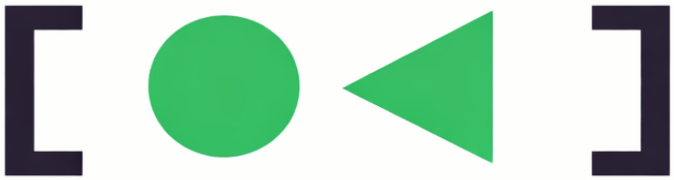
URADICAL

POCKET REFERENCE



systemd

Service Management & Boot Control

Alan Bradley

systemd Pocket Reference

A uRadical Production

Alan Bradley

uradical.io

systemd Pocket Reference

Alan Bradley

Table of Contents

Chapter 1: Core Concepts

What is systemd?

systemd is the init system and service manager for modern Linux distributions. It handles starting services at boot, managing dependencies between them, and supervising running processes. When you deploy an application to a Linux server, systemd is typically what keeps it running.

Units

Everything systemd manages is called a **unit**. Units are defined in configuration files with extensions that indicate their type:

- `.service` — A process or daemon (most common)
- `.timer` — Scheduled activation (replaces cron)
- `.socket` — Network or IPC socket for on-demand activation
- `.target` — A grouping of units (like a synchronisation point)
- `.mount` / `.path` / `.device` — Filesystem, path monitoring, and hardware units

Targets

Targets are collection points that represent system states. When the system boots, it progresses through targets until reaching the default target. The two you'll encounter most often are:

- `multi-user.target` — Normal multi-user system without graphical interface (typical for servers)
- `graphical.target` — Full graphical desktop environment

When you enable a service, you're typically adding it to one of these targets so it starts at boot.

The Unit File Search Path

systemd looks for unit files in multiple locations, with earlier paths taking precedence:

```
/etc/systemd/system/        # Local administrator config
(highest priority)
/run/systemd/system/        # Runtime units
/usr/lib/systemd/system/    # Package-installed units (lowest
priority)
```

Place your custom service files in `/etc/systemd/system/`. Never edit files in `/usr/lib/systemd/system/` directly—they'll be overwritten on package updates.

Essential systemctl Commands

These are the commands you'll use daily:

```
# Start a service immediately
systemctl start myapp.service
Stop a running service
systemctl stop myapp.service
Restart (stop then start)
systemctl restart myapp.service
Reload configuration without full restart (if supported)
systemctl reload myapp.service
Enable service to start at boot
systemctl enable myapp.service
Disable service from starting at boot
systemctl disable myapp.service
Check current status
systemctl status myapp.service
```

Note: `enable` and `start` are independent operations. Enabling a service doesn't start it, and starting doesn't enable it. Use `systemctl enable --now myapp.service` to do both at once.

daemon-reload: When You Need It

Whenever you create, modify, or delete a unit file, run:

```
systemctl daemon-reload
```

This tells systemd to re-scan its configuration. Without this step, systemd uses its cached version of your unit files. This is a common source of confusion when changes don't seem to take effect.

Chapter 2: Unit Files

A unit file is an INI-style configuration file with sections denoted by square brackets. For service units, three sections matter most: `[Unit]`, `[Service]`, and `[Install]`.

A Complete Example

Here's a typical service file for a Go web application:

```
[Unit]
Description=My Go Web Application
Documentation=https://example.com/docs
After=network.target postgresql.service
Wants=postgresql.service
[Service]
Type=simple
User=myapp
Group=myapp
WorkingDirectory=/opt/myapp
ExecStart=/opt/myapp/bin/server
ExecReload=/bin/kill -HUP $MAINPID
Restart=on-failure
RestartSec=5
EnvironmentFile=/opt/myapp/config/env
[Install]
WantedBy=multi-user.target
```

Let's break down each section.

The \[Unit\] Section

This section describes the unit and defines its relationships with other units.

Description — Human-readable name shown in logs and status output. Keep it concise but descriptive.

Documentation — URLs to documentation. Appears in `systemctl status` output.

After — Ordering dependency. This unit starts *after* the listed units. Doesn't require them to be running, just means "if both are starting, start me second."

Before — Reverse of After. This unit starts before the listed units.

Wants — Soft dependency. systemd will try to start these units alongside yours, but won't fail if they don't start.

Requires — Hard dependency. If these units fail, your unit fails too. Use sparingly—`Wants` is usually sufficient.

A critical point: `After` is about *ordering*, while `Wants`/`Requires` are about *activation*. You often need both. For example, if your app needs PostgreSQL:

```
After=postgresql.service    # Wait for PostgreSQL to start
first
Wants=postgresql.service    # Also try to start PostgreSQL if
it isn't running
```

Without `After`, both might start simultaneously. Without `Wants`, PostgreSQL won't be started automatically when your service starts.

The \[Service\] Section

This section defines how to run your service.

Type

The `Type` directive tells systemd how your service behaves:

- `simple` (default) — The process started by `ExecStart` is the main service process. Use this for most applications that stay in the foreground.

- `forking` — Traditional daemon behavior: process forks and parent exits. Requires `PIDFile` so systemd knows which process to track.
- `oneshot` — Process runs once and exits. Good for initialization scripts. Often paired with `RemainAfterExit=yes`.
- `notify` — Like simple, but the service sends a notification when ready. Requires `sd_notify()` support in your application.
- `exec` — Like simple, but service is considered started only after the binary is successfully executed.

For modern applications, `simple` is almost always correct. Only use `forking` for legacy daemons that insist on daemonizing themselves.

ExecStart, ExecStop, ExecReload

- `ExecStart` — The command to start the service. Must be an absolute path.
- `ExecStop` — Command to stop the service. If not specified, systemd sends SIGTERM.
- `ExecReload` — Command to reload configuration. `$MAINPID` expands to the main process ID.

Commands can be prefixed with special characters:

```
ExecStart=-/opt/myapp/bin/server    # '-' means ignore failure
ExecStart=+/opt/myapp/bin/server    # '+' means run with full
privileges
```

Restart Policies

Control what happens when your service exits:

- `no` — Don't restart (default)
- `on-failure` — Restart only on non-zero exit, signal, or timeout
- `on-abnormal` — Restart on signal, timeout, or watchdog
- `on-success` — Restart only on clean exit (exit code 0)

- `always` — Always restart, regardless of exit reason

For production services, `on-failure` is usually the right choice. Pair it with `RestartSec` to add a delay between restart attempts:

```
Restart=on-failure
RestartSec=5            # Wait 5 seconds before restarting
```

To limit restart attempts and prevent infinite loops:

```
StartLimitBurst=5       # Maximum 5 restart attempts...
StartLimitIntervalSec=60  # ...within 60 seconds
```

User and Group

Never run services as root unless absolutely necessary:

```
User=myapp
Group=myapp
```

Create a dedicated user for each service. This limits the blast radius if the service is compromised.

Working Directory

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

Sets the current working directory for the service. Relative paths in your application will resolve from here.

Environment Variables

Two ways to set environment variables:

```
# Inline (good for a few variables)
Environment="PORT=8080"
Environment="LOG_LEVEL=info"
From file (better for many variables or secrets)
```

```
EnvironmentFile=/opt/myapp/config/env
```

The environment file uses simple `KEY=value` format, one per line. Protect this file's permissions if it contains secrets.

The \[Install\] Section

This section defines what happens when you run `systemctl enable`:

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

`WantedBy` creates a symbolic link in the target's `.wants` directory, causing your service to start when that target is reached. For server applications, `multi-user.target` is almost always correct. For desktop applications that need a graphical environment, use `graphical.target` .

Chapter 3: Service Hardening

systemd provides security directives that sandbox your service with minimal effort. These use Linux kernel features like namespaces and capabilities to restrict what a compromised service can do.

Quick Wins

Add these to almost any service for immediate security improvement:

```
[Service]
Prevent gaining new privileges via setuid binaries
NoNewPrivileges=yes
Private /tmp directory, isolated from other services
PrivateTmp=yes
Make /usr, /boot, /efi read-only
ProtectSystem=strict
Make /home, /root, /run/user inaccessible
ProtectHome=yes
Hide /proc info about other processes
ProtectProc=invisible
New /dev with only pseudo-devices (null, zero, random, etc.)
PrivateDevices=yes
```

These directives have essentially no performance cost and protect against common attack patterns.

Filesystem Restrictions

ProtectSystem controls write access to system directories:

- `true` — `/usr` and `/boot` are read-only
- `full` — Also makes `/etc` read-only
- `strict` — Entire filesystem is read-only except explicit write paths

With `ProtectSystem=strict`, you must explicitly allow write access:

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

Capability Restrictions

Linux capabilities divide root's powers into granular permissions. Most services need very few:

```
# Drop all capabilities except those needed
CapabilityBoundingSet=CAP_NET_BIND_SERVICE
For services that don't need any special privileges:
CapabilityBoundingSet=
```

Common capabilities you might need to keep:

- `CAP_NET_BIND_SERVICE` — Bind to ports below 1024
- `CAP_CHOWN` — Change file ownership
- `CAP_SETUID` / `CAP_SETGID` — Change process user/group

Network Restrictions

```
# Allow only specific address families
RestrictAddressFamilies=AF_INET AF_INET6 AF_UNIX
Or disable networking entirely
PrivateNetwork=yes
```

A Hardened Example

Putting it together for a typical web service:

```
[Service]
User=myapp
Group=myapp
NoNewPrivileges=yes
PrivateTmp=yes
PrivateDevices=yes
ProtectSystem=strict
ProtectHome=yes
```

```
ProtectProc=invisible
ReadWritePaths=/var/lib/myapp
CapabilityBoundingSet=CAP_NET_BIND_SERVICE
RestrictAddressFamilies=AF_INET AF_INET6 AF_UNIX
```

Checking Security

systemd provides a security analysis tool:

```
systemd-analyze security myapp.service
```

This scores your service from 0 (most secure) to 10 (least secure) and suggests improvements. Don't aim for a perfect score—aim for meaningful protection that doesn't break your service.

Chapter 4: Timers

systemd timers are the modern replacement for cron jobs. They offer better logging, dependency handling, and resource control.

Basic Structure

A timer unit activates another unit (typically a service) on a schedule. You need two files:

backup.service — Defines what to run:

```
[Unit]
Description=Daily backup
[Service]
Type=oneshot
ExecStart=/opt/scripts/backup.sh
```

backup.timer — Defines when to run it:

```
[Unit]
Description=Run backup daily at 3am
[Timer]
OnCalendar=--* 03:00:00
Persistent=true
[Install]
WantedBy=timers.target
```

Enable the timer, not the service:

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

OnCalendar Syntax

Calendar expressions follow this format:

```
DayOfWeek Year-Month-Day Hour:Minute:Second
```

Examples:

```
--* 03:00:00          # Every day at 3am
Mon --* 09:00:00      # Every Monday at 9am
--01 00:00:00         # First day of every month at midnight
*-01,07-01 00:00:00     # January 1st and July 1st
Mon..Fri --* 09:00:00 # Weekdays at 9am
-- :00:00           # Every hour on the hour
-- :*:00            # Every minute
```

Test your expressions with:

```
systemd-analyze calendar "Mon --* 09:00:00"
Shows next trigger times
```

Shorthand Expressions

For common intervals, use the built-in shorthands:

```
OnCalendar=hourly       # -- :00:00
OnCalendar=daily        # --* 00:00:00
OnCalendar=weekly       # Mon --* 00:00:00
OnCalendar=monthly      # --01 00:00:00
OnCalendar=yearly       # *-01-01 00:00:00
```

Interval-Based Timers

For intervals rather than specific times:

```
[Timer]
15 minutes after system boot
OnBootSec=15min
1 hour after the service last finished
OnUnitActiveSec=1h
30 minutes after the timer was last activated
OnUnitInactiveSec=30min
```

These are useful for tasks that should run periodically but don't need a specific time.

Persistent Timers

```
Persistent=true
```

With `Persistent=true`, if the system was off when a timer should have triggered, the timer fires immediately on next boot. Essential for daily backups or maintenance tasks on systems that might be shut down.

Managing Timers

```
# List all timers and their next trigger time
systemctl list-timers
Include inactive timers
systemctl list-timers --all
Check timer status
systemctl status backup.timer
Manually trigger the associated service
systemctl start backup.service
```

Chapter 5: Socket Activation

Socket activation lets systemd listen on a port and start your service only when a connection arrives. This saves resources for rarely-used services and enables zero-downtime restarts.

Why Use It?

- **Resource efficiency** — Services that are rarely used don't consume memory until needed
- **Faster boot** — Services start on-demand rather than all at once
- **Smooth restarts** — systemd holds connections while the service restarts
- **Privilege separation** — systemd can bind privileged ports, then hand off to an unprivileged service

Basic Setup

myapp.socket — The socket definition:

```
[Unit]
Description=My App Socket
[Socket]
ListenStream=8080
Accept=no
[Install]
WantedBy=sockets.target
```

myapp.service — The service (note: no `[Install]` section):

```
[Unit]
Description=My App Service
[Service]
ExecStart=/opt/myapp/bin/server
```

Enable the socket, not the service:

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

How Your App Receives the Socket

With `Accept=no` (the common case), systemd passes the listening socket to your service as file descriptor 3. Most languages have libraries to handle this. In Go:

```
import "github.com/coreos/go-systemd/activation"
listeners, _ := activation.Listeners()
if len(listeners) > 0 {
http.Serve(listeners[0], handler)  // Use systemd's socket
} else {
http.ListenAndServe(":8080", handler)  // Fallback
}
```

This pattern lets your app work both with and without socket activation.

Accept Mode

With `Accept=yes`, systemd spawns a new service instance for each connection. The service name becomes `myapp@.service` (note the `@`) and receives the connected socket rather than the listening socket. This is useful for simple request-response services but adds overhead for high-traffic applications.

Chapter 6: journalctl

journalctl is your window into systemd's logging. It replaces the traditional syslog approach with a structured, indexed journal.

Basic Usage

```
# Show all logs (oldest first)
journalctl
Show logs in reverse order (newest first)
journalctl -r
Follow logs in real-time
journalctl -f
```

Filtering by Unit

The most common filter—show logs from a specific service:

```
# Logs from a specific service
journalctl -u myapp.service
Follow a specific service's logs
journalctl -u myapp.service -f
Multiple units
journalctl -u myapp.service -u nginx.service
```

Filtering by Time

```
# Since a specific time
journalctl --since "2024-01-15 09:00:00"
Until a specific time
journalctl --until "2024-01-15 17:00:00"
Both
journalctl --since "09:00" --until "10:00"
Relative times
journalctl --since "1 hour ago"
journalctl --since "yesterday"
journalctl --since "2 days ago"
```

Filtering by Priority

Log priorities follow syslog levels:

```
# Show only errors and above (0-3)
journalctl -p err
Show warnings and above (0-4)
journalctl -p warning
Priority range
journalctl -p err..warning
```

Priority levels: 0=emerg, 1=alert, 2=crit, 3=err, 4=warning, 5=notice, 6=info, 7=debug

Filtering by Boot

```
# Current boot only
journalctl -b
Previous boot
journalctl -b -1
List all boots
journalctl --list-boots
```

Useful for debugging issues that occurred before the last reboot.

Output Formats

```
# Short (default) - one line per entry
journalctl -o short
Short with precise timestamps
journalctl -o short-precise
JSON format (for parsing)
journalctl -o json
Pretty-printed JSON
journalctl -o json-pretty
Just the message, nothing else
journalctl -o cat
Verbose - all fields
```

```
journalctl -o verbose
```

The `json` format is invaluable for shipping logs to external systems or parsing with `jq`.

Pagination and Limits

```
# Show last N lines
journalctl -n 100
Disable pager (useful for scripts)
journalctl --no-pager
Combine with follow
journalctl -u myapp.service -n 50 -f
```

Disk Usage

```
# Check journal disk usage
journalctl --disk-usage
Vacuum by size (keep only 500MB)
journalctl --vacuum-size=500M
Vacuum by time (keep only 2 weeks)
journalctl --vacuum-time=2weeks
Vacuum by files
journalctl --vacuum-files=5
```

Consider adding vacuum commands to a timer for automatic log rotation.

Kernel Messages

```
# Show kernel messages (like dmesg)
journalctl -k
Kernel messages from previous boot
journalctl -k -b -1
```

Combining Filters

Filters combine with AND logic:

```
# Errors from myapp in the last hour
journalctl -u myapp.service -p err --since "1 hour ago"
Follow myapp logs, showing last 100 lines
journalctl -u myapp.service -n 100 -f
```

Chapter 7: Troubleshooting

Reading Status Output

`systemctl status` packs a lot of information into its output:

```
$ systemctl status myapp.service
. myapp.service - My App Service
Loaded: loaded (/etc/systemd/system/myapp.service; enabled;
preset: disabled)
Active: active (running) since Mon 2024-01-15 09:00:00 UTC; 2h
ago
Main PID: 1234 (server)
Tasks: 8 (limit: 4915)
Memory: 45.2M
CPU: 1min 23.456s
CGroup: /system.slice/myapp.service
..1234 /opt/myapp/bin/server
Jan 15 09:00:00 hostname server[1234]: Starting server on
:8080
```

Key things to look for: the Active state (running, failed, inactive), exit codes in parentheses if failed, and recent log lines at the bottom.

Common Failure States

- **activating** — Service is starting but hasn't signaled readiness
- **deactivating** — Service is shutting down
- **failed** — Service exited with an error or was killed
- **inactive (dead)** — Service is not running (may be normal for oneshot)
- **auto-restart** — Service is in restart delay interval

Investigating Failures

When a service fails, start with these commands:

```
# Check status and recent logs
systemctl status myapp.service
More detailed logs
journalctl -u myapp.service -n 100 --no-pager
Show the full unit file being used
systemctl cat myapp.service
Show runtime changes (drop-ins)
systemctl show myapp.service
```

Resetting Failed State

After fixing the issue, clear the failed state:

```
# Reset one service
systemctl reset-failed myapp.service
Reset all failed units
systemctl reset-failed
```

Dependency Inspection

When startup order seems wrong:

```
# Show what this unit depends on
systemctl list-dependencies myapp.service
Show what depends on this unit
systemctl list-dependencies --reverse myapp.service
Show full dependency tree
systemctl list-dependencies --all myapp.service
```

Boot Analysis

When boot is slow:

```
# Overall boot time
systemd-analyze time
Time spent per unit (blame the slow ones)
```

```
systemd-analyze blame
Critical path analysis
systemd-analyze critical-chain
Critical chain for a specific unit
systemd-analyze critical-chain myapp.service
Generate SVG boot chart
systemd-analyze plot > boot.svg
```

Configuration Verification

```
# Validate unit file syntax
systemd-analyze verify myapp.service
Show effective configuration (after drop-ins)
systemctl cat myapp.service
```

Common Mistakes

Forgetting daemon-reload:

```
# Always run after editing unit files
systemctl daemon-reload
```

Using relative paths: ExecStart must use absolute paths.

Wrong Type: If your app daemonizes itself, use `Type=forking`. If it stays in foreground (most modern apps), use `Type=simple`.

Permission issues: Check that the User/Group can access all paths and ports.

Missing dependencies: Add `After=` and `Wants=` for any services you depend on.

Chapter 8: Quick Reference Tables

systemctl Commands

Command	Description
systemctl start unit	Start a unit
systemctl stop unit	Stop a unit
systemctl restart unit	Restart a unit
systemctl reload unit	Reload configuration
systemctl enable unit	Enable at boot
systemctl disable unit	Disable at boot
systemctl enable --now unit	Enable and start
systemctl status unit	Show status and logs
systemctl cat unit	Show unit file
systemctl daemon-reload	Reload systemd config
systemctl list-units	List active units
systemctl list-unit-files	List all unit files
systemctl list-timers	List timers
systemctl reset-failed	Clear failed state

journalctl Options

Option	Description
-u unit	Filter by unit
-f	Follow (tail) logs

-n N	Show last N lines
-r	Reverse order (newest first)
-p priority	Filter by priority
-b	Current boot only
-b -1	Previous boot
--since "time"	Logs after time
--until "time"	Logs before time
-o format	Output format
--disk-usage	Show journal size
--vacuum-size=N	Limit journal to N bytes
--list-boots	List recorded boots
--no-pager	Disable pager

Common Unit File Options

Section	Option	Description
\[Unit\]	Description	Human-readable name
\[Unit\]	After	Start after these units
\[Unit\]	Before	Start before these units
\[Unit\]	Wants	Soft dependency
\[Unit\]	Requires	Hard dependency
\[Service\]	Type	simple, forking, oneshot, notify
\[Service\]	ExecStart	Command to start
\[Service\]	ExecStop	Command to stop
\[Service\]	ExecReload	Command to reload

\[Service\]	User/Group	Run as user/group
\[Service\]	WorkingDirectory	Set working directory
\[Service\]	Environment	Set environment variable
\[Service\]	EnvironmentFile	Load env from file
\[Service\]	Restart	Restart policy
\[Service\]	RestartSec	Delay between restarts
\[Install\]	WantedBy	Target to install into

OnCalendar Expressions

Expression	Description
--* 03:00:00	Daily at 3am
Mon --* 09:00:00	Mondays at 9am
Mon..Fri --* 09:00:00	Weekdays at 9am
--01 00:00:00	First of month
*-01,07-01 00:00:00	Jan 1 and Jul 1
-- `` :00:00	Every hour
-- `` :*:00	Every minute
hourly	Top of every hour
daily	Midnight daily
weekly	Monday midnight
monthly	First of month

Security Directives

Directive	Description
NoNewPrivileges=yes	Prevent privilege escalation
PrivateTmp=yes	Isolated /tmp
PrivateDevices=yes	Minimal /dev
ProtectSystem=strict	Read-only filesystem
ProtectHome=yes	No access to /home
ProtectProc=invisible	Hide other processes
ReadWritePaths=/path	Allow writes to path
CapabilityBoundingSet=	Drop all capabilities
RestrictAddressFamilies=	Limit network access
