and graceful configuration reloads — things your framework either lacks or does worse. The app gets to focus on being an app.

Start a stand-in application on loopback:

```
python3 -m http.server 3000 --bind 127.0.0.1
```

Then write the Caddyfile:

```
:8080 {
  reverse_proxy 127.0.0.1:3000
}
```

Understand what happens per request: the client connects to Caddy, then Caddy opens a second connection to `127.0.0.1:3000`, forwards the request, and relays the response back. Two connections, two log entries, one request.

Compare direct access against the proxied path:

```
curl -i http://127.0.0.1:3000/
curl -i http://127.0.0.1:8080/
```

Same body both times. Watch the Python terminal: it logged both requests, one straight from curl and one from Caddy. From the app's point of view, Caddy is just another client.

The most important detail in this setup is the app's bind address. It listens on `127.0.0.1`, not on all interfaces. If the app also listens publicly, anyone can skip Caddy — and with it your TLS, your logs, and any auth you add later. Verify it:

```
ss -lnt | grep 3000
```

You want `127.0.0.1:3000` in that output, not `*:3000` or `0.0.0.0:3000`.

Now break it on purpose. Stop the Python process and curl through Caddy again: you get `502 Bad Gateway`. The runtime log — `journalctl -u caddy`, or your foreground terminal — shows the dial error with the upstream address. Remember this signature: a 502 from Caddy almost always means the upstream didn't answer. Check the app first, not the proxy.

Preserve client identity safely

Use Caddy's forwarded headers and trusted proxy settings without accepting forged client addresses.

Put a proxy in front of an app and the app stops seeing real client addresses. Every connection now comes from Caddy, so the app logs `127.0.0.1` for everyone. Rate limiting, audit logs, anything keyed on client IP — all broken.

The fix is a convention: forwarded headers. Caddy sets **`X-Forwarded-For`** with the client's address, **`X-Forwarded-Proto`** with the original scheme, and **`X-Forwarded-Host`** on every proxied request. Your application reads those instead of the socket address. No configuration needed — `reverse_proxy` does it by default.

Now the security problem. Headers are just text, and any client can send one. Try forging an address through your proxy from the last lesson:

```
curl -H "X-Forwarded-For: 203.0.113.9" http://127.0.0.1:8080/
```

Log the headers in your application and look at what arrived. The forged value is gone: Caddy replaced it with the address it saw on the socket. That's because the request didn't come from a **trusted proxy**, so Caddy refuses to believe its identity claims. Forgery defeated by default.

But sometimes there is a legitimate proxy in front of Caddy — a load balancer or a CDN. Then the socket address Caddy sees belongs to the CDN, and the real client address is in the header the CDN sends. You tell Caddy which peers to believe with the global `trusted_proxies` option:

```
{
  servers {
    trusted_proxies static 10.0.0.0/8
  }
}
```

Now, when a connection arrives from an address in `10.0.0.0/8`, Caddy trusts its forwarded headers and passes the real client identity to your upstream. Connections from anywhere else still get the socket address.

The mistake to avoid is trusting too widely. Set `trusted_proxies` to ranges you control, never to the whole Internet. If everyone is a trusted proxy, anyone can claim any address, and every downstream system that keys off client IP — bans, throttles, audit trails — believes the lie.

When your logs show impossible client addresses, or every visitor appears to come from one CDN IP, this setting is the first place to look.

Secure an HTTPS upstream

Verify an upstream certificate with the correct name and trust pool instead of disabling TLS checks.

When your upstream is on localhost, plain HTTP between Caddy and the app is fine. But when the upstream lives on another machine — another VM, another datacenter — that hop crosses a network, and it deserves encryption too.

Caddy speaks TLS to the upstream when you write the scheme into the address:

```
app.example.com {
  reverse_proxy https://api.internal.example.com
}
```

With `https://`, Caddy encrypts the upstream connection and verifies the upstream's certificate: the chain must be trusted and the name must match, exactly the checks a browser would make.

That's where internal services get awkward. Internal APIs often carry certificates from a **private CA**, one that Caddy's trust store has never heard of. The proxy starts returning 502s, and the runtime log tells you why:

```
journalctl -u caddy | grep -i 'certificate'
```

You'll find a line like `tls: failed to verify certificate: x509: certificate signed by unknown authority`.

There are two ways out. The wrong one is `tls_insecure_skip_verify`, which turns verification off entirely. Never use `tls_insecure_skip_verify` in production. Without verification, Caddy will happily send headers, cookies, and request bodies to anyone who can intercept the connection. Encrypted-but-unverified TLS protects you from nobody who matters.

The right fix is to trust the private CA explicitly, keeping every check enabled:

```
app.example.com {
  reverse_proxy https://api.internal.example.com {
    transport http {
      tls_trust_pool file /etc/caddy/internal-ca.pem
    }
  }
```

```
}
```

`tls_trust_pool file` loads your CA's root certificate as the trust anchor for this upstream. Hostname verification stays on, so you still know you're talking to the right server, signed by the issuer you chose.

Verify the trust chain from the server itself before blaming Caddy:

```
curl --cacert /etc/caddy/internal-ca.pem https://api.internal.example.com/health
```

If curl succeeds with the same CA file, Caddy will too. If curl fails, the problem is the certificate or the name, not your Caddyfile. One common variant: you dial the upstream by IP address while the certificate names a host. Fix the address, or set `tls_server_name` in the transport so verification checks the name the certificate actually carries.

Balance and check upstreams

Add multiple backends, active health checks, bounded retries, and a failure drill without repeating unsafe requests.

One backend is a single point of failure. Run two, and Caddy can spread traffic between them and route around the one that dies.

List multiple upstreams and add health checking:

```
:8080 {
  reverse_proxy 127.0.0.1:4001 127.0.0.1:4002 {
    lb_policy round_robin
    health_uri /health
    lb_try_duration 3s
  }
}
```

Three subdirectives, three jobs. `lb_policy round_robin` alternates requests between the upstreams instead of the default random pick. `health_uri` enables **active health checks**: Caddy requests `/health` on each upstream on a regular interval, and an upstream that fails the check is marked unhealthy and removed from rotation. `lb_try_duration 3s` gives each incoming request a three-second budget to find a working upstream before Caddy gives up and returns an error.

Start two throwaway backends to test with:

```
caddy respond --listen 127.0.0.1:4001 "backend one" &
caddy respond --listen 127.0.0.1:4002 "backend two" &
```

Send a few requests and watch them alternate:

```
for i in 1 2 3 4; do curl -s http://127.0.0.1:8080/; echo; done
```

Now run the failure drill. Kill backend one and keep sending requests. At first, requests that land on the dead upstream get retried onto the survivor within the try duration — clients see slow responses, not errors. Once the active health check fails, the dead upstream leaves rotation entirely and responses are fast again. Restart it and watch it return to rotation after its health check passes. That full cycle — degrade, eject, recover — is what you're buying with three lines of config.

Retries carry a trap worth understanding. Retrying a failed GET is safe. Retrying a POST that charges a credit card is not: the upstream may have processed the request before dying, and a

retry runs it twice. Only allow retries of state-changing requests when the application handles them idempotently. A missing response never proves the upstream did nothing.

One failure mode to recognize: if the `/health` endpoint itself breaks — say a deploy removed it — Caddy marks every upstream down and serves 502s while the apps are actually fine. When the whole fleet goes unhealthy at the same moment, suspect the check before the backends.

Check your understanding: proxy applications

Check the commands, configuration choices, trust boundaries, and failure cases from proxy applications.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Operate Caddy

Run Caddy as a service, reload safely, use logs and metrics, protect the API, and preserve recovery data.

Run Caddy as a service

Use the official system service, dedicated account, configuration paths, and journal instead of an improvised background process.

`caddy run` in a terminal dies with your SSH session. A real server needs Caddy supervised: started at boot, restarted after a crash, logging somewhere durable. On Linux that supervisor is systemd, and the official package already did the setup for you.

See what got installed:

```
systemctl status caddy
systemctl cat caddy
```

`status` shows whether Caddy is running, since when, and its recent log lines. `cat` prints the unit file itself. Read it once — it answers most questions about how Caddy runs: as the dedicated `caddy` user, loading `/etc/caddy/Caddyfile`, restarting on failure. Notice the `ExecReload` line too: it means `systemctl reload caddy` maps to Caddy's graceful reload, so configuration changes don't require a restart.

Logs go to the journal:

```
journalctl -u caddy --since today
journalctl -u caddy -f
```

The first command shows today's runtime log: startup, certificate activity, errors. The second follows it live, which is what you want open in a second terminal while changing configuration.

The service user matters more than people expect. `caddy` is an unprivileged account. The unit grants it just enough capability to bind ports 80 and 443, it can read `/etc/caddy`, and it writes its own state under `/var/lib/caddy`. It cannot read your home directory. That is the number-one source of "it works with `caddy run` but not as a service": you tested as your own user, and the service user can't reach the files.

Fix permission problems narrowly. Move site files somewhere sensible like `/srv/www`, make them readable by the `caddy` group, and confirm with `sudo -u caddy ls /srv/www`. Don't chmod 777 a directory tree to make an error disappear — you'd be trading a visible failure for an invisible one.

My advice: on servers, skip `caddy start` (Caddy's own backgrounding) entirely. It leaves a process that nothing supervises. The systemd service gives you boot integration, restart policy, and one obvious place to look when something breaks.

Reload without downtime

Validate a changed Caddyfile, reload through the admin API, and verify behavior before accepting the change.

Configuration changes shouldn't drop requests. Caddy is built around **graceful reloads**: it loads the new configuration, starts serving with it, and lets in-flight requests finish on the old one. Zero downtime — if you follow a safe sequence.

Here's the sequence I use, every time:

```
caddy fmt --overwrite Caddyfile
caddy validate --config Caddyfile
caddy reload --config Caddyfile
curl -f https://app.example.com/health
```

Format, validate, reload, verify. Each step earns its place.

`caddy validate` catches a broken configuration on the spot, before the running server ever sees it. If validation fails, you just fixed a production incident that never happened.

`caddy reload` sends the new configuration to the running Caddy through its admin API on `localhost:2019`. The server applies it gracefully: listeners stay open and existing connections finish. And if the new config fails to load, Caddy keeps running the old one and the command reports the error. A bad reload gets you an error message, not an outage.

The final `curl -f` is behavior verification. A valid config is not necessarily a correct config — you can validly route all traffic to the wrong upstream. Hit a health endpoint and confirm the site does what you meant. The `-f` flag makes curl exit non-zero on HTTP errors, so the same line works in a deploy script.

Keep the previous known-good Caddyfile until that check passes. Version control makes this free: roll back by checking out the old file and running the same four commands.

What you should not do is `systemctl restart caddy` for routine changes. A restart kills connections mid-flight. Reload exists precisely so you never have to. Save restarts for upgrading the Caddy binary itself.

And the classic mistake, one more time because it costs people real debugging hours: editing the Caddyfile changes nothing on its own. Caddy doesn't watch files. If the behavior doesn't match the file, first ask yourself — did I actually reload?

Observe requests and runtime

Separate access logs from runtime logs, preserve useful fields, and expose metrics only to an authorized monitoring path.

When something breaks, you need to know what Caddy saw. Caddy keeps two kinds of logs, and

knowing which one to read saves real time.

Access logs record HTTP requests: method, path, status, duration. **Runtime logs** record Caddy's own operations: startup, reloads, certificate issuance, upstream errors. Runtime logs are on by default — that's what you've been reading in the journal. Access logs you enable per site with the `log` directive:

```
app.example.com {
  log {
    output file /var/log/caddy/app-access.log
    format json
  }
  reverse_proxy 127.0.0.1:3000
}
```

Reload, then make one good request and one bad one:

```
curl -s https://app.example.com/ > /dev/null
curl -s https://app.example.com/nope > /dev/null
```

Now read the log. JSON logs are built for tooling, so use `jq`:

```
jq "{status: .status, uri: .request.uri, duration: .duration}" /var/log/caddy/app-access.log
```

Two entries, one 200 and one 404, each with URI and duration. Everything a request-debugging session needs, greppable and parseable. The `file` output rotates logs automatically, so this won't silently fill the disk.

Here's the division of labor between the two logs. A 502 shows up in the access log as the status the client received. The reason — connection refused, TLS failure, timeout — lives in the runtime log. Status questions go to the access log; "why" questions go to `journalctl -u caddy`.

Caddy also exposes Prometheus **metrics**. Turn them on with the `metrics` global option:

```
{
  metrics
}
```

Then scrape them from the admin endpoint:

```
curl -s http://localhost:2019/metrics | grep caddy_http_requests
```

That endpoint lives on the admin port, which never belongs on the public Internet. Point your monitoring at it over a private network or a tunnel.

One caution about log contents. Caddy redacts common credential headers like `Authorization` and `Cookie` by default — keep that protection. And remember that URLs themselves leak: a token in a query string lands in the access log in plain text. Treat log files with the same care as the data behind them.

Protect configuration and recovery state

Keep the admin API private, understand native JSON updates, preserve Caddy storage, and finish a tested recovery runbook.

Two assets decide whether a bad day is a blip or a disaster: control of the running configuration, and the state Caddy needs to come back after a loss. This last lesson covers both.

Caddy is controlled at runtime through its **admin API**, listening on `localhost:2019`. Every `caddy reload` you've run in this course used it. You can read the live configuration directly:

```
curl http://localhost:2019/config/ | jq .
```

That JSON is the actual running configuration — the truth, even if someone edited the Caddyfile without reloading. The API also accepts POST and PATCH requests that change configuration on the fly, which is exactly why it must stay private. Whoever reaches this endpoint owns the server: they can reroute your traffic anywhere. The default localhost binding is correct. Leave it alone, and if you ever need remote administration, tunnel over SSH instead of exposing the port.

The second asset is Caddy's **storage**: the data directory holding certificates, private keys, and the ACME account. Lose it and Caddy re-issues everything from scratch — slow at best, rate-limited at worst. Back it up with the built-in command:

```
caddy storage export --output caddy-storage.tar
```

`storage export` produces a tar archive of the whole storage. Treat it like a private key, because it contains your private keys: encrypt it, store it apart from your configuration backups, and never commit it to a repository.

Restore is the mirror image:

```
caddy storage import --input caddy-storage.tar
```

A backup you haven't restored is a guess. Run the drill once in a throwaway VM or container: install Caddy, restore the Caddyfile, import the storage, start the service, and confirm the site comes up over HTTPS. Then check the runtime log — no new certificate issuance means the restored state actually worked.

Your full recovery kit is three things: the Caddyfile in version control, the storage export somewhere encrypted, and the notes from lesson one about your binary version and modules. With those, a dead server is an hour of calm work instead of a scramble against expiring certificates.

Check your understanding: operate caddy

Check the commands, configuration choices, trust boundaries, and failure cases from operate caddy.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/caddy/.