# systemd Course

Flavio Copes

Updated August 3, 2026

## Contents



Learn to run Linux server applications with systemd units, dependencies, restart policies, timers, resource limits, logs, and safe deployments.

**What you'll learn:** Create, harden, operate, and troubleshoot a systemd service and timer for a server application.

## Units and state

Understand the manager, unit files, load state, active state, enablement, and dependencies.

### What systemd manages

Understand PID 1, the service manager, units, and why systemd owns more than background processes.

When the kernel finishes booting, it starts exactly one process. On most Ubuntu servers that process is systemd, running as **PID 1**. Every other process on the machine descends from it.

Check it yourself:

```
ps -p 1 -o pid,comm,args
```

```
  PID COMMAND         COMMAND
    1 systemd         /sbin/init
```

`/sbin/init` is a symlink to the systemd binary. The name survives from older init systems, but the process is systemd.

#### Units, not scripts

Older init systems treated startup as a pile of shell scripts that ran in order. systemd models the system as **units**: named objects with a type, a state, and relationships to other units.

Services are the unit type you will touch most, but they are not the only one. Sockets, timers, mounts, paths, targets, and devices are units too. A timer can activate a service. A target can group dozens of units into one named system state, like `multi-user.target`.

This is why systemd owns more than background processes. It knows that your backup timer triggers `backup.service`, that `nginx.service` wants the network first, and which control group every process belongs to.

#### See what it tracks

List the services running right now:

```
systemctl list-units --type=service --state=running
```

```
  UNIT           LOAD   ACTIVE SUB     DESCRIPTION
  cron.service   loaded active running Regular background program processing daemon
  nginx.service  loaded active running A high performance web server
  ssh.service    loaded active running OpenBSD Secure Shell server
```

Swap `--type=service` for `--type=timer` or `--type=socket` and you see the other unit types the manager is tracking. The manager holds their state and relationships instead of treating startup as one long shell script.

Connect the two views: the process tree hanging off PID 1, and the unit list describing why each piece of it exists.

One habit to build early: when a process misbehaves, ask systemd about it before reaching for `ps`. `ps` tells you a process exists. systemd tells you which unit owns it, why it started, and what happens when it dies.

### Read unit state

Separate load, active, substate, and enablement before deciding what a service needs.

`systemctl status` combines several different facts in one screen. Read each fact instead of reducing the output to running or broken.

```
systemctl status nginx
```

```
 nginx.service - A high performance web server and a reverse proxy server
    Loaded: loaded (/usr/lib/systemd/system/nginx.service; enabled; preset: enabled)
    Active: active (running) since Mon 2026-08-03 09:14:02 UTC; 3h 12min ago
  Main PID: 812 (nginx)
     Tasks: 3 (limit: 4573)
    Memory: 8.4M
    CGroup: /system.slice/nginx.service
            812 "nginx: master process /usr/sbin/nginx"
            813 "nginx: worker process"
            814 "nginx: worker process"
```

#### The four facts

**Loaded** tells you whether systemd found and parsed the unit file, and shows the path it used. If this says `not-found` or `bad-setting`, nothing else matters yet.

**Active** is the broad runtime state: `active`, `inactive`, `failed`, `activating`.

The **substate** in parentheses adds detail. `running` means a live process. A `Type=oneshot` unit can be `active (exited)`: it ran, finished with success, and stayed logically active. That is normal, not a bug.

**Enablement** (`enabled` on the Loaded line) describes boot-time symlinks. It does not prove the unit is running now.

You can query the same facts as plain properties, which is handy in scripts:

```
systemctl show --property=LoadState,ActiveState,SubState,UnitFileState nginx
```

```
LoadState=loaded
ActiveState=active
SubState=running
UnitFileState=enabled
```

#### The classic confusion

People mix up enablement and activity constantly. A service can be `enabled` but `failed` because it crashed an hour ago. It can be `disabled` but `active` because someone started it by hand. After the next reboot that hand-started service will be gone, and you will wonder why.

So check both axes. `systemctl is-active nginx` answers "is it running now". `systemctl is-enabled nginx` answers "will it start at boot". Both commands exit non-zero on a negative answer, so they work in shell conditions.

Inspect one installed service with the `systemctl show` command above. Explain each of the four values in one sentence. If you can do that, you know what the service needs next.

## Find unit files and overrides

Locate vendor units and local drop-ins without editing package-owned files in place.

Unit files live in more than one place, and the location decides who owns them.

Packages normally install unit files under `/usr/lib/systemd/system` or `/lib/systemd/system` (on Debian and Ubuntu the two are the same directory). Treat those files as vendor property. Local administrator changes belong under `/etc/systemd/system`, and anything there wins over the vendor copy.

The classic mistake is editing the vendor file in place. It works until the next package upgrade silently replaces the file, and your change is gone with no warning.

### See the effective configuration

Use `systemctl cat` to see every fragment systemd combined for a unit, with the path of each one:

```
systemctl cat ssh.service
```

```
# /usr/lib/systemd/system/ssh.service
[Unit]
Description=OpenBSD Secure Shell server
...
# /etc/systemd/system/ssh.service.d/override.conf
[Service]
ExecStart=
ExecStart=/usr/sbin/sshd -D -p 2222
```

The second block is a **drop-in**: a small file that overrides only the settings it names. The vendor unit stays untouched, and the local difference is explicit.

### Create a drop-in with systemctl edit

`systemctl edit` opens an editor on the right file, creating it if needed:

```
sudo systemctl edit ssh.service
```

It writes to `/etc/systemd/system/ssh.service.d/override.conf` and runs `daemon-reload` for you when you save. `systemctl edit --full` instead copies the whole unit into `/etc/systemd/system`, which you rarely want.

One drop-in gotcha: list settings like `ExecStart=` append rather than replace. To change the command you must clear it first with an empty assignment, then set the new value — exactly what the override above does. If you skip the empty line, systemd complains that the service has two `ExecStart=` lines.

To audit a whole machine, `systemd-delta` lists every unit that differs from its vendor version.

Run `systemctl cat ssh.service` on your server and read where each fragment comes from. Then open `sudo systemctl edit ssh.service`, note the target path in the editor comment, and leave without saving.

## Dependencies and targets

Read requirement and ordering relationships without assuming that After also starts another unit.

systemd separates whether a unit is required from when it should start. This distinction prevents many confusing unit files, and misreading it causes many broken ones.

### Requirement vs ordering

`Requires=` and `Wants=` express **requirement** relationships. `Wants=` is the soft version: start that other unit too, but carry on if it fails. `Requires=` is hard: if the required unit fails to start, this unit fails with it. Prefer `Wants=` unless the service genuinely cannot exist without the dependency.

`After=` and `Before=` express **ordering only**. `After=postgresql.service` means "if both of us are starting, let PostgreSQL go first". It does not pull PostgreSQL in.

That leads to the classic bug: a unit with `After=network-online.target` but no `Wants=`. Nothing asked the target to be reached, so the ordering is meaningless. The two directives travel in pairs:

```
[Unit]
Description=Demo API server
Wants=network-online.target
After=network-online.target postgresql.service
```

This unit pulls in `network-online.target`, waits for it, and orders itself after PostgreSQL without demanding it exists.

### Targets group units

A target is a unit with no process of its own. It exists to group other units around a boot or operational state. `multi-user.target` is the normal server boot state; when you write `WantedBy=multi-user.target` in an `[Install]` section, enabling the service creates a symlink in that target's `.wants/` directory.

### Read the graph

You can walk relationships in both directions:

```
systemctl list-dependencies --reverse network-online.target
```

```
network-online.target
  demo-api.service
  nginx.service
```

The `--reverse` flag answers "who depends on this" instead of "what does this depend on".

To see ordering as systemd resolved it, query the unit's effective properties:

```
systemctl show demo-api.service -p Wants -p After
```

Run the `list-dependencies` command on your server. Find one unit that wants the target, then confirm one ordering relationship in that unit's effective configuration. If you find an `After=` with no matching `Wants=` or `Requires=`, you have found a latent bug.

## Check your understanding: units and state

Check units, service state, enablement, dependencies, overrides, and the system and user managers.

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

Take this interactive quiz online

## Build a service

Run an application with a dedicated account, explicit environment, and predictable lifecycle.

### Choose a service type

Choose Type=simple, exec, notify, forking, or oneshot from the process behavior you actually control.

A service type tells systemd two things: when startup has succeeded, and how to identify the main process. Pick it from how your program behaves, not from what you copied last time.

#### The types that matter

`Type=simple` is the default. systemd considers the service started the instant the process is forked, before your binary even runs. That means `systemctl start` can report success for a service whose executable does not exist.

Use `Type=exec` for a normal foreground process when an execution failure must fail startup. It behaves like `simple`, but `systemctl start` waits until the binary has actually been executed, so a wrong path or a permission problem fails the start command itself:

```
[Service]
Type=exec
ExecStart=/usr/bin/node /opt/app/server.js
```

`Type=notify` needs application support: the program calls `sd_notify()` to send `READY=1` when it can really serve traffic. That makes ordering meaningful for anything started `After=` this service. Only use it when the application documents that support.

`Type=forking` exists for traditional daemons that detach: the parent exits, a child keeps running. Pair it with `PIDFile=` so systemd can track the right process. New software should stay in the foreground instead.

`Type=oneshot` runs finite setup work. systemd waits for the process to exit before considering the unit started. Add `RemainAfterExit=yes` when the unit should stay `active (exited)` after finishing:

```
[Service]
Type=oneshot
RemainAfterExit=yes
ExecStart=/usr/local/bin/prepare-cache-dirs
```

#### The failure mode to recognize

The classic mismatch is `Type=forking` on a program that never forks. systemd waits for the parent to exit, the parent never does, and startup hangs until the timeout kills it. The journal shows a start timeout even though the process was healthy the whole time. The fix is not a longer timeout. It is the right type.

Read the startup behavior of an application you run: does it stay in the foreground, detach, or finish and exit? Choose a type and write down the event that should count as successful startup.

## Write a minimal service

Create a small unit with an explicit executable, working directory, user, and restart boundary.

A useful service unit states exactly what process systemd should run and which account should own it.

Use absolute paths in `ExecStart=`. Set `WorkingDirectory=` only when the program needs it. Run network applications as a dedicated unprivileged user rather than root.

Create a disposable `hello.service` that runs `/usr/bin/sleep infinity` as a regular service account. Validate it with `systemd-analyze verify` before starting it.

Create /etc/systemd/system/hello.service:

```
[Unit]
Description=Practice HTTP service

[Service]
ExecStart=/usr/bin/node /opt/hello/server.js
User=hello
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

Run systemd-analyze verify on the file before daemon-reload. Start it without enabling it first, inspect status and logs, then stop it. Use an absolute executable path and a dedicated user so the unit does not depend on an interactive shell. Finally, inspect the effective unit with systemctl cat hello.service and confirm it matches the file you intended to load.

## Pass configuration and secrets

Give a service its runtime configuration without embedding secrets in the unit or source repository.

A service does not inherit the full environment of your interactive shell. The `PATH` tweaks and exported variables in your `.bashrc` do not exist when systemd starts the process. Define every required value deliberately, in the unit.

### Environment and EnvironmentFile

`Environment=` suits small non-secret values, written directly in the unit:

```
[Service]
Environment=NODE_ENV=production
EnvironmentFile=/etc/demo-api/app.env
ExecStart=/usr/bin/node /opt/demo-api/server.js
```

`EnvironmentFile=` loads `KEY=VALUE` lines from a separate file. That keeps configuration out of the unit and out of your repository:

```
# /etc/demo-api/app.env
PORT=3000
DATABASE_URL=postgres://app@10.0.0.5/app
```

Make that file root-owned and unreadable to others:

```
sudo chown root:root /etc/demo-api/app.env
sudo chmod 600 /etc/demo-api/app.env
```

If the file is missing, the service fails to start. Prefix the path with a dash (`EnvironmentFile=-/etc/demo-api/local.env`) when the file is optional.

#### Why environment variables are weak for secrets

Even with a root-owned file, environment variables may still be exposed through process inspection or debugging. `systemctl show demo-api -p Environment` prints them. Child processes inherit them. Crash reports and debug tooling can dump them.

For sensitive data, prefer application-supported credential files. systemd has a mechanism built for this:

```
[Service]
LoadCredential=db-password:/etc/demo-api/db-password
```

The service reads the secret from `$CREDENTIALS_DIRECTORY/db-password`, a private per-service path. The value never appears in the environment or in `systemctl show`. The application needs to support reading a secret from a file, but most database drivers and libraries do.

#### Do the inventory

List every value your application needs to run. Classify each one as ordinary configuration or secret. Ordinary configuration goes in `Environment=` or an `EnvironmentFile=`. Secrets go in a credential file with tight ownership and mode, loaded through `LoadCredential=` or the application's own secret-file option.

Doing this classification once, on paper, is faster than discovering a leaked `DATABASE_URL` in a process listing later.

### Load and start a new unit

Validate a unit, reload the manager, start it, and confirm the process and logs before enabling boot startup.

systemd does not automatically reread every changed unit file. It keeps a parsed copy in memory, so you have to tell the manager when unit definitions change. Forgetting that step is the most common systemd mistake there is.

#### The sequence

```
systemd-analyze verify /etc/systemd/system/demo-api.service
sudo systemctl daemon-reload
sudo systemctl start demo-api.service
systemctl status demo-api.service
journalctl -u demo-api.service -b
```

`systemd-analyze verify` catches syntax errors, unknown directives, and missing executables before anything runs. Then `daemon-reload` makes the manager reread unit files, `start` runs the service once, and `status` plus the journal confirm the main PID exists and the application logged a healthy startup.

#### Two failures you will hit

If you edit a unit and restart without reloading, systemd runs the old definition and tells you so:

```
Warning: The unit file, source configuration file or drop-ins of
demo-api.service changed on disk. Run 'systemctl daemon-reload' to reload units.
```

Read the warnings in `systemctl status`. This one means the running service and the file on disk disagree.

The other classic is a wrong `WorkingDirectory=`. The process never starts, and the journal names the exact step that failed:

```
demo-api.service: Changing to the requested working directory failed: No such file or directo
demo-api.service: Failed at step CHDIR spawning /usr/bin/node: No such file or directory
demo-api.service: Main process exited, code=exited, status=200/CHDIR
```

Exit statuses in the 200 range come from systemd itself, before your program ran. `status=200/CHDIR` points at the directory, `203/EXEC` at the executable path or permissions.

#### Enable only after it works

Enabling a broken service just schedules a failure for the next boot. Once the current start works, wire it into boot and verify the state:

```
sudo systemctl enable demo-api.service
systemctl is-enabled demo-api.service   # enabled
```

Follow the complete sequence with a disposable service. Reboot only in a test environment; on a real server, `systemctl is-enabled` is enough to verify the intended boot state.

### Check your understanding: build a service

Check service types, execution rules, identities, environment, working directories, and validation.

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

Take this interactive quiz online

## Reliability and resources

Control restarts, startup order, timeouts, resources, and failure behavior.

### Design restart policy

Restart unexpected failures without turning a permanent configuration error into a tight restart loop.

`Restart=on-failure` can recover a service from crashes and non-zero exits. It cannot repair bad configuration. A service that dies because its config file is invalid will die again on every restart, forever, and the restart policy will hide that from a casual glance.

#### The knobs

```
[Unit]
```

```
StartLimitIntervalSec=60
StartLimitBurst=5

[Service]
Restart=on-failure
RestartSec=2
```

`Restart=on-failure` restarts on crashes, unclean signals, and non-zero exit codes, but not on a clean stop. `Restart=always` also restarts clean exits, which is right for daemons that should never exit on their own.

`RestartSec=` adds a delay between attempts. The default is 100 milliseconds, which turns a broken service into a tight loop. Two seconds is a saner floor.

`StartLimitIntervalSec=` and `StartLimitBurst=` (in the `[Unit]` section) rate-limit starts: with the values above, five failures within sixty seconds put the unit in a permanent `failed` state instead of looping.

#### How Restart masks a crash loop

Here is the trap. You check the service, it says `active (running)`, and you move on. But it has been crashing every few seconds and systemd keeps resurrecting it. Uptime in `systemctl status` resets on each restart — a service that is always "up since 4 seconds ago" is a loop, not a healthy daemon.

The restart counter makes it unambiguous:

```
systemctl show demo-api.service -p NRestarts
```

```
NRestarts=47
```

A non-zero, climbing `NRestarts` means the policy is papering over a real failure. The journal shows the same story as repeated `Scheduled restart job, restart counter is at N` lines. An operator should still see a service that repeatedly fails — alert on the counter or on the failed state, not just on "is it running".

#### Try it

Make a disposable test service exit with status 1 (`ExecStart=/bin/false` works). Start it and watch the restart count and the rate limit trigger:

```
demo-fail.service: Start request repeated too quickly.
demo-fail.service: Failed with result 'start-limit-hit'.
```

Fix the cause, then clear the state with `systemctl reset-failed demo-fail.service` before starting it again.

### Model startup order and readiness

Express real dependencies and application readiness instead of adding arbitrary sleep commands.

Ordering a service after the network target does not prove that a remote database or API is ready. `network-online.target` means the local network stack came up. Your database on another machine can still be rebooting, slow, or down.

That gap is where sleep hacks come from:

```
[Service]
ExecStartPre=/bin/sleep 15
ExecStart=/usr/bin/node /opt/demo-api/server.js
```

This works until the database takes sixteen seconds. Then it fails, someone bumps the sleep to thirty, and every deploy gets slower. The sleep encodes a guess, not a dependency.

#### Model what systemd can actually see

systemd can order units on the same machine. Use that for local dependencies:

```
[Unit]
Wants=network-online.target
After=network-online.target postgresql.service
```

If PostgreSQL runs on this host, `After=postgresql.service` is a real, checkable ordering. Use `Requires=` instead of `Wants=` only when your service is useless without the dependency and should fail with it.

For anything remote, the dependency is invisible to systemd. Let the application retry temporary remote failures with bounded backoff: try the connection, wait, try again, and give up with a clear error after a budget. Combined with the restart policy from the previous lesson, this recovers from slow dependencies without a single sleep.

#### Readiness is a separate claim

"Started" and "ready to serve" are different events. `Type=notify` closes that gap: the application calls `sd_notify()` with `READY=1` only when it can really handle traffic, and units ordered `After=` it wait for that signal. Use it only when the application can report readiness — check its documentation before switching the type.

You can see how long a service took to reach ready:

```
systemd-analyze blame | head -5
```

Services with sleep hacks stand out immediately in that list.

#### Try it

Take one sleep-based startup workaround from a design you know. Replace it with either a real local dependency (`After=` a unit on the same host) or an application-level retry with backoff. Then write one sentence stating the boundary: which part systemd guarantees, and which part the application handles.

### Set timeouts and stop cleanly

Give startup and shutdown bounded time while letting the application handle its normal termination signal.

When you stop a service, systemd sends a termination signal and waits before escalating. By default that is SIGTERM, a 90 second wait, then SIGKILL for anything still alive. Your application should use that window to stop accepting work, finish in-flight requests, and flush what it must.

SIGKILL cannot be caught. Anything the process had not written yet is lost. The goal of this lesson is to make sure your service never gets that far in normal operation.

### Set bounds from measured behavior

```
[Service]
TimeoutStartSec=30
TimeoutStopSec=20
```

Use `TimeoutStartSec=` and `TimeoutStopSec=` from measured behavior, not guesses. If your application drains connections in five seconds, twenty is a comfortable stop bound. If startup ever legitimately takes a minute, a 30 second start timeout will kill healthy starts.

`KillSignal=` changes the initial signal, but the default SIGTERM is normally correct — it is the signal every runtime and framework documents for graceful shutdown. Change it only for software that documents something else.

Avoid `KillMode=process`. It signals only the main process and can leave children behind, still holding ports and files after systemd considers the service stopped. The default (`control-group`) signals the whole process tree, which is almost always what you mean.

### Watch a stop happen

Follow the journal in one terminal while stopping the service in another:

```
journalctl -u demo-api.service -f
```

```
sudo systemctl stop demo-api.service
```

A clean stop logs the application's own shutdown messages and then `Deactivated successfully`. A service that ignores SIGTERM looks like this instead:

```
Stopping demo-api.service - Demo API server...
demo-api.service: State 'stop-sigterm' timed out. Killing.
demo-api.service: Killing process 1423 (node) with signal SIGKILL.
demo-api.service: Main process exited, code=killed, status=9/KILL
```

If you see `stop-sigterm timed out`, the fix is in the application first: handle SIGTERM and exit. Raising `TimeoutStopSec=` only hides the missing handler.

Stop a test service while following its journal. Confirm which signal it receives, how long it needs, and whether every child process exits.

## Control CPU, memory, and tasks

Use cgroup-backed unit controls to contain one service without confusing a limit with capacity planning.

systemd places every service in its own control group. That is what makes resource limits practical: unit properties constrain the whole process tree, including children and anything they spawn. A worker pool cannot escape the limit by forking.

### The four controls

```
[Service]
MemoryHigh=400M
MemoryMax=512M
CPUQuota=50%
TasksMax=64
```

Use `MemoryHigh=` for pressure: above it, the kernel throttles and reclaims memory from the service aggressively, but the process survives. Use `MemoryMax=` as a hard boundary: above it, processes in the group get killed. Setting both gives the service a warning zone before the cliff.

`CPUQuota=` limits CPU time — `50%` means half of one CPU. `TasksMax=` bounds process and thread creation, which is your defense against fork bombs and runaway worker pools.

Put these in a drop-in (`systemctl edit demo-api.service`) rather than the vendor unit, then restart the service.

#### Verify the limit and the current use

```
systemctl show --property=ControlGroup,MemoryCurrent,TasksCurrent demo-api.service
```

```
ControlGroup=/system.slice/demo-api.service
MemoryCurrent=61128704
TasksCurrent=11
```

That is roughly 58 MB and 11 tasks right now. `systemd-cgtop` gives you the same numbers live, across all services, sorted by usage.

#### A limit is not capacity planning

Limits need monitoring because hitting them is still a failure. If the service breaches `MemoryMax=`, the journal shows the kill:

```
demo-api.service: A process of this unit has been killed by the OOM killer.
demo-api.service: Main process exited, code=killed, status=9/KILL
```

and `systemctl show demo-api.service -p Result` reports `oom-kill`. The service did not get slower. It died. If that keeps happening, the answer is a bigger budget or a smaller workload, not deleting the limit — the limit is what kept the rest of the machine alive.

Inspect one real service with the `systemctl show` command above. Propose one limit based on measured use and expected peaks, with `MemoryHigh=` around the peak and `MemoryMax=` comfortably above it.

### Check your understanding: reliability and resources

Check restart policy, rate limits, dependencies, readiness, timeouts, resource controls, and OOM behavior.

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

Take this interactive quiz online

## Timers and operations

Replace fragile cron jobs, deploy safely, reload configuration, and inspect execution.

### Create a service and timer

Run scheduled server work as a paired oneshot service and timer with separate logs and state.

A systemd timer activates another unit, normally a service with the same base name. This is systemd's answer to cron, and the split into two files is the point: the service says what to run, the timer says when. You can run the service by hand any time, and swap the schedule without touching the command.

Compared to a crontab line, you also get the job's output in the journal, real dependencies, resource limits, and catch-up behavior for machines that were off.

### The service half

Put the command in a `Type=oneshot` service:

```
# /etc/systemd/system/backup.service
[Unit]
Description=Nightly PostgreSQL backup

[Service]
Type=oneshot
ExecStart=/usr/local/bin/backup-postgres.sh
```

No `[Install]` section. The service is never enabled on its own; the timer owns the schedule.

### The timer half

The schedule goes in a timer using `OnCalendar=` (or monotonic settings like `OnUnitActiveSec=` for "every N minutes after the last run"):

```
# /etc/systemd/system/backup.timer
[Unit]
Description=Run backup.service every night

[Timer]
OnCalendar=*-*-* 03:00:00
Persistent=true

[Install]
WantedBy=timers.target
```

Enable the timer, not the service:

```
sudo systemctl daemon-reload
sudo systemctl enable --now backup.timer
systemctl list-timers backup.timer
```

```
NEXT                         LEFT     LAST PASSED UNIT         ACTIVATES
Tue 2026-08-04 03:00:00 UTC 9h left  -    -      backup.timer backup.service
```

`list-timers` is your proof: it shows the next elapse, and after the first run, when it last fired.

The classic mistake is enabling `backup.service` instead of `backup.timer`. The backup then runs once at every boot and never on schedule, and `list-timers` shows nothing — which is exactly how you spot it.

### Check the schedule expression

Calendar expressions are easy to get subtly wrong. `systemd-analyze calendar` parses one and shows the next elapse:

```
systemd-analyze calendar "*-*-* 03:00:00"
```

Design `backup.service` and `backup.timer` as above without running a real backup — point `ExecStart=` at `/bin/true` first. Verify both files load, check `list-timers`, and confirm the schedule with `systemd-analyze calendar`.

## Handle missed and randomized runs

Choose what happens after downtime and avoid making every server start maintenance at the same instant.

A cron job that was scheduled while the machine was off just never happens. Timers let you choose the behavior instead of inheriting it.

### Catching up with Persistent=

`Persistent=true` lets a calendar timer catch up after the machine was off. systemd records the last trigger time on disk; if the machine boots and a run was missed, the timer fires once, immediately.

```
[Timer]
OnCalendar=*-*-* 03:00:00
Persistent=true
RandomizedDelaySec=30min
```

That behavior is useful only when a late run is safe. A missed backup should absolutely catch up. A billing job that charges customers should probably not fire as a surprise the moment a machine comes back — especially during incident recovery, when a boot triggering a pile of catch-up jobs is the last thing you want.

This is why **idempotent** jobs are safer: retries or catch-up runs do not duplicate damage. A backup that overwrites today's file is idempotent. A job that appends charges is not, and it needs its own run-once guard before you let a timer catch it up.

### Spreading the load

`RandomizedDelaySec=` spreads work across a window. With the 30 minute value above, each server picks a random delay after 03:00, so a fleet of a hundred machines does not hammer the backup target at the same instant.

`AccuracySec=` controls scheduling precision and defaults to one minute — systemd batches timer wakeups to save power. If a job genuinely must fire at 03:00:00, set `AccuracySec=1s`. Most jobs do not care, and the default is fine.

Verify the resulting schedule instead of trusting your reading of it:

```
systemctl list-timers backup.timer
```

```
NEXT                         LEFT      LAST                         PASSED  UNIT         ACTIVA
Tue 2026-08-04 03:11:42 UTC 9h left   Mon 2026-08-03 03:24:17 UTC 14h ago backup.timer backup
```

The randomized offset is visible right in the `NEXT` column.

### Decide per job

Work through three common jobs: a temp-file cleanup, a database backup, and a monthly billing run. For each one, decide whether a missed run should catch up, and give it a persistence and randomization policy. Cleanup and backup usually get `Persistent=true`. Billing usually gets `Persistent=false` and an alert when a run is missed, so a human decides.

## Deploy a service safely

Sequence artifact installation, validation, manager reload, restart, health checks, and rollback.

A deployment should change one known artifact and preserve a path back to the previous version. Everything else in this lesson follows from that sentence.

### Install atomically, validate before restarting

Install files atomically where possible. A versioned directory plus a symlink switch gives you atomic cutover and instant rollback:

```
sudo cp -r build/ /opt/demo-api/releases/v42
sudo ln -sfn /opt/demo-api/releases/v42 /opt/demo-api/current
```

The service's `ExecStart=` points at `/opt/demo-api/current/server.js`. Flipping the symlink back to `v41` is the whole rollback.

Validate configuration before restart, using whatever check the application offers — `nginx -t`, `node --check`, a config linter. A restart is the wrong moment to discover a typo.

### Reload the right thing

Use `daemon-reload` only for unit changes. New application code does not need it; a changed `.service` file or drop-in does. Cargo-culting `daemon-reload` into every deploy is harmless but hides the real rule, and then one day someone skips it after actually editing the unit — and the old definition keeps running.

Restart or reload, then check the service, logs, and an application-level health endpoint:

```
sudo systemctl restart demo-api.service
systemctl is-active demo-api.service          # active
journalctl -u demo-api.service --since "2 min ago"
curl -fsS http://localhost:3000/health        # {"status":"ok","version":"v42"}
```

`is-active` proves systemd's view. The health endpoint proves the application's view, including that the new version actually loaded. You want both, because a service can be `active` while serving errors.

### Decide the rollback before you need it

Write down the exact trigger that means ”roll back now” — for example: the health check fails for 60 seconds, or the error rate doubles. Vague triggers turn into long incidents while people debate.

```
sudo ln -sfn /opt/demo-api/releases/v41 /opt/demo-api/current
sudo systemctl restart demo-api.service
```

Write a deployment checklist for one of your services: artifact install, config validation, whether `daemon-reload` applies, restart, the exact health checks, the rollback trigger, and the command

that restores the previous known-good version. Keep it to one screen so it gets used under pressure.

## Run transient work

Use systemd-run for one-off or temporary work that still needs logs, limits, and lifecycle tracking.

`systemd-run` creates a transient unit without leaving a permanent unit file behind. You get the machinery of a service — journal capture, resource limits, cgroup tracking — for a command you will run once.

Start with a harmless one:

```
systemd-run --user --wait /usr/bin/true
```

```
Running as unit: run-r1f3a2b9c4d.service; invocation ID: 7b0f...
Finished with result: success
Main processes terminated with: code=exited, status=0/SUCCESS
Service runtime: 4ms
```

`--wait` makes the command block until the unit finishes and print the result. `--user` runs it in your user manager, no root needed.

### Where this earns its keep

A transient service is useful for an administrative command that needs resource controls or journal capture. Say you need to rebuild a search index, it eats memory, and you refuse to let it take down the API on the same box:

```
sudo systemd-run --unit=reindex -p MemoryMax=1G -p CPUQuota=50% \
  /usr/local/bin/reindex-search
```

The job now has a name, a memory ceiling, and its output lands in the journal:

```
systemctl status reindex.service
journalctl -u reindex.service -f
```

Compare that with running it from your shell: no limits, output lost when your SSH session drops, and the process dies with the session unless you remembered `nohup` or tmux.

A transient **scope** is the other variant: `systemd-run --scope` places an existing command tree under systemd management while it still runs attached to your terminal. Useful when you want limits on an interactive process.

### The cleanup gotcha

Successful transient units vanish when they finish. Failed ones stay visible in `systemctl --failed` until you clear them with `systemctl reset-failed reindex.service`. That is a feature — the failure evidence survives — but it surprises people who expect transient to mean traceless.

Run the `--user --wait` example above, then inspect what remains with `systemctl --user status`. Then decide the boundary for yourself: anything you run twice, or that a teammate must find and understand later, deserves a permanent unit file instead.

## Check your understanding: timers and operations

Check timers, persistence, deployment sequencing, reloads, transient units, and execution inspection.

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

Take this interactive quiz online

## Security and troubleshooting

Reduce privileges, analyze units, read the journal, and diagnose failed services.

### Reduce service privileges

Start with a dedicated account and add systemd sandboxing controls that match what the application needs.

A service should receive only the files, capabilities, devices, and kernel interfaces required for its job. If the process gets compromised, the sandbox decides what the attacker actually holds.

Step one is always a dedicated unprivileged account (`User=demo-api`). The sandboxing directives build on top of that.

### Measure before you harden

systemd will grade your unit for you:

```
systemd-analyze security demo-api.service
```

```
  NAME                        DESCRIPTION                                  EXPOSURE
  PrivateTmp=                 Service has access to other software's...
  ProtectSystem=              Service has full access to the OS file...
  NoNewPrivileges=            Service processes may acquire new privi...
...
→ Overall exposure level for demo-api.service: 9.2 UNSAFE
```

The score is a heuristic, not a verdict. Its value is the sorted list: it shows which missing control exposes the most.

### The high-value controls

Useful controls include `NoNewPrivileges=`, `PrivateTmp=`, `ProtectSystem=`, `ProtectHome=`, and `CapabilityBoundingSet=`. Put them in a drop-in (`sudo systemctl edit demo-api.service`), not the vendor unit:

```
[Service]
NoNewPrivileges=yes
PrivateTmp=yes
ProtectSystem=strict
ProtectHome=yes
ReadWritePaths=/var/lib/demo-api
```

`NoNewPrivileges=yes` blocks the process from gaining privileges through setuid binaries. `PrivateTmp=yes` gives it a private `/tmp`. `ProtectSystem=strict` mounts the entire filesystem read-only for this service, with `ReadWritePaths=` carving out the directories it genuinely writes. `ProtectHome=yes` hides home directories. `CapabilityBoundingSet=` with an empty value drops all capabilities — right for most services that just listen on a high port and talk to a database.

#### Apply gradually, and expect breakage

Apply them gradually because an incompatible control can prevent correct startup. The typical failure after `ProtectSystem=strict` looks like this in the journal:

```
demo-api.service: Error: EROFS: read-only file system, open '/var/lib/demo-api/cache.json'
```

That is the sandbox working. The fix is not removing the protection — it is adding the specific path to `ReadWritePaths=`, restarting, and retesting.

Run `systemd-analyze security` for one of your services. Choose one high-impact exposure, add a compatible control in a drop-in, restart, and verify the application still works before moving to the next one.

### Filter the journal

Query one unit, boot, time window, priority, or process instead of scrolling through unrelated server logs.

The journal stores structured fields alongside each message: the unit, the boot, the priority, the PID. Filters turn that structure into focused evidence, which beats scrolling through a mixed stream of everything the server logged.

#### Narrow by unit, boot, and time

```
journalctl -u demo-api.service -b
```

`-u` selects one unit's messages — both what the application printed and what systemd said about it. `-b` restricts to the current boot, which removes last week's noise from today's investigation.

Time windows stack on top:

```
journalctl -u demo-api.service --since "14:00" --until "14:30"
journalctl -u demo-api.service --since "10 min ago"
```

`--since` and `--until` accept timestamps and human phrases. Combine them with `-u` and you can isolate one incident on one service in seconds.

#### Narrow by priority, then follow

`-p` filters by syslog priority. `-p err` shows errors and worse; `-p warning` includes warnings:

```
journalctl -u demo-api.service -p err -b
```

An empty result here is meaningful: the service logged no errors this boot, so look elsewhere.

`-f` follows new entries, like `tail -f` for one unit:

```
journalctl -u demo-api.service -f
```

Keep that running while you restart a service or send a test request. Watching cause and log effect together is the fastest way to connect them.

#### Previous boots need persistent storage

Previous-boot logs require persistent journal storage and can be read with `-b -1`. If you instead see this, the journal lives in memory and died with the reboot:

```
Specifying boot ID or boot offset has no effect, no persistent journal was found.
```

The fix on Ubuntu is creating the on-disk directory and restarting journald:

```
sudo mkdir -p /var/log/journal
sudo systemctl restart systemd-journald
```

Do this before you need it. The most interesting boot is usually the one that just crashed.

Choose a recent service event — a restart, an error, a deploy. Produce the smallest journal query that includes the event and enough surrounding context to explain it. If your query returns more than a screen or two, add another filter.

### Diagnose a failed service

Read the manager result, exit status, effective unit, and application messages before editing configuration.

Start with `systemctl status`, but do not stop at its short log excerpt.

Inspect `systemctl show` for `Result`, `ExecMainCode`, and `ExecMainStatus`. Read the unit journal and `systemctl cat` output. Validate the executable and configuration as the service user.

Break a disposable unit with an invalid executable path. Record the exact evidence, repair the path, reload the manager, and confirm the failed state clears.

Collect the service state and the current boot's logs before restarting:

```
systemctl status hello.service --no-pager
journalctl -u hello.service -b --no-pager -n 100
systemctl show hello.service -p Result -p ExecMainStatus
```

Read the first relevant error, not only the final "failed" line. Check the executable, user, working directory, environment, and permissions. If the command works in your shell but fails as a service, run it as the service user with the same working directory and configuration. Change one cause, restart once, then verify the application from outside systemd.

### Complete the service review

Review one production-style unit from identity and dependencies through recovery, logging, limits, and deployment.

A good unit file is a small operational contract. Another person should understand how the service starts, stops, fails, and recovers — without asking you. This closing lesson turns everything from the course into one repeatable review.

#### Collect the evidence

Review the service's account, paths, configuration, dependencies, restart policy, timeouts, resources, sandboxing, logs, timer relationships, health check, and rollback procedure. Each item maps to a command you already know:

```
systemctl cat demo-api.service
systemctl show demo-api.service -p User,WorkingDirectory,Wants,After
systemctl show demo-api.service -p Restart,RestartSec,TimeoutStopSec,NRestarts
systemctl show demo-api.service -p MemoryMax,TasksMax,Result
systemd-analyze security demo-api.service
systemctl list-timers --all
```

```
journalctl -u demo-api.service -p warning -b
```

`systemctl cat` shows the contract as written, drop-ins included. The `show` queries reveal the effective values — including defaults nobody set on purpose, like a missing memory limit or the 90 second stop timeout. The journal query surfaces warnings the service has been logging while everyone looked away.

Two questions have no command and matter most: where is the health check, and what is the rollback procedure? If the answer to either is "ask Flavio", the contract is incomplete.

### Write it down, small

A review that lives in your head expires in a week. Keep the artifact tiny:

```
Service: demo-api.service           Reviewed: 2026-08-03
Strength: restart policy verified - killed the process,
  recovered in 2s, NRestarts incremented as expected
Risk: DATABASE_URL passed via Environment=,
  visible in `systemctl show` to any local user
Improvement (tested): moved the password to
  LoadCredential=, restarted, login path verified
```

Name one verified strength, one concrete risk, and one reversible improvement you tested. Each claim needs evidence: the strength was exercised, not assumed; the risk names how you would exploit or trigger it; the improvement was applied, verified, and could be rolled back with a single drop-in removal.

The trap in service reviews is producing opinions instead of observations. "Restart policy looks fine" is an opinion. "I killed the main process and watched it recover in two seconds" is a review.

Run this review against one real service you operate. It usually takes twenty minutes and almost always finds something.

## Check your understanding: security and troubleshooting

Check privilege reduction, sandboxing, journal filters, failed starts, effective configuration, and final review.

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/systemd/.