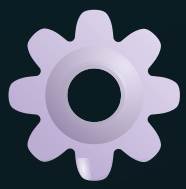

systemd in Depth

A Complete 11-Chapter Init System & Service Management Course

Topics covered:

Unit files & systemctl basics · Custom services & dependencies

Targets & journald · Timers & socket activation

cgroups & resource control · systemd-analyze & boot troubleshooting

Capstone: a real multi-unit service stack, fully explained

Exercises: 33 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · resolves systemctl/service calls used unexplained across ws1 and ansible1

Table of Contents

CHAPTER 1 OF 11

What systemd Actually Is & Why It Replaced init

systemd in Depth

Chapter 1 · What systemd Actually Is & Why It Replaced init

`ws1`'s own hardening course ran `systemctl restart apache2` and `systemctl enable fail2ban` without ever explaining what those words meant. `ansible1-10`'s own capstone wrapped `service: name: fail2ban state: restarted` around exactly the same underlying mechanism. `bash3` only gestured at it in passing. This course, starting now, is the actual explanation.

PID 1 — The First Process, With a Special Job

When the Linux kernel finishes its own initialization, it starts exactly one process directly: **PID 1**. Every other process on the system is, eventually, a descendant of it. PID 1 carries a genuinely special responsibility beyond just "the first thing that runs" — it also **reaps** orphaned processes, adopting them when their original parent process dies, so they never become permanent, un-cleaned-up zombies. On any modern Linux distribution — including Debian, `ws1`'s own target — PID 1 is **systemd**.

System V init — The Sequential, Runlevel-Based Model

The traditional init system read a sequence of numbered shell scripts (the classic `/etc/init.d/`, `rc.d`-style layout), organized into **runlevels** — 0 for halt, 1 for single-user mode, 3 or 5 for multi-user with or without a graphical environment, 6 for reboot. The real limitation: startup was fundamentally **sequential** — script *N+1* never started until script *N* finished, even when the two had nothing to do with each other, wasting real boot time waiting on completely unrelated services. Dependencies between services were expressed only informally, through the order scripts happened to be numbered — a fragile convention, not an explicit, checkable relationship.

systemd's Answer — Parallel, Dependency-Based Startup

systemd starts services in parallel wherever genuinely possible, and expresses real dependencies **explicitly** — previewed here, covered in full in `systemd1-4`'s own `Wants` / `Requires` / `After` / `Before` material — rather than relying on script-numbering convention. This directly resolves System V init's own sequential-boot cost: independent services no longer wait on each other purely because of where they happened to land in a numbered sequence.

A Concrete Comparison — Booting the Same System, Two Ways

```
# System V init — strictly sequential
S01networking  →  S02syslog  →  S03cron  →  S04apache2  →  ...

# systemd — parallel, dependency-aware
networking.service  ─┐
syslog.service      ─┼─→  (each starts as soon as ITS OWN dependencies are ready,
cron.service        ─┘      not waiting on unrelated services)
apache2.service     ──depends on networking.service specifically
```

`apache2` genuinely does need networking to be up first — that's a real dependency, expressed explicitly. `cron` has no such requirement, and under systemd it can start the moment its own dependencies are satisfied, without waiting for `apache2` or anything else unrelated to finish first.

	STARTUP MODEL	DEPENDENCIES
System V init	Strictly sequential, by script number	Implied by ordering convention only
systemd	Parallel wherever possible	Explicit, checkable relationships

CONFIRM PID 1 DIRECTLY ON ANY SYSTEM

`ps -p 1 -o comm=` reports exactly what process is running as PID 1 right now — `systemd` on any modern distro, a quick, concrete way to confirm this chapter's own opening claim rather than taking it on faith.

SYSTEMD IS GENUINELY BIGGER — AND MORE DEBATED — THAN PURE PROCESS SUPERVISION

An honest note: systemd does considerably more than start and stop services — it also handles logging (`journald`, `systemd1-6`), and on many systems, device management and other lower-level responsibilities too. This scope has been a real, ongoing point of debate within parts of the Linux community. This course focuses on the practical, universally-relevant core — services, timers, journald — not the debate itself.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why PID 1 needing to "reap" orphaned processes is a genuinely special responsibility that no other process on the system has.

View solution

EXERCISE 2

Using this chapter's own apache2/networking/cron example, explain specifically why System V init would have delayed cron unnecessarily, while systemd wouldn't.

View solution

EXERCISE 3

Explain what ws1's own "systemctl restart apache2" command was actually asking systemd to do, in terms of this chapter's own PID-1/process-supervision material — even before systemd1-2 covers systemctl's own full syntax.

View solution

CHAPTER 1 QUICK REFERENCE

- **PID 1** is the one process the kernel starts directly; it also reaps orphaned processes
- On modern Linux (including Debian/ws1's own target), PID 1 is **systemd**
- **System V init** — strictly sequential, numbered scripts, dependencies only implied by ordering convention
- **systemd** — parallel startup wherever possible, dependencies expressed explicitly (full depth in systemd1-4)
- This course exists to explain what ws1's and ansible1-10's own unexplained systemctl/service calls were actually doing
- `ps -p 1 -o comm=` — confirm PID 1's identity directly on any system
- systemd's real scope (journald, device management, etc.) is broader than pure process supervision — this course covers the practical core, not the surrounding debate

CHAPTER 2 OF 11

Unit Files — The Basic Building Block

systemd in Depth

Chapter 2 · Unit Files — The Basic Building Block

`systemd1-1` established what systemd is and why it exists. This chapter gets concrete — the actual files systemd reads, and the real `systemctl` commands behind everything `ws1` and `ansible1-4` already used, unexplained.

What a Unit Is

A **unit** is systemd's own generic term for anything it manages — not just services, but sockets, timers, mounts, devices, and targets too (`systemd1-3` covers `.service` in full depth, `systemd1-7` covers `.timer`, `systemd1-8` covers `.socket`). Every unit is described by a plain-text configuration file — its **unit file**.

Where Unit Files Actually Live — Location Precedence

- **/usr/lib/systemd/system/** (or `/lib/systemd/system/`) — units shipped by installed packages, the vendor default
- **/etc/systemd/system/** — local admin overrides and custom units, taking priority over the vendor copy
- **/run/systemd/system/** — runtime-only, volatile, never survives a reboot; used by systemd itself and some tools for temporary units

If the same unit name exists in more than one location, `/etc` wins over `/usr/lib` — a real, useful override mechanism. Copying a vendor unit into `/etc/systemd/system/` and editing it there lets you customize a service without touching the original package-provided file, which a future package update would otherwise silently overwrite.

Basic Unit File Syntax — A First Look

```
[Unit]
Description=OpenSSH server daemon

[Service]
ExecStart=/usr/sbin/sshd -D

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

Just enough to recognize the shape — `systemd1-3` does the real deep dive. `[Unit]` holds metadata (`Description=`); `[Service]` holds the actual execution details (`ExecStart=`, the command actually run); `[Install]` controls what happens when the unit is enabled — `WantedBy=multi-user.target` is exactly why this note matters back in `systemd1-1`'s own boot-target material.

systemctl — The Real Command, In Depth

- **systemctl start/stop/restart NAME** — an immediate, one-time action; does not by itself survive a reboot
- **systemctl status NAME** — current state, recent log lines, whether it's enabled
- **systemctl enable/disable NAME** — a genuinely different axis: controls whether the unit starts automatically at the next boot, by creating or removing a symlink tied directly to the unit's own `[Install]` / `WantedBy=` line — completely independent of whether it's running *right now*

The real, common beginner confusion: enabling a service does **not** start it immediately, and starting a service does **not** enable it. These are two separate actions, frequently needed together.

Revealing What ansible1-4's Own service Module Wraps

`ansible1-4`'s own `service: name: nginx state: started` is, underneath, literally invoking `systemctl start nginx`. Ansible's own idempotency check — is it already running? — corresponds directly to `systemctl status` being checked first, with `start` only actually called if genuinely needed. `state: started` maps to `systemctl start`; the module's own `enabled: true` option maps directly to `systemctl enable`. This is precisely what `systemd1-1` promised, and what `ansible1-1` / `ansible1-4` left unexplained.

	CONTROLS	SURVIVES A REBOOT ON ITS OWN?
start / stop / restart	Whether the unit is running right now	No
enable / disable	Whether the unit starts automatically at boot	Yes — that's the whole point

SYSTEMCTL ENABLE --NOW COMBINES BOTH ACTIONS

A single command that both enables (boots automatically going forward) and starts (running immediately) a unit — worth knowing, since "I enabled it but it's not running" or the reverse is a genuinely frequent point of confusion this shortcut sidesteps entirely.

NEVER HAND-EDIT A UNIT FILE IN /USR/LIB/SYSTEMD/SYSTEM/

A real mistake, in the same spirit as `grub1-2`'s own warning against hand-editing `grub.cfg` — a package update can silently overwrite it. The correct place for a customization is `/etc/systemd/system/`, either as a full override copy, or, more surgically, via a drop-in override file — a lighter-weight option covered later in this course.

Hands-On Exercises

EXERCISE 1

Explain the difference between running "systemctl start nginx" and "systemctl enable nginx" on a system where nginx is currently stopped and disabled, including what happens on the next reboot in each case.

View solution

EXERCISE 2

A unit named webapp.service exists in both /usr/lib/systemd/system/ and /etc/systemd/system/, with different ExecStart= values. Explain which one systemd actually uses, and why.

View solution

EXERCISE 3

Explain exactly what Ansible's service module is doing internally when it runs "service: name: nginx state: started enabled: true", in terms of the two systemctl actions this chapter describes.

View solution

CHAPTER 2 QUICK REFERENCE

- A **unit** is anything systemd manages, described by a plain-text unit file (.service, .timer, .socket, ...)
- Location precedence: **/etc/systemd/system/** overrides **/usr/lib/systemd/system/**; **/run/systemd/system/** is volatile
- Basic unit shape: **[Unit]** (metadata), **[Service]** (execution), **[Install]** (enable behavior)
- **start/stop/restart** — right now, doesn't survive reboot; **enable/disable** — automatic at boot, independent of current state

- `systemctl enable --now NAME` — does both at once
- ansible1-4's own service module is a thin wrapper: state: started → systemctl start; enabled: true → systemctl enable
- Never hand-edit a unit file in /usr/lib/systemd/system/ — customize via /etc/systemd/system/ instead

CHAPTER 3 OF 11

Writing a Custom Service Unit

systemd in Depth

Chapter 3 · Writing a Custom Service Unit

`systemd1-2` showed the shape of a unit file without explaining any field in depth. This chapter is that depth — and the real service written here is the same one `systemd1-11`'s own capstone actually deploys.

[Unit] Section In Depth

`Description=` is shown directly in `systemctl status` output — worth writing something genuinely useful, not a placeholder. `Documentation=` is optional, a URL or man page reference. `After=` / `Requires=` are previewed here; `systemd1-4` covers them in full.

[Service] Section — Type=

The field that determines how systemd knows whether the service actually started successfully:

- **Type=simple** (the default) — the process started by `ExecStart=` *is* the main process; systemd considers the service started the moment it forks
- **Type=forking** — for older-style daemons that fork into the background and exit the original process; systemd needs `PIDFile=` to know which PID to actually track
- **Type=oneshot** — a command that runs once and exits, not a long-running daemon at all; commonly paired with `RemainAfterExit=yes` so `systemctl status` still reports "active" after it finishes — directly relevant to `systemd1-7`'s own timer-paired service units
- **Type=notify** — the service itself signals "I'm actually ready now" back to systemd; the most precise option, but requires the application to support it

ExecStart=, ExecStop=, ExecReload=

`ExecStart=` is required — the actual command that *is* the service. It must use an **absolute path**, a genuinely common beginner mistake: no shell `PATH` lookup happens here the way it would in an interactive terminal. `ExecStop=` is optional — for `Type=simple`, systemd can usually just send `SIGTERM` to stop the process cleanly; only write one when a service needs a genuinely custom shutdown command. `ExecReload=` runs on

`systemctl reload NAME` — a real, common third action beyond just start/stop, typically telling a running process to re-read its config without a full restart.

Restart= — Automatic Recovery

```
Restart=on-failure
RestartSec=5
```

`Restart=on-failure` (or `always`, or the default `no`) tells systemd to automatically restart the service if it exits unexpectedly. `RestartSec=` adds a delay before the restart attempt, avoiding a tight crash-loop hammering the system. A genuinely valuable capability with no clean System V init equivalent — one more concrete payoff of `systemd1-1`'s own "why systemd replaced init" material.

[Install] Section — WantedBy=

`WantedBy=multi-user.target` is what `systemctl enable` actually acts on — the line determining which target's own `.wants/` directory receives the new symlink. Different `WantedBy=` values matter for different kinds of services, previewed here and covered fully in `systemd1-5`'s own targets chapter.

A Full Worked Example — Toward the Capstone

```
[Unit]
Description=My Application Service
After=network.target

[Service]
Type=simple
ExecStart=/usr/local/bin/myapp
Restart=on-failure
RestartSec=5
User=myapp

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

Every field here is explained above — this is the actual shape `systemd1-11`'s own capstone deploys for real. `User=myapp` is worth a light callback of its own: running a service as a dedicated, non-root user rather than root by default is a genuinely security-relevant habit, the same least-privilege instinct `owasp1` / `bc1` already established elsewhere on this site.

Applying a New or Changed Unit File

```
sudo systemctl daemon-reload
sudo systemctl enable --now myapp
```

`systemctl daemon-reload` is required after creating or editing any unit file — it tells systemd to re-read unit files from disk. Forgetting it is a real, common "why isn't my change taking effect" gotcha.

TYPE=	"STARTED" MEANS
simple	The ExecStart= process forked — immediate
forking	The tracked PID (via PIDFile=) is running, after the original process exits
oneshot	The command ran and exited; RemainAfterExit=yes keeps it reported as active
notify	The application itself explicitly signaled readiness

VERIFY A UNIT FILE BEFORE USING IT

`systemd-analyze verify myapp.service` checks a unit file for syntax errors before you actually try to start it — worth running on any hand-written unit, before `daemon-reload` and `start`.

FORGETTING DAEMON-RELOAD IS A REAL, COMMON MISTAKE

systemd keeps using its own cached, in-memory copy of unit files until explicitly told to reload — an edit to a unit file on disk can appear to have no effect at all if `systemctl daemon-reload` is skipped, in the same shape (a different subsystem, same underlying gotcha) as `grub1-3`'s own "forgot update-grub" mistake.

Hands-On Exercises

EXERCISE 1

Write a complete unit file for a service named "backup-agent" that runs /opt/backup/agent, restarts automatically on failure after a 10-second delay, and starts after the network is up.

View solution

EXERCISE 2

Explain why a one-time database migration script should use Type=oneshot with RemainAfterExit=yes rather than Type=simple, referencing what "started" actually means for each.

View solution

EXERCISE 3

Someone edits /etc/systemd/system/myapp.service to change its Restart= setting, then runs systemctl restart myapp, but the old behavior persists. Explain the most likely cause and the fix.

View solution

CHAPTER 3 QUICK REFERENCE

- **Type=simple** (default, immediate), **forking** (needs PIDFile=), **oneshot** (+ RemainAfterExit=yes), **notify** (app signals readiness)
- **ExecStart=** requires an absolute path — no shell PATH lookup; **ExecStop=** optional; **ExecReload=** for config reloads without a full restart
- **Restart=on-failure** + **RestartSec=** — automatic recovery, no System V init equivalent
- **WantedBy=** in [Install] determines which target's .wants/ gets the enable symlink
- **User=** runs a service as a dedicated non-root user — a real least-privilege habit
- `systemctl daemon-reload` is required after any unit file change — forgetting it is a real, common gotcha
- `systemd-analyze verify` catches syntax errors before you try to use a hand-written unit

CHAPTER 4 OF 11

Dependencies & Ordering

systemd in Depth

Chapter 4 · Dependencies & Ordering

`systemd1-3` used `After=network.target` without fully explaining it. This chapter is the real depth — and the single most common point of confusion anyone new to systemd runs into.

Two Genuinely Separate Axes — What vs. When

The core confusion this chapter exists to resolve: "dependency" (should this unit even be started) and "ordering" (when, relative to another unit, should it start) are **two separate, independently configured axes** in systemd — not one combined concept, however natural that assumption feels at first. `Wants=` / `Requires=` control **whether** another unit gets started at all, as a side effect of this one starting. `After=` / `Before=` control **only the order**, with zero effect on whether anything actually gets started.

Wants= vs. Requires= — How Strict Is the Dependency?

`Wants=` is a **soft** dependency — if the wanted unit fails to start, the wanting unit still proceeds normally. The more common, more resilient choice for most real-world dependencies. `Requires=` is a **hard** dependency — if the required unit fails, systemd treats this unit's own startup as a failure too, and can stop it if the required unit later stops. Genuinely stricter, appropriate only when a unit truly cannot function at all without the other one.

A concrete example: a web server might `Wants=` a caching service — nice to have, but the site can still serve requests, just more slowly, if the cache is down — while it `Requires=` its own database, since it genuinely cannot function at all without one.

After= vs. Before= — Ordering Only, No Dependency Implied

This directly resolves `systemd1-3`'s own `After=network.target`: it only says "if `network.target` is going to start anyway, for whatever reason, start after it." It does **not**, by itself, cause `network.target` to be started as a side effect. A unit with only `After=` and no `Wants=` / `Requires=` for the same target could, in principle, start with no network at all, if nothing else on the system independently wanted networking. This is exactly why real unit files commonly pair both together.

A Concrete Worked Example

```
# Correct — actually guarantees network-online.target starts, in the right order
[Unit]
Wants=network-online.target
After=network-online.target

# A real, subtle bug — no guarantee network-online.target ever starts at all
[Unit]
After=network-online.target
```

In the second, broken version, if nothing else on the system happens to `Want=` `network-online.target` independently, this unit could start immediately — network or no network — with no error, no warning, just silently wrong behavior on a system where networking happens to come up slowly or not at all.

Target Units as Synchronization Points

`systemd1-1` previewed target units replacing runlevels; `systemd1-5` covers them in full. A target unit doesn't "do" anything itself — it's purely a named synchronization point, a milestone other units can `Want=`/`Require=`/order themselves around. This lets many otherwise-unrelated units all agree on "wait until networking is genuinely usable" without each one needing its own bespoke detection logic.

Viewing the Real Dependency Tree

```
systemctl list-dependencies myapp
```

Shows the actual, resolved dependency tree for a real unit on a real system — a genuinely concrete way to see this chapter's abstract material in practice, rather than only reasoning about it on paper.

	CONTROLS	AFFECTS WHETHER THE OTHER UNIT STARTS?
Wants= / Requires=	Whether the other unit gets started at all	Yes — that's the entire point
After= / Before=	Ordering only, relative to another unit	No — purely sequencing

SEE WHAT DEPENDS ON A UNIT, NOT JUST WHAT IT DEPENDS ON

`systemctl list-dependencies --reverse NAME` shows the opposite direction — everything that depends *on* this unit. Genuinely useful for answering "what would actually break if I disabled this," before doing it.

THE MISTAKE THIS WHOLE CHAPTER EXISTS TO PREVENT

Writing only `After=` when a unit genuinely needs both ordering and an actual guarantee the other unit starts — or, less commonly, writing only `Wants=` / `Requires=` and assuming the "right" start order comes for free. Neither assumption holds; the two axes are independent, and most real dependencies need both lines, deliberately.

Hands-On Exercises

EXERCISE 1

A logging service should only run if a remote log-collector service is available, and must not start until after it. Write the two [Unit] lines needed to correctly express this.

View solution

EXERCISE 2

Explain the real difference between using Wants= and Requires= for a service's own database dependency, including what happens to the service in each case if the database fails to start.

View solution

EXERCISE 3

A unit has only After=postgresql.service in its [Unit] section, with no Wants= or Requires=. Explain a real scenario in which this unit could start successfully even though PostgreSQL never started at all.

View solution

CHAPTER 4 QUICK REFERENCE

- **Wants=/Requires=** control whether another unit starts; **After=/Before=** control only ordering — two genuinely separate axes
- **Wants=** — soft dependency, proceeds even if the wanted unit fails; **Requires=** — hard dependency, failure propagates
- After= alone never guarantees the other unit is started — pairing Wants=/Requires= with After= is the real, common pattern
- Target units are pure synchronization points — they don't "do" anything, just let unrelated units agree on a shared milestone
- `systemctl list-dependencies NAME` / `--reverse NAME` — see the real, resolved dependency tree in both directions

- The classic mistake: only After=, expecting it to also guarantee the other unit starts — it doesn't

CHAPTER 5 OF 11

Targets — systemd's Answer to Runlevels

systemd in Depth

Chapter 5 · Targets — systemd's Answer to Runlevels

`systemd1-4` defined a target as a synchronization point without going further. This chapter goes further — and resolves `grub1-4`'s own unexplained `systemd.unit=rescue.target` kernel parameter along the way.

What a Target Actually Is, Formally

`systemd1-4` established that a target doesn't "do" anything itself. Formally: a target unit is a named grouping of other units — often implemented as essentially an empty unit that other units `Want=` / `Require=` /order themselves relative to. "Reaching a target" means every unit that target itself requires or wants, directly or transitively, has actually been started.

The Common Targets, Mapped to Legacy Runlevels

SYSTEMD TARGET	LEGACY RUNLEVEL	MEANING
poweroff.target	0	Shut down
rescue.target	1	Single-user/minimal recovery mode
multi-user.target	3	Full multi-user, no GUI
graphical.target	5	Multi-user + display manager/GUI
reboot.target	6	Reboot

`graphical.target` itself `Wants=multi-user.target`, since graphical mode is a strict superset — everything multi-user mode provides, plus a display manager on top.

The Default Target

```
systemctl get-default
systemctl set-default multi-user.target
```

Which target boots by default is determined by a symlink — `/etc/systemd/system/default.target` pointing at the chosen target — genuinely similar in spirit to `systemd1-3`'s own `WantedBy=` symlink mechanism, just applied to the top-level boot target rather than an individual service. A headless server typically defaults to `multi-user.target`; a desktop distro typically defaults to `graphical.target`.

systemctl isolate — Switching Targets Live

```
systemctl isolate multi-user.target
```

Run on a live graphical system, this stops everything **not** required by `multi-user.target` — the display manager included — and starts anything `multi-user.target` requires that wasn't already running. A genuine live runlevel-style switch, no reboot required. A real, practical use: temporarily dropping out of a graphical environment to troubleshoot a display/GPU driver issue, then isolating back to `graphical.target` once fixed.

Direct Legacy Comparison

`systemd1-1`'s own System V material named `telinit 3` as the legacy way to switch runlevels live. That maps directly to `systemctl isolate multi-user.target` — the same real-world action, but a genuinely different, more expressive mechanism underneath: an actual dependency-resolution operation, not a legacy numbered switch with no explicit relationship to what it starts or stops.

Resolving grub1-4's Own rescue.target

`grub1-4` taught appending `systemd.unit=rescue.target` at the GRUB menu to reach a recovery shell, without explaining what that parameter actually meant. Now it's clear: this tells systemd to boot straight to *this* target instead of the normal default, bypassing `graphical.target`/`multi-user.target`'s own much larger set of `Wants=`/`Requires=` dependencies entirely, landing in a genuinely minimal environment with far fewer units started — exactly why it's usable for recovery even when something elsewhere in the normal boot chain is broken.

SEE EVERY ACTIVE TARGET RIGHT NOW

`systemctl list-units --type=target` lists every target currently active on the system — a concrete way to see this chapter's own material reflected in real, live system state rather than only reasoning about it abstractly.

SYSTEMCTL ISOLATE IS A REAL, IMMEDIATELY DISRUPTIVE ACTION

Isolating to `rescue.target` on a live production system stops nearly everything non-essential immediately — network services included, in many configurations. Genuinely disruptive to anyone actively using the system; not something to run casually just to "see what happens."

Hands-On Exercises

EXERCISE 1

Write the command that sets a headless server's default boot target to multi-user.target, and explain why a headless server should never default to graphical.target.

View solution

EXERCISE 2

Explain what happens to a running display manager when an administrator runs systemctl isolate multi-user.target on a desktop system currently in graphical.target, and why.

View solution

EXERCISE 3

Explain, in terms of this chapter's own material, why booting with systemd.unit=rescue.target (grub1-4's own kernel parameter) can succeed even when something is broken elsewhere in the normal boot chain that would prevent reaching graphical.target or multi-user.target.

View solution

CHAPTER 5 QUICK REFERENCE

- A target is a named synchronization point — "reached" once everything it requires/wants has started
- **poweroff.target** (0), **rescue.target** (1), **multi-user.target** (3), **graphical.target** (5), **reboot.target** (6) — direct legacy runlevel mapping
- graphical.target Wants= multi-user.target — a strict superset, GUI on top of full multi-user mode
- `systemctl get-default` / `set-default` — controlled by a symlink, the same mechanism as an individual unit's own WantedBy=
- `systemctl isolate TARGET` — a live runlevel-style switch, the modern equivalent of `telinit N`
- grub1-4's own systemd.unit=rescue.target works precisely because it skips graphical/multi-user's own much larger dependency set

- isolate is immediately disruptive on a live system — never run casually

CHAPTER 6 OF 11

journald & Reading Logs

systemd in Depth

Chapter 6 · journald & Reading Logs

`systemd1-1`'s own warn-box named journald as part of systemd's broader scope beyond process supervision. This chapter is that promised depth — and it's also the single-host counterpart to `obs1-7`'s own centralized logging material.

What journald Actually Is

systemd's own built-in logging daemon. It automatically captures stdout/stderr from every unit systemd manages, plus kernel messages, plus anything using the standard `syslog()` call — with no separate logging setup required for a systemd-managed service. Storage is structured and indexed, not plain text like a traditional `/var/log/*.log` file — queryable directly by unit, priority, or time range, without reaching for `grep`/`awk` gymnastics.

journalctl — The Basic Query Tool

```
journalctl                # the whole journal, oldest first
journalctl -u myapp       # logs for one specific unit only
journalctl -f             # follow mode — live-tail new entries
journalctl -b             # logs since the current boot only
journalctl -b -1          # the PREVIOUS boot's logs
```

`-u myapp` ties directly to `systemd1-3`'s own worked example. `-f` is genuinely comparable to `tail -f` on a traditional log file. `-b -1` is a real, practical tool for diagnosing a crash or unexpected reboot after the fact — the previous boot's own logs, isolated from everything since.

Persistent vs. Volatile Logging

By default on many systems, the journal lives in `/run/log/journal/` — RAM-backed, volatile, wiped completely on reboot. This has a real, practical consequence: `journalctl -b -1` only works at all if persistent storage is actually configured; without it, the previous boot's logs are simply gone.

```
sudo mkdir -p /var/log/journal
sudo systemd-tmpfiles --create --prefix /var/log/journal
```

Creating `/var/log/journal/` with correct ownership is the actual, real-world trigger that switches journald into persistent mode automatically — logs now survive reboots. This matters directly for the same reason `ws1`'s own security material cared about logging at all: a system with only volatile logs loses everything the moment it's rebooted, a real, practical gap for any kind of incident investigation after the fact.

Priority Filtering

```
journalctl -p err
```

The classic 8-level syslog priority scale — `emerg`, `alert`, `crit`, `err`, `warning`, `notice`, `info`, `debug`, ordered from most to least severe. `-p err` shows entries at error level **or more severe** — the scale is ordered, so this filters "at least this severe," not "exactly this level." A real, practical way to find genuinely serious issues in a noisy, chatty log without reading every routine info-level line.

The Single-Host Counterpart to obs1-7's Own Centralized Logging

`obs1-7` covered structured logging and centralized aggregation (Loki) across many hosts and services, at real scale. journald solves a genuinely related but narrower problem — one host's own logs, structured and queryable, with no separate aggregation system required at all. The same underlying idea — structured, queryable logs beat unstructured, grep-able text files — applies at both scales, just with different tooling for different scopes. Real production setups often use **both**: journald locally on each host, with an agent (the same DaemonSet-based approach `obs1-10` described) shipping journal entries onward to centralized storage. journald isn't replaced by Loki — it's frequently the local *source* a Loki-equivalent agent reads from in the first place.

	FORMAT	QUERYABLE BY FIELD?	PERSISTENT BY DEFAULT?
Traditional /var/log/*.log	Plain text	No — grep/awk required	Yes
journald	Structured, indexed, binary	Yes — by unit, priority, time, ...	No — must be enabled

A REAL, TARGETED INCIDENT-RESPONSE QUERY

`journalctl -u myapp -p err -b` combines everything in this chapter — one specific unit, error-severity-or-worse only, current boot only. Exactly the kind of narrow, deliberate query you'd actually run mid-incident, rather than scrolling through the entire journal by hand.

THE JOURNAL CAN GROW LARGE — REAL MAINTENANCE IS NEEDED

Especially with verbose, chatty services, journal storage can grow substantially over time. `journalctl --disk-usage` shows current size; `journalctl --vacuum-size=500M` (or `--vacuum-time=`) actually reclaims space. A real, practical maintenance task worth knowing about before disk space becomes a genuine problem, not after.

Hands-On Exercises

EXERCISE 1

Write the journalctl command to view only error-or-worse log entries for a unit named "backup-agent" from the current boot, matching this chapter's own combined example pattern.

View solution

EXERCISE 2

A server unexpectedly rebooted overnight. An administrator runs journalctl -b -1 the next morning to investigate, but gets no results at all. Explain the most likely cause.

View solution

EXERCISE 3

Explain, using this chapter's own material, why a real production setup might reasonably run both journald locally and ship logs to a centralized system like obs1-7's own Loki, rather than choosing one or the other.

View solution

CHAPTER 6 QUICK REFERENCE

- journald automatically captures unit stdout/stderr, kernel messages, and syslog() calls — structured, indexed, no separate setup needed
- **-u UNIT**, **-f** (follow), **-b** (current boot), **-b -1** (previous boot) — the core journalctl flags
- Volatile by default (/run/log/journal/, wiped on reboot); creating /var/log/journal/ enables persistent storage
- **-p LEVEL** filters to that severity or worse, using the 8-level syslog scale
- journald is the single-host counterpart to obs1-7's own centralized aggregation — often the local source a Loki-equivalent agent reads from

- `--disk-usage` / `--vacuum-size=` / `--vacuum-time=` — real, necessary journal maintenance

CHAPTER 7 OF 11

Timers — systemd's Replacement for Cron

systemd in Depth

Chapter 7 · Timers — systemd's Replacement for Cron

`systemd1-3`'s own `Type=oneshot` material previewed exactly this chapter. Timers are systemd's own scheduled-task mechanism — genuinely different from cron, not just a reskinned version of it.

The Two-Unit Pattern — .timer + .service

A systemd timer is genuinely **two** separate unit files working together: a `.timer` unit defining *when* to run, and a `.service` unit defining *what* to run — by convention, sharing the same base name (`backup.timer` + `backup.service`). A deliberate design decision distinct from cron: the "what" is a real, independently-runnable systemd service — exactly `systemd1-3`'s own `Type=oneshot` material — cleanly separated from the "when." You can run and test the service directly at any time with `systemctl start backup.service`, with no need to wait for or fake the timer at all.

OnCalendar= — Scheduling Syntax

```
OnCalendar=daily
OnCalendar=*-*-* 03:00:00
OnCalendar=Mon,Wed,Fri 09:00
OnCalendar=weekly
```

The general shape: `DayOfWeek Year-Month-Day Hour:Minute:Second`, with wildcards (`*`) and shorthand keywords (`daily` / `weekly` / `monthly` / `hourly`) covering common cases without needing the full explicit syntax.

```
systemd-analyze calendar "Mon,Wed,Fri 09:00"
```

Verifies a calendar expression means what you think it means **before** deploying it, printing the next several actual trigger times — genuinely worth running on any expression before trusting it.

A Full Worked Timer Example

```
# backup.service
[Unit]
Description=Run backup script

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

```
# backup.timer
[Unit]
Description=Run backup.service daily at 3am

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

[Install]
WantedBy=timers.target
```

```
systemctl enable --now backup.timer
```

Notice: you enable and start the **timer**, not the service directly — the timer itself then triggers the service at the scheduled time. `WantedBy=timers.target` is `systemd1-5`'s own target material made concrete for a specific new case — a dedicated target that exists purely to group active timers.

Persistent=true — Catching Up Missed Runs

The real, standout advantage over cron. `Persistent=true` tells systemd to check, at boot, whether this timer's scheduled trigger was **missed** while the system was off — a laptop shut down overnight through the scheduled 3am backup, for instance — and if so, run it once, immediately, upon boot, rather than silently skipping that day entirely. Traditional cron has no native equivalent: a missed cron job, with the system off at the scheduled time, is simply missed, silently, until the next scheduled occurrence.

Direct Comparison Against Traditional Cron

Real, concrete advantages: dependency-awareness (a timer's own service can `Wants=`/`After=` other units — waiting for a network mount before running a backup, genuinely impossible to express cleanly in a crontab); automatic journald logging (`systemd1-6`'s own material — every timer-triggered run is automatically logged and queryable via `journalctl -u backup.service`, unlike cron's own historical reliance on separate mail-based or ad-hoc logging); `Persistent=true`'s own missed-run catch-up; and direct testability, per this

chapter's own opening point. An honest note: cron's own crontab syntax is genuinely more compact for the simplest cases, and cron remains extremely widely deployed and understood — this isn't a "cron is bad" chapter, just an honest accounting of what systemd timers genuinely add.

	MISSED RUN (SYSTEM OFF)	DEPENDENCY-AWARE?	LOGGING
cron	Silently skipped	No	Separate, ad-hoc
systemd timer	Caught up on boot with Persistent=true	Yes — via the paired service's own Wants=/After=	Automatic, via journald

SEE EVERY SCHEDULED TIMER AND ITS NEXT RUN

`systemctl list-timers` shows every active timer on the system, along with its next scheduled trigger time — a genuinely useful, quick "what's actually scheduled on this box" overview.

FORGETTING WANTEDBY=TIMERS.TARGET BREAKS PROPER ENABLING

Without it in the `.timer` unit's own `[Install]` section, `systemctl enable` has nothing to actually attach the timer to. It may appear to work when started manually, but won't behave the way a genuinely enabled unit should — always include it.

Hands-On Exercises

EXERCISE 1

Write a complete .timer unit (matching this chapter's own backup.timer structure) that runs a paired service every Sunday at 02:30, with missed-run catch-up enabled.

View solution

EXERCISE 2

Explain why the two-unit .timer/.service pattern makes testing a scheduled task genuinely easier than testing a traditional cron job, referencing this chapter's own opening point directly.

View solution

EXERCISE 3

A laptop is shut down at the exact moment its nightly backup.timer would have triggered, and stays off until the next afternoon. Explain what happens when it's finally turned back on, assuming Persistent=true is set.

📄 View solution

CHAPTER 7 QUICK REFERENCE

- A systemd timer is two paired units: **.timer** (when) + **.service** (what, matching systemd1-3's own Type=oneshot)
- **OnCalendar=** — daily/weekly/monthly/hourly shorthand, or explicit DayOfWeek Year-Month-Day Hour:Min:Sec syntax
- `systemd-analyze calendar "expr"` — verify a schedule before deploying it
- Enable/start the **timer**, not the service; **WantedBy=timers.target** is required for proper enabling
- **Persistent=true** — catches up a missed run on boot, no cron equivalent
- Real advantages over cron: dependency-awareness, automatic journald logging, missed-run catch-up, direct testability
- `systemctl list-timers` — every active timer and its next scheduled run

CHAPTER 8 OF 11

Socket Activation

systemd in Depth

Chapter 8 · Socket Activation

`systemd1-7` covered time-based activation. This chapter covers a genuinely different trigger — on-demand, connection-based activation, with no real precedent in traditional init or cron.

The Problem — Services That Are Rarely Used, But Must Always Be Ready

Some services are needed only occasionally — an SSH connection arrives once in a while, a rarely-used network service gets a request once a day. Traditionally, "must always be ready to respond" meant "must always be running," consuming memory and resources continuously, even through long idle stretches with zero actual traffic.

What Socket Activation Actually Does

systemd itself opens and listens on the network (or Unix domain) socket on behalf of a service, **before** that service is even running. Any connection attempt is queued by the kernel/systemd; **only then** does systemd actually start the real service, handing it the already-open socket. From the client's own point of view this is completely invisible — the connection appears to succeed immediately either way, whether the service was already running or had to be started on-demand just now.

A .socket Unit, Paired With a .service Unit

The same two-unit pattern as `systemd1-7`'s own `.timer` / `.service` pairing — here, `.socket` + `.service`, sharing a base name.

```
# myapp.socket
[Unit]
Description=Socket for myapp

[Socket]
ListenStream=8080

[Install]
WantedBy=sockets.target
```

`myapp.service` itself needs no port-binding logic of its own — systemd hands the already-listening socket to the service when it starts it, via a specific, well-known file descriptor (conventionally file descriptor 3). `WantedBy=sockets.target` is `systemd1-5`'s own targets material, one more dedicated target — directly parallel to `systemd1-7`'s own `timers.target`.

Enabling and Testing Socket Activation

```
systemctl enable --now myapp.socket
systemctl status myapp.socket    # listening
systemctl status myapp.service   # inactive (dead) — until a real connection arrives
```

Again, enable and start the **socket**, not the service directly — the same pattern `systemd1-7` already established for timers. The before/after state is genuinely visible: the socket shows active/listening immediately, while the service itself stays inactive until the first real connection triggers it.

Real Efficiency Benefits

Real memory and resource savings for genuinely rarely-used services — no process running at all during idle periods. A real, historical example worth naming: SSH itself was traditionally socket-activatable this way on some systems (`sshd.socket` alongside `sshd.service`) — though many modern distros run `sshd` as an always-on service instead, for latency reasons on a frequently-used service. Socket activation genuinely shines for rarely-used services specifically, not universally. Faster boot, too: a socket-activated service doesn't need to actually finish starting up during the boot sequence itself — only the socket (cheap to open) is needed at boot; the real service startup is deferred until first actually used.

An Honest Limitation

Not every service is a good fit. A service with genuine startup latency — loading a large dataset, warming a cache — means the **first** connection after any idle period genuinely waits for that startup to finish, a real,

noticeable delay an always-running service wouldn't have. Socket activation trades "always consuming resources" for "occasionally slower first response" — a genuine tradeoff, not a pure, unconditional win.

	RESOURCE USE WHILE IDLE	FIRST-REQUEST LATENCY
Always-running service	Continuous, whether used or not	None — already warm
Socket-activated service	None while idle — only the socket itself	Real startup delay on the first connection after idle time

SEE EVERY SOCKET-ACTIVATED UNIT AT A GLANCE

`systemctl list-sockets` shows every socket unit and what it's listening on — the socket-activation equivalent of `systemd1-7`'s own `systemctl list-timers`.

NEVER ALSO ENABLE THE SERVICE DIRECTLY WITH WANTEDBY=MULTI-USER.TARGET

A socket-activated service's own unit file generally should **not** also carry `WantedBy=multi-user.target` in its own `[Install]` section — that would start it unconditionally at boot anyway, defeating the entire point of on-demand activation. The socket unit's own `WantedBy=sockets.target` is what gets enabled — not the service.

Hands-On Exercises

EXERCISE 1

Write a complete .socket unit (matching this chapter's own myapp.socket structure) for a service listening on port 9000, and the systemctl command needed to enable and start it.

View solution

EXERCISE 2

Explain why a client connecting to a socket-activated service that isn't currently running experiences no visible difference from connecting to one that's already running, referencing what systemd actually does with the connection.

View solution

EXERCISE 3

A team considers socket-activating a service that loads a 4GB dataset into memory on startup, taking about 30 seconds. Using this chapter's own honest limitation, explain why this is likely a poor fit for socket activation.

View solution

CHAPTER 8 QUICK REFERENCE

- Socket activation: systemd listens on a socket **before** the service runs, starting it only when a real connection arrives
- **.socket** + **.service** — the same two-unit pairing pattern as timers, sharing a base name
- The service receives the already-open socket via a well-known file descriptor — no port-binding logic of its own needed
- Enable/start the **socket**, not the service — WantedBy=sockets.target, parallel to timers.target
- Real benefits: no idle resource use, faster boot (only the cheap socket needs to be ready at boot)
- Real tradeoff: the first connection after idle time pays the service's own real startup cost
- Never also give the service its own WantedBy=multi-user.target — that defeats on-demand activation entirely

CHAPTER 9 OF 11

Resource Control with cgroups

systemd in Depth

Chapter 9 · Resource Control with cgroups

`perf1`'s own CPU and Memory Bottlenecks chapters covered diagnosing a resource problem system-wide. This chapter closes the loop — once you know which specific service is the actual cause, systemd's own per-service limits are the concrete tool to contain it.

What cgroups Actually Are

Control groups (cgroups) are a Linux kernel feature that groups processes together and lets the kernel enforce resource limits — CPU, memory, I/O — on the group as a whole, not just individual processes. A genuinely important, often-unknown fact: as PID 1 and the ultimate ancestor of essentially every process on the system, systemd automatically places **every** unit into its own cgroup, by default, whether or not any explicit limit is ever configured. Cgroup accounting already exists per-unit before you ever write a single `CPUQuota=` line.

Setting Limits — CPUQuota= and MemoryMax=

```
[Service]
CPUQuota=50%
MemoryMax=512M
```

`CPUQuota=50%` means this service can use at most 50% of a single CPU core's worth of processing time — even on a multi-core system, even with idle capacity available elsewhere — genuinely enforced by the kernel, not a polite request. `MemoryMax=` is a hard ceiling; exceeding it triggers the kernel's own OOM killer specifically against this cgroup, rather than the whole system hunting for an arbitrary victim process. `MemoryHigh=` is the softer companion — a throttling threshold below `MemoryMax=` that encourages memory reclaim before the hard kill ever becomes necessary.

Why This Matters — the Direct Tie to perf1

`perf1`'s own bottleneck chapters diagnosed resource problems at the whole-system level — a genuinely common real outcome of that kind of investigation is discovering one specific misbehaving service is the

actual cause. This chapter's own limits are the concrete containment tool once that diagnosis is made. Concrete scenario: a runaway process — a memory leak in a custom app — that would otherwise eventually exhaust all system memory, potentially taking down unrelated services too. `MemoryMax=` turns "the whole system's memory" into "just this cgroup's own memory," genuinely containing the blast radius to the one service actually responsible.

systemd-cgtop — Real-Time Per-Unit Resource Usage

```
systemd-cgtop
```

A live, `top`-style view — but organized by cgroup/unit rather than individual process, showing CPU/memory/IO usage per **service**, in real time. Immediately actionable alongside this chapter's own `CPUQuota=` / `MemoryMax=` settings — observe first, then set a limit informed by real usage.

Verifying a Limit Is Actually Enforced

```
systemctl show myapp -p MemoryMax
```

Confirms the configured value actually took effect. `systemd1-3`'s own `daemon-reload` gotcha applies directly here again — forgetting to reload and restart after adding a resource limit means the old, unrestricted cgroup settings are still what's actually enforced.

	BEHAVIOR WHEN EXCEEDED
MemoryHigh=	Soft — throttling and reclaim pressure encouraged, no kill
MemoryMax=	Hard — the cgroup-scoped OOM killer triggers

LIMITS CAN BE APPLIED INSTANTLY TO AN ALREADY-RUNNING UNIT

`systemctl set-property myapp MemoryMax=512M` applies a limit immediately, live, with no unit file edit or restart required — genuinely useful for urgent, live containment mid-incident, a direct parallel to `grub1-4`'s own live-edit-vs-permanent-config distinction.

SETTING LIMITS TOO AGGRESSIVELY CAN KILL A PERFECTLY HEALTHY SERVICE

A `MemoryMax=` set without first observing real usage via `systemd-cgtop` can OOM-kill a service during entirely legitimate, normal peak load — the same "measure before you optimize" discipline `perf1` itself would recognize, applied here to setting a hard resource ceiling rather than diagnosing an existing one.

Hands-On Exercises

EXERCISE 1

Write the [Service] section additions needed to limit a unit to 25% of a CPU core and a hard 1GB memory ceiling, and the command to confirm the memory limit actually took effect afterward.

View solution

EXERCISE 2

Explain the real difference in what happens to a service when it exceeds MemoryHigh= versus when it exceeds MemoryMax=.

View solution

EXERCISE 3

A team sets MemoryMax=256M on a service without ever checking its real memory usage first, and the service gets OOM-killed every day during its own normal peak traffic window. Explain what went wrong, and the correct process that should have been followed.

View solution

CHAPTER 9 QUICK REFERENCE

- Every systemd unit is placed into its own cgroup automatically, whether or not limits are ever configured
- **CPUQuota=** — a hard, kernel-enforced percentage-of-a-core cap
- **MemoryHigh=** (soft throttle) vs. **MemoryMax=** (hard, cgroup-scoped OOM kill)
- Per-service limits are the concrete containment tool once perf1's own system-wide bottleneck diagnosis points at one specific service
- `systemd-cgtop` — real-time, per-unit resource usage, observe before setting a limit
- `systemctl set-property` applies a limit instantly to an already-running unit, no restart required
- Never set a hard memory ceiling without first observing real usage — a real, common cause of killing a healthy service

CHAPTER 10 OF 11

systemd-analyze & Troubleshooting Boot

systemd in Depth

Chapter 10 · systemd-analyze & Troubleshooting Boot

`systemd1-1` claimed systemd's parallel, dependency-based startup is faster than System V init's sequential one. This chapter gives you the tools to actually see that claim in practice — and to diagnose it when something goes wrong.

systemd-analyze — The Basic Boot Time Report

```
systemd-analyze
```

A first-glance breakdown: kernel time, initrd time, and userspace time — the portion systemd itself is responsible for. A genuinely useful starting point before diving deeper into any specific tool below.

systemd-analyze blame — Which Units Took the Longest

```
systemd-analyze blame
```

A sorted list of every unit and how long it **individually** took to initialize, longest first. A real, important caveat, tying directly back to `systemd1-1`'s own parallel-startup material: a unit taking a long time doesn't necessarily delay the *overall* boot, if other unrelated units were starting in parallel at the same time. `blame` shows individual unit duration — not necessarily "this is why your boot is slow."

systemd-analyze critical-chain — The Actual Bottleneck Path

```
systemd-analyze critical-chain
```

This is the tool that actually answers "what delayed the overall boot": the **critical path** — the specific chain of units where each one genuinely had to wait for the one before it, via real dependencies (`systemd1-4`'s own material) — tracing back to the actual longest dependency chain, not just the longest individual unit. A unit

can show up near the top of `blame` (genuinely slow on its own) but never appear on the critical path at all, if it was running in parallel with other things and nothing else was actually waiting on it. This is exactly the resolution to `blame`'s own caveat above.

systemd-analyze plot — Visualizing the Whole Boot

```
systemd-analyze plot > boot.svg
```

Generates an actual visual, Gantt-style timeline of every unit's start time and duration — genuinely useful for spotting parallel vs. sequential patterns at a glance, rather than reading through a table of numbers.

Masking vs. Disabling — A Real, Important Distinction

`systemd1-2` covered `disable` — it removes the boot-time symlink, but a disabled unit can still be started manually, or pulled in as a dependency of something else that `Wants=`/`Requires=` it. `systemctl mask` is genuinely stronger: it creates a symlink to `/dev/null` in place of the unit, making it impossible to start **at all** — not manually, not as anyone else's dependency, nothing — until explicitly unmasked. The real use case: a unit that's actively causing boot problems and needs to be completely, unconditionally prevented from running while you investigate, not just "not started by default."

A Real Diagnostic Workflow — Slow Boot

1. `systemd-analyze` — confirm the boot is genuinely slow
2. `systemd-analyze critical-chain` — find the actual bottleneck chain, not just guess from `blame`
3. Identify the specific slow unit on that chain
4. `journalctl -u THATUNIT -b` (`systemd1-6`'s own tool) — investigate what it was actually doing
5. If it's not needed at boot at all, `systemctl disable` it; if it's actively causing harm, `systemctl mask` it while investigating further

A Real Diagnostic Workflow — Failed Boot / A Unit That Won't Start

```
systemctl --failed
```

Lists every unit currently in a failed state — a genuinely fast first step. Then `systemctl status THATUNIT` (`systemd1-2`'s own tool) for the specific error, then `journalctl -u THATUNIT -b` for the full detail behind it.

	SHOWS	PROVES THE BOOT WAS ACTUALLY DELAYED?
blame	Each unit's own individual initialization time	No — could be running in parallel with nothing waiting on it
critical-chain	The actual dependency chain that determined total boot time	Yes — this is the real bottleneck path

A "SLOW" OR "FAILED" UNIT IS SOMETIMES JUST BROKEN

`systemd-analyze verify` (`systemd1-3`'s own tool, revisited) is worth running on any unit suspected of being the culprit — a genuine syntax error can look, from the outside, like a mysterious slowdown or failure.

MASKING A UNIT OTHERS DEPEND ON CAN CAUSE A CASCADING FAILURE

Mask deliberately, understanding what depends on the unit first — `systemd1-4`'s own `systemctl list-dependencies --reverse` is directly relevant again here — not as a reflexive "make the error go away" action.

Hands-On Exercises

EXERCISE 1

A unit appears near the top of systemd-analyze blame's own output, but does not appear anywhere in systemd-analyze critical-chain's output. Explain what this combination actually tells you about that unit's real impact on boot time.

View solution

EXERCISE 2

Explain the real difference between systemctl disable myapp and systemctl mask myapp, including a scenario where disable alone would not actually prevent myapp from starting.

View solution

EXERCISE 3

Write out the full diagnostic workflow (matching this chapter's own numbered sequence) for investigating a boot that has become noticeably slower over the past week, from first command to final action.

View solution

CHAPTER 10 QUICK REFERENCE

- **systemd-analyze** — total boot time breakdown (kernel/initrd/userspace)
- **blame** — individual unit duration, not proof of overall boot impact
- **critical-chain** — the actual dependency chain that determined total boot time; the real diagnostic tool
- **plot** — a visual, Gantt-style timeline of the whole boot
- **disable** removes the boot-time symlink but allows manual/dependency starts; **mask** blocks starting entirely, unconditionally
- Slow boot workflow: analyze → critical-chain → journalctl on the culprit → disable or mask
- Failed boot workflow: `systemctl --failed` → status → journalctl
- Check what depends on a unit (list-dependencies --reverse) before masking it

CHAPTER 11 OF 11

Capstone: Building a Real Multi-Unit Service Stack

systemd in Depth

Chapter 11 · Capstone — Building a Real Multi-Unit Service Stack

The final chapter. A real small app — **notesapp** — deployed as a properly dependent, resource-limited service with a companion maintenance timer, logged and verified using every tool this course has built. This is where `ws1`'s and `ansible1-10`'s own unexplained `systemctl`/`service` calls finally get their full answer.

The Target — A Real Small App: notesapp

A small notes-taking web service: a long-lived process, needs the network genuinely up before starting, running on a modest, shared VPS where resource limits matter, and needing a periodic cleanup job to purge notes older than 90 days.

Step 1 — The Service Unit

```
# notesapp.service
[Unit]
Description=Notes App Service
After=network-online.target
Wants=network-online.target

[Service]
Type=simple
ExecStart=/usr/local/bin/notesapp
Restart=on-failure
RestartSec=5
User=notesapp

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

Note the paired `Wants=` **and** `After=` for `network-online.target` — `systemd1-4`'s own central lesson, applied for real: `After=` alone would never guarantee networking is actually started; the pair together genuinely guarantees both that it starts and that it starts first.

Step 2 — Resource Limits

```
CPUQuota=50%
MemoryMax=512M
MemoryHigh=384M
```

These values were chosen only after observing notesapp's own real memory and CPU usage via `systemd-cgtop` over a representative period — `systemd1-9`'s own warn-box, applied honestly here rather than picking arbitrary numbers.

Step 3 — Applying and Verifying

```
sudo systemctl daemon-reload
sudo systemctl enable --now notesapp
systemctl status notesapp
systemctl show notesapp -p MemoryMax
```

`daemon-reload` is not optional (`systemd1-3`'s own gotcha); the final command confirms the resource limit actually took effect, exactly `systemd1-9`'s own verification step.

Step 4 — The Companion Maintenance Timer

```
# notesapp-cleanup.service
[Unit]
Description=Purge notes older than 90 days

[Service]
Type=oneshot
ExecStart=/usr/local/bin/notesapp-cleanup
```

```
# notesapp-cleanup.timer
[Unit]
Description=Run notesapp-cleanup.service daily

[Timer]
OnCalendar=daily
Persistent=true

[Install]
WantedBy=timers.target
```

`systemd1-7`'s own two-unit pattern, applied for real — the cleanup logic is a genuine, independently-testable `Type=oneshot` service; the timer only decides when it runs.

Step 5 — Reading the Logs

```
journalctl -u notesapp -f
journalctl -u notesapp-cleanup -b
```

The first watches notesapp live after its first start; the second confirms the daily cleanup timer's own most recent run actually executed and logged correctly — `systemd1-6`'s own tools, put to real use.

Step 6 — Verifying Boot Impact

```
systemd-analyze critical-chain notesapp.service
```

An honest use of `systemd1-10`'s own tool — confirming notesapp's real position (or lack of one) in the boot's actual critical path, rather than assuming a new service must automatically be a boot-time concern just because it's new.

Closing the Loop on ws1 and ansible1-10

Every `systemctl` command run throughout this capstone — `status`, `restart`, `enable`, `daemon-reload` — is now fully understood. This resolves exactly what `ws1`'s own hardening chapters and `ansible1-10`'s own capstone left unexplained at the very start of this course.

PIECE	CHAPTER
Unit file structure, systemctl basics	**systemd1-2**

PIECE	CHAPTER
Type=simple, Restart=, daemon-reload	systemd1-3
Paired Wants=/After=	systemd1-4
WantedBy=multi-user.target / timers.target	systemd1-5
journalctl verification	systemd1-6
The .timer + .service pattern	systemd1-7
CPUQuota=/MemoryMax=/MemoryHigh=, verified via systemctl show	systemd1-9
systemd-analyze critical-chain verification	systemd1-10

STILL OUT OF SCOPE, HONESTLY

notesapp itself deliberately does **not** use socket activation — `systemd1-8` 's own material is a real, honest non-fit here: an always-on, frequently-hit web-facing app doesn't benefit from on-demand startup the way that chapter's own rarely-used-service example did. No systemd-managed sandboxing directives (`ProtectSystem=` , `NoNewPrivileges=` , and similar real, common production-hardening options) were covered anywhere in this course and aren't used here either. Logging stays local to journald — no centralized Loki-style shipping is actually configured, only discussed as a real possibility back in `systemd1-6` . And this capstone is single-host only, with no clustering or multi-host coordination — genuine next steps, not oversights.

Hands-On Exercises

EXERCISE 1

Explain why notesapp.service uses both Wants= and After= for network-online.target rather than just After= alone, using this chapter's own reasoning.

View solution

EXERCISE 2

Explain why the notesapp-cleanup task is written as a separate Type=oneshot service paired with a timer, rather than being built directly into notesapp.service's own ExecStart= somehow.

View solution

EXERCISE 3

A colleague suggests socket-activating notesapp.service to save resources. Give an honest answer, using this chapter's own scope note, on whether that's a good idea for this specific app.

View solution

CHAPTER 11 QUICK REFERENCE — SYSTEMD IN DEPTH COMPLETE

- notesapp.service combines paired Wants=/After=, Restart=on-failure, and a dedicated User= — every piece from systemd1-2 through systemd1-4
- CPUQuota=/MemoryMax=/MemoryHigh= chosen only after observing real usage via systemd-cgtop, per systemd1-9's own discipline
- notesapp-cleanup.timer/.service — a real .timer + .service pairing, daily, with Persistent=true catch-up
- journalctl and systemd-analyze critical-chain both verify the deployment honestly, not just assumed to be correct
- Every systemctl command from ws1's and ansible1-10's own unexplained calls is now fully understood
- Honest scope note: no socket activation (a real non-fit here), no sandboxing directives, journald only (no centralized shipping configured), single-host
- **The full systemd in Depth course — 11 chapters — is now complete.**