systemd i systemctl: jednostki, zależności, usługi

◫◫ Poziom: średnio

◫◫ Dotyczy: RHEL/Rocky 9–10 (domyślnie), różnice Debian/Ubuntu i SLES zaznaczone

▯ Czas: ~22 min

systemd to proces PID 1 i cały ekosystem narzędzi wokół niego — startuje system, nadzoruje usługi, zarządza zależnościami między nimi i pilnuje, żeby każdy proces dało się jednoznacznie przypisać do usługi, z której pochodzi. `systemctl` *to komenda, którą będziesz wpisywać codziennie.*

◫◫ Skróty użyte na tej stronie (6)

Skrót	Co znaczy
CPU	Central Processing Unit: procesor
PID	Process ID: numer procesu w systemie
RHEL	Red Hat Enterprise Linux: komercyjna dystrybucja Linuksa
SLES	SUSE Linux Enterprise Server: płatna dystrybucja Linuksa firmy SUSE, konkurent Rocky/RHEL
SSH	Secure Shell: szyfrowane zdalne logowanie do serwera
WWW	World Wide Web: strony i aplikacje otwierane w przeglądarce po HTTP albo HTTPS

Fiszka

◫◫ TYLKO ODCZYT

Sprawdzenie stanu jednostki: `systemctl status`

```
systemctl status NazwaUslugi
```

Najczęstsza komenda z tej strony. Sprawdzamy nią, czy jednostka jest załadowana, czy działa, od kiedy, jaki proces jest za nią uznawany, ile zasobów zużywa i jakie ostatnie linie

trafiły do dziennika.

Przykład:

```
● nginx.service - The nginx HTTP and reverse proxy server
     Loaded: loaded (/usr/lib/systemd/system/nginx.service; enabled; preset: disabled)
    Drop-In: /etc/systemd/system/nginx.service.d
             └─override.conf
     Active: active (running) since Thu 2026-09-17 08:02:11 UTC; 3h 12min ago
   Main PID: 1842 (nginx)
      Tasks: 5 (limit: 4915)
     Memory: 9.8M
        CPU: 412ms
     CGroup: /system.slice/nginx.service
             ├─1842 "nginx: master process /usr/sbin/nginx"
             ├─1843 "nginx: worker process"
             └─1844 "nginx: worker process"

Sep 17 08:02:11 srv01 systemd[1]: Starting The nginx HTTP and reverse proxy server...
Sep 17 08:02:11 srv01 nginx[1842]: nginx: configuration file /etc/nginx/nginx.conf test
is successful
Sep 17 08:02:11 srv01 systemd[1]: Started The nginx HTTP and reverse proxy server.
```

Pełne rozbicie tego wyniku pole po polu jest w sekcji „Jak to działa" niżej.

systemctl = system control: sterowanie usługami systemd

ZMIENIA SYSTEM

Uruchomienie jednostki teraz: `systemctl start`

```
sudo systemctl start NazwaUslugi
```

Uruchamiamy jednostkę natychmiast. Nie zmienia to niczego w kwestii autostartu przy następnym rozruchu — to osobna decyzja, patrz `enable`.

systemctl = system control: sterowanie usługami systemd

ZMIENIA SYSTEM

Zatrzymanie jednostki teraz: `systemctl stop`

```
sudo systemctl stop NazwaUslugi
```

Zatrzymujemy jednostkę natychmiast, tak samo bez wpływu na autostart.

systemctl = system control: sterowanie usługami systemd

ZMIENIA SYSTEM

Zatrzymanie i ponowne uruchomienie jednostki: `systemctl restart`

```
sudo systemctl restart NazwaUslugi
```

Zatrzymujemy i uruchamiamy jednostkę od nowa. Krótka przerwa w dostępności jest nieunikniona — jeśli usługa to wspiera, `reload` jest łagodniejszą alternatywą.

systemctl = system control: sterowanie usługami systemd

ZMIENIA SYSTEM

Przeładowanie konfiguracji bez przerywania pracy: `systemctl reload`

```
sudo systemctl reload NazwaUslugi
```

Każe procesowi wczytać nowy config bez zamykania i otwierania go od nowa — istniejące połączenia klientów nie są zrywane. Działa tylko, jeśli aplikacja sama to wspiera (nginx tak, wiele prostszych demonów — nie). Gdy nie wiadomo, czy usługa to wspiera, `sudo systemctl reload-or-restart NazwaUslugi` zrobi reload, a jeśli się nie da — pełny restart.

systemctl = system control: sterowanie usługami systemd

ZMIENIA SYSTEM

Włączenie autostartu przy boocie: `systemctl enable`

```
sudo systemctl enable NazwaUslugi
```

Ustawia autostart przy kolejnym rozruchu systemu. **Nie uruchamia usługi teraz** — jeśli akurat nie działa, dalej nie będzie działać, dopóki nie zrobimy `start` albo nie zrestartujemy maszyny.

systemctl = system control: sterowanie usługami systemd

ZMIENIA SYSTEM

Włączenie autostartu i uruchomienie jedną komendą: `systemctl enable --now`

```
sudo systemctl enable --now NazwaUslugi
```

To, czego prawie zawsze potrzebujemy zamiast samego `enable`: ustawia autostart i uruchamia jednostkę od razu, w jednym poleceniu.

systemctl = system control: sterowanie usługami systemd

ZMIENIA SYSTEM

Wyłączenie autostartu: `systemctl disable`

```
sudo systemctl disable NazwaUslugi
```

Usuwa symlink autostartu. Jednostkę wciąż da się uruchomić ręcznie przez `start` — `disable` nie blokuje tego, tylko wyłącza uruchamianie przy boocie.

systemctl = system control: sterowanie usługami systemd

ZMIENIA SYSTEM

Całkowite zablokowanie jednostki: `systemctl mask`

```
sudo systemctl mask NazwaUslugi
```

Podmienia jednostkę na symlink do `/dev/null`. Od tej chwili nawet ręczny `systemctl start NazwaUslugi` się nie powiedzie — i tak samo nie uruchomi jej żadna inna jednostka, która próbowałaby jej użyć jako zależności. To mocniejsze narzędzie niż `disable`.

systemctl = system control: sterowanie usługami systemd

TYLKO ODCZYT

Zdjęcie blokady: `systemctl unmask`

```
sudo systemctl unmask NazwaUslugi
```

Przywraca jednostkę do normalnego stanu po `mask` — dopiero po tym `start` znowu zadziała.

systemctl = system control: sterowanie usługami systemd

TYLKO ODCZYT

Szybkie sprawdzenie stanu: `systemctl is-active`

```
systemctl is-active NazwaUslugi
```

Przykład:

```
active
```

Zwraca jedno słowo (`active`, `inactive`, `failed`...) i kod wyjścia dopasowany do wyniku — wygodne w skryptach, które mają tylko sprawdzić stan, bez parsowania pełnego `status`. Analogicznie `systemctl is-enabled NazwaUslugi` sprawdza samą kwestię autostartu, a `systemctl is-failed NazwaUslugi` pyta wyłącznie o stan `failed`.

systemctl = system control: sterowanie usługami systemd

TYLKO ODCZYT

Szybkie sprawdzenie autostartu: `systemctl is-enabled`

```
systemctl is-enabled NazwaUslugi
```

Przykład:

```
enabled
```

Odpowiada na pytanie „czy ta jednostka wystartuje sama przy następnym boocie", niezależnie od tego, czy akurat teraz działa.

systemctl = system control: sterowanie usługami systemd

TYLKO ODCZYT

Wylistowanie jednostek załadowanych do pamięci: `systemctl list-units`

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

Pokazuje jednostki, które systemd aktualnie zna i śledzi — załadowane, niekoniecznie działające. Bez `--type` pokaże wszystkie rodzaje na raz, co bywa przytłaczające.

systemctl = system control: sterowanie usługami systemd

TYLKO ODCZYT

Wylistowanie wszystkich zainstalowanych jednostek: `systemctl list-unit-files`

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

W przeciwieństwie do `list-units`, pokazuje też jednostki, które nigdy nie były uruchamiane w tej sesji systemd — wszystko, co leży na dysku i mogłoby zostać załadowane, razem z kolumną mówiącą, czy jest `enabled`, `disabled` czy `static`.

systemctl = system control: sterowanie usługami systemd

TYLKO ODCZYT

Wypisanie tylko jednostek, które padły: `--failed`

```
systemctl list-units --failed
```

Przykład:

```
UNIT                    LOAD   ACTIVE SUB    DESCRIPTION
● backup-nocny.service  loaded failed failed Nocna kopia bazy danych

LOAD   = Reflects whether the unit definition was properly loaded.
ACTIVE = The high-level unit activation state, i.e. generalization of SUB.
SUB    = The low-level unit activation state, values depend on unit type.

1 loaded units listed.
```

Pierwsza komenda do odpalenia po alercie „coś nie działa, ale nie wiadomo co" — filtruje od razu do tego, co realnie wymaga uwagi.

systemctl = system control: sterowanie usługami systemd

TYLKO ODCZYT

Pokazanie drzewa zależności: `systemctl list-dependencies`

```
systemctl list-dependencies nginx
```

Pokazuje, od czego dana jednostka zależy (i pośrednio — co ją zablokuje, jeśli zależność nie wystartuje). Przydaje się, gdy usługa nie chce ruszyć, a `status` samej usługi wygląda niewinnie.

> **Pojęcie: drop-in.** Osobny plik `.conf` w katalogu `nazwa.service.d/`, który dopisuje albo nadpisuje pojedyncze ustawienia jednostki, nie ruszając jej oryginalnego pliku. systemd wczytuje najpierw plik bazowy, a potem nakłada na niego wszystkie pliki drop-in z tego katalogu, w kolejności alfabetycznej. To pozwala zmienić jedną dyrektywę bez kopiowania i utrzymywania całego pliku jednostki.

systemctl = system control: sterowanie usługami systemd

TYLKO ODCZYT

Pokazanie złożonego efektu wszystkich warstw: `systemctl cat`

```
systemctl cat nginx
```

Pokazuje plik jednostki dokładnie tak, jak widzi go systemd — bazę i wszystkie nałożone na nią drop-iny, po kolei, z nazwami plików źródłowych.

Przykład:

```
# /usr/lib/systemd/system/nginx.service
[Unit]
Description=The nginx HTTP and reverse proxy server
After=network.target remote-fs.target nss-lookup.target

[Service]
```

```
Type=forking
PIDFile=/run/nginx.pid
ExecStartPre=/usr/sbin/nginx -t
ExecStart=/usr/sbin/nginx
ExecReload=/bin/kill -s HUP $MAINPID
KillSignal=SIGQUIT
TimeoutStopSec=5
KillMode=mixed

[Install]
WantedBy=multi-user.target

# /etc/systemd/system/nginx.service.d/override.conf
[Service]
LimitNOFILE=65535
```

Drugi blok (`LimitNOFILE=65535`) pochodzi z drop-ina widocznego wcześniej w `Drop-In:` w wyniku `systemctl status` — `cat` pokazuje, skąd dokładnie się wziął.

□□ **systemctl** = system control: sterowanie usługami systemd

□□ TYLKO ODCZYT

Bezpieczna edycja jednostki przez drop-in: `systemctl edit`

```
sudo systemctl edit nginx
```

TAK się edytuje jednostkę zainstalowaną z pakietu. Otwiera edytor nad nowym plikiem drop-in w `nginx.service.d/override.conf`, zamiast dotykać oryginału — przykład: `Environment=PGDATA=/postgres/16/data` dla usługi bazy danych. Po zapisaniu `systemctl edit` sam wykonuje `daemon-reload`.

□□ **systemctl** = system control: sterowanie usługami systemd

□□ TYLKO ODCZYT

Utworzenie pełnej kopii jednostki do edycji: `systemctl edit --full`

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

Kopiuje **cały** plik jednostki do `/etc/systemd/system/` i zasłania nim wersję z pakietu na stałe. Rzadziej właściwe niż zwykły `edit` — sięgamy po to tylko wtedy, gdy zmiana dotyczy większości pliku, a nie pojedynczej dyrektywy.

systemctl = system control: sterowanie usługami systemd

ZMIENIA SYSTEM

Przeładowanie jednostek po ręcznej zmianie pliku: `systemctl daemon-reload`

```
sudo systemctl daemon-reload
```

Każe systemd ponownie wczytać pliki jednostek z dysku. Obowiązkowe po **każdej** ręcznej edycji pliku jednostki (`vim`, `nano`, `sed -i`) — bez tego `restart` użyje starej, zapamiętanej wersji. `systemctl edit` robi to automatycznie za nas.

systemctl = system control: sterowanie usługami systemd

TYLKO ODCZYT

Sprawdzenie domyślnego targetu: `systemctl get-default`

```
systemctl get-default
```

Przykład:

```
multi-user.target
```

systemctl = system control: sterowanie usługami systemd

ZMIENIA SYSTEM

Zmiana domyślnego targetu: `systemctl set-default`

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

Zmienia, który target wystartuje przy **następnym** rozruchu. Nie przełącza niczego teraz.

systemctl = system control: sterowanie usługami systemd

TYLKO ODCZYT

Natychmiastowe przełączenie targetu: `systemctl isolate`

```
sudo systemctl isolate rescue.target
```

Przełącza system na inny target od razu, zatrzymując wszystko, co nie należy do docelowego targetu. Ostrzejsze niż `set-default` — na produkcji to operacja świadoma, nie pomyłka w Tab-completion.

systemctl = system control: sterowanie usługami systemd

TYLKO ODCZYT

Sprawdzenie, co było wolne przy rozruchu: `systemd-analyze blame`

Samo `systemd-analyze` (bez podkomendy) wypisuje tylko łączny czas rozruchu, rozbity na firmware/loader/kernel/userspace. `blame` idzie dalej i sortuje jednostki według czasu inicjalizacji:

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

Przykład:

```
33.812s postgresql.service
12.401s NetworkManager-wait-online.service
 4.220s systemd-udev-settle.service
 2.005s firewalld.service
 1.310s tuned.service
  897ms systemd-journald.service
  621ms polkit.service
  455ms nginx.service
```

Pokazuje, co było *wolne*, niekoniecznie co wydłużyło cały rozruch — jednostka wolna, ale uruchamiana równolegle z czymś jeszcze wolniejszym, nie liczy się do czasu całkowitego.

TYLKO ODCZYT

Sprawdzenie, co naprawdę wydłużyło rozruch: `systemd-analyze critical-chain`

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

Pokazuje najdłuższy łańcuch zależności, który realnie zdecydował o czasie rozruchu — to jest lista do optymalizacji, nie `blame`.

TYLKO ODCZYT

Zarządzanie jednostkami bez roota: `systemctl --user`

```
systemctl --user status
systemctl --user list-units
```

Ten sam `systemctl`, ale działający na osobnym, prywatnym menedżerze systemd uruchamianym dla naszego konta przy logowaniu — bez `sudo`, bez wpływu na resztę systemu. Jednostki `--user` trzymamy w `~/.config/systemd/user/`.

> **Pojęcie: jednostka (unit).** W systemd wszystko, czym on zarządza — usługa, gniazdo sieciowe, punkt montowania, harmonogram — jest jednostką: plikiem tekstowym opisującym, jak coś uruchomić, od czego to zależy i kiedy ma działać. Rozszerzenie nazwy pliku mówi, jakiego rodzaju jest to jednostka. Poniższa tabela pokazuje wszystkie rodzaje, jakie na co dzień spotykamy.

systemctl = system control: sterowanie usługami systemd

TYLKO ODCZYT

Rodzaje jednostek

Typ	Rozszerzenie	Do czego służy
service	`.service`	proces demona — najczęściej używany typ
socket	`.socket`	gniazdo sieciowe/Unix; uruchamia usługę przy pierwszym połączeniu
target	`.target`	grupa jednostek, punkt synchronizacji (odpowiednik runlevela)
timer	`.timer`	harmonogram, odpowiednik wpisu w cronie

Typ	Rozszerzenie	Do czego służy
mount	`.mount`	punkt montowania, generowany też z `/etc/fstab`
path	`.path`	uruchamia jednostkę przy zmianie pliku/katalogu (inotify)
slice	`.slice`	grupa cgroup do limitowania zasobów wielu jednostek
scope	`.scope`	proces spoza systemd, objęty jego nadzorem post factum

Uważaj

- `enable` bez `--now` **nie uruchamia usługi** — tylko ustawia autostart przy następnym boocie. Po zmianie pliku jednostki zawsze `daemon-reload`, inaczej systemd dalej trzyma starą wersję w pamięci.
- `mask` to nie to samo co `disable` — zamaskowanej jednostki nie uruchomisz nawet ręcznym `systemctl start`, dopóki jej nie odmaskujesz.
- Edycja pliku w `/usr/lib/systemd/system/` przepada przy najbliższej aktualizacji pakietu. Właściwe miejsce na zmiany administratora to drop-in przez `systemctl edit`.

Z życia wzięte

Z ŻYCIA WZIĘTE

O 10:00 Tomek zmienia `ExecStart` w jednostce usługi na app01, żeby wskazać nową wersję binarki. Robi `systemctl restart`, a usługa dalej odpala starą wersję, bo zapomniał o `daemon-reload` po edycji pliku. Dopiero po `systemctl daemon-reload && systemctl restart` nowa wersja faktycznie startuje.

Po co to istnieje

Pojęcie: demon i system inicjalizacyjny. Demon (ang. *daemon*) to proces działający w tle, bez terminala i bez interakcji z użytkownikiem — serwer WWW, baza danych, usługa SSH . System inicjalizacyjny to pierwszy proces, jaki jądro Linuksa uruchamia po starcie (dostaje

> PID 1) i który z kolei uruchamia oraz nadzoruje wszystkie pozostałe demony. Zanim pojawił się systemd, tę rolę pełnił SysV init — dużo prostszy program, który tylko odpalał skrypty po kolei i nie pilnował, co się z ich procesami dzieje dalej.

Przed systemd standardem był SysV init: skrypty powłoki w `/etc/init.d/`, uruchamiane po kolei w kolejności wyznaczonej numerkami w nazwach symlinków (`S01network`, `S02sshd`, `S50nginx`...). Kolejność była płaska i ręcznie utrzymywana — nikt nie deklarował „potrzebuję sieci", tylko ktoś musiał zadbać, żeby numerek sieci był niższy niż numerek wszystkiego, co jej potrzebuje. Skrypty startowały jeden po drugim, nawet gdy nie miały ze sobą nic wspólnego. A kiedy demon forkował się i uciekał w tło, init tracił go z oczu — jeśli proces spłodził dzieci, które go przeżyły, nikt formalnie ich nie pilnował.

systemd (2010, Lennart Poettering i Kay Sievers) zastąpił to jako proces PID 1 z zupełnie innym modelem. Realne zmiany, które to wprowadziło, to nie „szybszy boot" jako slogan, tylko cztery konkretne rzeczy:

> **Pojęcie: cgroup i nadzór nad procesami.** *Control group* (cgroup) to mechanizm jądra Linuksa, który grupuje procesy i pozwala jądru mierzyć oraz ograniczać ich wspólne zużycie CPU, pamięci i innych zasobów jako jednej całości. systemd tworzy jedną cgroup na każdą uruchomioną jednostkę i wrzuca do niej każdy proces, który ta jednostka odpali — łącznie z dziećmi i wnukami, nawet jeśli oryginalny proces dawno się zakończył. Dzięki temu systemd wie dokładnie, co należy do której usługi, i potrafi zabić całe drzewo procesów naraz, zamiast gonić za uciekającymi potomkami.

- **Start równoległy** — systemd buduje graf zależności z plików jednostek i wypuszcza naraz wszystko, co może wystartować już teraz.
- **Zależności deklaratywne** — piszesz `After=network-online.target`, a nie numerek. Kolejność wynika z realnej potrzeby, nie z arytmetyki nazw plików.
- **Realny nadzór przez cgroups** — każdy proces jednostki, łącznie z potomkami, trafia do jej grupy kontrolnej. systemd wie, co do czego należy, nawet gdy demon się rozgałęzia, i zabija całe drzewo naraz przy zatrzymaniu.
- **Jednolite logowanie** — `journald` zbiera stdout/stderr każdej jednostki i komunikaty jądra w jednym miejscu (strona *journalctl — logi systemd*).

Nawet kilkanaście lat po przejściu na systemd trafiają się aplikacje, które przy instalacji wciąż podpinają się pod warstwę kompatybilności SysV zamiast tworzyć własną, natywną jednostkę. Typowy objaw: `systemctl enable nazwa.service` kończy się błędem `Failed to execute operation: No such file or directory`, mimo że usługa startuje bez problemu po ręcznym `systemctl start`. Winny bywa brakujący albo uszkodzony skrypt kompatybilności `/usr/lib/systemd/systemd-sysv-install`. Naprawa doraźna to jego odtworzenie; naprawa właściwa to napisanie dla aplikacji zwykłej jednostki `.service` — upstream systemd oznaczył całą warstwę SysV jako przestarzałą w wydaniu 258 (koniec 2025) i usunął jej obsługę w wydaniu 260 (2026). RHEL 9 i 10 niosą własne, starsze wersje systemd (odpowiednio ok. 252 i 257), ale kierunek upstreamu jest jasny.

Analogia

Pomyśl o **dyspozytorze w dużej zajezdni autobusowej**.

Naiwny dyspozytor wypuszczałby autobusy po kolei według numerków linii: najpierw 1, potem 2, potem 3, każdy dopiero, gdy poprzedni zjedzie z placu — nieważne, że linia 7 nie potrzebuje niczego, co robi linia 3, i mogłaby wyjechać dużo wcześniej. Tak działał SysV init.

systemd to dyspozytor z mapą zależności zamiast numerków: „linia 12 potrzebuje otwartej bramy i zatankowanego autobusu; linia 7 nie potrzebuje niczego oprócz kierowcy". Wypuszcza równolegle wszystko, co może jechać już teraz, i pilnuje, żeby zależności faktycznie zostały spełnione, zanim coś ruszy.

Druga część roboty dyspozytora — tu analogia sięga do cgroups — to pilnowanie, żeby każdy autobus dało się rozliczyć, nawet jeśli po drodze przesiadł pasażerów na inny pojazd. SysV-owy dyspozytor tracił autobus z oczu, gdy tylko skrypt startowy się kończył. systemd przypisuje każdy proces, łącznie z potomkami, do konkretnej „trasy" (cgroup jednostki) i wie dokładnie, co do czego należy.

Ograniczenie tej analogii: dyspozytor nie wymyśla tras sam — potrzebuje, żeby ktoś (administrator albo autor pakietu) najpierw napisał rozkład jazdy, czyli plik jednostki z poprawnie zadeklarowanymi zależnościami. Zła mapa zależności (brakujące `After=`, złe `Wants=`) to błąd rozkładu jazdy, nie błąd dyspozytora — systemd wykona dokładnie to, co napisano, nawet jeśli kolejność wynikowa nie ma sensu.

Jak to działa

Rozwiń: jak to działa pod spodem

Czytanie `systemctl status` linia po linii. To najważniejsza umiejętność z tej strony — reszta to głównie warianty tego samego pytania: „co się dzieje z tą jednostką". Wróć do wyniku `systemctl status nginx` pokazanego w Fiszce i przeczytaj go pole po polu:

- **Loaded** — skąd systemd wziął jednostkę, czy jest `enabled`, i czy ma zastosowany drop-in (`Drop-In`).
- **Active** — stan bieżący (`active`, `inactive`, `failed`, `activating`...), od kiedy i jak długo.
- **Main PID** — proces uznany za „ten" proces jednostki — na nim opiera się logika `Type=` (patrz dalej).
- **Tasks** — liczba wątków/procesów jednostki względem limitu (`TasksMax`); przekroczenie to częsta przyczyna awarii pod obciążeniem.

- **Memory**/**CPU** — zużycie liczone przez cgroup jednostki, nie ręcznym sumowaniem PID-ów.
- **CGroup** — realne drzewo procesów nadzorowane jako ta jednostka; dowód, że nadzór działa przez cgroups, nie przez jeden zapamiętany PID.

Hierarchia nadpisywania. `/usr/lib/systemd/system/` (Debian/Ubuntu: `/lib/systemd/system/`) to pliki pakietu, nadpisywane bez ostrzeżenia przy aktualizacji; `/etc/systemd/system/` to pliki administratora — plik tu o tej samej nazwie **całkowicie zastępuje** wersję z `/usr/lib`; katalogi drop-in `*.service.d/*.conf` (w obu lokalizacjach) **dopisują się** do istniejącego pliku, niezależnie od tego, gdzie on leży.

`daemon-reload` — dlaczego jest obowiązkowy. systemd trzyma sparsowaną reprezentację jednostek w pamięci od momentu wczytania. Ręczna edycja pliku na dysku nie zmienia tej reprezentacji, dopóki nie każesz mu wczytać jej ponownie — bez `daemon-reload` `restart` użyje **starej** konfiguracji. `systemctl edit` robi to automatycznie po zapisaniu; ręczna edycja przez `vim`/`nano` — nie.

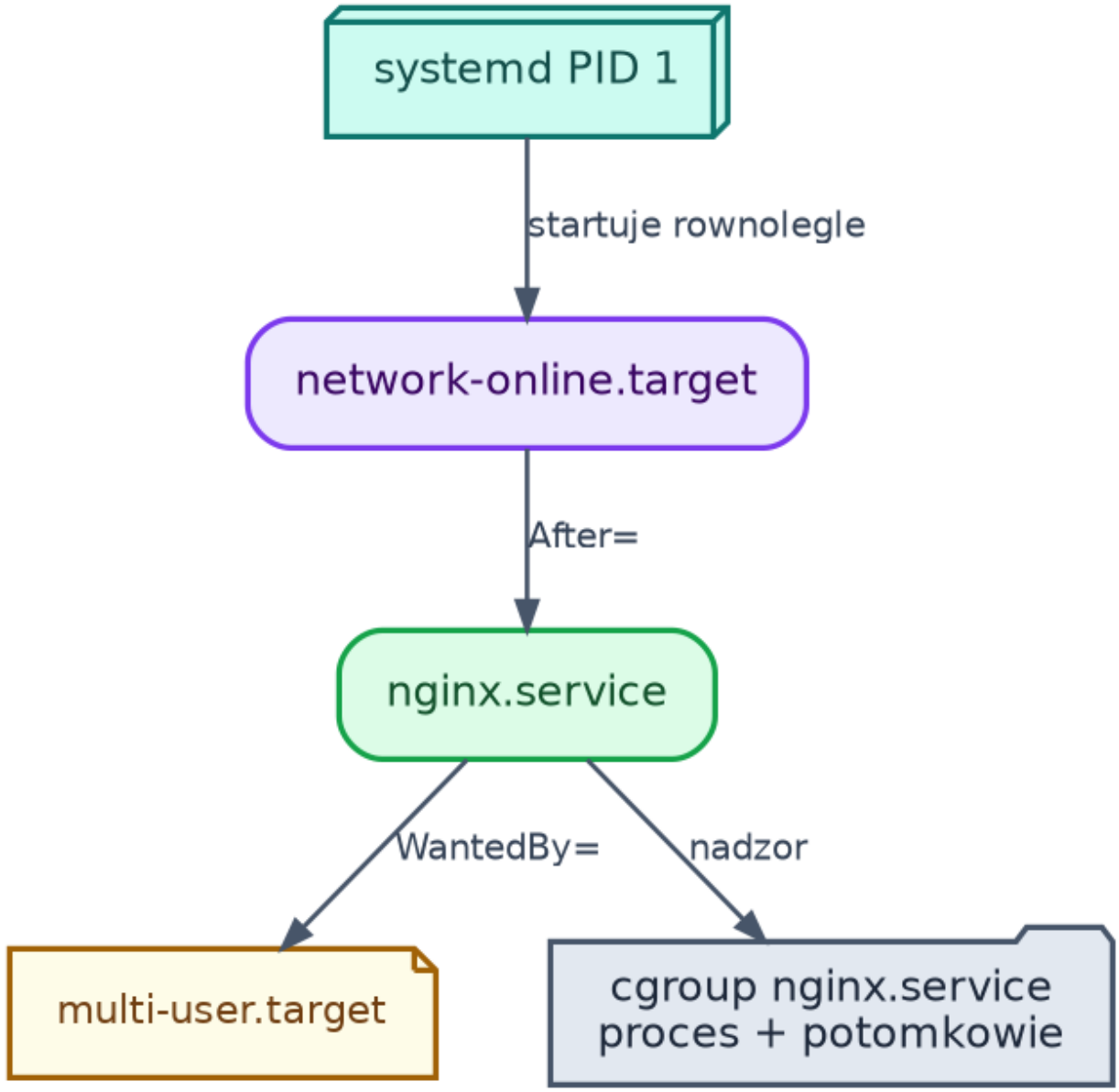
Anatomia pliku `.service`. Kompletny, w pełni działający przykład z komentarzem przy każdej ważnej dyrektywie:

```
[Unit]
Description=Prosta usługa demonstracyjna
After=network-online.target
Wants=network-online.target

[Service]
Type=simple
ExecStartPre=/usr/bin/echo "startuję demo"
ExecStart=/usr/local/bin/demo-app --port=8080
ExecStop=/usr/bin/kill -TERM $MAINPID
User=demo
Group=demo
WorkingDirectory=/opt/demo
Environment=APP_ENV=production
EnvironmentFile=-/etc/demo/env
Restart=on-failure
RestartSec=5
TimeoutStopSec=20

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

`After=`/`Wants=` to luźna kolejność; `Requires=` wymusza start zależności; `BindsTo=` idzie dalej — jednostka kończy się razem z zależnością; `Before=` odwraca kierunek.



Jednostki tworzą graf zależności: Wants i Requires decydują, co musi wystartować, After i Before o kolejności, a systemd startuje równolegle wszystko, co może już ruszyć (kliknięcie otwiera pełny rozmiar)

`ExecStop=` woła `kill -TERM` tylko dla demonstracji (sygnały: *Sygnały: kill, pkill, killall, trap*) — domyślny `systemctl stop` zwykle wystarcza bez tego. Myślnik w `EnvironmentFile=-/etc/demo/env`

oznacza „wczytaj, jeśli plik istnieje, nie failuj jeśli go nie ma".

Typy z `Type=` różnią się tym, **kiedy** systemd uznaje jednostkę za wystartowaną:

Type	Kiedy systemd uznaje start za zakończony
`simple` (domyślny)	natychmiast po uruchomieniu `ExecStart=`
`exec`	po udanym `execve()` procesu `ExecStart=` — precyzyjniej niż `simple`
`forking`	gdy proces macierzysty się zakończy (klasyczny demon forkujący w tło; zwykle wymaga `PIDFile=`)
`oneshot`	po zakończeniu `ExecStart=`; z `RemainAfterExit=yes` jednostka zostaje `active` mimo braku procesu
`notify`	gdy proces zgłosi gotowość przez `sd_notify()` — wymaga wsparcia w kodzie aplikacji

Dlaczego usługa nie widzi Twojego PATH-a. Proces systemd nie dziedziczy `~/.bashrc` ani `/etc/profile`, tylko wbudowaną minimalną wartość `PATH`, chyba że jawnie ustawisz `Environment=PATH=...`. Objaw: skrypt działa z terminala, a jako usługa pada z `command not found`. Pełny mechanizm: *Zmienne środowiskowe i PATH*.

> **Pojęcie: target.** Jednostka typu `.target` nie uruchamia żadnego procesu — grupuje inne jednostki i działa jako punkt synchronizacji: „dotarliśmy tutaj, gdy wszystko wymagane już wystartowało". systemd używa targetów tam, gdzie SysV init używał numerków runlevela, z tą różnicą, że target może zależeć od dowolnej kombinacji usług, a nie tylko od jednego numeru.

Target	Odpowiednik runlevela SysV	Co robi
`poweroff.target`	0	wyłączenie systemu
`rescue.target`	1	tryb jednego użytkownika, minimalny zestaw usług
`multi-user.target`	3	pełny system wieloużytkownikowy, bez GUI
`graphical.target`	5	pełny system z GUI (obejmuje `multi-user.target`)
`reboot.target`	6	restart systemu

```
systemctl get-default             # jaki target startuje domyślnie
sudo systemctl set-default multi-user.target
sudo systemctl isolate rescue.target   # przełącz NATYCHMIAST — zatrzyma usługi spoza
```

```
rescue.target
```

`isolate` jest ostrzejsze niż zmiana defaultu — zatrzymuje od razu wszystko spoza docelowego targetu; na produkcji to operacja świadoma, nie pomyłka w Tab-completion.

`systemd-analyze blame` i **`critical-chain`**. `blame` sortuje jednostki według czasu inicjalizacji — pokazuje, co było *wolne*, niekoniecznie co wydłużyło cały rozruch (wolna jednostka czekająca równolegle z czymś jeszcze wolniejszym nie liczy się do czasu całkowitego). `critical-chain` pokazuje właśnie to: najdłuższy łańcuch zależności, który realnie zdecydował o czasie rozruchu — to lista do optymalizacji.

Krok po kroku

Budujemy kompletną, działającą jednostkę `--user` od pliku do uruchomienia — bez roota, bez wpływu na resztę systemu.

1. Przygotuj katalog i napisz plik jednostki.

```
mkdir -p ~/.config/systemd/user
cat > ~/.config/systemd/user/demo.service <<'EOF'
[Unit]
Description=Zegar demonstracyjny do ćwiczeń z systemd

[Service]
Type=simple
ExecStart=/bin/bash -c 'while true; do date >> /tmp/demo-systemd.log; sleep 5; done'
Restart=on-failure
RestartSec=3

[Install]
WantedBy=default.target
EOF
```

2. Wczytaj nową jednostkę i uruchom ją.

```
systemctl --user daemon-reload
systemctl --user start demo.service
systemctl --user status demo.service
```

```
● demo.service - Zegar demonstracyjny do ćwiczeń z systemd
     Loaded: loaded (/home/adm_user/.config/systemd/user/demo.service; disabled; preset:
enabled)
     Active: active (running) since Thu 2026-09-17 15:10:02 UTC; 4s ago
   Main PID: 28841 (bash)
     CGroup: /user.slice/.../demo.service
             ├─28841 /bin/bash -c while true; do date >> /tmp/demo-systemd.log; sleep 5; done
             └─28855 sleep 3
```

Zauważ `disabled` przy `Loaded` — `start` uruchomił usługę na teraz, ale nie ustawił autostartu.

3. Sprawdź różnicę między `start` a `enable`, a potem na żywo, dlaczego `daemon-reload` jest obowiązkowy.

```
systemctl --user enable demo.service
systemctl --user is-enabled demo.service
```

```
enabled
```

`enable` tylko dopisał symlink autostartu na kolejne logowanie — usługa działa dalej dzięki wcześniejszemu `start`. Teraz zmień plik i zrestartuj bez przeładowania:

```
sed -i 's/sleep 5/sleep 1/' ~/.config/systemd/user/demo.service
systemctl --user restart demo.service
```

Interwał w logu wciąż wynosi 5 sekund — `restart` używa reprezentacji w pamięci, nie tego, co leży na dysku. Dopiero `systemctl --user daemon-reload && systemctl --user restart demo.service` wczytuje nową wersję i log zaczyna rosnąć co sekundę.

4. Dołóż override przez drop-in, bez dotykania pliku bazowego.

```
systemctl --user edit demo.service
```

W otwartym edytorze dopisz:

```
[Service]
Environment=DEMO_ENV=cwiczenie
```

Po zapisaniu `systemctl --user edit` sam robi za Ciebie `daemon-reload`. Sprawdź złożony efekt:

```
systemctl --user cat demo.service
```

5. Posprzątaj po ćwiczeniu.

```
systemctl --user disable --now demo.service
rm ~/.config/systemd/user/demo.service
rm -rf ~/.config/systemd/user/demo.service.d
rm -f /tmp/demo-systemd.log
systemctl --user daemon-reload
```

Częste błędy

`enable` bez `start` — usługa „włączona", a nie działa.

```
$ sudo systemctl enable nginx
Created symlink /etc/systemd/system/multi-user.target.wants/nginx.service →
/usr/lib/systemd/system/nginx.service.
$ curl -s -o /dev/null -w '%{http_code}\n' http://localhost
curl: (7) Failed to connect to localhost port 80
```

`enable` tylko podpina symlink w `.wants/` docelowego targetu — zadziała przy **następnym** rozruchu, nie teraz. Jedno i drugie od razu: `systemctl enable --now nginx`.

Brak `daemon-reload` po edycji jednostki.

```
$ sudo vim /etc/systemd/system/moja-app.service     # zmieniasz ExecStart=
$ sudo systemctl restart moja-app
```

Usługa restartuje się starą komendą — systemd nadal trzyma w pamięci to, co wczytał przy poprzednim `daemon-reload`. Naprawa: `sudo systemctl daemon-reload && sudo systemctl restart moja-app`.

Edycja pliku w `/usr/lib/systemd/system/` zamiast drop-ina.

```
$ sudo vim /usr/lib/systemd/system/nginx.service     # dopisujesz własne ustawienia
```

Działa — do najbliższej aktualizacji pakietu, która nadpisze ten plik bez ostrzeżenia, kasując zmiany. Właściwe miejsce to `systemctl edit nginx`, które tworzy drop-in w `nginx.service.d/override.conf` — ten katalog przetrwa aktualizację, bo menedżer pakietów go nie dotyka.

`Type=forking` bez `PIDFile=` (albo zły `Type=` w ogóle) — start „wisi", a potem timeout.

```
Job for legacy-app.service failed because a timeout was exceeded.
See "systemctl status legacy-app.service" and "journalctl -xeu legacy-app.service" for
details.
```

Jeśli demon forkuje się w tło (proces startowy kończy się, prawdziwy demon zostaje), a jednostka ma `Type=simple` zamiast `Type=forking`, systemd uznaje za „główny proces" ten, który natychmiast się kończy — i albo widzi jednostkę jako martwą, albo (przy `forking` bez `PIDFile=`) nie wie, którego procesu pilnować. Sprawdź w dokumentacji aplikacji, jak się zachowuje, i dobierz `Type=` zgodnie z tym.

Oczekiwanie, że usługa zobaczy `PATH` z `.bashrc` administratora — kod wyjścia `127` (patrz sekcja wyżej). Rozwiązanie: jawny `Environment=PATH=...` albo pełna ścieżka do binarki w `ExecStart=`, nie poleganie na środowisku interaktywnej powłoki.

□□ Ćwiczenia

> Wszystko poniżej wykonujesz na własnej maszynie, w katalogu `/tmp`. Nic nie wymaga uprawnień administratora.

Ćwiczenie 1 — timer i `Type=oneshot`

Zbuduj timer `--user`, żeby zobaczyć w praktyce jednostkę typu `timer` i `Type=oneshot` z tabeli w Fiszce — bez roota.

```
cat > ~/.config/systemd/user/znacznik.service <<'EOF'
[Unit]
Description=Zapisz znacznik czasu (oneshot, wywoływany przez timer)

[Service]
Type=oneshot
ExecStart=/bin/bash -c 'echo "znacznik: $(date)" >> /tmp/demo-timer.log'
EOF

cat > ~/.config/systemd/user/znacznik.timer <<'EOF'
[Unit]
Description=Uruchamiaj znacznik.service co minutę

[Timer]
```

```
OnCalendar=*-*-* *:*:00
Persistent=true

[Install]
WantedBy=timers.target
EOF

systemctl --user daemon-reload
systemctl --user enable --now znacznik.timer
systemctl --user list-timers
```

Odczekaj dwie–trzy minuty i sprawdź wynik:

```
cat /tmp/demo-timer.log
journalctl --user -u znacznik.service --since "-5min"
```

Czego się spodziewać

`list-timers` pokaże `znacznik.timer` z kolumnami `NEXT` (kiedy odpali się następnym razem) i `LAST` (kiedy ostatnio odpalił `znacznik.service`). Plik `/tmp/demo-timer.log` powinien mieć nowy wpis co minutę. Zauważ, że sam `znacznik.service` nigdy nie jest `enabled` ani uruchamiany bezpośrednio — to `znacznik.timer` jest jednostką z autostartem, a on z kolei aktywuje powiązaną usługę o tej samej nazwie bazowej. Posprzątaj: `systemctl --user disable --now znacznik.timer && rm ~/.config/systemd/user/znacznik.* /tmp/demo-timer.log`.

Pytania kontrolne

1. `systemctl enable postgresql` zwraca sukces, ale baza danych nie odpowiada na porcie. Dlaczego, i jaka jedna komenda naprawiłaby to od razu?

Odpowiedź

`enable` tylko ustawia autostart na kolejny rozruch — nie uruchamia niczego teraz. `systemctl enable --now postgresql` zrobiłoby jedno i drugie naraz.

2. Zmieniłeś `ExecStart=` w `/etc/systemd/system/moja-app.service`, zrobiłeś `systemctl restart moja-app`, a usługa dalej uruchamia się starą komendą. Co pominąłeś?

Odpowiedź

`systemctl daemon-reload`. systemd trzyma sparsowaną wersję jednostki w pamięci od ostatniego wczytania i `restart` używa tego, co już wie, dopóki nie każesz mu wczytać pliku ponownie.

3. Czym różni się `mask` od `disable`, i kiedy sięgniesz po `mask`?

Odpowiedź

`disable` usuwa symlink autostartu — usługę wciąż da się uruchomić ręcznie. `mask` podmienia jednostkę na symlink do `/dev/null`, więc nawet ręczny `start` się nie powiedzie, dopóki nie zrobisz `unmask`. Sięgasz po `mask`, gdy inna jednostka próbuje uruchomić ją jako zależność, a Ty świadomie chcesz to zablokować.

4. Twoja jednostka `.service` ma `Type=simple`, a aplikacja w rzeczywistości forkuje się w tło i kończy proces startowy. Co zobaczysz w `systemctl status` i dlaczego?

Odpowiedź

Przy `Type=simple` systemd uznaje proces z `ExecStart=` za „główny" natychmiast. Gdy ten proces kończy się po forknięciu właściwego demona, systemd widzi to jako zakończenie jednostki — status pokaże ją jako nieaktywną, mimo że demon dalej działa w tle, niezauważony. Rozwiązanie: `Type=forking` z ustawionym `PIDFile=`.

Powiązane: Zmienne środowiskowe i PATH · journalctl: logi systemd · Sygnały: kill, pkill, killall, trap · Kod wyjścia, $?, &&, ||, średnik

Źródła: `man 1 systemctl` · `man 5 systemd.unit` · `man 5 systemd.service` · `man 7 systemd.special` · `man 1 systemd-analyze` · systemd 258 — Linuxiac · systemd 258: cgroup v1 removal — linuxiac · systemd 261 — Help Net Security

---

Revision #19
Created 2026-09-17 15:07:24 UTC by Tomasz Konieczny
Updated 2026-10-03 19:08:22 UTC by Tomasz Konieczny