Reverse Proxies and Load Balancers Course

Flavio Copes

Updated August 3, 2026

Contents

Proxy foundations **2**
- Run two backend services . . . 2
- Run a one-command reverse proxy . . . 3
- Write a Caddyfile . . . 4
- Inspect forwarded headers . . . 5
- Check your understanding: proxy foundations . . . 6

Routing and rewrites **6**
- Route by path . . . 6
- Route by hostname . . . 7
- Rewrite an upstream path . . . 8
- Proxy a WebSocket . . . 9
- Check your understanding: routing and rewrites . . . 10

TLS and trust boundaries **10**
- Terminate local HTTPS . . . 10
- Use HTTPS upstreams . . . 11
- Define trusted proxies . . . 12
- Protect the backend path . . . 13
- Check your understanding: tls and trust boundaries . . . 14

Load balancing **15**
- Balance two upstreams . . . 15
- Compare balancing policies . . . 16
- Add health checks . . . 17
- Bound retries and side effects . . . 18
- Check your understanding: load balancing . . . 19

Operate the proxy **19**
- Enable structured access logs . . . 19
- Set upstream time bounds . . . 20
- Reload configuration safely . . . 21
- Run the final failover lab . . . 22
- Check your understanding: operate the proxy . . . 24

Use Caddy to route HTTP traffic, terminate TLS, balance multiple backends, test failure behavior, and document every trust boundary.

What you'll learn: Run and operate a reverse proxy that routes requests, protects client identity, balances healthy backends, and fails predictably.

Proxy foundations

Run two backends, put Caddy in front, and inspect the separate client and upstream exchanges.

Run two backend services

Start two tiny HTTP servers on different loopback ports and make their identity visible in every response.

A reverse proxy needs an upstream application. Something has to sit behind the proxy and answer requests. In this course we use two small backends, because two is the minimum number that makes routing and load balancing visible without a framework.

The important design decision is identity. Every lesson that follows will ask a version of the same question: which backend handled this request? So each backend must say who it is in every response.

Write one backend, run it twice

Create `backend.mjs`:

```
import http from 'node:http'

const port = Number(process.argv[2])
http.createServer((request, response) => {
  response.setHeader('Content-Type', 'application/json')
  response.end(JSON.stringify({ port, path: request.url }))
}).listen(port, '127.0.0.1')

// node backend.mjs 4001
// node backend.mjs 4002
```

The port comes from the command line, so one file gives you as many instances as you want. Each response carries the port that produced it and the path the backend actually received. Those two fields are the evidence you'll rely on for the whole course.

Start both instances in two terminals:

```
node backend.mjs 4001
node backend.mjs 4002
```

Verify direct access

Request each backend directly and record its port:

```
curl http://127.0.0.1:4001/hello
# {"port":4001,"path":"/hello"}

curl http://127.0.0.1:4002/hello
# {"port":4002,"path":"/hello"}
```

Direct loopback access is the baseline before adding the proxy. Later, when a request travels through Caddy, you'll compare what the client sent with what the backend reports here.

If `curl` hangs or refuses the connection, the usual cause is a typo in the port argument. `node backend.mjs` without a port makes `port` become `NaN`, and the server never binds. Check the

terminal running the backend for an error.

One deliberate choice in the code: the servers listen on `127.0.0.1`, not on all interfaces. Do not bind the practice backends publicly. The proxy should be their intended entry point, and a later lesson checks that nothing can reach them any other way.

Run a one-command reverse proxy

Put Caddy in front of one backend and confirm the client no longer connects directly to the application port.

A **reverse proxy** accepts the client request and creates a separate upstream request. There are two distinct connections: client to proxy, and proxy to backend. The client sees the proxy address, not the backend listener, and everything the proxy does — routing, TLS, logging, balancing — happens between those two connections.

Caddy can run this whole arrangement from a single command, which makes it perfect for a first experiment.

Start the proxy

With the backend from the previous lesson running on port 4001:

```
caddy reverse-proxy --from http://127.0.0.1:8080 --to 127.0.0.1:4001 --access-log
```

`--from` is the address Caddy listens on. `--to` is the upstream. `--access-log` prints every proxied request to the terminal, which is exactly the visibility we want right now.

Verify the two connections

Request port 8080, the proxy's port:

```
curl http://127.0.0.1:8080/hello
# {"port":4001,"path":"/hello"}
```

The response body says port 4001, so the backend handled it. But you connected to 8080. Match this request with the line Caddy printed in its access log: same path, same status. That log line is the proof the proxy was in the middle.

Break the upstream

Now stop the backend process and repeat the request:

```
curl -i http://127.0.0.1:8080/hello
# HTTP/1.1 502 Bad Gateway
```

The proxy is still up — it answered you. It just couldn't complete its own connection to the upstream, and it reported that as a **502 Bad Gateway**. This distinction matters for the rest of the course: a proxy error and a backend error look different, and the status code tells you which connection failed.

Restart the backend and confirm the 200 comes back without touching Caddy at all.

Keep the first listener on loopback. Public proxying needs firewall, TLS, application authentication, and origin protection — all of which come later in this course. Exposing port 8080 to a network now would mean anyone on it can reach your backend through the proxy.

Write a Caddyfile

Move the proxy configuration into a readable file and validate it before loading.

The one-command proxy is great for experiments, but it disappears when the terminal closes and nobody can review it. A configuration file makes routing and operational policy reviewable. Caddy's file format is the **Caddyfile**, and Caddy can validate it before serving it.

Create the file

Create a file named `Caddyfile` (no extension) in your working directory:

```
http://127.0.0.1:8080 {
  reverse_proxy 127.0.0.1:4001
}

# validate with:
# caddy validate --config Caddyfile
```

The first line is the **site address**. Everything inside the braces applies to requests matching that address. The `reverse_proxy` directive does the same job as `--to` did in the previous lesson.

The explicit `http://` prefix matters here. Without a scheme, Caddy assumes you want HTTPS and tries to provision a certificate. For a loopback lab on a custom port, plain HTTP is what we want for now.

Format, validate, run

Caddy ships tooling for treating this file with care:

```
caddy fmt --overwrite Caddyfile
caddy validate --config Caddyfile
caddy run --config Caddyfile
```

`caddy fmt --overwrite Caddyfile` normalizes indentation so diffs stay clean. `caddy validate` parses the file and checks it can actually load, without binding any ports. `caddy run` starts the server in the foreground.

Validation is the step people skip, and it's the cheapest insurance you'll get. Try it: delete the closing brace, run `caddy validate --config Caddyfile` again, and read the error. It names the line where parsing failed. Fix the brace and validation passes silently with an exit code of 0.

Verify

```
curl http://127.0.0.1:8080/hello
# {"port":4001,"path":"/hello"}
```

Same behavior as the command-line version, but now the configuration is an artifact you can commit, review, and roll back.

Treat configuration as code. Keep secrets out, review changes, and preserve the last known good version. A Caddyfile in version control with a passing `caddy validate` is the foundation the operations lessons build on later.

Inspect forwarded headers

See how Host, X-Forwarded-For, X-Forwarded-Proto, and X-Forwarded-Host describe the original request.

The backend receives a new request from the proxy, not the client's original one. From the backend's point of view, every request now comes from the proxy's address. That breaks anything that needs the real client IP or the original protocol — rate limiting, audit logs, redirect URLs.

Forwarded headers preserve selected client-facing context that the application may need. Caddy adds them to the upstream request automatically.

Log what the backend sees

Change the lab backend to print its incoming headers:

```
http.createServer((request, response) => {
  console.log(request.headers)
  console.log('peer:', request.socket.remoteAddress)
  response.end('ok\n')
}).listen(4001, '127.0.0.1')
```

The extra `request.socket.remoteAddress` line matters. It shows the **immediate network peer** — who actually opened the TCP connection — which is a separate fact from anything a header claims.

Compare direct and proxied requests

Request the backend directly, then through Caddy:

```
curl http://127.0.0.1:4001/
curl http://127.0.0.1:8080/
```

The direct request shows a plain header set with `host: 127.0.0.1:4001`. The proxied request looks different:

```
host: '127.0.0.1:8080',
x-forwarded-for: '127.0.0.1',
x-forwarded-proto: 'http',
x-forwarded-host: '127.0.0.1:8080'
```

Read each one. `Host` is passed through unchanged, so the backend sees the hostname the client used. `X-Forwarded-For` carries the client's IP address. `X-Forwarded-Proto` records whether the client-facing connection was `http` or `https` — essential once the proxy terminates TLS but talks plain HTTP to the backend. `X-Forwarded-Host` preserves the original host in setups that rewrite it.

In both cases the peer address printed by the backend is a loopback address. Through the proxy, that peer is Caddy, no matter what the headers say.

The trust problem

Now the uncomfortable part. Send a forged header directly to the backend:

```
curl -H "X-Forwarded-For: 203.0.113.99" http://127.0.0.1:4001/
```

The backend prints `203.0.113.99` as if it were real. Headers are just text the sender chose. Forwarded headers are trustworthy only when they come through a proxy boundary you control and configure. A later lesson makes that boundary explicit with trusted proxy configuration.

Check your understanding: proxy foundations

Check the commands, decisions, safety boundaries, and failure cases from proxy foundations.

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

Take this interactive quiz online

Routing and rewrites

Route by path and host, change upstream paths deliberately, combine static files, and proxy WebSockets.

Route by path

Send API requests to one backend while returning a separate response for other paths.

One proxy usually fronts more than one thing. The classic split: API requests go to an application server, everything else gets the frontend. **Request matchers** choose which handler receives traffic.

Add a path matcher

Update the Caddyfile:

```
http://127.0.0.1:8080 {
  handle /api/* {
    reverse_proxy 127.0.0.1:4001
  }

  handle {
    respond "frontend\n"
  }
}
```

Each `handle` block is mutually exclusive — the first matching one wins, and Caddy orders them by matcher specificity. The `/api/*` matcher catches paths under `/api/`. The bare `handle` with no matcher is the fallback for everything else.

Reload and test each route:

```
curl http://127.0.0.1:8080/api/users
# {"port":4001,"path":"/api/users"}

curl http://127.0.0.1:8080/
# frontend
```

The API request reached the backend, and note what the backend saw: the full path `/api/users`, unchanged. `handle` routes the request but doesn't modify it. The next lesson covers stripping the prefix when the backend expects clean paths.

Test the edges

Now the requests people forget to try:

```
curl http://127.0.0.1:8080/api
# frontend

curl http://127.0.0.1:8080/api/
# {"port":4001,"path":"/api/"}
```

`/api` without a trailing slash does not match `/api/*` — the pattern requires the slash. It falls through to the frontend handler. Whether that's a bug or the behavior you want depends on your API contract, but you should decide it, not discover it. If you want both, use two matchers: `handle /api /api/*`.

Notice that matcher details affect edge paths such as `/api` without a trailing slash. These boundaries are where routing bugs live.

My advice is to write tests for near matches: the prefix without a slash, a path that shares a prefix like `/apiv2`, an encoded slash. A broad route can accidentally expose an internal handler, and you find out when someone else does.

Route by hostname

Serve two local hostnames through one listener and preserve distinct application boundaries.

A single machine with a single IP address can serve many sites. The HTTP `Host` header and TLS SNI let one proxy serve multiple sites on one address: the client names the site it wants, and the proxy routes on that name. Each site block can have its own upstream and policy.

Map the lab hostnames

The names need to resolve first. Add them to `/etc/hosts`:

```
127.0.0.1 app.lab.test admin.lab.test
```

Both names point to loopback, so the same Caddy listener receives both.

Create two site blocks

```
http://app.lab.test:8080 {
  reverse_proxy 127.0.0.1:4001
}

http://admin.lab.test:8080 {
  reverse_proxy 127.0.0.1:4002
}
```

Two site addresses, one port, two different upstreams. Caddy picks the block whose hostname matches the request's `Host` header.

Reload and request each name:

```
curl http://app.lab.test:8080/
# {"port":4001,"path":"/"}
```

```
curl http://admin.lab.test:8080/
# {"port":4002,"path":"/"}
```

Same IP, same port, different backends. You can prove the routing decision is purely the `Host` header by faking it:

```
curl -H "Host: admin.lab.test" http://127.0.0.1:8080/
# {"port":4002,"path":"/"}
```

That request never mentioned `admin.lab.test` in the URL. The header alone selected the admin backend.

The unmatched host

Now send a name Caddy doesn't know:

```
curl -i -H "Host: other.lab.test" http://127.0.0.1:8080/
```

Caddy returns an empty response rather than picking a site for you. That's the behavior you want. An unmatched Host should not silently reach a privileged backend — if it did, anyone who guessed your IP could skip the hostname entirely.

That last experiment is also the warning. Host routing is not user authorization. The client chooses the `Host` header, exactly like it chose the forged `X-Forwarded-For` earlier in the course. Routing `admin.lab.test` to a separate upstream organizes traffic; it doesn't authenticate anyone. Protect an admin application inside the application or trusted access layer, and treat the hostname split as convenience, not as a security boundary.

Rewrite an upstream path

Remove a public prefix before proxying and confirm what the backend receives.

Public URLs and backend URLs don't always agree. Clients call `/api/users`, but the backend was written to serve `/users` — it doesn't know it lives behind an `/api` prefix. A **rewrite** changes the URI used by later handlers, so the proxy can translate between the two contracts.

Strip the prefix

In the path-routing lesson, `handle /api/*` forwarded the full path. Swap it for `handle_path`:

```
handle_path /api/* {
  reverse_proxy 127.0.0.1:4001
}
```

`handle_path` behaves exactly like `handle` with one addition: it strips the matched prefix from the path before running its handlers.

Reload and verify with the lab backend, which echoes the path it receives:

```
curl http://127.0.0.1:8080/api/users
# {"port":4001,"path":"/users"}
```

The client asked for `/api/users`. The backend saw `/users`. That one-line change is the difference between ”backend must know its public prefix” and ”backend is prefix-agnostic”.

Compare `handle_path` with a plain `handle` by switching back briefly — the backend reports `/api/users` again. Keeping both behaviors clear in your head prevents a whole class of 404s

where the backend receives a path it never defined.

The contract problem

Rewrites are easy to write and easy to get subtly wrong. Make the external and internal contracts explicit so redirects and generated links remain correct.

Here's the classic failure. The backend at `/users` issues a redirect to `/users/42`. The client receives `Location: /users/42` — but the public path is `/api/users/42`. The rewrite happened on the way in, and nothing translated the way out. The client follows the redirect and gets the frontend handler instead of the API.

So test round trips, not just single requests:

```
curl -i http://127.0.0.1:8080/api/redirect-test
# check the Location: header the client actually receives
```

If the backend generates absolute paths, either configure it with a base path or keep the public and internal paths identical.

Two more edges worth testing while you're here: encoded paths and trailing slashes. And be careful that a prefix rewrite never crosses a boundary it shouldn't. Do not rewrite authentication or tenant boundaries accidentally — a rewrite that strips `/tenant-a` could let a request land on another tenant's routes.

Proxy a WebSocket

Pass an HTTP upgrade through the proxy and verify that a long-lived bidirectional connection survives.

WebSockets start with an HTTP upgrade, then continue as a long-lived connection. The client sends a normal GET with `Upgrade: websocket` and `Connection: Upgrade` headers. The server answers `101 Switching Protocols`, and from that moment the connection stops being HTTP — it's a bidirectional byte stream that stays open for minutes or hours.

That's a different beast from the request/response traffic we've proxied so far, and it's worth proving your proxy handles it. Caddy's HTTP reverse proxy supports the upgrade path without any special configuration.

Proxy the socket path

Use a WebSocket-capable lab backend — the `ws` package gives you one in a few lines — then proxy its path:

```
handle /socket {
  reverse_proxy 127.0.0.1:4001
}
```

Nothing WebSocket-specific in the config. Caddy detects the upgrade headers and switches to streaming mode for that connection on its own.

Verify the upgrade

You can watch the handshake succeed with plain curl by sending the upgrade headers yourself:

```
curl -i -N \
```

```
  -H "Connection: Upgrade" \
  -H "Upgrade: websocket" \
  -H "Sec-WebSocket-Version: 13" \
  -H "Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==" \
  http://127.0.0.1:8080/socket
# HTTP/1.1 101 Switching Protocols
```

The `101` proves the upgrade traveled through the proxy to the backend and back. For real message exchange, connect with a WebSocket client through port 8080 and send a few messages both ways.

Restart the backend

Here's the part that makes long-lived connections operationally different. With a client connected, restart the backend and observe how existing and new connections fail differently.

The existing connection dies immediately — the proxy can't preserve a TCP stream to a process that's gone. New connections fail until the backend is back, then succeed. Your open client doesn't automatically reconnect; that's the application's job.

This is why WebSocket-heavy systems obsess over reconnect logic and connection draining. Every proxy reload, backend deploy, or health-based removal severs live connections that HTTP clients would never notice. Plan connection draining and reconnect behavior before deploying a proxy change around long-lived clients — the config change is trivial, the client experience is not.

Check your understanding: routing and rewrites

Check the commands, decisions, safety boundaries, and failure cases from routing and rewrites.

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

Take this interactive quiz online

TLS and trust boundaries

Terminate local HTTPS, protect upstream TLS, and configure forwarded client identity safely.

Terminate local HTTPS

Let Caddy issue a local development certificate and proxy protected client traffic to a loopback backend.

TLS termination means the client's protected connection ends at the proxy. The proxy decrypts the traffic, then chooses how to reach the upstream — often over plain HTTP on a private network. The application never touches certificates, and the proxy becomes the one place where TLS is configured, renewed, and debugged.

This split is the main reason reverse proxies exist in so many stacks. Let's build it locally.

Use Caddy's internal issuer

Caddy ships with a built-in certificate authority for development. Use Caddy's internal issuer for the lab:

```
app.lab.test {
  tls internal
  reverse_proxy 127.0.0.1:4001
}
```

No port in the site address means Caddy serves HTTPS on 443 and redirects HTTP on 80. `tls internal` tells it to issue the certificate from its own local CA instead of contacting a public authority — which would fail anyway, since `app.lab.test` isn't a public name.

Map the hostname to loopback in `/etc/hosts` if you haven't already, then start Caddy.

Inspect what was issued

```
curl -v https://app.lab.test/ 2>&1 | grep -E "subject|issuer|SSL certificate"
```

Depending on your system, this either succeeds or fails with an unknown-issuer error. Both outcomes are informative. The certificate's subject is `app.lab.test`, and the issuer is a Caddy local authority — not Let's Encrypt, not anything your OS trusts by default.

If curl rejects it, Caddy can install its root into your local trust store:

```
caddy trust
```

After that, `curl https://app.lab.test/` returns the backend response over a verified connection. Trust the local root only on your own development client when needed — that's the entire intended audience.

Confirm what the backend sees

The backend still receives plain HTTP on 4001. Check its logged headers: `X-Forwarded-Proto` now says `https`. That header is how the application learns the client-facing connection was encrypted, even though its own listener never was.

One warning to take seriously. A local CA is powerful: anything that trusts its root will accept any certificate it signs, for any name. Do not copy its root key or install its certificate on unrelated systems. Keep it on your machine, for your lab, and remove the trust when the course is done.

Use HTTPS upstreams

Verify an HTTPS backend instead of disabling certificate checks between the proxy and application.

So far the proxy talked plain HTTP to its upstreams. On a loopback interface that's fine. But when the backend lives on another machine — a different host, a cloud VM, someone else's network — the second connection deserves the same protection as the first.

An **HTTPS upstream** adds encryption and identity verification on the proxy-to-backend connection. And identity is the part people forget: the certificate's name must match the upstream TLS server name, and it must chain to a CA the proxy trusts. Otherwise you're encrypting traffic to a server you never verified.

Proxy to a named HTTPS backend

Give the lab backend a name and a certificate signed by a lab CA (the TLS course on this site walks through creating one). Then proxy to it by name:

```
reverse_proxy https://backend.lab.test:4443 {
```

```
  transport http {
    tls_trust_pool file lab-ca.crt
  }
}
```

Two things changed from the plain setup. The upstream address now carries `https://` and the TLS port. And the transport declares which CA to trust: `tls_trust_pool file lab-ca.crt` points Caddy at the lab CA certificate instead of the system trust store, which knows nothing about your lab.

Verify both directions of the check

First the success path:

```
curl https://app.lab.test/
# {"port":4443,"path":"/"}
```

Now make it fail on purpose, because a check you've never seen fail isn't verified. Point `tls_trust_pool` at the wrong CA file and reload. The proxy returns a 502, and its log shows a certificate verification error: it refused to talk to an upstream it couldn't authenticate. Do the same by proxying to the backend's IP address instead of `backend.lab.test` — name mismatch, same refusal.

Confirm the proxy fails with the wrong CA or hostname and succeeds with the intended trust root and certificate. Those two failures are the security feature working.

The shortcut to refuse

When this fails in production, you'll find advice suggesting `tls_insecure_skip_verify`. It makes the error disappear by skipping verification entirely — the proxy will then accept any certificate from anyone, including an attacker between it and the backend.

Avoid `tls_insecure_skip_verify`. The error is telling you trust or identity is misconfigured. Repair trust and identity so the upstream connection remains authenticated: fix the trust pool, fix the certificate name, or reissue the certificate. Every one of those is a real fix. The flag is not.

Define trusted proxies

Accept client-address headers only from known upstream proxies and preserve the immediate peer separately.

You already saw that anyone can send an `X-Forwarded-For` header. Now consider Caddy's position in a bigger chain. If another CDN or load balancer sits before Caddy, it may supply the original client address in exactly that header — and Caddy should believe it. The same header from an arbitrary direct client is untrusted and should be discarded.

The difference between those two cases is not the header. It's who the immediate peer is. A **trusted proxy** configuration lists the peer addresses whose forwarded headers you accept.

Configure the trusted range

Trusted proxies are a server-level setting, so they go in the Caddyfile's global options block. Configure a placeholder private lab range:

```
{
```

```
  servers {
    trusted_proxies static private_ranges
  }
}
```

The `static` module takes a fixed list. `private_ranges` is a shorthand for the RFC 1918 private networks plus loopback — reasonable for a lab where your "upstream proxy" is another local process. In production you'd list the specific ranges your CDN publishes, nothing wider.

Watch the behavior change

The lab backend logs its incoming headers. Send a forged client address through Caddy and watch what arrives upstream:

```
curl -H "X-Forwarded-For: 203.0.113.50" http://127.0.0.1:8080/
```

Run this before and after adding the `trusted_proxies` block, and compare what the backend logs.

Without trust configured, Caddy treats your curl as an untrusted client: it discards the incoming forwarded headers and writes its own, so the backend sees the real peer address. With loopback trusted, Caddy accepts your claim and passes `203.0.113.50` along, appending the connecting address to the chain.

That second behavior is correct when the peer really is your CDN. It's a spoofing hole when the peer is the open internet. Send a forged header directly and compare behavior before and after a correctly modeled trusted proxy — the exercise makes the boundary visible in a way no diagram does.

Keep the real peer separate

Whatever the headers say, the TCP peer address is a separate fact. Log it separately at the backend (`request.socket.remoteAddress` in the lab backend). When forwarded data and the peer disagree unexpectedly, the peer is the one that can't lie.

Two rules to carry into production. Use real provider ranges only from current official sources — CDN ranges change, and stale lists silently break attribution. And never trust every address merely to make logs look right. That turns user input into security identity, and rate limits, bans, and audit trails all inherit the lie.

Protect the backend path

Keep application listeners private so clients cannot bypass proxy TLS, routing, logging, or access policy.

Everything this course has built — TLS termination, routing, forwarded-header trust, access logs — assumes traffic goes through the proxy. A proxy policy is effective only when the backend is not reachable through an alternate public path. If a client can hit port 4001 directly, every rule you wrote on port 443 is a suggestion.

This is the most commonly broken assumption in real deployments. An app gets bound to `0.0.0.0` during debugging, a cloud firewall rule opens "temporarily", and the backend quietly serves unencrypted, unlogged traffic to anyone who scans for it.

Check what the backend exposes

Bind locally or restrict network access to the proxy. Start with what's actually listening. Verify the backend listener:

```
ss -lntp | grep 4001
# macOS: lsof -nP -iTCP:4001 -sTCP:LISTEN
```

Read the local address column carefully:

```
LISTEN 0 511 127.0.0.1:4001 0.0.0.0:*
```

`127.0.0.1:4001` means loopback only — unreachable from any other machine. If you see `0.0.0.0:4001` or `*:4001`, the backend accepts connections on every interface, and the only thing between it and the network is a firewall you may or may not have.

Our lab backend passes because `backend.mjs` binds explicitly: `.listen(port, '127.0.0.1')`. Delete that second argument and re-run `ss` to see the difference. Node, like most runtimes, defaults to all interfaces.

Test from the outside

A listener check tells you intent. An external probe tells you truth. From an authorized second device on the same network:

```
curl --max-time 3 http://192.168.1.20:4001/
# curl: (28) Connection timed out  <- what you want

curl --max-time 3 http://192.168.1.20:8080/
# proxy answers  <- also what you want
```

The backend port should be unreachable while the proxy remains available. Only probe machines you own or administer.

Layer the controls

When backends must bind beyond loopback — separate hosts, containers — binding alone isn't enough, and obscurity is worth nothing. A hidden port is not protection: scanners find nonstandard ports in minutes. Use explicit binding and firewall rules together:

```
sudo ufw allow from 192.168.1.10 to any port 4001 proto tcp
sudo ufw deny 4001/tcp
```

That allows the proxy host (192.168.1.10 here) and rejects everyone else, even if the bind address is wide. Two layers, so one mistake doesn't become an exposure.

Check your understanding: tls and trust boundaries

Check the commands, decisions, safety boundaries, and failure cases from tls and trust boundaries.

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

Take this interactive quiz online

Load balancing

Distribute requests, add health checks, bound retries, and understand stateful application behavior.

Balance two upstreams

Send traffic to two backend instances and observe Caddy's upstream selection.

A **load balancer** chooses an available upstream for each request. That's the whole job: one incoming stream of requests, several backends, a selection decision per request. In Caddy, you get one by giving `reverse_proxy` more than one upstream address.

Configure both backends

With both loopback backends from the first lesson running:

```
reverse_proxy 127.0.0.1:4001 127.0.0.1:4002
```

That's the entire change. Caddy's default selection policy is `random` — each request picks an upstream at random. The default policy is suitable for many stateless labs, but policy must match application behavior, which the next lesson digs into.

Observe the distribution

Our backends report their port in every response, so distribution is measurable. Send repeated requests and count each backend port in the responses:

```
for i in {1..20}; do
  curl --silent http://127.0.0.1:8080/ | grep -o '"port":[0-9]*'
done | sort | uniq -c
#  11 "port":4001
#   9 "port":4002
```

Roughly even, not exactly even — it's random selection, not strict alternation. The point isn't the exact split. The point is that you measured it instead of assuming it, a habit that pays off when a policy misbehaves in production.

Kill one backend

Now stop the 4002 process and run the loop again:

```
#  13 "port":4001
#   7 failed
```

About half your requests fail with 502. Caddy doesn't yet know 4002 is dead — it keeps selecting it and hitting a connection error each time. Record failures before health policy is added: this is the baseline that health checks (two lessons from now) exist to fix, and seeing the failure rate first makes the fix meaningful.

Restart the backend and confirm both ports reappear in the counts.

What this doesn't buy you

Two upstreams on one laptop demonstrate selection mechanics, nothing more. Two processes are not high availability if they share the same failing machine, storage, configuration, and release. Real redundancy separates failure domains — different hosts at minimum, and the same thinking applied

to whatever database both instances share. A load balancer distributes requests; it can't distribute away a single point of failure sitting behind both upstreams.

Compare balancing policies

Switch between round robin, least connections, and sticky selection and connect each policy to workload behavior.

How should the balancer pick an upstream? There's no universally right answer, only policies that fit or fight your workload.

The three families you'll actually use: **round robin** rotates requests through upstreams in order. **Least connections** favors the currently less busy upstream. **Hash-based policies** can keep related clients or requests sticky to the same upstream, by hashing the client IP, a header, or a cookie.

Make selection deterministic

Start with round robin, because its behavior is predictable enough to verify by eye. Make selection visibly deterministic:

```
reverse_proxy 127.0.0.1:4001 127.0.0.1:4002 {
  lb_policy round_robin
}
```

Send an even request count and count the ports:

```
for i in {1..10}; do
  curl --silent http://127.0.0.1:8080/ | grep -o '"port":[0-9]*'
done | sort | uniq -c
#   5 "port":4001
#   5 "port":4002
```

Exactly even, unlike the random default from the previous lesson. Round robin's weakness is that it's blind: a request going to a struggling upstream counts the same as one going to an idle upstream.

Expose round robin's blind spot

Create a slow endpoint and compare `least_conn`. Add a route to the lab backend that sleeps for five seconds, then hold a few slow requests open against round robin — quick requests keep landing on the busy upstream anyway, right behind the stuck ones.

Switch the policy:

```
lb_policy least_conn
```

Repeat the experiment. While one upstream holds the slow connections, new requests drain to the other one. `least_conn` reacts to actual load, which makes it a better default when request costs vary a lot.

Whichever policy you test, record distribution instead of assuming it. The counting loop is your instrument.

Sticky selection, and its cost

Sometimes requests aren't interchangeable because the server holds per-client session state in memory. Then you need stickiness: `lb_policy client_ip_hash` keeps a client on one upstream, and `lb_policy cookie` pins browsers via a cookie.

Use it knowingly, not by default. Sticky routing can hide server-side session design problems and produce uneven load — one heavy client saturates one upstream while others idle, and when that upstream dies, its sessions die with it. Prefer shared or externalized state when practical (a database, Redis), so any upstream can serve any client and the balancer stays free to balance.

Add health checks

Remove unhealthy upstreams from selection using a narrow endpoint that represents readiness.

Two lessons ago, killing one backend made half your requests fail. The balancer kept selecting a dead upstream because nothing told it to stop. **Health checks** close that gap: the proxy probes each upstream and removes failing ones from selection.

A health check should answer whether the instance can safely receive new work. That's a narrower question than "is the process running". A backend can be alive but useless — database connection lost, disk full, still warming up. It should be cheap and distinguish process life from readiness.

Give the backend a health endpoint

Add a `/health` route to the lab backend that you can flip on demand:

```
let healthy = true

// inside the request handler:
if (request.url === '/health') {
  response.statusCode = healthy ? 200 : 503
  return response.end()
}
if (request.url === '/break') {
  healthy = false
  return response.end('broken\n')
}
```

The `/break` route lets you simulate the interesting failure: a process that's alive but not ready.

Add an active check

```
reverse_proxy 127.0.0.1:4001 127.0.0.1:4002 {
  health_uri /health
  health_interval 5s
  health_timeout 2s
}
```

`health_uri` is the path Caddy probes on each upstream. `health_interval 5s` sets the probe frequency, and `health_timeout 2s` counts a slow answer as a failure — an upstream too overloaded to answer its health check in two seconds shouldn't get new work either.

Watch removal and recovery

Start the counting loop from the earlier lessons, then make one backend's health endpoint fail while its process remains alive:

```
curl http://127.0.0.1:4002/break
```

Within one probe interval, port 4002 disappears from the response counts. No failed client requests this time — Caddy checked before selecting. Confirm Caddy stops selecting it and later restores it: restart the backend (fresh state, `healthy` is true again) and 4002 rejoins within a probe cycle. Compare this with the raw failure rate you recorded before health checks existed.

Design the contract

The endpoint's meaning deserves a decision, not an accident. Do not make health checks perform expensive writes or depend on every remote service. A check that queries five dependencies turns any dependency blip into full fleet removal — every upstream fails the check at once, and the balancer has nowhere left to send traffic. A check that does real writes generates load precisely when the system is weakest.

Define the readiness contract intentionally: verify the few things this instance needs to serve correctly, keep it read-only, keep it fast.

Bound retries and side effects

Retry failed upstream selection without duplicating unsafe application operations.

Retries look like free reliability. An upstream fails, the proxy quietly tries another one, the client never notices. Sometimes that's exactly what happens. Sometimes the retry charges a customer twice.

The line between those outcomes is where the failure happened. A proxy may safely retry a failed connection before sending a request — nothing reached the backend, so nothing happened. But retrying after an upstream received a state-changing request can duplicate work. If a POST creates an order and the connection drops before the response arrives, the proxy can't know whether the order exists. Retrying it might create a second one.

This is why HTTP method semantics matter here. GET, HEAD, PUT, and DELETE are defined as **idempotent** — repeating them shouldn't change the outcome. POST is not.

Configure a bounded retry window

Configure a short selection window:

```
reverse_proxy 127.0.0.1:4001 127.0.0.1:4002 {
  lb_try_duration 2s
  lb_try_interval 250ms
  lb_retry_match GET
}
```

`lb_try_duration 2s` gives Caddy up to two seconds to find a working upstream for a request. `lb_try_interval 250ms` spaces the attempts. `lb_retry_match GET` restricts which requests may be retried after reaching an upstream: only GETs, the safest class.

Verify both sides of the boundary

Stop one backend and test GET recovery:

```
curl --silent http://127.0.0.1:8080/
# {"port":4001,"path":"/"}
```

Even when the balancer first picks the dead upstream, the request lands on the live one within the try window. No visible failure.

Now the other side. Send a POST while one upstream is down and watch some fail rather than retry — that's `lb_retry_match` doing its job. For a POST, use an application idempotency key instead of assuming the proxy can infer safety: the client sends a unique key per logical operation, the backend stores it, and a duplicate arrival returns the original result instead of repeating the work. That moves retry safety into the layer that actually knows what the request does.

Respect the caller's deadline

Keep retry budgets below the caller deadline. If your client gives up at 2 seconds and your `lb_try_duration` is also 2 seconds, the proxy's heroics finish after the audience has left — the client saw a timeout anyway, and may retry on its own.

That stacking is the real danger. Layered retries can multiply load during an outage: client retries three times, proxy tries multiple upstreams per attempt, and a struggling backend receives several times its normal traffic at the worst possible moment. Budget retries at one layer, keep the window short, and let the layers above fail fast.

Check your understanding: load balancing

Check the commands, decisions, safety boundaries, and failure cases from load balancing.

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

Take this interactive quiz online

Operate the proxy

Read logs, set deadlines, reload safely, test failover, and document the complete traffic path.

Enable structured access logs

Record requests with useful identity, status, duration, size, and upstream evidence without logging secrets.

When a user says ”the site is slow” or ”I got an error”, the proxy is the best witness you have. It saw the client request, the routing decision, the upstream answer, and the timing of all three. Access logs connect client symptoms with proxy decisions — but only if you can query them.

That's the case for structure. A JSON log line is a record with named fields, not a sentence to parse with regexes. Structured logs are easier to filter and correlate with backend events.

Enable a local log

```
http://127.0.0.1:8080 {
```

```
  log {
    output file access.json
    format json
  }
  reverse_proxy 127.0.0.1:4001
}
```

The `log` directive enables access logging for this site. `output file` writes to a file instead of stderr, and `format json` emits one JSON object per request.

Generate evidence worth reading

Make successful and failed requests, then find status, duration, request ID if present, and upstream failure details:

```
curl http://127.0.0.1:8080/hello
# stop the backend, then:
curl http://127.0.0.1:8080/hello
```

Now read the log with `jq`:

```
jq '{status, duration, uri}' access.json
# {"status": 200, "duration": 0.003, "uri": "/hello"}
# {"status": 502, "duration": 0.001, "uri": "/hello"}
```

The success line tells you the path, the outcome, and how long the whole exchange took. The failure line is more interesting: status 502 with a tiny duration means the upstream connection failed immediately — a refused connection, not a slow backend. A 502 after several seconds points at timeouts instead. You just diagnosed two different failure modes from two numbers.

Try a filter you'd actually run during an incident:

```
jq 'select(.status >= 500)' access.json
```

One line, every server-side failure. That query is the reason `format json` earns its place.

Log carefully

Access logs are a data liability as much as a debugging tool. They capture headers, and headers carry credentials. Redact authorization, cookies, sensitive query values, and private bodies — Caddy's `log` directive accepts filters that replace or delete named fields before they're written, so tokens never land on disk in the first place.

Then treat the file itself as sensitive. Restrict log access and retention: tighten file permissions, and rotate with a defined lifetime. A year of forgotten access logs is a breach waiting for its disclosure date.

Set upstream time bounds

Bound connection and response-header waits according to the application contract.

Without time bounds, a stuck upstream can hold proxy and client resources. Every hung request keeps a connection open, a client waiting, and proxy memory allocated. Enough of them and a single misbehaving backend drags the proxy — and every other site it serves — down with it.

Timeouts should distinguish connecting from waiting for response headers, because those failures

mean different things. A connect that takes more than a couple of seconds on a healthy network means the host is unreachable or overwhelmed — waiting longer won't help. A backend that accepted the connection but hasn't sent headers is doing work, and how long that legitimately takes depends entirely on your application.

Add explicit transport limits

```
reverse_proxy 127.0.0.1:4001 {
  transport http {
    dial_timeout 3s
    response_header_timeout 10s
  }
}
```

`dial_timeout 3s` bounds the TCP connection attempt. `response_header_timeout 10s` bounds the wait between sending the request and receiving the upstream's response headers. Two knobs, two distinct failure modes.

Verify the bound

Create a backend route that never responds:

```
if (request.url === '/hang') {
  return // accept the request, answer nothing
}
```

Then confirm the proxy returns within the expected window, and time it from the client side:

```
curl -o /dev/null -s -w "%{http_code} %{time_total}s\n" http://127.0.0.1:8080/hang
# 502 10.003s
```

Ten seconds, then a 502 — that's `response_header_timeout` firing. Without it, this curl would sit forever, and so would every real client stuck behind that route. Check the access log too: the entry logs the boundary, showing the duration and the upstream error, which is how you'll tell ”timeout” from ”connection refused” during a real incident.

Test `dial_timeout` separately by pointing the proxy at a port nobody listens on and watching it give up in three seconds.

Choose values from evidence

Do not choose arbitrary tiny timeouts. A 2-second header timeout feels snappy until it kills every legitimate report generation and large query your application performs — you've converted your slowest features into guaranteed failures. Measure legitimate slow operations first, from access logs, then set the bound above the honest worst case.

And when one endpoint genuinely needs minutes, the fix usually isn't a huge timeout. Design asynchronous work when a request should not wait: accept the job, return immediately, let the client poll or get notified. Timeouts protect the request path; long work belongs off it.

Reload configuration safely

Validate and reload Caddy without discarding active connections or losing the known good file.

The proxy is the front door for everything behind it. Restarting it to change a route means dropping every connection to every site it serves — which is why proxies support **graceful reloads**: apply new configuration without stopping the process or refusing traffic.

But zero-downtime machinery doesn't protect you from loading a broken config. Process discipline does. A safe change has a validated candidate, exact deployment action, smoke test, and rollback target. Miss any of the four and you're gambling with the front door.

Validate, then reload

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

`caddy validate` parses and provisions the candidate without binding ports — syntax errors and bad directives fail here, at zero cost. Only after it passes does `caddy reload` hand the file to the running process. Caddy loads the new config, swaps it in, and keeps serving; a config that fails to load is rejected while the old one stays active.

Make the sequence unskippable by chaining it:

```
caddy validate --config Caddyfile && caddy reload --config Caddyfile
```

The `&&` means an invalid file never even reaches the reload step.

Watch a live connection cross the reload

The claim behind ”graceful” deserves a test. Keep a WebSocket or slow request open during the reload and observe its behavior:

```
curl --max-time 30 http://127.0.0.1:8080/hang &
caddy reload --config Caddyfile
```

The in-flight request keeps running on the old configuration while new requests get the new one. That's the mechanism behind zero-downtime deploys at the proxy layer — old connections drain, new connections land on the new config, and no client sees a refused connection.

Smoke test and keep the way back

A reload that ”worked” only proves the config loaded, not that it's right. Run the route and health smoke tests afterward:

```
curl --silent --fail http://127.0.0.1:8080/api/users || echo "route broken"
```

A couple of curls against your critical routes catch the misplaced `handle` block that validation can't — validation checks form, not intent.

Finally, the rollback target. Do not overwrite the only known good configuration without version control or a recovery copy. The Caddyfile lives in git, so the recovery procedure is boring: check out the previous version, validate, reload. Boring is the goal. The worst place to reconstruct a working config from memory is mid-outage with the front door broken.

Run the final failover lab

Demonstrate routing, TLS, balancing, health removal, recovery, logging, and backend privacy in one controlled exercise.

Every lesson so far isolated one behavior. Real proxies run all of them at once, and the interesting failures happen at the seams. The final lab joins every boundary. Start two private backends, one HTTPS proxy, structured logs, health checks, and bounded retries.

Assemble the full configuration

This Caddyfile is the course in one file:

```
app.lab.test {
  tls internal
  log {
    output file access.json
    format json
  }
  reverse_proxy 127.0.0.1:4001 127.0.0.1:4002 {
    lb_policy round_robin
    health_uri /health
    health_interval 5s
    health_timeout 2s
    lb_try_duration 2s
    lb_retry_match GET
  }
}
```

Start both backends bound to loopback, validate, run. Confirm the baseline before injecting any fault: both ports appear in responses, the certificate is the local one, `access.json` grows.

Run the observation loop

Run a repeatable request loop:

```
for number in {1..20}; do
  curl --silent --show-error --fail https://app.lab.test/ || echo failed
  sleep 0.5
done
```

Keep it running in one terminal. It's your client's-eye view for the whole exercise — every fault you inject either shows up here or it doesn't, and both results are findings.

Inject faults, one at a time

Work through the sequence: stop one backend, restore it, reload one routing change, and inspect distribution and failures.

Stop backend 4002 mid-loop. You may see a `failed` line or two before the next health probe, then clean responses from 4001 only. Restore it and watch 4002 rejoin the rotation within a probe interval. Then edit one route, `caddy validate`, `caddy reload`, and confirm the loop never misses.

After each fault, write the first failed boundary for every intentional fault. Which layer noticed first — the health check, the retry window, the client loop? If the answer surprises you, that's the lab working. A fault the loop never saw is a resilience win worth understanding; a fault it saw for ten seconds reveals a probe interval worth tuning.

Finish with the privacy check from the trust module: the backend ports must be unreachable from

outside while `https://app.lab.test` answers.

Clean up deliberately

Keep the whole lab local or on infrastructure you own. And when finished, remove what you installed: the `/etc/hosts` entries, and the local CA root — `caddy untrust` removes it from the trust store. Remove local CA trust and stop listeners when finished. A forgotten trusted root on a development machine is a real credential, and cleanup is part of operating a proxy, not an afterthought.

Check your understanding: operate the proxy

Check the commands, decisions, safety boundaries, and failure cases from operate the proxy.

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/reverse-proxies-load-balancers/.