How to Create and Manage Systemd Services in Linux

Written By: *Saikat Goswami*

This article is a complete, in-depth guide—suitable for beginners, system administrators, DevOps engineers, and developers—on creating, enabling, managing, debugging, and optimizing **systemd services**.

Contents

1. What is the *init* process in Linux ?

The **init** process in Linux is the ***first process*** started by the kernel after booting. Always runs as process ID 1, making it the root of the process tree. It is responsible for booting the system, managing services, tracking logs, supervising processes, and ensuring that the system remains stable and predictable.

2. Introduction to systemd

Systemd has become the standard init system on most modern Linux distributions, including Ubuntu, Debian, CentOS, Fedora, RHEL, AlmaLinux, Rocky Linux and many others. Systemd is an init system and service manager designed to overcome the limitations of traditional init scripts (SysVinit). Earlier Linux systems relied on `/etc/init.d/` shell scripts that executed sequentially. As systems grew more complex and boot speeds became a priority, a more modern, parallelized, event-driven system was required.

Systemd provides:

- Faster boot times through parallelization
- A standardized service configuration format
- Built-in logging via **journald**
- Automatic service restarts and supervision
- Dependency-based startup
- Integration with **cgroups** for resource management

Most importantly, systemd introduces **unit files**—small declarative configuration files that define services, mount points, timers, devices, and more.

This article focuses specifically on **systemd service units** and teaches you how to write, install, run, monitor, troubleshoot, and optimize them.

2. Understanding Systemd Unit Files

A **systemd 'unit file'** acts as a blueprint that tells systemd what to start, when to start it, and under what conditions it should run. For `.service` files specifically, the unit defines details such as the service's executable command, its startup behavior, dependencies, restart policies, and logging configuration. These files make service management predictable and consistent because all operational rules are declared in one structured place.

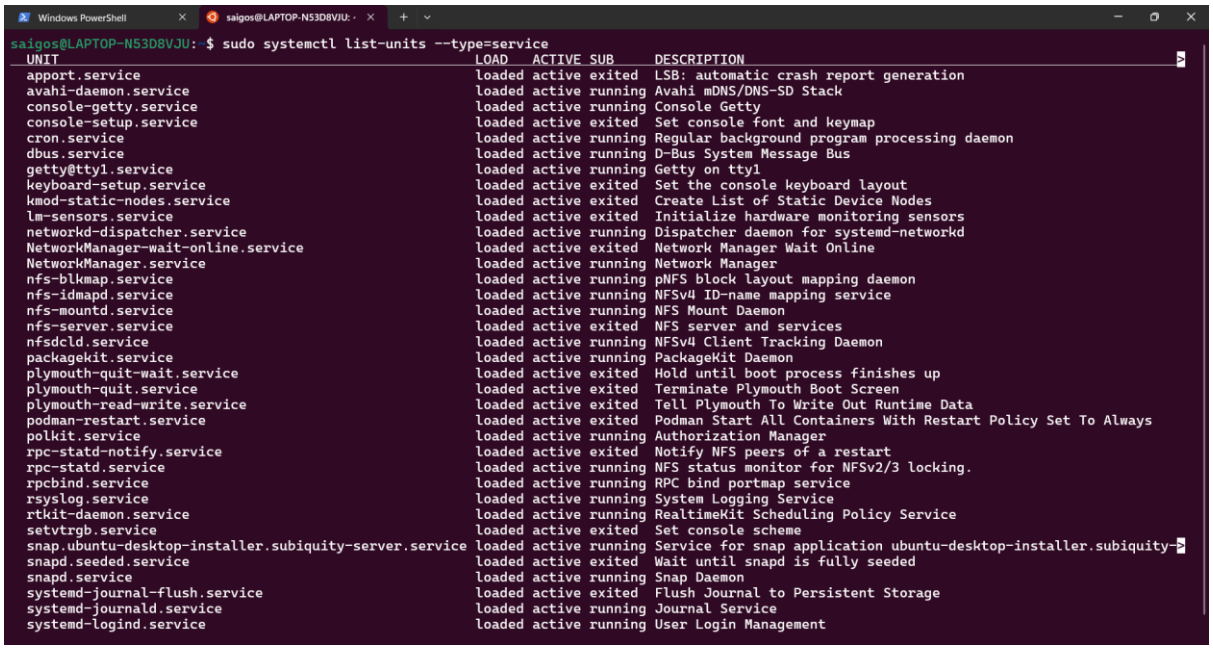
A unit file is a simple text file describing how systemd should manage a resource. For services, the file extension is `.service`.

Unit files typically reside in:

Directory	Purpose
`/usr/lib/systemd/system/`	Distribution-provided services
`/etc/systemd/system/`	Administrator-created or overridden services
`/run/systemd/system/`	Runtime-generated units

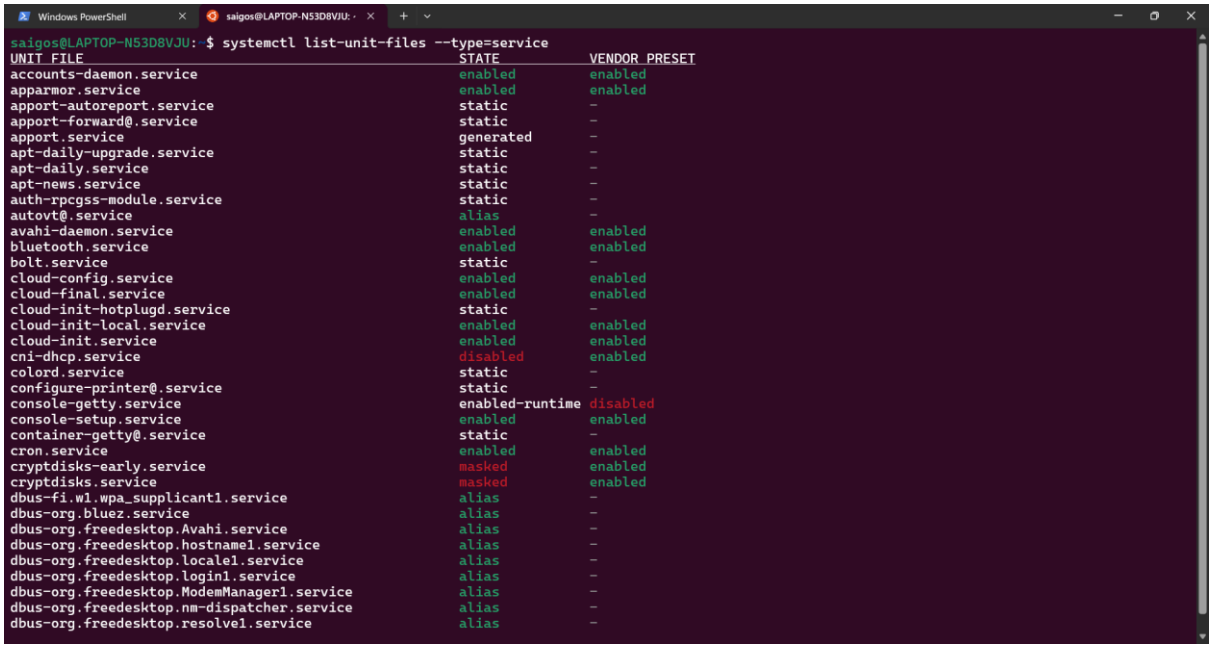
To list all active services, type:

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



To list all installed services:

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



3. What is a unit file?

- A **unit file** is a configuration file that tells systemd how to manage a resource (service, socket, device, mount, etc.).
- All unit files live under `/etc/systemd/system/`, `/lib/systemd/system/`, or `/usr/lib/systemd/system/`.
- Each unit file has a **type**, indicated by its suffix.

3.1 Common unit file types

- `.service` → defines how to start/stop/manage a service (e.g., `nginx.service`).
- `.socket` → describes a network socket or IPC socket.
- `.target` → groups units together (like runlevels).
- `.mount` → controls filesystem mounts.
- `.timer` → schedules tasks (like cron).
- `.device`, `.path`, etc. → specialized resources.

3.2 Sections of a unit file

A unit file contains sections such as:

Explanation

- `[Unit]` **section** → metadata (description, dependencies).

- `[Service]` **section** → runtime configuration for the service process (command, environment variables, working directory, etc.).
- `[Install]` **section** → installation details (wanted-by targets).

3.3 A sample systemd Service File

A sample service file looks like this:

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

[Service]
Type=simple
ExecStart=/usr/bin/python3 /home/user/app.py

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

Explanation of Sections

[Unit] Section

The `[Unit]` section describes:

- A short description
- Dependencies
- Behavior before starting the service

Common directives:

Directive	Description
`Description=`	Human-readable description
`After=`	Start service after another unit
`Before=`	Opposite of After
`Requires=`	Hard dependency; fails if missing
`Wants=`	Soft dependency

Example:

```
After=network-online.target
Requires=docker.service
```

[Service] Section

The heart of the service.

Common Directives:

Directive	Purpose
`Type=`	How the process behaves
`ExecStart=`	The command to start the service
`ExecReload=`	Command to reload
`ExecStop=`	Command to stop
`Restart=`	Restart policies
`Environment=`	Set environment variables
`User=` / `Group=`	Non-root service execution
`WorkingDirectory=`	Set app directory

Restart policies examples:

```
Restart=on-failure
RestartSec=3
```

Service Types:

Type	Meaning
`Simple`	Default; runs a foreground process
`Forking`	Background/daemonized processes
`Oneshot`	Runs to completion (e.g., scripts)
`Notify`	App notifies systemd when ready
`Idle`	Starts after all jobs are loaded

[Install] Section

Under this section, consider an example entry:

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

It means when enabled, start this service whenever the system reaches multi-user mode (normal boot).

Common targets:

Target	Description
`multi-user.target`	Normal CLI system

Target	Description
`graphical.target`	Desktop
`network.target`	Networking available
`local-fs.target`	Filesystems ready

Systemd follows dependencies. In the file if you have:

```
After=network-online.target
```

This means your service will start *after* `network-online.target` has reached "active".

Another option in the file is '`Requires=`':

```
Requires=network-online.target
```

If `network-online.target` service fails, your service won't start**.** Think of **`Requires=`** as a mandatory dependency. Like saying: "Start with me, and if you fail, I fail too."

Systemd services often need environment variables for need for things like:

- application mode (`APP_ENV=production`)
- ports (`APP_PORT=9000`)
- database URLs (`DB_URL=mysql://user:password@localhost:3306/mydatabase`)

Adding Inline in the Service File. Add the following under [Service] section:

```
Environment="APP_ENV=production"
```

Load from File

Create a file:

```
vi/etc/myapp/env
```

Add variables:

```
[Service]
APP_ENV=production
APP_PORT=9000
```

Reference it in your unit file:

```
EnvironmentFile=/etc/myapp/env
```

The content looks like this:

```
[Service]
Environment="APP_ENV=production"
Environment="APP_PORT=9000"
```

Running as root is unsafe. To run as a non-root user, add:

```
User=appuser
Group=appuser
```

Restrict File Access

For restricting file access, add the following entries in your service file:

```
ProtectSystem=full
ProtectHome=true
```

Disable Networking

For enabling and disabling networking in your service, add the following entry in your service file:

```
PrivateNetwork=true
```

Restrict Read/Write Access

For restricting read/write access, add the following entries in your service file:

```
ReadOnlyPaths=/usr
ReadWritePaths=/var/log/myapp
```

Disable Capability Escalation

```
NoNewPrivileges=true
```

Use systemd sandboxing

```
ProtectKernelTunables=yes
ProtectProc=invisible
RestrictSUIDSGID=yes
```

These features significantly harden services in production environments.

Using systemd for Automatic Restarts and Resilience

Systemd can restart your app automatically under many circumstances.

Examples:

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

Other types:

Policy	Meaning
`No`	Never restart
`on-success`	Restart on exit code 0
`on-failure`	Restart on non-zero exit
`on-abnormal`	Restart on signal
`Always`	Always restart

Example for high reliability:

```
Restart=always
RestartSec=1
StartLimitInterval=400
StartLimitBurst=10
```

4 Managing Systemd Services

4.1 To **start** service, type:

```
sudo systemctl start myapp.service
```

4.2 To **stop** service, type:

```
sudo systemctl stop myapp.service
```

4.3 To **restart** service type:

```
sudo systemctl restart myapp.service
```

Stops and then starts the service. Required when:

- You've changed binaries or environment variables.
- The service is stuck or misbehaving.
- You want a clean slate.

4.4 To **reload** service type:

```
sudo systemctl reload myapp.service
```

Sends a reload signal (usually `SIGHUP`) to the running process. The service keeps running but re-reads its configuration files. Required when:
• You've updated config files
• You want changes applied without downtime.

Limitation: Reloading only works if `ExecReload` is defined in the Unit file. If not, the command may fail or do nothing.

4.5 If you want your service to startup at boot, type:

```
sudo systemctl enable myapp.service
```

4.6 If you do not want your service to startup at boot, type:

```
sudo systemctl disable myapp.service
```

4.7 If you want to check if your service is configured to startup at boot:

```
systemctl is-enabled myapp.service
```

5. Creating a Custom Systemd Service

5.1 Let's create a real service for a simple shell script.

Step 1: Write Your Application

Example script `/home/saigos/hello/hello.sh`:

```
#!/bin/bash
while true; do
  echo "Hello from systemd service!" >> /home/saigos/hello/output.log
  sleep 5
done
```

Make it executable:

```
chmod a+x /home/saigos/hello/hello.sh
```

Step 2: Create the Service File

Create a file:

```
sudo nano /etc/systemd/system/hello.service
```

Paste:

```
[Unit]
Description=Hello Test Service
After=network.target

[Service]
ExecStart=/home/saigos/hello/hello.sh
Restart=always
RestartSec=2
User=root

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

Step 3: Reload systemd

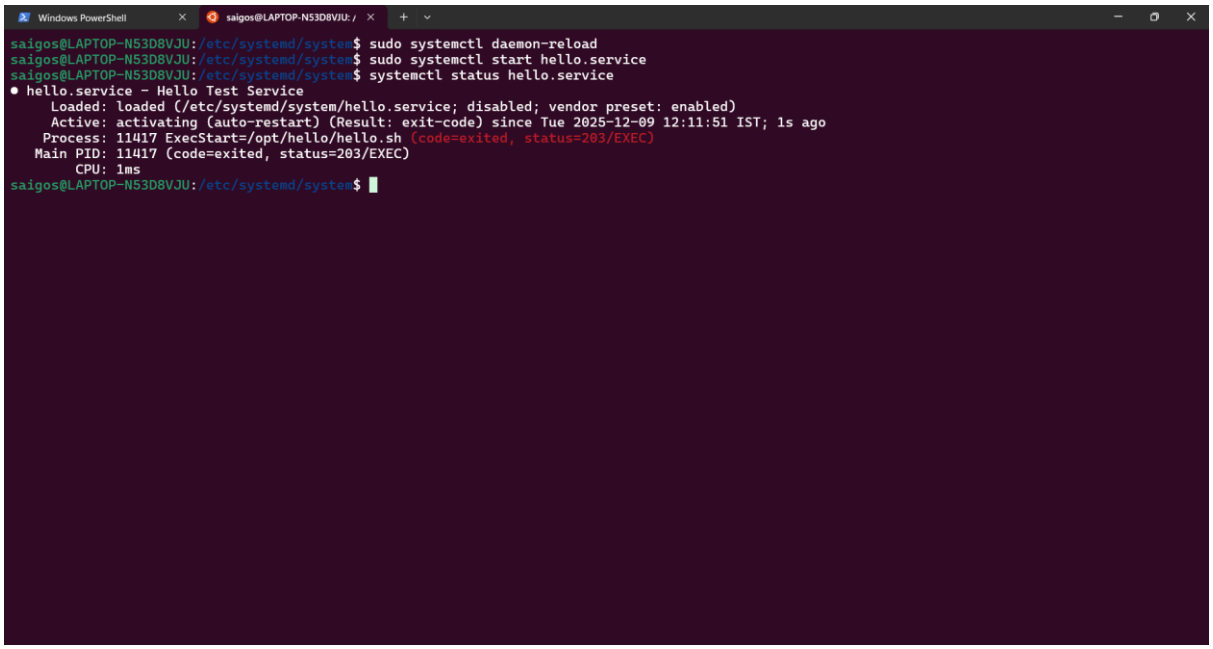
```
sudo systemctl daemon-reload
```

Step 4: Start the Service

```
sudo systemctl start hello.service
```

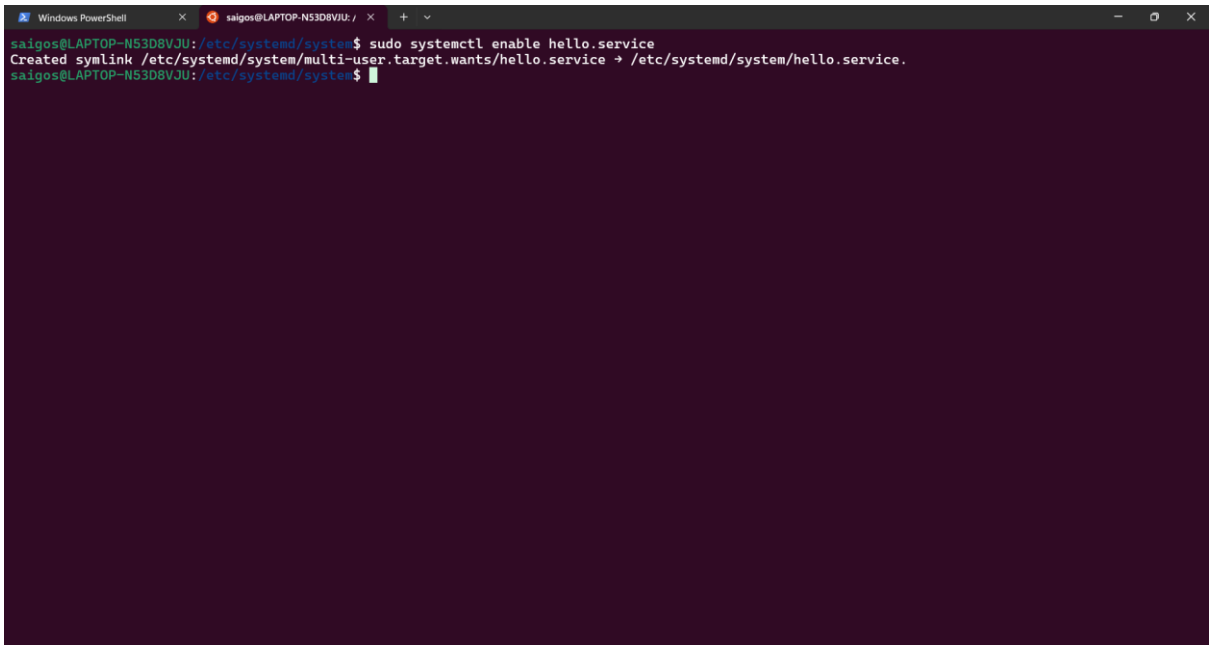
To check the status, check status:

```
systemctl status hello.service
```

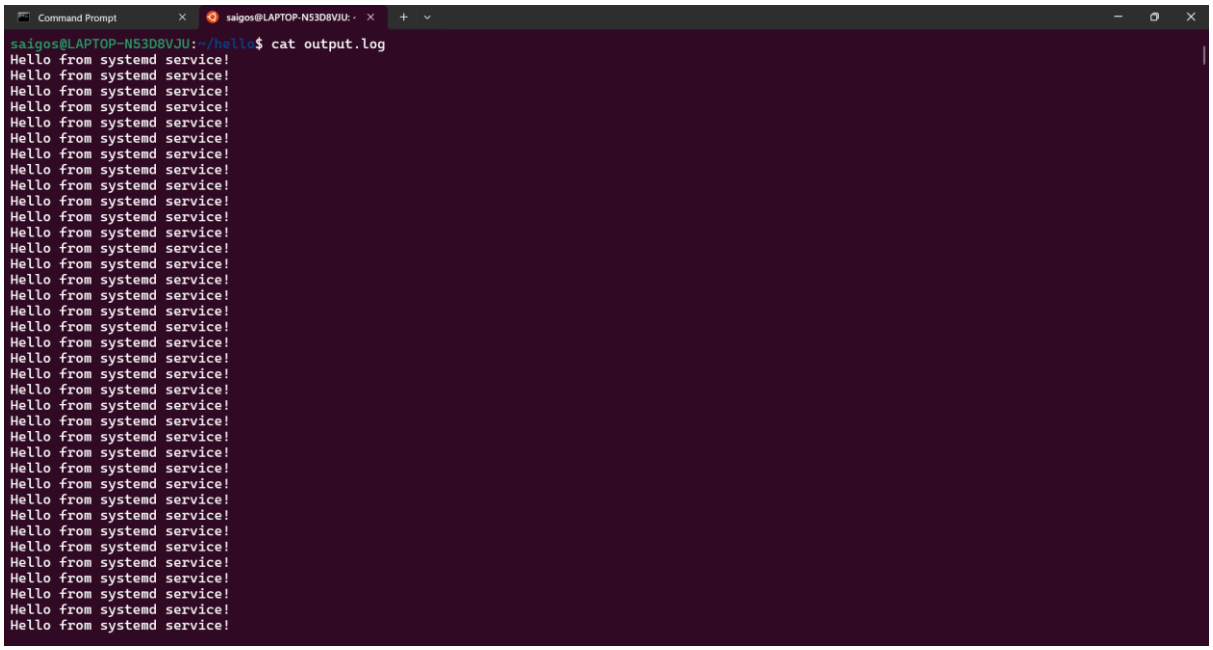


Step 5: If you want to start hello.service at boot, type

```
sudo systemctl enable hello.service
```



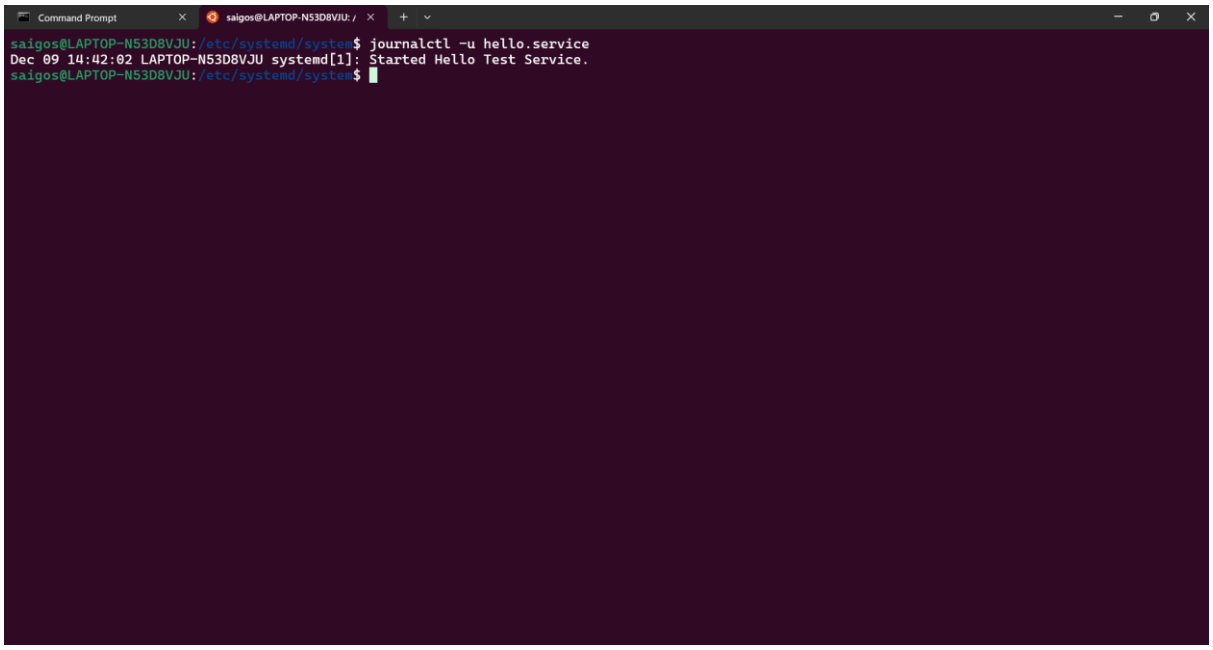
See output at /home/saigos/hello/output.log:



5.2 Viewing Logs with Journald

Systemd integrates with journald:

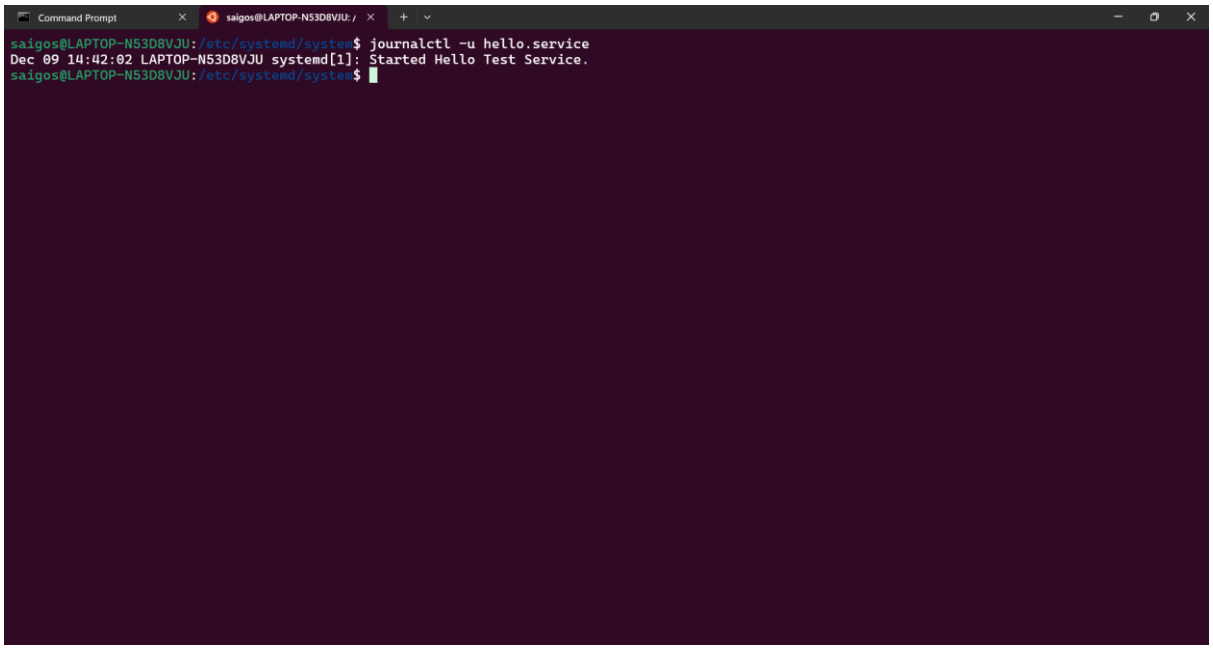
```
journalctl -u hello.service
```



You can view logs. Systemd integrates with **journald.** Type:

```
journalctl -u hello.service
```

You will see the output:



Follow logs live:

```
journalctl -u hello.service -f
```

Show only today's logs:

```
journalctl -u hello.service --since=today
```

6. Using systemd Timers Instead of Cron Jobs

Systemd timers replace cron jobs and integrate better with logs, dependencies, and failures.

Example timer unit (`backup.timer`):

```
[Unit]
Description=Run backup every day

[Timer]
OnCalendar=daily
Persistent=true

[Install]
WantedBy=timers.target
```

Timer calls `backup.service`.

Enable:

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

List timers:

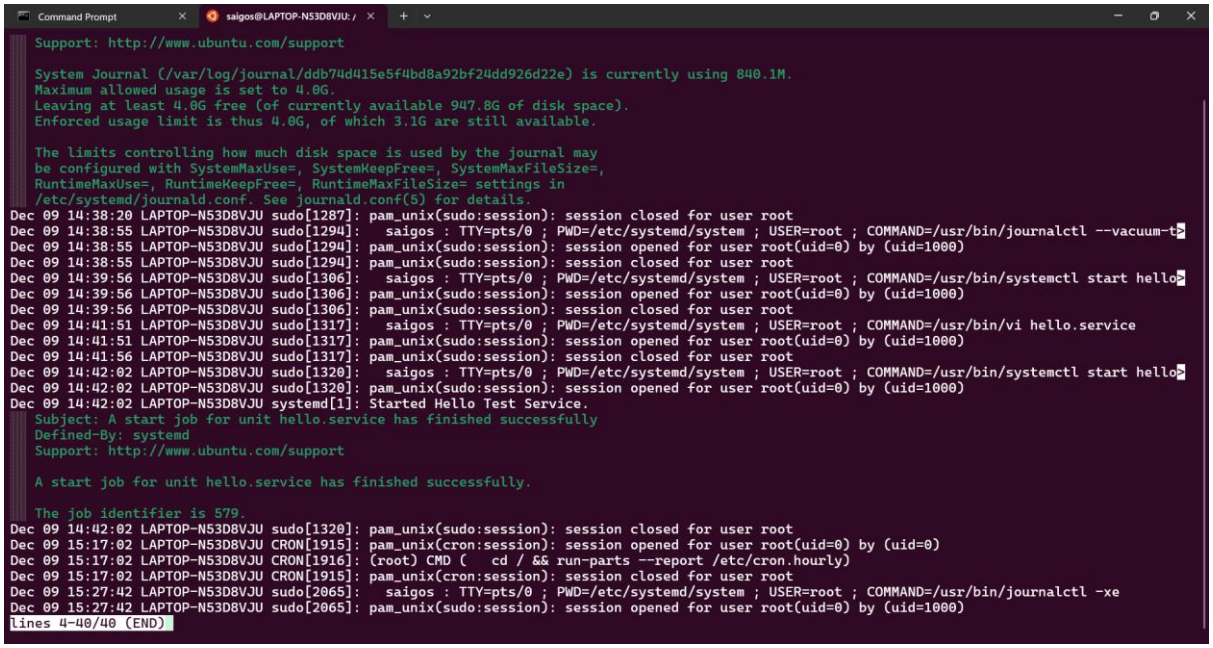
```
systemctl list-timers
```

---

7. Debugging and Troubleshooting systemd Services

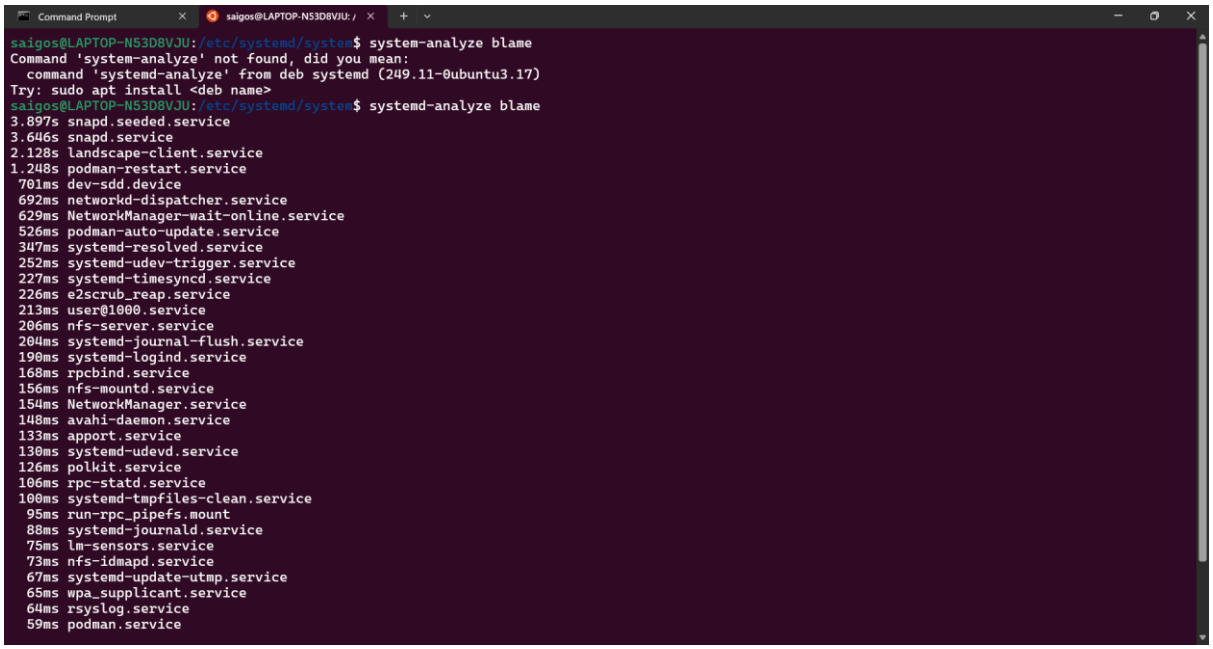
7.1 Check Logs

```
journalctl -xe
```



7.2 Check Startup Time and Dependencies

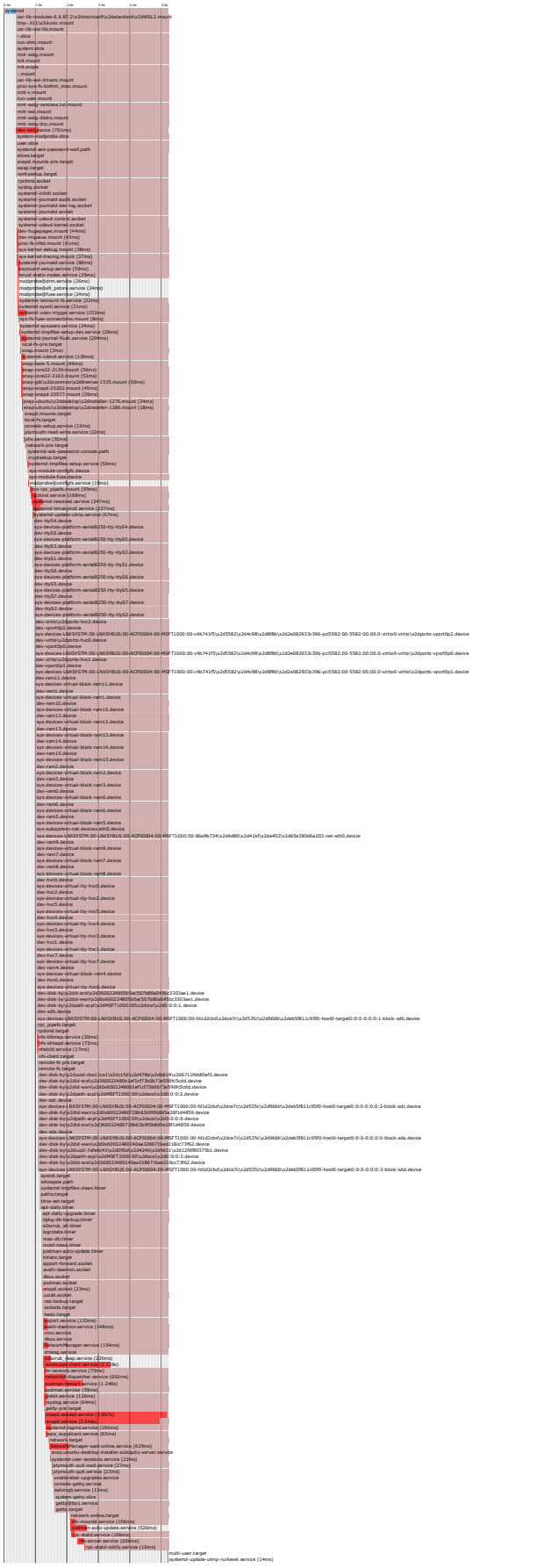
```
systemd-analyze blame
```



Graph the boot process:

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

Ubuntu 22.04.5 LTS LAPTOP-N53D8VJU (Linux 6.6.87.2-microsoft-standard-WSL2 #1 SMP PREEMPT_DYNAMIC Thu Jun 5 18:30:46 UTC 2025) x86-64 wsl
Startup finished in 5.236s (userspace) multi-user.target reached after 5.193s in userspace



8. Real-World Examples

8.1 Docker Container Auto-Start Using systemd

```
[Unit]
Description=Run My Docker Container
After=docker.service
Requires=docker.service

[Service]
ExecStart=/usr/bin/docker run --rm --name myapp -p 8080:8080 myimage
ExecStop=/usr/bin/docker stop myapp
Restart=always

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

9. Best Practices for Writing Systemd Service Files

1. Never run apps as root unless necessary
2. Use `Restart=on-failure` to ensure reliability
3. Keep unit files in `/etc/systemd/system/`
4. It is recommended to use logs via journald instead of custom logging
5. Use `EnvironmentFile` instead of hardcoding values
6. Prefer absolute paths in unit files
7. Use `systemd-analyze verify` before enabling

Following such practices ensures stable, maintainable services.

10.Conclusion

Creating and managing systemd services is one of the most essential skills for working with modern Linux systems. Systemd is not just a replacement for SysVinit—it is a powerful framework that handles service lifecycles, supervision, logging, security, sandboxing, scheduling, and system orchestration.

By understanding unit file structures, dependencies, environment variables, service types, logging, and security features, you can build robust and reliable services tailored to your system or production environment.

Whether you're deploying a Python web app, managing Docker containers, setting up automated backups, or writing a lightweight daemon, **systemd provides the speed, stability, and control you need**.