systemd:

The Complete Reference • 2026

Service Management • Unit Files • Targets • Timers
journalctl • Socket Activation • Security • Troubleshooting

Covers systemd 257–259 • Debian 13 Trixie • Ubuntu 26.04 LTS • RHEL 10.2 • 2026 Edition

Tux image on cover created by Larry Ewing (lewing@isc.tamu.edu) and Simon Budig using GIMP. Redistribution permitted with this notice.

systemd is actively developed software. Commands, directives, and behavior described in this book are accurate as of the versions noted (systemd 257–259, 2026). Future versions of systemd or your Linux distribution may introduce changes that differ from what is documented here. Always consult the official man pages for your installed version when in doubt.

First Edition, 2026

Contents

Preface

systemd is the init system and service manager running on almost every major Linux distribution in production today. It manages the startup sequence of your system, every service running on it, the system journal, network configuration, time synchronization, login sessions, temporary files, containers, and more. If you administer Linux servers, you interact with systemd daily — whether you know it or not.

This book exists because the official systemd documentation, while comprehensive, is written for developers who already know the system. Man pages are reference material, not tutorials. The best online guides cover individual commands well but do not give you a complete picture. This book aims to be the single volume you reach for whether you are starting out, troubleshooting a failed service at 2 AM, designing a secure service architecture, or migrating old cron jobs to systemd timers.

Every command in this book has been verified against real systems running systemd versions 257 through 259, which covers Debian 13 (Trixie), Ubuntu 26.04 LTS, Rocky Linux 10, AlmaLinux 10, and RHEL 10.2. RHEL 9.8 is also noted throughout where it remains the enterprise standard. Where behavior differs between versions, those differences are noted.

Commands that require root are shown with the sudo prefix. Commands that work without root are shown without it. Placeholders that you should replace with your own values appear in angle brackets like this: <service-name>.

Good luck, and may your services always be active (running).

How to Use This Book

This book is organized in seven parts. You do not need to read it front to back, though doing so will give you the most complete mental model of how systemd works as a whole.

If you are new to systemd, start with Part I (Chapters 1–3) to understand the architecture, then move to Part II for the core systemctl commands. Part III covers unit files in depth and is where you will spend the most time if you are creating or customizing services.

If you are an experienced Linux administrator who is already comfortable with basic systemctl usage, skip directly to the chapter most relevant to your current need. The table of contents is specific enough to get you there.

If you are troubleshooting a problem right now, go to Chapter 27 (Troubleshooting Failed Units) or Chapter 29 (Common Real-World Scenarios). These chapters are written to be used under pressure.

Code blocks throughout the book use this format:

```
$ command-run-as-regular-user
# command-run-as-root (or with sudo)
```

The dollar sign ($) prefix indicates a command run as a regular user. The hash (#) prefix indicates a command requiring root or sudo. In practice, many systemctl commands require sudo on most systems unless you are already root.

> **NOTE:** *Throughout this book, <unit> means substitute your unit name, such as nginx or my-app.service. Extensions (.service, .timer, etc.) are included in examples where they matter for clarity.*

PART I

FOUNDATIONS

Understanding What systemd Is and How It Works

Chapter 1: What systemd Is and Why It Replaced SysV Init

1.1 The Problem with SysV Init

Before systemd, the dominant init system on Linux was SysV init, inherited from Unix System V. SysV init managed startup through shell scripts located in /etc/init.d/, executed sequentially based on numbered symlinks in /etc/rc*.d/ directories. The number determined order: S20networking ran before S21apache2, for example.

SysV init had served Linux well for decades, but by the mid-2000s its limitations were causing real problems in modern computing environments:

- Sequential startup: each init script waited for the previous one to complete before starting. On modern hardware with fast SSDs and multiple CPU cores, booting still took 30–60 seconds because nothing ran in parallel.
- No dependency tracking: scripts had no way to express 'I need the network to be up before I start.' Order was implied by script numbers. Getting the numbers wrong meant broken boots.
- No restart logic: if a service crashed, SysV init did not restart it. You needed a separate tool like monit or supervisord to handle process supervision.
- No unified logging: each init script wrote to its own log file in whatever format the developer chose. Correlating logs across services required piecing together files from /var/log/, syslog, dmesg, and individual application logs.
- No cgroup integration: SysV init had no way to track all child processes of a service. A service that forked background processes could leave orphan processes running after the service was stopped.
- Shell script complexity: init scripts were Bash scripts ranging from simple wrappers to complex state machines. They were hard to write correctly, hard to read, and hard to debug.

1.2 The Rise of systemd

Lennart Poettering and Kay Sievers, working at Red Hat, announced systemd in April 2010. The design was radical in the context of existing Linux init systems: rather than shell scripts, systemd used declarative unit files. Rather than sequential startup, systemd parallelized service startup aggressively using socket activation. Rather than separate logging tools, systemd integrated a structured binary journal.

The adoption was not without controversy. systemd was criticized for its scope (init systems should do one thing), its binary journal format (not human-readable with cat), and for absorbing functions that critics argued belonged in separate tools. The debate was real and in some corners of the Linux world continues today.

But pragmatically, the results were compelling. Boot times dropped dramatically. Service reliability improved with automatic restart. Log correlation became possible with a single tool. The major distributions adopted systemd in sequence:

- Fedora 15 (2011) — first major distro to ship systemd by default
- openSUSE 12.1 (2011)
- Arch Linux (2012)
- Debian 8 Jessie (2015) — after a contentious vote and a GR
- Ubuntu 15.04 (2015) — replacing Upstart
- RHEL/CentOS 7 (2014)
- Debian 13 Trixie (2025) — current stable, systemd 257
- Ubuntu 26.04 LTS (2026) — current LTS, systemd 259

1.3 What systemd Actually Is

systemd is not just an init system. It is a suite of programs that together form a complete system and session manager. The core components you will use regularly:

- systemd — the main daemon, PID 1, the parent of all other processes on the system
- systemctl — the primary command-line tool for controlling systemd and its units
- journald — the logging daemon that collects and stores structured log data
- journalctl — the tool for reading and querying journal logs
- systemd-analyze — tools for inspecting boot performance
- networkd — network configuration daemon
- resolved — DNS resolver
- timesyncd — NTP client for time synchronization
- logind — login session management
- tmpfiles — temporary file management
- nspawn — lightweight container manager

This book covers all of them. The core focus is systemctl and journalctl, which you will use every day, but the ancillary tools are covered in Part VII.

1.4 systemd Versions and Distribution Mapping

systemd version numbers matter because features are added and behavior changes across versions. Here is a practical mapping for current distributions:

```
Distribution                    systemd Version
──────────────────────────────────────────────
Debian 12 (Bookworm)            252
Debian 13 (Trixie)              257
Ubuntu 24.04 LTS                255
Ubuntu 26.04 LTS (Resolute Raccoon)  259
RHEL 9.8 / AlmaLinux / Rocky 9  252
RHEL 10.2 / AlmaLinux / Rocky 10  256
Fedora 41                       257
Fedora 42                       257
Arch Linux (rolling)            259+
```

To check the version on your system:

```
$ systemctl --version
```

Example output:

```
systemd 259 (259.7-1ubuntu2)
+PAM +AUDIT +SELINUX +APPARMOR +IMA +SMACK +SECCOMP +GCRYPT -GNUTLS
+OPENSSL +ACL +BLKID +CURL +ELFUTILS +FIDO2 +IDN2 -IDN +IPTC +KMOD
+LIBCRYPTSETUP +LIBFDISK +PCRE2 +PWQUALITY +P11KIT +QRENCODE +TPM2
+BZIP2 +LZ4 +XZ +ZLIB +ZSTD +BPF_FRAMEWORK +XKBCOMMON +UTMP
+SYSVINIT default-hierarchy=unified
```

The first line shows the version. The compile-time options on subsequent lines tell you which optional features were built in. The +/- prefix indicates whether the feature is enabled.

Chapter 2: systemd Architecture — PID 1, Units, and the Manager

2.1 PID 1: The First Process

When the Linux kernel finishes initializing hardware, it hands control to the first user-space process. This process gets PID 1. It is the ancestor of every other process on the system. If PID 1 dies, the kernel panics.

On systemd systems, PID 1 is the systemd process itself. You can verify this:

```
$ ps -p 1 -o comm=
systemd
```

As PID 1, systemd is responsible for bringing up the entire system. It reads unit files, calculates dependency graphs, starts units in the correct order with maximum parallelism, and then transitions the system to the default target (the equivalent of a runlevel) where it waits and manages services for the lifetime of the system.

2.2 Units: The Basic Building Block

Everything systemd manages is represented as a unit. A unit is a configuration file that describes a system resource and how systemd should manage it. Units have names with a type suffix that tells systemd what kind of resource it represents:

- .service — a daemon or one-shot process (most common)
- .socket — a network or IPC socket for socket activation
- .target — a group of units, used for synchronization and as boot milestones
- .timer — a timer that triggers another unit (replaces cron)
- .mount — a filesystem mount point
- .automount — an automount point (mounts on first access)
- .path — monitors a path for changes and triggers a unit
- .device — a hardware device exposed by udev
- .swap — a swap file or partition
- .slice — a cgroup hierarchy node for resource management
- .scope — an externally created process group

Unit files are stored in several directories, checked in a specific priority order:

```
/etc/systemd/system/        # Admin-created, highest priority
/run/systemd/system/        # Runtime units, not persistent
/usr/lib/systemd/system/    # Package-installed units
/lib/systemd/system/        # Same as above on some distros
```

> **NOTE:** *Units in /etc/systemd/system/ override units with the same name in /usr/lib/systemd/system/. This is how you customize package-provided services without modifying their files.*

2.3 The Dependency Graph

systemd reads all unit files and builds an in-memory dependency graph before starting anything. This graph determines which units must start before others (ordering), which units must be running for another to start (requirements), and which units are optional companions (wants).

The key relationship keywords in unit files are:

- Requires= — hard dependency. If the required unit fails to start, this unit also fails.
- Wants= — soft dependency. The wanted unit is started alongside this one, but failure of the wanted unit does not cause this unit to fail.
- After= — ordering. This unit starts after the listed unit(s) are active.
- Before= — ordering. This unit starts before the listed unit(s).
- BindsTo= — strong binding. If the bound unit stops for any reason, this unit stops too.
- PartOf= — if the listed unit is stopped or restarted, stop or restart this unit too.

Dependencies and ordering are separate concerns. Requires= without After= means both units start at the same time (in parallel). Requires= with After= means this unit waits for the required unit to be fully active before starting.

2.4 Cgroups: Process Tracking and Resource Control

systemd uses Linux control groups (cgroups) to track and manage all processes belonging to each unit. When you start a service, systemd places it and all its child processes into a cgroup named after the service. This provides two important capabilities:

- Clean stopping: when you run systemctl stop myapp.service, systemd sends SIGTERM to every process in the service's cgroup, not just the main process. No orphan processes escape.
- Resource limits: you can set CPU, memory, and I/O limits on a service using cgroup directives in the unit file, and they apply to all processes spawned by that service.

To see the cgroup tree for running services:

```
$ systemctl status
$ systemd-cgls
```

Chapter 3: Installing and Verifying systemd

3.1 systemd on Debian 13

Debian 13 (Trixie) ships systemd 257 as the default init system. No installation is required. On a fresh Debian install, systemd is already running. Notable Trixie changes include /tmp now being a tmpfs by default, run0 as a privilege escalator, and stricter AppArmor 4.0 containment for services. Verify:

```
$ systemctl --version
$ ps -p 1 -o comm=
```

Ensure the systemd package and its utilities are installed:

```
$ dpkg -l systemd
# apt install systemd systemd-sysv
```

The systemd-sysv package provides compatibility wrappers so that legacy commands like service and invoke-rc.d continue to work by calling systemctl underneath.

3.2 systemd on Ubuntu 26.04

Ubuntu 26.04 LTS (Resolute Raccoon) ships systemd 259. This is a significant release: cgroup v1 support is completely removed, journald is persistent by default, /tmp is now tmpfs, chrony replaces systemd-timesyncd as the default NTP daemon, and Ubuntu 26.04 is the last release to support SysV init script compatibility. Verify:

```
$ systemctl status systemd-resolved
```

3.3 Checking systemd Health

After verifying the version, check overall system health:

```
$ systemctl status
```

This shows a brief summary: whether the system state is running, how many jobs are queued, how many units have failed, and how long the system has been running. A healthy system shows:

```
State: running
Jobs: 0 queued
Failed: 0 units
Since: Wed 2025-01-15 09:23:11 UTC; 3 days ago
```

If Failed is nonzero, list the failed units:

```
$ systemctl --failed
```

This shows every unit in a failed state and is your starting point for troubleshooting.

Chapter 3 Supplement: Breaking Changes in systemd 257–259

If you are upgrading an existing system to Debian 13 or Ubuntu 26.04, or if you have scripts and unit files written for older distributions, this chapter documents the changes that will break existing configurations. Read this before upgrading any production system.

3.3.1 cgroup v1 Is Gone — cgroup v2 Is Mandatory

The single most important change in this generation of systemd is the complete removal of cgroup v1 support. systemd 258 dropped cgroup v1, and Ubuntu 26.04 enforces this: systems still booting with cgroup v1 cannot upgrade to 26.04 LTS at all. Debian 13 ships systemd 257 which still transitions gracefully, but 259 on Ubuntu 26.04 is uncompromising.

What cgroup v2 means in practice:

- The unified cgroup hierarchy at /sys/fs/cgroup/ is now the only hierarchy. The legacy per-controller hierarchies under /sys/fs/cgroup/memory/, /sys/fs/cgroup/cpu/, etc. no longer exist.
- Docker has supported cgroup v2 since version 20.10. If you are running Docker 20.10 or later with standard configurations, your containers will work without modification.
- Older container images or Kubernetes configurations that reference v1-specific controller paths like memory.memsw.limit_in_bytes or cpuset paths structured under the v1 hierarchy will fail.
- LXC containers targeting Ubuntu earlier than 18.04 are not supported on a 26.04 host.

Verify your system is already running cgroup v2 before upgrading:

```
$ stat -fc %T /sys/fs/cgroup/
cgroup2fs    # Good: already on v2

# If you see 'tmpfs', you are still on v1 or hybrid
# Do not upgrade to Ubuntu 26.04 until this is resolved
```

The systemd resource control directives covered in Chapter 28 (MemoryMax=, CPUQuota=, IOWeight=, etc.) all operate on cgroup v2 and work identically on all distributions covered in this book.

> **NOTE:** *All resource control examples in this book use cgroup v2 syntax exclusively. There are no cgroup v1 examples. If you are running an older distribution that still uses v1, the syntax differs significantly.*

3.3.2 journald Is Now Persistent by Default

In systemd 259 (Ubuntu 26.04), the journal storage default changed from auto to persistent. Previously, the presence or absence of /var/log/journal/ determined whether logs survived a reboot. Now, journald creates /var/log/journal/ automatically on first boot and stores logs persistently without any administrator action.

Practical implications:

- On a fresh Ubuntu 26.04 install, journalctl -b -1 works immediately to show previous boot logs, with no configuration required.
- Disk usage will grow over time. Set SystemMaxUse= in /etc/systemd/journald.conf to cap it (see Chapter 21).
- On Debian 13 (systemd 257), the old auto default still applies. Create /var/log/journal/ manually to enable persistence, or set Storage=persistent in journald.conf.

```
# Ubuntu 26.04: persistent by default, verify with:
$ journalctl --disk-usage

# Debian 13: enable persistent logging manually:
# mkdir -p /var/log/journal
# systemctl restart systemd-journald

# Or set in /etc/systemd/journald.conf:
# [Journal]
# Storage=persistent
```

3.3.3 chrony Replaces systemd-timesyncd on Ubuntu 26.04

Ubuntu 26.04 ships chrony as the default network time synchronization daemon instead of systemd-timesyncd. This is a more capable NTP implementation better suited for server environments.

On Ubuntu 26.04 fresh installs:

```
# Check what is managing time sync:
$ systemctl status chrony
$ systemctl status systemd-timesyncd

# chrony commands:
$ chronyc tracking          # Current sync status
$ chronyc sources -v        # NTP sources
$ chronyc sourcestats       # Source statistics

# chrony config file:
# /etc/chrony/chrony.conf
```

The timedatectl command still works for checking and setting the timezone and NTP enable status regardless of which daemon is running:

```
$ timedatectl                          # Still works on all distros
$ sudo timedatectl set-timezone America/Chicago
$ sudo timedatectl set-ntp true
```

> **NOTE:** *Chapter 32 covers systemd-timesyncd and timedatectl. On Ubuntu 26.04, use chronyc for detailed NTP diagnostics. The timedatectl commands in Chapter 32 work unchanged on all distributions.*

3.3.4 /tmp Is Now tmpfs by Default

Both Debian 13 and Ubuntu 26.04 now mount /tmp as a tmpfs filesystem by default. This means /tmp is RAM-backed rather than disk-backed.

Implications:

- /tmp contents are lost on reboot. This has always been the expected behavior, but was not always enforced.
- The size of /tmp is limited. By default it is capped at 50% of physical RAM. A system with 8 GB RAM gets a 4 GB /tmp.
- Services that write large temporary files to /tmp may fail or behave unexpectedly on systems with limited RAM.

Check /tmp mount:

```
$ df -h /tmp
Filesystem      Size  Used Avail Use% Mounted on
tmpfs           3.9G     0  3.9G   0% /tmp

# Adjust tmp size in /etc/systemd/system/tmp.mount.d/override.conf:
# [Mount]
# Options=mode=1777,strictatime,nosuid,nodev,size=10G
```

For unit files that use PrivateTmp=true (covered in Chapter 23), this change has no effect. The private /tmp is always a separate tmpfs regardless of the system default.

3.3.5 SysV Init Scripts: Last Chance

Ubuntu 26.04 is explicitly the last Ubuntu LTS release that supports System V init script compatibility. Debian 13 includes the compatibility layer but systemd 259 has deprecated it, and it will be removed in systemd 260.

If you still have init scripts in /etc/init.d/, now is the time to convert them. See Appendix E for a practical SysV-to-systemd conversion guide.

```
# Check for legacy init scripts still in use:
$ ls /etc/init.d/

# Check which are managed by systemd vs legacy:
```

```
$ systemctl list-units --type=service | grep -i sysv
```

> **WARNING:** Do not delay converting SysV scripts. The next Ubuntu LTS after 26.04 will not support them at all. The next systemd release (260) removes the compatibility layer entirely.

3.3.6 networkd and nspawn: nftables Only

systemd-networkd and systemd-nspawn no longer support NAT rules via iptables/libiptc. Only nftables is now supported for network address translation. If you have custom NAT configurations using iptables in networkd or nspawn setups, they must be converted to nftables syntax.

Check which firewall backend your system uses:

```
$ nft list ruleset          # nftables rules
$ iptables -L               # Legacy (may not function in nspawn)
```

3.3.7 run0: The systemd sudo Replacement

systemd 256 introduced run0, a privilege escalation tool that is an alternative to sudo. It is available on Debian 13 and Ubuntu 26.04. Unlike sudo, run0 creates a completely fresh process tree rather than preserving the calling user's environment.

```
# Run a command as root:
$ run0 systemctl restart nginx

# Run as a specific user:
$ run0 --user=postgres psql

# Temporarily empower an unprivileged user (systemd 259+):
$ run0 --empower bash

# run0 is not a drop-in sudo replacement in all cases.
# sudo remains available and fully functional on all distributions.
```

> **NOTE:** *run0 is not covered in depth in this book as it is outside the core systemd service management scope. For production systems, sudo remains the standard. run0 is worth knowing about for interactive debugging sessions on modern systems.*

PART II

SYSTEMCTL

The Complete Command Reference

Chapter 4: Service Lifecycle Commands

These are the commands you will use most often — starting, stopping, restarting, and checking services. Every example in this chapter uses nginx as a stand-in. Replace it with whatever service you are managing.

4.1 Starting a Service

```
$ sudo systemctl start nginx
```

Starts nginx immediately. Has no effect on boot behavior — if nginx was not enabled, it will still not start at next boot. This command blocks until the service is active or has failed.

For services of Type=oneshot, start waits until the process exits. For Type=simple and Type=forking, start returns when the main process is running.

4.2 Stopping a Service

```
$ sudo systemctl stop nginx
```

Sends SIGTERM to all processes in the service cgroup, waits for them to exit cleanly, then sends SIGKILL to anything still running after the TimeoutStopSec timeout (default 90 seconds). The service is deactivated but remains enabled for boot if it was enabled.

4.3 Restarting a Service

```
$ sudo systemctl restart nginx
```

Equivalent to stop followed by start. The service is stopped completely and a fresh instance is started. All connections are dropped. Use this when you need a clean slate.

4.4 Reloading a Service

```
$ sudo systemctl reload nginx
```

Sends SIGHUP (or whatever reload signal the unit defines with ExecReload=) to the main process, telling it to re-read its configuration file without stopping. Connections are not dropped. Only works if the service supports in-process reload. For nginx, this reloads nginx.conf without interrupting active connections. If you are unsure whether a service supports reload, use reload-or-restart:

```
$ sudo systemctl reload-or-restart nginx
```

This attempts reload first. If the service does not support reload, it falls back to a full restart.

4.5 Enabling and Disabling at Boot

```
$ sudo systemctl enable nginx
$ sudo systemctl disable nginx
```

enable creates symlinks in the appropriate .wants/ or .requires/ target directory so the unit starts at boot. It does not start the service immediately. disable removes those symlinks. Neither command starts or stops the service right now.

To enable and start in one command:

```
$ sudo systemctl enable --now nginx
```

To disable and stop in one command:

```
$ sudo systemctl disable --now nginx
```

> **NOTE:** *enable --now is the recommended way to deploy a new service. It both configures it for future boots and starts it immediately, reducing the chance of forgetting one step.*

4.6 Checking Service Status

```
$ systemctl status nginx
```

This is your most-used diagnostic command. It shows:

- The service description from the unit file
- Load state: whether the unit file was loaded successfully
- Active state: active (running), inactive (dead), failed, activating, deactivating

- Main PID and process name
- Memory usage and CPU time
- CGroup tree showing all processes in the service
- Last 10 journal log lines for the service

Example output:

```
● nginx.service - A high performance web server and a reverse proxy
server
     Loaded: loaded (/lib/systemd/system/nginx.service; enabled;
preset: enabled)
     Active: active (running) since Mon 2025-01-13 08:15:42 UTC; 2 days
ago
       Docs: man:nginx(8)
   Main PID: 1247 (nginx)
      Tasks: 5 (limit: 4915)
     Memory: 12.4M
        CPU: 4.521s
     CGroup: /system.slice/nginx.service
             ├─1247 "nginx: master process /usr/sbin/nginx -g daemon
on; master_process on;"
             ├─1248 "nginx: worker process"
             ├─1249 "nginx: worker process"
             └─1250 "nginx: cache manager process"

Jan 13 08:15:42 server nginx[1246]: nginx: configuration file
/etc/nginx/nginx.conf test ok
Jan 13 08:15:42 server systemd[1]: Started A high performance web
server and a reverse proxy server.
```

4.7 Checking Active, Enabled, and Failed State

Sometimes you need a quick programmatic check rather than full status output:

```
$ systemctl is-active nginx
active

$ systemctl is-enabled nginx
enabled

$ systemctl is-failed nginx
failed    # or 'activating', 'inactive', depending on state
```

These commands return a single word and also set the shell exit code: 0 for true, nonzero for false. This makes them useful in scripts:

```
if systemctl is-active --quiet nginx; then
    echo "nginx is running"
else
    echo "nginx is NOT running"
fi
```

The --quiet flag suppresses all output, leaving only the exit code.

4.8 Freezing and Thawing Services

```
$ sudo systemctl freeze nginx
$ sudo systemctl thaw nginx
```

freeze sends SIGSTOP to all processes in the service cgroup, pausing them without stopping the service. thaw sends SIGCONT, resuming execution. Useful for debugging or temporarily halting a resource-intensive process without killing it. Available in systemd 252+. Supported on all distributions covered in this book. Supported on all distributions covered in this book.

4.9 Killing Processes in a Service

```
$ sudo systemctl kill nginx
$ sudo systemctl kill --kill-whom=all --signal=SIGKILL nginx
```

kill sends a signal to processes in the service. By default sends SIGTERM to the main process. Options:

- --kill-whom=main — send to main process only (default)
- --kill-whom=control — send to control process (ExecReload, ExecStop, etc.)
- --kill-whom=all — send to all processes in the cgroup
- --signal=SIGKILL — specify any signal by name or number

> **WARNING:** systemctl kill bypasses the normal stop sequence and does not run ExecStop= or ExecStopPost= hooks. Use systemctl stop for normal service stopping. Use kill only when stop is unresponsive.

Chapter 5: Listing, Querying, and Inspecting Units

5.1 List All Units

```
$ systemctl list-units
```

Lists all units that systemd has loaded, regardless of state. Output columns:

- UNIT — the unit name
- LOAD — loaded, not-found, masked, error
- ACTIVE — active, inactive, failed, activating, deactivating
- SUB — the detailed sub-state (running, exited, dead, waiting, etc.)
- DESCRIPTION — the Description= from the unit file

Filter by type:

```
$ systemctl list-units --type=service
$ systemctl list-units --type=timer
$ systemctl list-units --type=socket
$ systemctl list-units --type=mount
```

Filter by state:

```
$ systemctl list-units --state=running
$ systemctl list-units --state=failed
$ systemctl list-units --state=inactive
```

Combine type and state:

```
$ systemctl list-units --type=service --state=failed
```

5.2 List Unit Files

```
$ systemctl list-unit-files
```

Different from list-units: this shows all unit files on disk, regardless of whether they are currently loaded. The STATE column shows:

- enabled — configured to start at boot
- disabled — not configured to start at boot
- static — no [Install] section; enabled by other units
- masked — blocked from starting
- generated — auto-generated by systemd generators
- indirect — enabled through an alias

Filter:

```
$ systemctl list-unit-files --type=service
```

```
$ systemctl list-unit-files --state=enabled
$ systemctl list-unit-files --state=masked
```

5.3 Showing Unit Properties

```
$ systemctl show nginx
```

Outputs every property of the unit in key=value format — hundreds of properties including all directives, their current values, and systemd-computed values. This is the machine-readable companion to systemctl status.

Show a specific property:

```
$ systemctl show nginx --property=MainPID
MainPID=1247

$ systemctl show nginx --property=ActiveState
ActiveState=active

$ systemctl show nginx --property=FragmentPath
FragmentPath=/lib/systemd/system/nginx.service
```

Show multiple properties:

```
$ systemctl show nginx --property=MainPID,ActiveState,MemoryCurrent
```

5.4 Viewing Unit File Contents

```
$ systemctl cat nginx
```

Prints the actual unit file content as systemd sees it, including any drop-in overrides, clearly labeled with the source file path. This is the best way to see the effective configuration of a unit, as it shows both the main unit file and any /etc/systemd/system/nginx.service.d/*.conf overrides.

5.5 Editing Unit Files

```
$ sudo systemctl edit nginx
$ sudo systemctl edit --full nginx
```

systemctl edit opens a drop-in override file at /etc/systemd/system/nginx.service.d/override.conf in your default editor. Only the directives you add in this file override the original. The original unit file is not modified. This is the safe way to customize package-provided units.

systemctl edit --full copies the entire unit file to /etc/systemd/system/nginx.service and opens it for editing. You are now editing a complete replacement for the package unit file. Be aware that package updates will not update /etc/systemd/system/ copies, only /usr/lib/systemd/system/.

After any edit, reload the daemon to pick up changes:

```
$ sudo systemctl daemon-reload
```

> **WARNING:** Always run daemon-reload after creating or modifying unit files. Without it, systemd continues using the cached version of the unit file and will not see your changes.

5.6 Viewing Dependencies

```
$ systemctl list-dependencies nginx
$ systemctl list-dependencies nginx --reverse
$ systemctl list-dependencies nginx --all
```

list-dependencies shows what nginx depends on (the units that must be active for nginx to start). --reverse shows what depends on nginx (the units that would be affected if nginx stopped). --all expands all dependencies recursively.

To see only the immediate ordering relationships:

```
$ systemctl list-dependencies nginx --before
$ systemctl list-dependencies nginx --after
```

Chapter 6: Targets — Replacing Runlevels

6.1 What Targets Are

In SysV init, runlevels were integers (0–6) representing system states: 0 for halt, 1 for single-user mode, 3 for multi-user without graphics, 5 for multi-user with a graphical display. Scripts in /etc/rc3.d/ ran when entering runlevel 3.

systemd replaces runlevels with targets. A target is a unit file with the .target extension that groups other units. Targets are named descriptively rather than numbered:

```
SysV Runlevel    systemd Target              Description
─────────────────────────────────────────────────────────────
0                poweroff.target            Halt the system
1                rescue.target              Single-user / rescue mode
2, 3, 4          multi-user.target           Multi-user, no GUI
5                graphical.target            Multi-user with GUI
6                reboot.target              Reboot
                 emergency.target           Minimal emergency shell
```

6.2 Checking and Setting the Default Target

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

$ sudo systemctl set-default graphical.target
$ sudo systemctl set-default multi-user.target
```

set-default creates a symlink at /etc/systemd/system/default.target pointing to the specified target. The system boots to this target on next restart.

For servers, multi-user.target is correct. For desktops, graphical.target is correct. Setting a headless server to graphical.target wastes resources on a display manager that can never display anything.

6.3 Switching Targets Without Rebooting

```
$ sudo systemctl isolate multi-user.target
$ sudo systemctl isolate rescue.target
```

isolate switches the system to the specified target immediately, stopping all units that are not part of the new target and starting all units that are. On a graphical system, isolating multi-user.target kills the display manager and drops you to a console.

> **WARNING:** systemctl isolate rescue.target puts the system in single-user mode. Most services will stop. Use this for maintenance only.

6.4 Listing Available Targets

```
$ systemctl list-units --type=target
```

Shows all loaded targets and their state. Notable targets beyond the runlevel equivalents:

- network.target — indicates network interfaces are configured (not necessarily connected)
- network-online.target — indicates at least one network interface is online with connectivity
- local-fs.target — local filesystems are mounted
- remote-fs.target — remote filesystems (NFS, CIFS) are mounted
- sysinit.target — basic system initialization is complete
- basic.target — basic system services are running (sockets, paths, timers)
- shutdown.target — system is shutting down
- timers.target — all enabled timers are loaded
- sockets.target — all enabled sockets are loaded

> **NOTE:** *network.target and network-online.target are different. network.target means the network configuration has been applied but the link may not be up. network-online.target means at least one interface is fully online. For services that require actual network connectivity, use After=network-online.target, not After=network.target.*

Chapter 7: Boot Control and System State Commands

7.1 Rebooting, Halting, and Powering Off

```
$ sudo systemctl reboot
$ sudo systemctl poweroff
$ sudo systemctl halt
$ sudo systemctl suspend
$ sudo systemctl hibernate
$ sudo systemctl hybrid-sleep
```

These commands trigger the corresponding system state transitions. They go through the proper shutdown sequence — stopping all services cleanly, syncing filesystems, and then performing the requested action.

- reboot — stop all services, sync, reboot
- poweroff — stop all services, sync, power off
- halt — stop all services, sync, halt the CPU (does not power off on some hardware)
- suspend — suspend to RAM (S3 sleep)
- hibernate — suspend to disk (S4 sleep, requires swap space >= RAM)
- hybrid-sleep — suspend to RAM and simultaneously write hibernation image to disk

7.2 Scheduled Reboot

```
$ sudo systemctl reboot --when=+10
$ sudo systemctl reboot --when=02:00
$ sudo systemctl reboot --when=cancel
```

--when schedules a reboot for a specific time. +10 means 10 minutes from now. 02:00 means 2 AM. cancel cancels a scheduled reboot. Available in systemd 252+. Supported on all distributions covered in this book. Supported on all distributions covered in this book.

7.3 Daemon Reload and Reexec

```
$ sudo systemctl daemon-reload
$ sudo systemctl daemon-reexec
```

daemon-reload tells systemd to re-read all unit files from disk. Always run this after creating or modifying unit files. It does not restart any services.

daemon-reexec tells systemd to reexecute itself — the systemd binary itself is reloaded from disk. Used after upgrading the systemd package itself. Services are not restarted, and the system state is serialized and restored.

7.4 Rescue and Emergency Mode

```
$ sudo systemctl rescue
$ sudo systemctl emergency
```

rescue transitions to rescue.target — single-user mode with a root shell, most services stopped but filesystems mounted.

emergency transitions to emergency.target — the most minimal possible state, typically a root shell with the root filesystem mounted read-only. Use when rescue mode is not available.

Both require the root password. These commands are intended for system recovery.

Chapter 8: Masking, Overriding, and Drop-In Files

8.1 Masking a Unit

```
$ sudo systemctl mask nginx
$ sudo systemctl unmask nginx
```

Masking a unit creates a symlink from /etc/systemd/system/nginx.service to /dev/null. A masked unit cannot be started manually or by dependency, even if it is listed in another unit's Wants= or Requires=. This is the strongest way to prevent a unit from running.

Use masking for services that should never run on a particular machine — for example, masking cups.service on a server that will never have a printer.

> **WARNING:** Masking prevents even manual starting with systemctl start. Make sure you mean to prevent ALL starting, not just boot starting. For just preventing boot start, use disable.

8.2 Drop-In Files

Drop-in files are the correct way to customize a package-provided unit file without replacing it entirely. They are placed in a directory named after the unit:

```
/etc/systemd/system/nginx.service.d/override.conf
/etc/systemd/system/ssh.service.d/custom-timeout.conf
/etc/systemd/system/postgresql.service.d/memory.conf
```

A drop-in file has the same syntax as a unit file. Directives in drop-in files are merged with the original unit file. For list-type directives like ExecStart=, you must first clear the list with an empty assignment, then set your value:

```
[Service]
ExecStart=
ExecStart=/usr/sbin/nginx -c /etc/nginx/custom.conf
```

Without the empty ExecStart= line, you would be appending to the existing ExecStart, which is not valid for ExecStart (it must have exactly one value).

For non-list directives, you simply set the new value:

```
[Service]
TimeoutStartSec=120
Restart=always
RestartSec=5
```

Create drop-ins with systemctl edit (recommended) or manually:

```
$ sudo mkdir -p /etc/systemd/system/nginx.service.d
$ sudo nano /etc/systemd/system/nginx.service.d/override.conf
```

```
$ sudo systemctl daemon-reload
```

8.3 Revert to Default

```
$ sudo systemctl revert nginx
```

Removes any drop-in files in /etc/systemd/system/nginx.service.d/ and the unit override at /etc/systemd/system/nginx.service if present, restoring the unit to its package-provided state. Runs daemon-reload automatically.

PART III

UNIT FILES IN DEPTH

Writing, Reading, and Customizing Every Directive

Chapter 9: Unit File Structure and the [Unit] Section

9.1 Unit File Anatomy

Every unit file is an INI-format text file divided into sections. Sections are marked with a name in square brackets. Directives within a section are key=value pairs. Comments begin with # or ;. Blank lines are ignored.

A typical service unit file has three sections:

```
[Unit]
# Metadata, documentation, dependencies

[Service]
# How to start, stop, and run the process

[Install]
# How to enable this unit at boot
```

Other unit types use different sections: [Socket] for .socket units, [Timer] for .timer units, [Mount] for .mount units, and so on. The [Unit] and [Install] sections are common to all unit types.

9.2 [Unit] Section: Every Directive

Description=

A short human-readable description of the unit. Shown in systemctl status output and journalctl logs.

```
Description=NGINX HTTP Server
```

Documentation=

Space-separated list of URIs for documentation. Shown in systemctl status. Can be man: URIs, http:// URLs, or file:// paths.

```
Documentation=man:nginx(8) https://nginx.org/en/docs/
```

Wants=

Soft dependency. The listed units will be started alongside this unit, but if they fail, this unit is not affected. Most dependencies should use Wants= rather than Requires=.

```
Wants=network-online.target
```

Requires=

Hard dependency. If the required unit fails to start, this unit also fails. If the required unit is stopped while this unit is running, this unit is stopped too.

```
Requires=postgresql.service
```

Requisite=

Like Requires=, but the required unit must already be active when this unit starts. If it is not already running, this unit fails immediately rather than trying to start the required unit first.

```
Requisite=docker.service
```

BindsTo=

Stronger than Requires=. If the bound unit stops for any reason — including being explicitly stopped — this unit is stopped too. Used when two units are tightly coupled.

```
BindsTo=dev-sda1.device
```

PartOf=

When the listed unit is restarted or stopped, this unit is also restarted or stopped. Does not create a dependency for starting.

```
PartOf=docker.service
```

Conflicts=

This unit cannot run simultaneously with the listed unit(s). Starting this unit will stop the conflicting unit, and vice versa.

```
Conflicts=apache2.service
```

After= and Before=

Ordering directives. After=network.target means this unit starts after network.target is active. Before=shutdown.target means this unit starts before shutdown.target. These are ordering only — they do not create a dependency.

```
After=network-online.target
After=postgresql.service
Before=apache2.service
```

OnFailure=

When this unit enters the failed state, start the listed unit. Commonly used to trigger a notification or recovery script.

```
OnFailure=notify-failed@%n.service
```

OnSuccess=

When this unit completes successfully (for Type=oneshot units), start the listed unit. Available in systemd 249+.

```
OnSuccess=cleanup.service
```

PropagatesReloadTo= and ReloadPropagatedFrom=

When this unit is reloaded, also reload the listed units (PropagatesReloadTo=). When the listed unit reloads, also reload this unit (ReloadPropagatedFrom=).

StopWhenUnneeded=

If set to yes, this unit is automatically stopped when no other active unit requires it. Useful for shared resources.

```
StopWhenUnneeded=yes
```

RefuseManualStart= and RefuseManualStop=

Prevents manual starting (RefuseManualStart=yes) or stopping (RefuseManualStop=yes) with systemctl. The unit can still be started or stopped by dependency.

ConditionPathExists=

The unit starts only if the specified path exists. Prefix with ! to invert: start only if the path does not exist.

```
ConditionPathExists=/etc/myapp/config.yaml
ConditionPathExists=!/var/lock/myapp-disabled
```

ConditionFileNotEmpty=

Start only if the specified file exists and is not empty.

```
ConditionFileNotEmpty=/etc/myapp/config.yaml
```

ConditionHost=

Start only on the specified hostname. Useful for units deployed from a shared configuration management system that should only run on specific machines.

```
ConditionHost=web-server-01
```

ConditionVirtualization=

Start only if the system is running in a specific virtualization environment. Values: yes (any VM), container, kvm, qemu, vmware, xen, lxc, docker, private-users, no (bare metal).

```
ConditionVirtualization=no        # Only on bare metal
ConditionVirtualization=container # Only in containers
```

AssertPathExists=

Like ConditionPathExists= but causes a hard failure rather than a silent skip if the condition is not met.

SourcePath=

Specifies the path of a configuration file this unit was generated from. Used by generators, not normally set manually.

DefaultDependencies=

If set to no, default implicit dependencies are not added. Default is yes. Setting to no is sometimes needed for units that run very early in the boot sequence, before default dependencies are available. Expert use only.

```
DefaultDependencies=no
```

Chapter 10: The [Service] Section — Every Directive Explained

The [Service] section is the largest and most complex part of a service unit file. It defines how the service process is started, stopped, supervised, and configured. This chapter covers every directive you are likely to encounter.

10.1 Process Execution

ExecStart=

The command to run when the service starts. Must be an absolute path. Arguments can follow. Environment variables can be used. For Type=oneshot, multiple ExecStart= lines are allowed and execute in sequence.

```
ExecStart=/usr/sbin/nginx -g 'daemon off;'
ExecStart=/usr/bin/python3 /opt/myapp/server.py --port 8080
```

Special prefixes for ExecStart= and other Exec* directives:

- - (hyphen) — ignore exit code: ExecStart=-/bin/command means failure is not an error
- @ (at) — pass argv[0] as a different value than the binary path
- ! (exclamation) — run without privilege escalation even if User= is set
- !! (double exclamation) — run with all privilege escalation (ambient capabilities)
- + (plus) — run with full privileges regardless of User= setting

ExecStartPre= and ExecStartPost=

Commands to run before (Pre) or after (Post) the main ExecStart= command. Multiple lines are allowed. Useful for setup and verification steps.

```
ExecStartPre=/usr/bin/nginx -t
ExecStart=/usr/sbin/nginx -g 'daemon off;'
ExecStartPost=/usr/bin/curl -sf http://localhost/health
```

ExecStop=

The command to run to stop the service. If not specified, systemd sends the KillSignal (usually SIGTERM) to all processes in the cgroup. For services that need a specific stop sequence, define ExecStop=.

```
ExecStop=/usr/bin/nginx -s quit
```

ExecStopPost=

Command to run after the service has stopped, even if ExecStop= failed or the service crashed. Used for cleanup.

```
ExecStopPost=/usr/bin/rm -f /run/myapp/myapp.pid
```

ExecReload=

The command to run when systemctl reload is called. For most services this sends SIGHUP to the main process.

```
ExecReload=/bin/kill -HUP $MAINPID
```

The special variable $MAINPID contains the PID of the main service process.

10.2 Service Type

Type=

Defines how systemd determines when the service has finished starting. This is one of the most important directives and is covered in full detail in Chapter 13.

- simple — default. Service is considered started when ExecStart process is launched.
- exec — like simple, but waits until the executable is actually exec'd (not just forked).
- forking — ExecStart forks and the parent exits. systemd follows the child.
- oneshot — ExecStart runs and exits. Service is considered done when it exits.
- dbus — service is ready when it acquires a D-Bus name.
- notify — service sends sd_notify(READY=1) when ready.
- notify-reload — like notify, also uses sd_notify for reload completion.
- idle — like simple, but delayed until all jobs are dispatched.

10.3 Restart Behavior

Restart=

When to automatically restart the service. Default is no (never restart).

- no — never restart
- on-success — restart only when the service exits cleanly (exit code 0)
- on-failure — restart on failure (nonzero exit, killed by signal, timeout). Most common for production services.
- on-abnormal — restart on signal, timeout, or watchdog timeout (not on clean exit or explicit stop)
- on-abort — restart only when killed by an uncaught signal
- on-watchdog — restart only on watchdog timeout
- always — always restart, regardless of exit status

```
Restart=on-failure
```

RestartSec=

Time to wait before restarting. Accepts a time value with unit: 5s, 1min, 500ms.

```
RestartSec=5s
```

StartLimitIntervalSec= and StartLimitBurst=

Rate-limits restarts. If the service starts more than StartLimitBurst times within StartLimitIntervalSec seconds, it enters the failed state and stops restarting. Prevents a broken service from spinning in a tight loop.

```
StartLimitIntervalSec=60s
StartLimitBurst=5
```

With these settings, if the service fails and restarts 5 times within 60 seconds, systemd gives up and leaves it in the failed state. Use systemctl reset-failed to clear this and try again.

RestartSteps= and RestartMaxDelaySec=

For exponential restart backoff (systemd 254+). RestartSteps= sets the number of steps from RestartSec= to RestartMaxDelaySec=. systemd increases the delay between each restart attempt up to the maximum.

```
RestartSec=1s
RestartSteps=5
RestartMaxDelaySec=60s
```

10.4 Timeouts

TimeoutStartSec=

How long systemd waits for the service to report itself as started. If exceeded, the service is killed and considered failed. Default is 90 seconds on most distributions.

```
TimeoutStartSec=120s
```

Set to infinity to disable the start timeout (for services that can take arbitrarily long to initialize):

```
TimeoutStartSec=infinity
```

TimeoutStopSec=

How long systemd waits for the service to stop after sending SIGTERM before sending SIGKILL. Default is 90 seconds.

```
TimeoutStopSec=30s
```

TimeoutSec=

Sets both TimeoutStartSec= and TimeoutStopSec= to the same value.

```
TimeoutSec=60s
```

TimeoutAbortSec=

Time to wait for the service to terminate after receiving SIGABRT before sending SIGKILL. Used for coredump collection.

WatchdogSec=

Enables the watchdog. The service must send sd_notify(WATCHDOG=1) at least once per WatchdogSec= period. If it fails to do so, systemd kills and restarts it (if Restart= is configured). Used for services that implement their own health checking.

```
WatchdogSec=30s
```

RuntimeMaxSec=

Maximum time the service may run. After this period, it is stopped. Useful for services that should be time-limited (batch jobs, maintenance tasks).

```
RuntimeMaxSec=1h
```

10.5 Identity and Permissions

User= and Group=

The user and group the service process runs as. If not set, runs as root.

```
User=www-data
Group=www-data
```

Setting User= is one of the most important security practices for services. A service running as a dedicated low-privilege user cannot access files belonging to root or other users.

DynamicUser=

If set to yes, systemd allocates a temporary user and group for the service that do not exist in /etc/passwd. The UID/GID are chosen dynamically, are not reused across restarts in a predictable way, and are deallocated when the service stops. Excellent for one-shot or batch tasks that need a user identity without requiring a system account.

```
DynamicUser=yes
```

SupplementaryGroups=

Additional groups to assign to the service process in addition to the primary group.

```
SupplementaryGroups=ssl-cert video
```

WorkingDirectory=

The working directory for the service process.

```
WorkingDirectory=/opt/myapp
```

RootDirectory=

Chroot the service to this directory. The service sees this as its root filesystem.

```
RootDirectory=/var/lib/myapp/chroot
```

UMask=

The umask for the service process. Default is 0022.

```
UMask=0027
```

10.6 Environment

Environment=

Set environment variables for the service. Can be specified multiple times.

```
Environment=PORT=8080
Environment=LOG_LEVEL=info
Environment="DB_URL=postgresql://localhost/mydb"
```

EnvironmentFile=

Read environment variables from a file. The file contains KEY=VALUE lines. Prefix with - to make the file optional.

```
EnvironmentFile=/etc/myapp/environment
EnvironmentFile=-/etc/myapp/local.env
```

The file format is simple:

```
PORT=8080
LOG_LEVEL=info
DB_PASSWORD=secret
```

PassEnvironment=

Pass specific environment variables from the systemd manager's environment to the service.

```
PassEnvironment=HOME LANG TZ
```

UnsetEnvironment=

Remove specific variables from the service environment even if they were set by Environment= or EnvironmentFile=.

```
UnsetEnvironment=LD_PRELOAD
```

10.7 Standard I/O

StandardOutput= and StandardError=

Where stdout and stderr from the service process are sent.

- inherit — inherited from systemd (goes to the journal)

- null — /dev/null
- tty — to a TTY
- journal — to the systemd journal (default)
- kmsg — to the kernel log buffer
- journal+console — to both the journal and the console
- file:/path/to/file — to a specific file
- append:/path/to/file — appended to a specific file
- truncate:/path/to/file — overwritten at each start
- socket — to the socket that activated the service (for socket-activated services)

```
StandardOutput=journal
StandardError=journal
```

SyslogIdentifier=

The identifier used for journal and syslog entries from this service. Defaults to the service name. Set this to distinguish logs from multiple instances or to match the application's own log identifier.

```
SyslogIdentifier=myapp
```

10.8 Resource Limits

LimitNOFILE=

Maximum number of open file descriptors. Default is the system limit (usually 1024). Many applications need this raised.

```
LimitNOFILE=65536
```

LimitNPROC=

Maximum number of processes the service can create.

```
LimitNPROC=512
```

LimitCORE=

Maximum coredump file size. Set to infinity to allow unlimited coredumps for debugging.

```
LimitCORE=infinity
```

LimitMEMLOCK=

Maximum bytes of memory that can be locked into RAM. Relevant for applications using mlock() like VPN software or in-memory databases.

```
LimitMEMLOCK=infinity
```

10.9 Other Service Directives

PIDFile=

Path to the PID file created by the service. Used with Type=forking so systemd knows which process is the main process after the parent forks and exits.

```
PIDFile=/run/nginx/nginx.pid
```

BusName=

The D-Bus name the service acquires when ready. Required for Type=dbus.

```
BusName=org.freedesktop.NetworkManager
```

NotifyAccess=

Controls which processes can send sd_notify messages to systemd.

- none — no process can send notifications (default unless Type=notify)
- main — only the main process
- exec — main process and any exec'd processes
- all — all processes in the cgroup

```
NotifyAccess=main
```

RemainAfterExit=

If yes, the service is considered active even after all its processes have exited. Useful for Type=oneshot services that perform setup steps: the service is shown as active until explicitly stopped.

```
RemainAfterExit=yes
```

GuessMainPID=

If systemd cannot reliably determine the main PID (for Type=forking without PIDFile=), it guesses. Set to no to disable this.

```
GuessMainPID=no
```

SuccessExitStatus=

Exit codes in addition to 0 that are considered success. Accepts exit codes and signal names.

```
SuccessExitStatus=0 1 SIGTERM
```

PermissionsStartOnly=

If yes, the User=, Group=, and capability settings only apply to the ExecStart= command, not to ExecStartPre= or ExecStartPost=. Those run as root.

```
PermissionsStartOnly=yes
```

Chapter 11: The [Install] Section

The [Install] section determines what happens when you run systemctl enable or systemctl disable. It is not read by systemd during normal operation — only when enable/disable processes the unit.

WantedBy=

When enabled, systemd creates a symlink in the .wants/ directory of the specified target. This is the most common way to enable a service.

```
WantedBy=multi-user.target
```

This creates: /etc/systemd/system/multi-user.target.wants/nginx.service -> /lib/systemd/system/nginx.service

Multi-user.target is correct for server services. Use graphical.target for services that only need to run in a graphical desktop session.

RequiredBy=

Like WantedBy= but creates a symlink in the .requires/ directory, making it a hard dependency of the target.

```
RequiredBy=multi-user.target
```

Also=

Additional units to enable/disable alongside this one. When you enable this unit, the listed units are also enabled.

```
Also=myapp-cleanup.timer
```

Alias=

Alternate names for this unit. When enabled, symlinks with these names are created pointing to this unit.

```
Alias=httpd.service www.service
```

Chapter 12: Writing Your First Custom Service

This chapter walks through creating a custom systemd service from scratch, explaining every decision. We will create a service for a fictional Python web application called myapp.

12.1 The Application

Our application is a Python script at /opt/myapp/server.py that runs a web server on port 8080. It reads its configuration from /etc/myapp/config.yaml and writes logs to stdout. It runs as a dedicated user named myapp.

12.2 Create the System User

```
# useradd --system --no-create-home --shell /bin/false myapp
```

A system user has a UID below 1000, has no home directory, and cannot log in. This is the correct account type for service processes.

12.3 Create the Unit File

```
# nano /etc/systemd/system/myapp.service
```

Content:

```
[Unit]
Description=MyApp Web Application
Documentation=https://docs.myapp.example.com
After=network-online.target
Wants=network-online.target

[Service]
Type=simple
User=myapp
Group=myapp
WorkingDirectory=/opt/myapp
ExecStart=/usr/bin/python3 /opt/myapp/server.py --config
/etc/myapp/config.yaml
Restart=on-failure
RestartSec=5s
StartLimitIntervalSec=60s
StartLimitBurst=5
```

```
# Logging
StandardOutput=journal
StandardError=journal
SyslogIdentifier=myapp

# Limits
LimitNOFILE=65536

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

12.4 Enable and Start

```
# systemctl daemon-reload
# systemctl enable --now myapp.service
# systemctl status myapp.service
```

12.5 Verify It Works

```
$ systemctl is-active myapp.service
active

$ curl http://localhost:8080/health
{"status": "ok"}

$ journalctl -u myapp.service -n 20
```

12.6 Test the Restart Behavior

Kill the process directly to verify systemd restarts it:

```
$ systemctl show myapp.service --property=MainPID
MainPID=4821

$ sudo kill -9 4821

# Wait 5 seconds (RestartSec=5s), then:
$ systemctl status myapp.service
Active: active (running) since...
```

Chapter 13: Service Types — simple, forking, oneshot, notify, dbus, idle

13.1 Type=simple (Default)

The service is considered started the moment ExecStart= forks the process. systemd does not wait for the process to signal readiness. This is correct for most modern daemons that stay in the foreground.

```
[Service]
Type=simple
ExecStart=/usr/bin/myapp --foreground
```

Use simple when: the process runs in the foreground, you do not need ordering guarantees (other services starting 'after' this one will start as soon as the process is forked, not when it is actually ready), and the process itself is its own main process.

13.2 Type=exec

Like simple, but systemd waits until the binary is actually exec'd (not just forked) before considering the service started. This ensures that the process name shown in ps output is the actual binary, not the shell. Available in systemd 240+.

```
[Service]
Type=exec
ExecStart=/usr/bin/myapp
```

13.3 Type=forking

The traditional Unix daemon model. The parent process forks a child, the child continues running as the daemon, and the parent exits. systemd considers the service started when the parent process exits. Use PIDFile= to tell systemd which process is the main daemon.

```
[Service]
Type=forking
PIDFile=/run/myapp/myapp.pid
ExecStart=/usr/sbin/myapp -D
```

Use forking for: older daemons that daemonize themselves (Apache 2 without the -DFOREGROUND option, older versions of nginx), when you have no control over the daemon's behavior.

> **WARNING:** Many modern services that historically used Type=forking now support a foreground mode. Prefer Type=simple or Type=notify when possible. forking is harder for systemd to track correctly.

13.4 Type=oneshot

The service runs a command that exits, and systemd waits for it to exit before considering the service done. Unlike simple, systemd knows the service is 'done' when the process exits. Multiple ExecStart= lines are executed in sequence. Often used with RemainAfterExit=yes to show the service as active after it completes.

```
[Service]
Type=oneshot
RemainAfterExit=yes
ExecStart=/usr/bin/setup-thing.sh
ExecStart=/usr/bin/verify-thing.sh
ExecStop=/usr/bin/teardown-thing.sh
```

Use oneshot for: initialization scripts, setup tasks, one-time migrations, systemd replacement for rc.local.

13.5 Type=notify

The service explicitly tells systemd when it is ready by calling sd_notify(READY=1). systemd waits for this notification before marking the service as started and letting dependent services proceed. This is the most reliable type for services that need non-trivial startup time.

```
[Service]
Type=notify
NotifyAccess=main
ExecStart=/usr/sbin/myapp
```

The application must call sd_notify(3) with READY=1 in its code. From a shell script:

```
systemd-notify --ready
```

Use notify for: any service that can be modified to signal readiness, modern well-behaved daemons. This is the recommended type for new services.

13.6 Type=dbus

The service is considered ready when it acquires a specific D-Bus name on the system bus. Used for services that expose a D-Bus interface.

```
[Service]
Type=dbus
BusName=org.freedesktop.NetworkManager
ExecStart=/usr/sbin/NetworkManager --no-daemon
```

13.7 Type=idle

Like simple, but the service is not started until all pending jobs are dispatched. Prevents the service output from mixing with boot-time output. Used for services that display text on the console.

```
[Service]
Type=idle
ExecStart=/usr/bin/console-message
```

Chapter 14: Dependencies, Ordering, and Conditions

14.1 Requirements vs. Ordering: A Critical Distinction

The most common misunderstanding in systemd unit files is conflating requirements with ordering. They are separate concepts.

Requires=B means: if B is not running, start it. If B fails, fail A.

After=B means: when starting A, wait until B is active first.

Requires=B without After=B means: start both A and B at the same time (in parallel). A requires B to eventually be running, but does not wait for it.

The correct pattern for 'A needs B to be running before A starts':

```
Requires=B.service
After=B.service
```

14.2 Wants= vs. Requires=

Prefer Wants= over Requires= in almost all cases. The difference:

- Wants=B: systemd tries to start B. If B fails, A still starts.
- Requires=B: systemd tries to start B. If B fails, A also fails.

Requires= creates brittleness. A database that temporarily fails to start will take down every service that Requires= it. With Wants=, your services try to start and then fail on their own if they cannot reach the database, which gives you better error messages and more flexibility.

14.3 network-online.target vs. network.target

This is the most common dependency mistake in custom service units.

network.target is reached when the network configuration has been applied. The links may not be up. DNS may not be resolving. This is too early for most services.

network-online.target is reached when at least one network interface is fully online with connectivity. This is what most services need.

```
# WRONG for most services:
After=network.target

# CORRECT for services that need network connectivity:
After=network-online.target
```

```
Wants=network-online.target
```

> **NOTE:** *network-online.target slows down boot because it waits for network connectivity. Do not use it for services that do not actually need network connectivity at startup.*

14.4 Condition Directives in Practice

Conditions are checked before the service starts. If a condition fails, the service is skipped (not failed) and dependent services continue.

```
[Unit]
# Only start if the config file exists
ConditionPathExists=/etc/myapp/config.yaml

# Only start on bare metal, not in VMs or containers
ConditionVirtualization=no

# Only start if this file does NOT exist
ConditionPathExists=!/var/lib/myapp/.disabled
```

Assert directives work like Condition but cause a hard failure instead of a silent skip:

```
# Fail with an error if /data is not a mount point
AssertPathIsMountPoint=/data
```

Chapter 15: Timers — Replacing Cron with systemd

systemd timers are unit files with the .timer extension. They trigger another unit (usually a .service unit with the same name) at specified times or intervals. Timers have several advantages over cron:

- Logging: timer activations appear in the journal
- Dependency handling: timers can have the same dependencies as services
- Missed execution handling: timers can run immediately if a scheduled time was missed while the system was off (AccuracySec=)
- Monotonic timers: trigger relative to system startup, not wall clock time
- Centralized management: all timers visible with systemctl list-timers

15.1 Timer and Service Pair

Timers almost always work with a companion service unit of the same name. For myapp-backup.timer, create myapp-backup.service:

The service unit (/etc/systemd/system/myapp-backup.service):

```
[Unit]
Description=MyApp Daily Backup

[Service]
Type=oneshot
User=myapp
ExecStart=/opt/myapp/scripts/backup.sh
```

The timer unit (/etc/systemd/system/myapp-backup.timer):

```
[Unit]
Description=MyApp Daily Backup Timer

[Timer]
OnCalendar=daily
Persistent=true

[Install]
WantedBy=timers.target
```

Enable the timer (not the service — the timer activates the service):

```
# systemctl enable --now myapp-backup.timer
# systemctl list-timers --all
```

15.2 [Timer] Section Directives

OnCalendar=

Wall clock time specification for when to trigger. Accepts a flexible calendar expression:

```
OnCalendar=daily              # midnight every day
OnCalendar=weekly             # Monday midnight
OnCalendar=monthly            # first day of month midnight
OnCalendar=hourly             # top of every hour
OnCalendar=*:0/15             # every 15 minutes
OnCalendar=Mon..Fri 09:00     # weekdays at 9 AM
OnCalendar=Sat 14:30          # Saturday at 2:30 PM
OnCalendar=2025-01-01 00:00   # specific date and time
OnCalendar=*-*-* 02:00:00     # every day at 2 AM (explicit)
OnCalendar=Mon *-*-* 00:00    # Monday midnight
```

Verify a calendar expression:

```
$ systemd-analyze calendar 'Mon..Fri 09:00'
  Original form: Mon..Fri 09:00
Normalized form: Mon..Fri *-*-* 09:00:00
    Next elapse: Mon 2025-01-13 09:00:00 UTC
       (in UTC): Mon 2025-01-13 09:00:00 UTC
       From now: 23h left
```

OnBootSec=

Trigger a specified time after the system boots. Used for tasks that should run once after startup.

```
OnBootSec=15min    # 15 minutes after boot
OnBootSec=1h       # 1 hour after boot
```

OnStartupSec=

Like OnBootSec= but relative to when systemd started (which may differ from system boot when systemd is restarted).

OnUnitActiveSec=

Trigger a specified time after the unit was last activated. Creates a repeating timer based on actual execution time rather than wall clock.

```
OnUnitActiveSec=1h  # 1 hour after last run
```

OnUnitInactiveSec=

Trigger a specified time after the unit was last deactivated (finished running).

```
OnUnitInactiveSec=30min
```

Persistent=

If yes and the timer missed a scheduled run (system was off), it triggers immediately on startup. Essential for backup timers that should not be silently skipped if the server was down at the scheduled time.

```
Persistent=true
```

AccuracySec=

How precisely the timer fires. Default is 1 minute. systemd uses this to group timers together for efficiency. If you need a timer to fire precisely at a specific second, set AccuracySec=1s. If approximate timing is fine, a larger value reduces wake-ups and saves power.

```
AccuracySec=1s     # Fire precisely at the scheduled time
AccuracySec=1h     # Can be off by up to an hour (for low-priority
tasks)
```

RandomizedDelaySec=

Add a random delay up to the specified duration before firing. Useful when you have many servers all running the same timer — randomizing prevents them from hammering a shared resource simultaneously.

```
RandomizedDelaySec=10min
```

Unit=

The unit to activate when the timer fires. Defaults to the service with the same name as the timer.

```
Unit=myapp-backup.service
```

15.3 Listing Active Timers

```
$ systemctl list-timers
```

Shows all active timers with columns NEXT (when it fires next), LEFT (time remaining), LAST (when it last fired), PASSED (how long ago), UNIT (timer name), and ACTIVATES (service it triggers).

```
$ systemctl list-timers --all
```

Shows all timers including inactive ones.

15.4 Running a Timer Immediately

To run the timer's service right now without waiting:

```
$ sudo systemctl start myapp-backup.service
```

Or trigger the timer immediately (bypassing time constraints):

```
$ sudo systemctl start myapp-backup.timer --trigger-now
```

Chapter 16: Socket Activation

Socket activation is one of systemd's most powerful features and one of the least understood. The idea: instead of a service listening on a port directly, systemd holds the socket open. When a connection arrives, systemd starts the service and hands it the socket. Services that have never received a connection use zero resources.

Benefits:

- Services start on demand: no wasted RAM for services that never receive connections
- Faster boot: all sockets are available immediately, even before their services start
- Reliable restarts: if a service crashes, the socket stays open and queues connections. When the service restarts, it picks up where it left off.
- Dependency simplification: services that communicate via sockets can start in any order

16.1 Socket and Service Pair

Socket activation requires a .socket unit and a corresponding .service unit. The socket unit (/etc/systemd/system/myapp.socket):

```
[Unit]
Description=MyApp Socket

[Socket]
ListenStream=8080

[Install]
WantedBy=sockets.target
```

The service unit (/etc/systemd/system/myapp.service) — note it does not need to bind the port:

```
[Unit]
Description=MyApp (socket activated)

[Service]
ExecStart=/opt/myapp/server
StandardInput=socket
```

Enable the socket (not the service):

```
# systemctl enable --now myapp.socket
```

16.2 [Socket] Section Directives

ListenStream=

TCP socket address to listen on. Can be a port number (all interfaces), an IP:port pair, or a Unix socket path.

```
ListenStream=8080               # All interfaces, port 8080
ListenStream=127.0.0.1:8080   # Localhost only
ListenStream=/run/myapp.sock  # Unix socket
```

ListenDatagram=

UDP socket address.

```
ListenDatagram=514  # syslog UDP
```

ListenFIFO=

A named pipe (FIFO).

```
ListenFIFO=/run/myapp.fifo
```

Accept=

If yes, a new instance of the service is spawned for each incoming connection. If no (default), the service handles connections itself. yes is used for inetd-style services.

```
Accept=no
```

Backlog=

The backlog for the listen() call. Default is 4096 on modern kernels.

```
Backlog=128
```

SocketUser= and SocketGroup=

Ownership of Unix domain sockets.

```
SocketUser=myapp
SocketGroup=myapp
```

SocketMode=

Permission mode of Unix domain sockets.

```
SocketMode=0600
```

Chapter 17: Path, Mount, Automount, Swap, and Device Units

17.1 Path Units (.path)

Path units monitor filesystem paths for changes and trigger other units when changes occur. Useful for watching a directory for new files and processing them automatically.

```
[Unit]
Description=Watch /var/spool/myapp for new files

[Path]
PathExistsGlob=/var/spool/myapp/*.job
Unit=myapp-process.service

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

Path unit directives:

- PathExists= — activate when the path exists
- PathExistsGlob= — activate when a glob pattern matches
- PathChanged= — activate when the path is modified
- PathModified= — activate on any change (including access time)
- DirectoryNotEmpty= — activate when the directory contains files
- MakeDirectory= — create the monitored directory if it does not exist

17.2 Mount Units (.mount)

Mount units represent filesystem mount points. They are often auto-generated from /etc/fstab, but can be written manually for mounts with complex systemd dependencies.

Unit file naming: the mount point path with slashes replaced by dashes. /mnt/data becomes mnt-data.mount.

```
[Unit]
Description=Mount /mnt/data

[Mount]
What=/dev/sdb1
Where=/mnt/data
Type=ext4
Options=defaults,noatime
```

```
[Install]
WantedBy=local-fs.target
```

17.3 Automount Units (.automount)

Automount units pair with mount units to mount filesystems on demand, when the mount point is first accessed. This prevents slow NFS mounts from blocking boot.

```
[Unit]
Description=Automount /mnt/nfs

[Automount]
Where=/mnt/nfs
TimeoutIdleSec=300

[Install]
WantedBy=remote-fs.target
```

Pair with mnt-nfs.mount. The filesystem mounts when /mnt/nfs is first accessed and unmounts after 300 seconds of idle time.

17.4 Swap Units (.swap)

Swap units manage swap files or partitions. Named after the swap path with slashes replaced by dashes.

```
[Unit]
Description=Swap on /dev/sda2

[Swap]
What=/dev/sda2
Priority=10

[Install]
WantedBy=swap.target
```

17.5 Device Units (.device)

Device units represent hardware devices. They are auto-generated by udev when devices appear. You do not usually write device unit files manually, but you reference them in dependencies:

```
After=dev-sdb1.device
BindsTo=dev-sdb1.device
```

Device unit names: /dev/sdb1 becomes dev-sdb1.device.

Chapter 18: Instantiated (Template) Units

Template units allow a single unit file to be used as a template for multiple instances with different parameters. The unit file name contains @ and the instance name is filled in when the unit is started.

18.1 Template Syntax

A template unit file has @ in its name before the extension: myapp@.service. An instance is: myapp@instance1.service, myapp@production.service, myapp@8080.service.

Template file (/etc/systemd/system/myapp@.service):

```
[Unit]
Description=MyApp instance %i
After=network-online.target

[Service]
Type=simple
User=myapp
ExecStart=/opt/myapp/server --port %i
Restart=on-failure

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

Specifiers available in template units:

- %i — the instance name as-is (e.g., 8080)
- %I — the instance name with escape sequences unescaped
- %n — full unit name including instance (e.g., myapp@8080.service)
- %N — unit name without type suffix
- %p — prefix name (the part before @)
- %u — the User= setting
- %h — the home directory of the User=
- %H — the hostname

18.2 Starting Template Instances

```
# systemctl start myapp@8080.service
# systemctl start myapp@8081.service
# systemctl enable myapp@8080.service
# systemctl status myapp@8080.service
```

18.3 Real-World Example: Multiple SSH Port Listeners

To run SSH on both port 22 and port 2222:

```
# systemctl start ssh@22.service
# systemctl start ssh@2222.service
```

This works because Debian ships ssh@.service as a template.

PART IV

JOURNALCTL

The Complete Log Reference

Chapter 19: How the Journal Works

19.1 The systemd Journal vs. Traditional Syslog

Traditional Linux logging wrote plain text to files in /var/log/. Each application chose its own format. Correlating events across services required grep, awk, and significant shell scripting. Timestamps were strings in various formats. Log rotation was a separate daemon (logrotate) with its own configuration.

The systemd journal (managed by journald) stores logs in a structured binary format. Every log entry includes:

- Timestamp with microsecond precision
- Service unit name
- Process ID
- User ID
- Priority / log level
- Message text
- Dozens of additional metadata fields (hostname, machine ID, kernel boot ID, etc.)

This structured format makes filtering fast and precise. journalctl -u nginx --since '10 minutes ago' -p err does in one command what would require multiple grep and awk pipelines on traditional log files.

19.2 Journal Storage

Journal files are stored in /run/systemd/journal/ (volatile, cleared on reboot) by default on many systems, or /var/log/journal/ (persistent) when persistent logging is enabled.

Check current journal storage:

```
$ journalctl --disk-usage
```

```
Archived and active journals take up 512.0M in the file system.
```

Check where the journal is stored:

```
$ ls /var/log/journal/ 2>/dev/null || echo 'Not persistent'
```

19.3 Who Can Read the Journal

By default on most distributions:

- root can read all journal entries
- Members of the adm group can read system journal entries
- Members of the systemd-journal group can read all journal entries
- Regular users can read their own user journal

To allow a user to read the full system journal:

```
# usermod -aG systemd-journal username
```

Chapter 20: journalctl — Every Flag and Filter

journalctl is the command for reading the systemd journal. Without any arguments, it shows all entries from oldest to newest, paged through less. This chapter covers every significant option.

20.1 Basic Usage

```
$ journalctl              # All entries, oldest first
$ journalctl -r           # All entries, newest first
$ journalctl -e           # Jump to end of journal
$ journalctl -f           # Follow (like tail -f)
$ journalctl -n 50        # Last 50 lines
$ journalctl -n 50 -r     # Last 50 lines, newest first
```

20.2 Filtering by Unit

```
$ journalctl -u nginx                      # All nginx logs
$ journalctl -u nginx -f                   # Follow nginx logs
$ journalctl -u nginx -n 100               # Last 100 nginx lines
$ journalctl -u nginx -u postgresql        # nginx AND postgresql
$ journalctl _SYSTEMD_UNIT=nginx.service   # Explicit field filter
```

20.3 Filtering by Time

```
$ journalctl --since '1 hour ago'
$ journalctl --since today
$ journalctl --since yesterday
$ journalctl --since '2025-01-13'
$ journalctl --since '2025-01-13 10:00' --until '2025-01-13 11:00'
$ journalctl --since '10 minutes ago'
$ journalctl -u nginx --since '2025-01-13 09:00' --until '2025-01-13
10:00'
```

20.4 Filtering by Priority

Log priorities follow the syslog convention (0 = most severe, 7 = least severe):

```
0 emerg    # System is unusable
1 alert    # Action must be taken immediately
2 crit     # Critical conditions
```

```
3 err      # Error conditions
4 warning  # Warning conditions
5 notice   # Normal but significant condition
6 info     # Informational
7 debug    # Debug messages
$ journalctl -p err           # Priority err and higher (0-3)
$ journalctl -p warning       # Priority warning and higher (0-4)
$ journalctl -p 3             # Same as -p err
$ journalctl -p err -p info   # Priority between err and info
$ journalctl -u nginx -p err  # nginx errors only
```

20.5 Filtering by Boot

```
$ journalctl -b               # Current boot
$ journalctl -b -1            # Previous boot
$ journalctl -b -2            # Two boots ago
$ journalctl --list-boots     # Show all recorded boots
$ journalctl -b 0             # Current boot (same as -b)
```

list-boots output:

```
IDX BOOT ID                          FIRST ENTRY                 LAST
ENTRY
 -2 dc722c908d5a43b4b83724ac87251295 Wed 2025-01-08 09:00:11 UTC Thu
2025-01-09 22:11:03 UTC
 -1 46b024e358c847e7a40bc936de0764ab Fri 2025-01-10 08:43:37 UTC Sun
2025-01-12 14:28:11 UTC
  0 5d3d17b012c94a5f9d3c13c8c5c7d1a2 Mon 2025-01-13 08:15:42 UTC Mon
2025-01-13 14:37:22 UTC
```

20.6 Filtering by Process, User, and Group

```
$ journalctl _PID=1247          # Logs from specific PID
$ journalctl _UID=1000          # Logs from specific user ID
$ journalctl _GID=1000          # Logs from specific group ID
$ journalctl _COMM=nginx        # Logs from processes named nginx
$ journalctl _EXE=/usr/sbin/nginx  # Logs from specific binary
```

20.7 Filtering by Kernel Messages

```
$ journalctl -k               # Kernel messages only (dmesg equivalent)
$ journalctl -k -b -1         # Kernel messages from previous boot
$ journalctl -k --since today # Today's kernel messages
$ journalctl -k -p err        # Kernel errors
```

20.8 Output Formats

```
$ journalctl -o short          # Default: short human-readable
$ journalctl -o short-iso      # ISO 8601 timestamps
$ journalctl -o short-precise  # Microsecond timestamps
$ journalctl -o short-full     # Full timestamps and all metadata
$ journalctl -o json           # JSON, one object per line
$ journalctl -o json-pretty    # JSON, formatted
$ journalctl -o verbose        # All fields, verbose
$ journalctl -o cat            # Message text only, no metadata
$ journalctl -o export         # Binary journal export format
```

JSON output is extremely useful for piping into jq or log aggregation systems:

```
$ journalctl -u nginx -o json-pretty -n 5 | jq '.MESSAGE'
```

20.9 Searching and Grepping

```
$ journalctl -g 'error'          # Grep for 'error' (case-sensitive)
$ journalctl -g 'error' -i       # Case-insensitive grep
$ journalctl -g 'timeout|refused' # Regex: timeout or refused
$ journalctl -u nginx | grep -i 'upstream'  # Pipe to grep
```

20.10 Catalog and Explanations

```
$ journalctl -x                   # Add explanatory catalog text
$ journalctl -xe                  # -x plus jump to end
$ journalctl -xe -u nginx         # Verbose nginx errors with
explanations
```

The -x flag adds human-readable explanations from the systemd message catalog for well-known messages. When a service fails with a known error code, -x often shows a description of what went wrong and suggestions for fixing it.

20.11 Combining Filters

Filters can be combined. Multiple -u flags are OR'd. Most other filters are AND'd.

```
# nginx errors in the last hour:
$ journalctl -u nginx -p err --since '1 hour ago'

# All errors from current boot, newest first:
$ journalctl -b -p err -r

# SSH login failures in the last day:
$ journalctl -u ssh --since yesterday -g 'Failed'

# Kernel errors from the previous boot:
$ journalctl -k -b -1 -p err
```

20.12 Field Filters

You can filter on any journal field by adding it as a positional argument in KEY=VALUE format:

```
$ journalctl PRIORITY=3                          # All errors
$ journalctl _HOSTNAME=web-01                    # Logs from specific
host
$ journalctl _SYSTEMD_UNIT=nginx.service PRIORITY=3  # nginx errors
```

List all available journal fields:

```
$ journalctl -N
```

Chapter 21: Persistent Logging and Journal Configuration

21.1 Enabling Persistent Logging

By default on some systems, the journal is stored in /run/systemd/journal/ which is volatile — lost on reboot. To enable persistent logging:

```
# mkdir -p /var/log/journal
# systemd-tmpfiles --create --prefix /var/log/journal
# systemctl restart systemd-journald
```

Or set Storage=persistent in journald.conf and restart journald.

21.2 /etc/systemd/journald.conf

The journal daemon is configured at /etc/systemd/journald.conf. Key settings:

```
[Journal]
Storage=persistent       # auto, volatile, persistent, none
Compress=yes             # Compress journal files (default yes)
SystemMaxUse=500M        # Max disk space for persistent journal
SystemKeepFree=100M      # Minimum free disk space to keep
SystemMaxFileSize=50M    # Max size of a single journal file
RuntimeMaxUse=100M       # Max disk space for volatile journal
MaxRetentionSec=1month   # Delete entries older than this
MaxFileSec=1week         # Rotate journal files at least weekly
ForwardToSyslog=no       # Whether to forward to syslog socket
ForwardToKMsg=no         # Forward to kernel log buffer
ForwardToConsole=no      # Forward to console
ForwardToWall=yes        # Forward emergency messages to wall
RateLimitIntervalSec=30s # Rate limit window
RateLimitBurst=10000     # Max messages per window
```

After editing journald.conf:

```
# systemctl restart systemd-journald
```

21.3 Vacuuming Old Journal Data

```
# journalctl --vacuum-size=500M    # Reduce to 500MB
# journalctl --vacuum-time=1month  # Delete entries older than 1 month
# journalctl --vacuum-files=10     # Keep only 10 journal files
```

21.4 Rotating the Journal

```
# journalctl --rotate
```

Forces journal rotation immediately, archiving current journal files and starting fresh.

Chapter 22: Log Forwarding and Remote Journals

22.1 Forwarding to Traditional Syslog

To forward journal entries to a traditional syslog daemon (rsyslog, syslog-ng), ensure ForwardToSyslog=yes in journald.conf. Most distributions ship this enabled by default for compatibility.

22.2 systemd-journal-remote

systemd-journal-remote allows you to collect journal entries from multiple machines on a central log server.

On the remote machines (journal senders), install systemd-journal-upload:

```
# apt install systemd-journal-remote
# systemctl enable --now systemd-journal-upload
```

Configure /etc/systemd/journal-upload.conf:

```
[Upload]
URL=http://logserver:19532
```

On the log server:

```
# apt install systemd-journal-remote
# systemctl enable --now systemd-journal-remote.socket
```

Journals from remote machines are stored in /var/log/journal/remote/ and can be read with journalctl -D:

```
$ journalctl -D /var/log/journal/remote/
```

PART V

SECURITY AND HARDENING

Sandboxing Services with systemd

Chapter 23: Service Sandboxing Directives

systemd provides a rich set of sandboxing directives that restrict what a service can access, significantly reducing the impact of a compromised service. These directives are applied in the [Service] section.

23.1 Filesystem Sandboxing

ProtectSystem=

Makes system directories read-only for the service.

- strict — /usr, /boot, and /etc are read-only. Best choice for most services.
- full — /usr and /boot are read-only.
- true — /usr and /boot are read-only (same as full).

```
ProtectSystem=strict
```

ProtectHome=

Restricts access to user home directories.

- true — /home, /root, and /run/user are inaccessible
- read-only — home directories are visible but read-only
- tmpfs — home directories are replaced with an empty tmpfs

```
ProtectHome=true
```

ReadWritePaths= and ReadOnlyPaths=

Used with ProtectSystem= to grant exceptions. If ProtectSystem=strict makes /etc read-only, use ReadWritePaths= to grant write access to a specific subdirectory.

```
ProtectSystem=strict
ReadWritePaths=/etc/myapp /var/lib/myapp /var/log/myapp
```

InaccessiblePaths=

Makes specific paths completely inaccessible to the service.

```
InaccessiblePaths=/home /root /boot /etc/ssh
```

PrivateTmp=

Gives the service a private /tmp and /var/tmp, isolated from the rest of the system. Files created in /tmp by the service are not visible to other processes and are cleaned up when the service stops.

```
PrivateTmp=true
```

PrivateDevices=

Gives the service a private /dev with only pseudo-devices (null, zero, random, urandom, tty, pts). Physical devices are inaccessible.

```
PrivateDevices=true
```

PrivateNetwork=

Gives the service a private network namespace with no network access (only loopback). For services that do not need network access.

```
PrivateNetwork=true
```

PrivateUsers=

Gives the service a private user namespace where the service sees only itself as a user. External UIDs appear as nobody.

```
PrivateUsers=true
```

23.2 Capability Restrictions

CapabilityBoundingSet=

Limits which Linux capabilities the service can have. Use ~ prefix to indicate 'all except these':

```
# Allow only network binding capability
CapabilityBoundingSet=CAP_NET_BIND_SERVICE

# Strip all capabilities
CapabilityBoundingSet=

# Strip specific capabilities
CapabilityBoundingSet=~CAP_SYS_ADMIN CAP_NET_RAW
```

AmbientCapabilities=

Grants capabilities to the service even when running as a non-root user. Use this to allow a non-root service to bind to ports below 1024 without running as root.

```
User=myapp
AmbientCapabilities=CAP_NET_BIND_SERVICE
```

23.3 System Call Filtering

SystemCallFilter=

Restricts which system calls the service can make. An application that does not need to fork() or exec() should not be allowed to. systemd provides predefined sets:

```
SystemCallFilter=@system-service    # Calls appropriate for normal
daemons
SystemCallFilter=@network-io        # Network I/O calls
SystemCallFilter=~@debug            # Deny debug system calls
SystemCallFilter=~@reboot           # Deny reboot calls
SystemCallFilter=~@raw-io           # Deny raw I/O (disk access)
```

List available system call sets:

```
$ systemd-analyze syscall-filter
```

SystemCallArchitectures=

Restricts which CPU architectures the service can use for system calls. Prevents 32-bit system call exploits on 64-bit systems.

```
SystemCallArchitectures=native
```

23.4 Network Restrictions

RestrictAddressFamilies=

Restricts which network address families the service can use.

```
RestrictAddressFamilies=AF_INET AF_INET6   # Only IPv4 and IPv6
RestrictAddressFamilies=AF_UNIX            # Only Unix sockets
RestrictAddressFamilies=none               # No networking at all
```

23.5 Recommended Baseline for Any Service

This block provides a solid security baseline for a typical web service running as a dedicated user:

```
[Service]
User=myapp
Group=myapp
PrivateTmp=true
PrivateDevices=true
ProtectSystem=strict
ProtectHome=true
ReadWritePaths=/var/lib/myapp /var/log/myapp
NoNewPrivileges=true
CapabilityBoundingSet=
SystemCallFilter=@system-service
SystemCallArchitectures=native
RestrictAddressFamilies=AF_INET AF_INET6
RestrictNamespaces=true
LockPersonality=true
MemoryDenyWriteExecute=true
RestrictRealtime=true
```

23.6 Checking the Security Score

```
$ systemd-analyze security nginx.service
```

systemd-analyze security rates a service's sandboxing from 0 (fully hardened) to 10 (exposed). It lists which security features are not enabled and the risk each represents. Use it to guide hardening efforts.

Chapter 24: User Services and the User Session

systemd supports per-user service managers, separate from the system service manager. User services run without root privileges and are managed by the user's own systemd instance.

24.1 User Unit File Locations

```
~/.config/systemd/user/         # User-created units
/usr/lib/systemd/user/          # Package-installed user units
/etc/systemd/user/              # Admin-installed user units
```

24.2 Managing User Services

```
$ systemctl --user status
$ systemctl --user start myapp.service
$ systemctl --user enable myapp.service
$ journalctl --user -u myapp.service
```

The --user flag tells systemctl to operate on the user's session manager rather than the system manager.

24.3 Lingering: Running After Logout

By default, a user's services stop when they log out. To keep services running after logout:

```
# loginctl enable-linger username
# loginctl disable-linger username
```

Lingering starts the user's systemd manager at boot, even without a login session. Essential for user-level services that should run on a server.

Chapter 25: Credentials, Secrets, and LoadCredential

systemd 247+ introduced a credentials system for passing secrets to services without storing them in unit files (which may be world-readable) or environment variables (which are visible in /proc).

25.1 LoadCredential=

Load a credential from a file and pass it to the service. The service accesses it as a file at $CREDENTIALS_DIRECTORY/<name>.

```
[Service]
LoadCredential=db-password:/etc/myapp/secrets/db-password
```

In the service, read the credential:

```
#!/bin/bash
DB_PASSWORD=$(cat "$CREDENTIALS_DIRECTORY/db-password")
```

25.2 SetCredential=

Set a credential value directly in the unit file. The credential is not stored as a plaintext file but is passed securely to the service. Less secure than LoadCredential= (it appears in the unit file) but more convenient for non-sensitive values.

```
[Service]
SetCredential=config-port:8080
```

25.3 ImportCredential=

Import credentials passed to systemd at boot (for example, from a TPM or early boot secret injection).

```
ImportCredential=*
```

PART VI

PERFORMANCE AND TROUBLESHOOTING

Diagnosing and Fixing systemd Problems

Chapter 26: systemd-analyze — Boot Performance Profiling

26.1 Basic Boot Time

```
$ systemd-analyze
Startup finished in 1.423s (kernel) + 3.871s (userspace) = 5.294s
graphical.target reached after 3.841s in userspace
```

Shows total boot time split between kernel initialization and userspace (systemd) initialization.

26.2 Blame: Slowest Units

```
$ systemd-analyze blame
          3.421s NetworkManager-wait-online.service
          1.234s snapd.service
           891ms dev-sda1.device
           654ms apt-daily.service
           421ms systemd-resolved.service
           312ms motd-news.service
...
```

Lists units sorted by how long they took to start, from slowest to fastest. The top entries are your targets for boot time improvement.

> **NOTE:** *NetworkManager-wait-online.service is almost always in the top 3 on servers with DHCP. If your services do not need the network at boot, disable it: sudo systemctl disable NetworkManager-wait-online.service*

26.3 Critical Chain

```
$ systemd-analyze critical-chain
The time when unit became active or started is printed after the '@'
character.
The time the unit took to start is printed after the '+' character.

graphical.target @3.841s
  └─multi-user.target @3.841s
    └─nginx.service @3.654s +0.186s
      └─network-online.target @3.650s
        └─NetworkManager-wait-online.service @0.231s +3.418s
          └─NetworkManager.service @0.221s +0.009s
            └─dbus.service @0.198s +0.021s
              └─basic.target @0.195s
```

Shows the longest dependency chain that determined when the default target was reached. The unit at the top of the chain is what you need to optimize.

26.4 Analyzing a Specific Unit

```
$ systemd-analyze critical-chain nginx.service
```

Shows the critical chain leading to a specific unit.

26.5 Generating a Boot SVG

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

Generates a Gantt chart SVG showing all units and their start times overlaid on a timeline. Open in a browser or SVG viewer to visualize parallelization and identify bottlenecks.

26.6 Verifying Unit Files

```
$ systemd-analyze verify myapp.service
$ systemd-analyze verify /etc/systemd/system/myapp.service
```

Checks a unit file for syntax errors and common mistakes without requiring a daemon-reload. Returns errors and warnings. Run this before deploying a new unit file.

26.7 Checking Security Score

```
$ systemd-analyze security nginx.service
  NAME
DESCRIPTION
✗ RootDirectory=/RootImage=                                    Service
runs within the host's root directory
```

```
✗ SupplementaryGroups=                                          Service
runs with supplementary groups
...
→ Overall exposure level for nginx.service: 9.6 UNSAFE 🔥
```

Rates the service on 0–10 security scale and lists unimplemented hardening features. Use this to prioritize security improvements.

Chapter 27: Troubleshooting Failed Units

This is the chapter you reach for when something is broken. Follow this sequence for any failed unit.

Step 1: Identify Failed Units

```
$ systemctl --failed
```

Lists every unit in a failed state. Start here.

Step 2: Check Unit Status

```
$ systemctl status myapp.service
```

Read the Active line carefully. Failed services often show the exit code:

```
Active: failed (Result: exit-code) since Mon 2025-01-13 10:23:44 UTC;
5min ago
Process: 4821 ExecStart=/opt/myapp/server (code=exited,
status=1/FAILURE)
```

The last 10 journal lines are shown at the bottom of status output. These often contain the actual error message.

Step 3: Read the Full Logs

```
$ journalctl -u myapp.service -n 100
$ journalctl -u myapp.service -b
$ journalctl -u myapp.service --since '1 hour ago'
$ journalctl -xe -u myapp.service
```

The -xe flag adds catalog explanations which often include suggestions for common errors.

Step 4: Check Exit Code and Signal

The exit code or signal in the status output is your next clue:

- status=1 — application exited with error. Check application logs for the reason.
- status=2 — often a shell built-in error. Check ExecStart= path is correct.
- status=203/EXEC — cannot execute binary. Check the path exists and is executable.

- status=217/USER — User= or Group= does not exist.
- status=218/CAPABILITIES — capability error. Service needs a capability it does not have.
- status=226/NAMESPACE — namespace setup failed. Check PrivateTmp= and related settings.
- killed with SIGKILL (9) — timeout exceeded. Increase TimeoutStartSec= or TimeoutStopSec=.
- killed with SIGSEGV (11) — application crashed. Enable LimitCORE=infinity for coredump.

Step 5: Test the Command Manually

Run the ExecStart= command directly as the service user to see errors interactively:

```
$ sudo -u myapp /opt/myapp/server --config /etc/myapp/config.yaml
```

If it fails, you see the error directly. If it succeeds, the problem is in the systemd environment (missing environment variables, different working directory, etc.).

Step 6: Reset Failed State

After fixing the issue, clear the failed state and start the service:

```
$ sudo systemctl reset-failed myapp.service
$ sudo systemctl start myapp.service
```

Common Problems and Solutions

Service fails immediately with status=203/EXEC

```
# Check the binary exists and is executable:
$ ls -la /opt/myapp/server
$ which /opt/myapp/server

# Check the shebang line if it is a script:
$ head -1 /opt/myapp/server
```

Service fails due to permissions

```
# Check ownership of files the service needs:
$ ls -la /var/lib/myapp /var/log/myapp /etc/myapp
```

```
# Fix ownership:
# chown -R myapp:myapp /var/lib/myapp
```

Service keeps restarting (start limit hit)

```
$ systemctl status myapp.service
# Look for: 'start request repeated too quickly'

# Reset and try again:
$ sudo systemctl reset-failed myapp.service
$ sudo systemctl start myapp.service

# Or increase the limit in a drop-in:
# [Unit]
# StartLimitBurst=20
# StartLimitIntervalSec=300s
```

Service starts but is not ready (timing issues)

```
# If using Type=simple and the service is not fully ready
# when dependents try to connect, change to Type=notify
# and add sd_notify(READY=1) to the application,
# or use ExecStartPost= to wait for readiness:

[Service]
Type=simple
ExecStart=/opt/myapp/server
ExecStartPost=/usr/bin/curl --retry 5 --retry-delay 1 -sf
http://localhost:8080/health
```

Chapter 28: cgroups and Resource Control

Control groups (cgroups) allow you to limit CPU, memory, and I/O usage for individual services. systemd exposes cgroup controls as unit file directives. All distributions covered in this book use the unified cgroup v2 hierarchy exclusively. There is no cgroup v1 on Debian 13 or Ubuntu 26.04. Every directive and path described in this chapter uses the v2 API.

28.1 Memory Limits

MemoryMax=

Hard memory limit. If the service exceeds this, the OOM killer kills processes in the service. Accepts bytes, K, M, G suffixes, or a percentage of physical RAM.

```
MemoryMax=512M
```

MemoryHigh=

Soft memory limit. If the service exceeds this, it is throttled and the kernel applies memory pressure, but processes are not killed.

```
MemoryHigh=400M
```

MemorySwapMax=

Maximum swap usage. Set to 0 to disable swap for this service.

```
MemorySwapMax=0
```

28.2 CPU Limits

CPUQuota=

Maximum CPU time as a percentage of one CPU core. 100% means one full core. 200% means two cores. 50% means half a core.

```
CPUQuota=50%
```

CPUWeight=

Relative CPU weight for scheduling. Default is 100. Higher values get more CPU when there is contention.

```
CPUWeight=500  # High priority service
```

AllowedCPUs=

Pin the service to specific CPU cores.

```
AllowedCPUs=0-3  # Only use cores 0, 1, 2, 3
```

28.3 I/O Limits

IOWeight=

Relative I/O scheduling weight. Default is 100.

```
IOWeight=10  # Low I/O priority
```

IOReadBandwidthMax= and IOWriteBandwidthMax=

Maximum read/write bandwidth for a device. Format: device path followed by bandwidth.

```
IOReadBandwidthMax=/dev/sda 50M
IOWriteBandwidthMax=/dev/sda 20M
```

28.4 Viewing cgroup Resource Usage

```
$ systemctl status nginx.service
# Shows Memory and CPU in the status output

$ systemd-cgtop
# Live view of cgroup resource usage (like top for cgroups)

$ cat /sys/fs/cgroup/system.slice/nginx.service/memory.current
# Raw cgroup memory usage in bytes
```

Chapter 29: Common Real-World Scenarios and Solutions

Scenario 1: Run a Script at Boot

Create a oneshot service that runs your script at boot and does not need to stay running:

```
[Unit]
Description=Set system parameters at boot
After=local-fs.target

[Service]
Type=oneshot
RemainAfterExit=yes
ExecStart=/usr/local/bin/boot-setup.sh

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

Scenario 2: Service That Depends on a Database

```
[Unit]
Description=Web Application
After=network-online.target postgresql.service
Wants=network-online.target
Requires=postgresql.service

[Service]
Type=notify
User=webapp
ExecStart=/opt/webapp/server
Restart=on-failure
RestartSec=10s
```

Scenario 3: Run a Task Every Hour

myapp-cleanup.service:

```
[Unit]
Description=MyApp Hourly Cleanup

[Service]
Type=oneshot
User=myapp
ExecStart=/opt/myapp/cleanup.sh
```

myapp-cleanup.timer:

```
[Unit]
Description=Run MyApp cleanup hourly

[Timer]
OnCalendar=hourly
Persistent=true
RandomizedDelaySec=5min

[Install]
WantedBy=timers.target
```

Scenario 4: Notify on Service Failure

Create a notification service template (notify-failed@.service):

```
[Unit]
Description=Notify on failed service %i

[Service]
Type=oneshot
ExecStart=/usr/local/bin/notify-failure.sh %i
```

Then add to any service:

```
OnFailure=notify-failed@%n.service
```

Scenario 5: Override a Package Service Timeout

```
# Increase start timeout for a slow-starting service
# /etc/systemd/system/postgresql.service.d/timeout.conf:

[Service]
TimeoutStartSec=300s
# systemctl daemon-reload
# systemctl restart postgresql
```

Scenario 6: Run Service in a Specific Network Namespace

```
[Service]
PrivateNetwork=true
NetworkNamespacePath=/run/netns/restricted
```

Scenario 7: Bind Service to a Hardware Device

Start a service only when a USB device is connected:

```
[Unit]
Description=Service for USB device
BindsTo=dev-bus-usb-001-002.device
After=dev-bus-usb-001-002.device

[Service]
ExecStart=/usr/local/bin/handle-device
```

PART VII

ADVANCED TOPICS

Ancillary systemd Tools and Production Practices

Chapter 30: Containers and systemd-nspawn

systemd-nspawn is a lightweight container manager built into systemd. It is more capable than a chroot and simpler than Docker, providing namespace isolation, cgroup integration, and full systemd support inside the container.

30.1 Creating a Container

```
# Install debootstrap to create a Debian container:
# apt install debootstrap

# Create a Debian Trixie container:
# debootstrap trixie /var/lib/machines/mycontainer
http://deb.debian.org/debian

# Start the container:
# systemd-nspawn -D /var/lib/machines/mycontainer
```

30.2 Starting Containers as Services

```
# machinectl start mycontainer
# machinectl status mycontainer
# machinectl stop mycontainer
# machinectl enable mycontainer  # Start at boot
```

Containers in /var/lib/machines/ are automatically managed as systemd-nspawn@<name>.service instances.

30.3 Executing Commands in a Container

```
# machinectl shell mycontainer
# machinectl shell mycontainer /bin/bash
# systemd-nspawn -D /var/lib/machines/mycontainer /usr/bin/apt update
```

Chapter 31: systemd-networkd and systemd-resolved

31.1 systemd-networkd

systemd-networkd is a network configuration daemon, an alternative to NetworkManager for servers. It is especially well-suited to servers and containers where simple, predictable network configuration is needed.

Enable it:

```
# systemctl enable --now systemd-networkd
```

Network configurations go in /etc/systemd/network/ with .network extension:

```
# /etc/systemd/network/10-ether.network
[Match]
Name=eth0

[Network]
DHCP=yes
```

For static IP:

```
# /etc/systemd/network/10-ether.network
[Match]
Name=eth0

[Network]
Address=192.168.1.100/24
Gateway=192.168.1.1
DNS=1.1.1.1 8.8.8.8
# networkctl status
```

31.2 systemd-resolved

systemd-resolved is a DNS resolver with caching, DNSSEC validation, and mDNS support.

```
# systemctl enable --now systemd-resolved
```

Configure in /etc/systemd/resolved.conf:

```
[Resolve]
DNS=1.1.1.1 8.8.8.8
FallbackDNS=9.9.9.9
DNSSEC=yes
Cache=yes
```

Check DNS resolution status:

```
$ resolvectl status
```

```
$ resolvectl query example.com
$ resolvectl statistics
```

Chapter 32: Time Synchronization — systemd-timesyncd, chrony, and timedatectl

32.1 systemd-timesyncd

A lightweight NTP client for time synchronization. On most systems it is enabled by default.

```
# systemctl status systemd-timesyncd
$ timedatectl timesync-status
```

Configure NTP servers in /etc/systemd/timesyncd.conf:

```
[Time]
NTP=time.cloudflare.com 0.pool.ntp.org 1.pool.ntp.org
FallbackNTP=ntp.ubuntu.com
RootDistanceMaxSec=5
PollIntervalMinSec=32
PollIntervalMaxSec=2048
```

32.2 timedatectl

```
$ timedatectl                        # Show current time settings
$ timedatectl list-timezones         # List available timezones
$ sudo timedatectl set-timezone America/Chicago
$ sudo timedatectl set-ntp true      # Enable NTP sync
$ sudo timedatectl set-time '2025-01-13 10:30:00'  # Set time manually
```

Chapter 33: systemd-logind and loginctl

systemd-logind manages user login sessions, seats, and power management.

33.1 loginctl Commands

```
$ loginctl                          # List current sessions
$ loginctl list-sessions            # Same
$ loginctl session-status 3         # Details on session 3
$ loginctl list-users               # List logged-in users
$ loginctl user-status username     # Details on a user
$ loginctl terminate-session 3      # Kill a session
$ loginctl terminate-user username  # Kill all sessions for user
# loginctl enable-linger username   # Enable linger
# loginctl disable-linger username  # Disable linger
```

33.2 Power Management via logind

logind handles power button, lid switch, and idle actions. Configure in /etc/systemd/logind.conf:

```
[Login]
HandlePowerKey=poweroff
HandleSuspendKey=suspend
HandleLidSwitch=suspend
HandleLidSwitchDocked=ignore
IdleAction=ignore
IdleActionSec=30min
```

Chapter 34: systemd-tmpfiles

systemd-tmpfiles manages temporary files and directories: creating, deleting, and cleaning them at the right times.

34.1 tmpfiles.d Configuration

Configuration files go in /etc/tmpfiles.d/ with .conf extension. The format is:

```
<type> <path> <mode> <user> <group> <age> <argument>
```

Common types:

- d — create directory
- D — create directory and clean contents older than age
- f — create file if it does not exist
- L — create symlink
- z — set permissions on existing path
- t — set extended attributes

Example: create a runtime directory for myapp:

```
d /run/myapp 0755 myapp myapp -
```

Create a directory and clean files older than 7 days:

```
D /var/cache/myapp 0755 myapp myapp 7d
```

Apply immediately:

```
# systemd-tmpfiles --create /etc/tmpfiles.d/myapp.conf
# systemd-tmpfiles --clean /etc/tmpfiles.d/myapp.conf
```

Chapter 35: systemd in Production — Best Practices

35.1 Unit File Best Practices

- Always run systemd-analyze verify before deploying a new unit file
- Use systemctl edit (drop-ins) to customize package units, never edit /usr/lib/systemd/system/ files directly
- Always run systemctl daemon-reload after any unit file change
- Use Type=notify for services you control, so systemd knows when they are truly ready
- Set Restart=on-failure with a reasonable RestartSec= for all long-running services
- Set StartLimitBurst= and StartLimitIntervalSec= to prevent crash loops
- Use After=network-online.target only when the service genuinely needs network connectivity at startup
- Document your units: use Description=, Documentation=, and comments

35.2 Security Best Practices

- Always run services as dedicated non-root users with User= and Group=
- Always set PrivateTmp=true on services that use temp files
- Always set NoNewPrivileges=true unless the service explicitly needs to gain privileges
- Run systemd-analyze security <service> on every service and address high-risk items
- Use CapabilityBoundingSet= to strip unneeded capabilities
- Use ProtectSystem=strict and ReadWritePaths= to minimize filesystem access
- Use DynamicUser=yes for stateless oneshot services

35.3 Logging Best Practices

- Enable persistent logging on all production servers
- Set SystemMaxUse= in journald.conf to prevent journal from consuming all disk space
- Use SyslogIdentifier= to set a clear log identifier for custom services

- On multi-server environments, configure log forwarding to a central log server
- Use journalctl -b -p err as a daily check for unexpected errors
- Set up alerts based on journal priority levels for critical services

35.4 Operational Best Practices

- Monitor failed units regularly: systemctl --failed
- Use systemctl status to check a service before and after changes
- Test timer calendars with systemd-analyze calendar before deploying
- Back up /etc/systemd/system/ as part of system configuration backup
- Use machinectl or systemd-nspawn for isolation rather than running multiple services in one unit
- Read journalctl -xe after any failed system operation — the catalog explanations save hours

Appendix A: Complete systemctl Command Reference

Service Lifecycle (All commands verified on systemd 257–259)

```
systemctl start <unit>              Start a unit
systemctl stop <unit>               Stop a unit
systemctl restart <unit>            Stop then start a unit
systemctl reload <unit>             Reload configuration (SIGHUP)
systemctl reload-or-restart <unit>  Reload if supported, else restart
systemctl enable <unit>             Enable at boot
systemctl disable <unit>            Disable at boot
systemctl enable --now <unit>       Enable and start immediately
systemctl disable --now <unit>      Disable and stop immediately
systemctl mask <unit>               Prevent any starting
systemctl unmask <unit>             Remove mask
systemctl preset <unit>             Apply vendor default enable/disable
```

Status and Inspection

```
systemctl status <unit>             Full status with recent logs
systemctl is-active <unit>          Check if active (exit code too)
systemctl is-enabled <unit>         Check if enabled
systemctl is-failed <unit>          Check if failed
systemctl show <unit>               All properties (machine-readable)
systemctl cat <unit>                Print unit file and drop-ins
systemctl list-dependencies <unit>  Show dependency tree
systemctl list-units                All loaded units
systemctl list-unit-files           All unit files on disk
systemctl list-timers               All active timers
systemctl --failed                  All failed units
```

Editing and Reloading

```
systemctl edit <unit>               Create/edit drop-in override
systemctl edit --full <unit>        Edit full unit file copy
systemctl revert <unit>             Remove overrides, restore original
systemctl daemon-reload             Reload all unit files from disk
systemctl daemon-reexec             Reexecute systemd binary
systemctl reset-failed              Clear all failed states
systemctl reset-failed <unit>       Clear failed state for one unit
```

System State

```
systemctl reboot                    Reboot the system
systemctl poweroff                  Power off the system
systemctl halt                      Halt the system
systemctl suspend                   Suspend to RAM
systemctl hibernate                 Suspend to disk
systemctl get-default               Show default target
systemctl set-default <target>      Set default target
systemctl isolate <target>          Switch to target now
systemctl rescue                    Switch to rescue mode
systemctl emergency                 Switch to emergency mode
```

Process Control

```
systemctl kill <unit>               Send signal to service
systemctl kill --kill-whom=all <unit>  Kill all processes in unit
systemctl kill --signal=SIGKILL <unit> Send SIGKILL
systemctl freeze <unit>             Pause all processes (systemd 252+)
systemctl thaw <unit>               Resume paused processes
```

Appendix B: Complete journalctl Flag Reference

```
DISPLAY
  -n N, --lines=N           Show last N lines
  -r, --reverse             Newest entries first
  -e, --pager-end           Jump to end of journal
  -f, --follow              Follow new entries (like tail -f)
  -a, --all                 Show all fields, including long ones
  -x, --catalog             Add explanatory catalog text
  -q, --quiet               Suppress informational messages
  --no-pager                Don't use a pager
  --no-hostname             Don't show hostname
  --utc                     Show timestamps in UTC

OUTPUT FORMAT
  -o short           Default short format
  -o short-iso       ISO 8601 timestamps
  -o short-precise   Microsecond timestamps
  -o short-full      Full timestamps
  -o json            JSON one object per line
  -o json-pretty     Formatted JSON
  -o verbose         All fields
  -o cat             Message text only
  -o export          Binary export format

FILTERING BY SOURCE
  -u, --unit=UNIT     Filter by systemd unit
  -t, --identifier=ID Filter by syslog identifier
  -k, --dmesg         Show kernel messages only
  --user              Show user session journal
  --system            Show system journal
  -m, --merge         Show all available journals merged
  -M, --machine=NAME  Show journal from container NAME

FILTERING BY TIME
  -b, --boot[=OFFSET] Show entries from boot (offset 0,-1,-2...)
  --list-boots        List all recorded boots
  --since=DATE        Show entries since DATE
  --until=DATE        Show entries until DATE

FILTERING BY PRIORITY
  -p, --priority=LEVEL  Show entries at LEVEL and above
                        0=emerg 1=alert 2=crit 3=err
                        4=warning 5=notice 6=info 7=debug

SEARCHING
  -g, --grep=PATTERN  Filter by message pattern (regex)
  -i                  Case-insensitive grep

DISK MANAGEMENT
```

```
  --disk-usage         Show disk space used by journal
  --vacuum-size=SIZE   Reduce journal to SIZE
  --vacuum-time=TIME   Delete entries older than TIME
  --vacuum-files=N     Keep only N journal files
  --rotate             Rotate journal files
  --sync               Flush unwritten data to disk

OTHER
  -N                   List all known journal fields
  -F FIELD             Show all values for a field
  -D DIR               Read journal from directory DIR
  --file=FILE          Read specific journal file
```

Appendix C: Unit File Directive Quick Reference

[Unit] Section

```
Description=       Human-readable description
Documentation=     URI list for documentation
Wants=             Soft dependency
Requires=          Hard dependency
Requisite=         Must already be active
BindsTo=           Stop when bound unit stops
PartOf=            Stop/restart with listed unit
Conflicts=         Cannot run simultaneously
After=             Start after listed unit(s)
Before=            Start before listed unit(s)
OnFailure=         Unit to start on failure
OnSuccess=         Unit to start on success
ConditionPathExists=  Start only if path exists
ConditionHost=     Start only on named host
ConditionVirtualization=  Start based on virt type
DefaultDependencies=  Enable/disable auto-deps
```

[Service] Section

```
Type=              simple|exec|forking|oneshot|notify|dbus|idle
ExecStart=         Command to start the service
ExecStartPre=      Command before ExecStart
ExecStartPost=     Command after ExecStart
ExecStop=          Command to stop the service
ExecStopPost=      Command after stop
ExecReload=        Command for reload
Restart=           no|on-success|on-failure|always
RestartSec=        Delay before restart
StartLimitBurst=   Max restarts before giving up
StartLimitIntervalSec= Window for StartLimitBurst
TimeoutStartSec=   Start timeout
TimeoutStopSec=    Stop timeout
WatchdogSec=       Watchdog timeout
User=              Run as this user
Group=             Run as this group
DynamicUser=       Allocate temporary user
WorkingDirectory=  Working directory
Environment=       Set environment variable
EnvironmentFile=   Read environment from file
StandardOutput=    Stdout destination
StandardError=     Stderr destination
SyslogIdentifier=  Journal identifier
LimitNOFILE=       Open file descriptor limit
PIDFile=           PID file location (forking)
```

```
RemainAfterExit=   Stay active after exit
PrivateTmp=        Private /tmp
PrivateDevices=    Private /dev
PrivateNetwork=    Private network namespace
ProtectSystem=     Read-only system paths
ProtectHome=       Restrict home dirs
NoNewPrivileges=   Cannot gain privileges
CapabilityBoundingSet=  Allowed capabilities
SystemCallFilter=  Allowed system calls
MemoryMax=         Hard memory limit
CPUQuota=          CPU usage limit (percentage)
```

[Install] Section

```
WantedBy=          Target to link to on enable
RequiredBy=        Required by target on enable
Also=              Enable alongside this unit
Alias=             Alternative unit name
```

Appendix D: systemd Exit Codes

When a service fails, systemctl status shows an exit code. Here are the most common:

```
Exit Code    Meaning
──────────────────────────────────────────────────────────────────────
0            Success
1            Generic failure (application-specific)
2            Shell built-in misuse / argument error
203/EXEC     Cannot execute binary (wrong path, not executable)
205/NAMESPACE  Namespace setup failed
206/CHDIR    Cannot change to WorkingDirectory=
207/CHROOT   Cannot chroot to RootDirectory=
210/USER     User= or Group= does not exist
211/CAPABILITIES  Required capability not available
212/CGROUP   cgroup setup failed
216/GROUP    Group= does not exist
217/USER     User= does not exist
218/CAPABILITIES  CapabilityBoundingSet= error
219/NAMESPACE  PrivateNetwork/PrivateTmp setup failed
220/SECCOMP  SystemCallFilter= error
221/SETSCHED Scheduling setup failed
222/CPUAFFINITY  AllowedCPUs= setup failed
223/NUMA     NUMA setup failed
224/IOPRIO   IOSchedulingClass= setup failed
226/NAMESPACE  Namespace error (detailed)
228/EXEC     Executable not found
238/RUNTIME-DIRECTORY  RuntimeDirectory= creation failed
239/ENVIRONMENT  EnvironmentFile= reading failed
240/CREDENTIAL  LoadCredential= failed
241/FDSTORE  File descriptor store error
242/STATE-DIRECTORY  StateDirectory= creation failed
```

Appendix E: Migrating SysV Init Scripts to systemd Unit Files

If you are converting an old SysV init script to a systemd unit file, this appendix maps the old patterns to the new ones. This is now urgent: Ubuntu 26.04 is the last LTS release with SysV compatibility, and systemd 260 removes the compatibility layer entirely. Any remaining /etc/init.d/ scripts must be converted before your next major distribution upgrade.

E.1 Mapping chkconfig / update-rc.d to systemd

```
SysV Command                    systemd Equivalent
─────────────────────────────────────────────────────────────
chkconfig myapp on              systemctl enable myapp.service
chkconfig myapp off             systemctl disable myapp.service
chkconfig --list myapp          systemctl is-enabled myapp.service
service myapp start             systemctl start myapp.service
service myapp stop              systemctl stop myapp.service
service myapp restart           systemctl restart myapp.service
service myapp status            systemctl status myapp.service
service myapp reload            systemctl reload myapp.service
```

E.2 Converting an init.d Script

A typical SysV init script for a Python daemon:

```
#!/bin/bash
# chkconfig: 345 85 15
# description: MyApp web server
case "$1" in
  start)
    su - myapp -c '/opt/myapp/server &'
    ;;
  stop)
    kill $(cat /var/run/myapp.pid)
    ;;
esac
```

Becomes this unit file:

```
[Unit]
Description=MyApp web server
After=network-online.target
Wants=network-online.target

[Service]
Type=simple
```

```
User=myapp
ExecStart=/opt/myapp/server
Restart=on-failure

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

Key differences:

- No shell script needed — just the binary path
- User= replaces 'su - myapp'
- Restart= replaces manual process monitoring
- After= and Wants= replace the chkconfig runlevel numbers
- systemd handles PID tracking, stopping, and restart logic automatically

Appendix F: Useful One-Liners and Shell Recipes

```
# List all enabled services:
$ systemctl list-unit-files --type=service --state=enabled

# Show all services that failed today:
$ journalctl -p err --since today -o short-precise

# Find what is eating the most disk space in the journal:
$ journalctl --disk-usage

# See what changed in the last 5 minutes:
$ journalctl --since '5 minutes ago'

# Watch all service events in real time:
$ journalctl -f -p notice

# Find all units that depend on network-online.target:
$ systemctl list-dependencies network-online.target --reverse

# Check if a service will start at next boot:
$ systemctl is-enabled myapp.service && echo 'Starts at boot'

# List all running timers and their next fire time:
$ systemctl list-timers --all --no-pager

# Temporarily run a command as a systemd service:
$ sudo systemd-run --scope --unit=my-task /path/to/command

# Run a transient one-shot service and capture its output:
$ sudo systemd-run --pipe --wait /bin/ls /etc

# Show the effective unit file including all drop-ins:
$ systemctl cat nginx.service

# Show all properties of a unit in a searchable format:
$ systemctl show nginx.service | grep -i restart

# Test a journald rate limit without waiting:
$ sudo journalctl --rotate && sudo journalctl --vacuum-size=100M

# Find which service is listening on a port:
$ ss -tlnp | grep :8080

# Verify all unit files in /etc/systemd/system/:
$ for f in /etc/systemd/system/*.service; do
    echo "=== $f ==="
    systemd-analyze verify "$f" 2>&1
  done
```

```
# Get the last 5 boots with their durations:
$ systemd-analyze --list-boots 2>/dev/null || journalctl --list-boots |
tail -5
```

Appendix G: Further Reading and Official Resources

Official Documentation

- systemd man pages online: https://www.freedesktop.org/software/systemd/man/latest/
- systemd GitHub repository: https://github.com/systemd/systemd
- systemd.directives index: https://www.freedesktop.org/software/systemd/man/latest/systemd.directives.html
- Lennart Poettering's original systemd blog posts (2010): http://0pointer.de/blog/projects/systemd.html

Distribution Documentation

- Debian systemd guide: https://wiki.debian.org/systemd
- Ubuntu systemd documentation: https://documentation.ubuntu.com/server/explanation/systemd/
- Arch Linux systemd wiki: https://wiki.archlinux.org/title/systemd
- Red Hat systemd guide: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/9/html/using_systemd_unit_files_to_customize_and_optimize_your_system/

Man Pages to Know

- man systemctl — all systemctl commands
- man systemd.service — service unit directives
- man systemd.unit — [Unit] and [Install] section directives
- man systemd.exec — execution environment directives (User=, Environment=, etc.)
- man systemd.resource-control — cgroup resource control directives
- man systemd.timer — timer unit directives
- man systemd.socket — socket unit directives
- man journalctl — all journalctl options
- man journald.conf — journal daemon configuration
- man systemd-analyze — boot analysis tool

- man sd_notify — readiness notification from services

Essential Online References

- DigitalOcean systemd tutorials: https://www.digitalocean.com/community/tutorials/systemd-essentials-working-with-services-units-and-the-journal
- Linux Audit systemctl cheat sheet: https://linux-audit.com/cheat-sheets/systemctl/
- Linux Audit journalctl cheat sheet: https://linux-audit.com/cheat-sheets/journalctl/
- systemd by example: https://systemd-by-example.com/
- Computing for Geeks systemctl guide: https://computingforgeeks.com/systemctl-commands-linux/

---

systemd: The Complete Reference

End of Book