Systemd

"**System and service manager**" - an init system used to bootstrap user space and manage user processes.

- Can be found on any modern Linux OS versions at or above some version
  - Ubuntu 15.04, Debian 8, CentOS 7, Fedora 15, and many more…

History

- **2010** - Lennart Poettering and Kay Sievers, the software engineers working for RedHat who initially developed systemd, started a project to replace Linux's conventional "System V init".
- **April 2010** blog post from Poettering, titled **Rethinking PID 1**, introduced an experimental version of what would later become **systemd**.
- **May 2011** Fedora Linux became the first major Linux distribution to enable systemd by default, replacing upstart.
- **October 2012** Arch Linux made systemd the default, switching from SysVinit.
- **2013 - 2014** Debian undergoes a massive discussion in the respective mailing list about pros/cons of switching to systemd…. resulting in definitive switching to systemd (from SysVinit).
- **2014** - Ubuntu switches to systemd too, mostly based on previous Debian decision (dropping their own **upstart**)
- There was(and still is) a solid portion of controversy among the Linux world about switching to systemd, more or less abandoning other alternatives. We try not to take any stands in this course, but for those who are interested:
  - The closest to a complex all in one technical & historical summary we were able to find(2022) is probably this .

Not everybody has switched.

Linux distributions without systemd here.

Design

- In May 2014, Poettering described systemd as unifying *"pointless differences between distributions"*, by providing the following three general functions:
  - A system and service manager (manages both the system, by applying various configurations, and its services)
  - A software platform (serves as a basis for developing other software)
  - The glue between applications and the kernel (provides various interfaces that expose functionalities provided by the kernel)
- Systemd is definitely much more than a bunch of shell scrips we used to know form SysVinit.

Basic info

- Systemd is a suite of basic building blocks for a Linux system
- System and service manager that runs as PID 1 and starts the rest of the system

  - aggressive parallelization capabilities
  - uses socket and D-Bus activation for starting services
  - offers on-demand starting of daemons
  - keeps track of processes using Linux control groups (cgroups)
  - maintains mount and automount points,
  - implements an elaborate transactional dependency-based service control logic
  - supports SysV and LSB init scripts - works as a replacement for SysVinit

- ### Other functions of systemd
  - logging daemon
  - utilities to control basic system configuration like the hostname, date, locale,
  - maintain a list of logged-in users `loginctl`
  - start & run containers and virtual machines
  - system accounts
  - runtime directories and settings and daemons
  - manage simple network configuration
  - network time synchronization
  - log forwarding
  - name resolution (`resolvectl status`)

- ## Units introduction
  - The basic object that systemd manages and acts upon is a “**unit**”.
  - Units commonly include, but are not limited to,
    - services (.service)
    - mount points (.mount)
    - devices (.device)
    - sockets (.socket)
  - To manage services on a systemd enabled machine
    - main tool is the **`systemctl`** command
      - (**Warning**: not to be confused with `sysctl`)

```
user@machine$ man systemctl
systemctl
SYSTEMCTL(1)
NAME
       systemctl - Control the systemd system and service manager
SYNOPSIS
       systemctl [OPTIONS...] COMMAND [UNIT...]
DESCRIPTION
       systemctl may be used to introspect and control the state of the
"systemd" system and service manager. Please refer to systemd(1) for an
introduction into the basic concepts and functionality this tool
manages.
```

- ### Available unit types
  - **Target** - A group of units that defines a synchronization point. The synchronization point is used at boot time to start the system in a particular state.
  - **Service** - Starts, stops, restarts or reloads a service daemon, such as Apache webserver.

  - **Socket** - Activates a service when the service receives incoming traffic on a listening socket.
  - **Device** - Implements device-based activation such as a device driver.
  - **Mount** - Controls the file-system mount point.
  - **Automount** - Provides and controls on-demand mounting of file systems.
  - **Swap** - Encapsulates/activates/deactivates swap partition.
  - **Path** - Monitors files/directories and activates/deactivates a service if the specified file or directory is accessed. Uses `inotify` (Basically like socket, but for files)
  - **Timer** - Activates/deactivates specified service based on a timer or when the set time is elapsed.
  - **Snapshot** - Creates and saves the current state of all running units. This state can be used to restore the system later.
  - **Slice** - A group of units that manages system resources such as CPU, and memory… allowing resources to be restricted or assigned to any processes associated with the slice. The name reflects its hierarchical position within the `cgroup` tree.
  - **Scope** - Organizes and manages foreign processes.
  - **..busname..** - A unit that controls DBus system.

-
  ```
  user@machine:~$ systemctl -t help
  Available unit types:
  service
  mount
  swap
  socket
  target
  device
  automount
  timer
  path
  slice
  scope
  user@machine:~$
  ```

- ## Basic usage

  - start the service

    ```
    user@machine$sudo systemctl start nginx.service
    ```

    - or

      ```
      user@machine$sudo systemctl start nginx
      ```

  - If you do not specify the suffix, systemctl will assume .service.
    - `netctl` and `netctl.service` are equivalent
  - Mount points will automatically be translated into the appropriate `.mount` unit.
    - specifying `/home` is equivalent to `home.mount`
  - Devices are automatically translated into the appropriate `.device` unit
    - specifying `/dev/sda2` is equivalent to `dev-sda2.device`
  - Commands by default operate on system units since `–system` is the implied default for `systemctl`.

    - To operate on *user units* (for the calling user), use `systemctl —user` without root privileges.

The @ sign

- Some unit names contain an **@** sign (e.g. `name@string.service`).
- This means that they are **instances** of a **template** unit, whose actual file name does not contain the string part (e.g. `name@.service`).
- "`string`" is called the **instance identifier**, and is similar to an argument that is passed to the template unit when called with the `systemctl` command:
    - in the unit file it will substitute the `%i` specifier.
    - To be more accurate, before trying to instantiate the `name@.suffix` template unit, systemd will actually look for a unit with the exact name@string.suffix file name,
        - although by convention such a "clash" happens rarely,
        - i.e. most unit files containing an @ sign are **meant to be templates**.
        - Also, if a template unit is called **without an instance identifier, it will generally fail**
            - (except with certain systemctl commands, like `cat`).

Basic Unit Management

- start the service

```
$ sudo systemctl start nginx.service
```

- stop it

```
$ sudo systemctl stop nginx.service
```

- restart the service

```
$ sudo systemctl restart nginx.service
```

- attempt to reload the service without interrupting normal functionality

```
$ sudo systemctl reload nginx.service
```

- if unsure whether the service has the functionality to reload its configuration

```
$sudo systemctl reload-or-restart application.service
```

Enabling or Disabling Units

To configure unit to start automatically at boot, you need to "enable" the unit. This hooks it up to a certain boot "target", causing it to be triggered when that target is started.

- enable a service to start automatically at boot

```
$ sudo systemctl enable nginx.service
```

- disable the service

```
$ sudo systemctl disable nginx.service
```

- **Unit status overview**
  - To see an overview of the current state of a unit.

```
$ systemctl status application.service
```

This will show you whether the unit is active, information about the process, and the latest journal entries.

  - check to see if a unit is currently active (running)

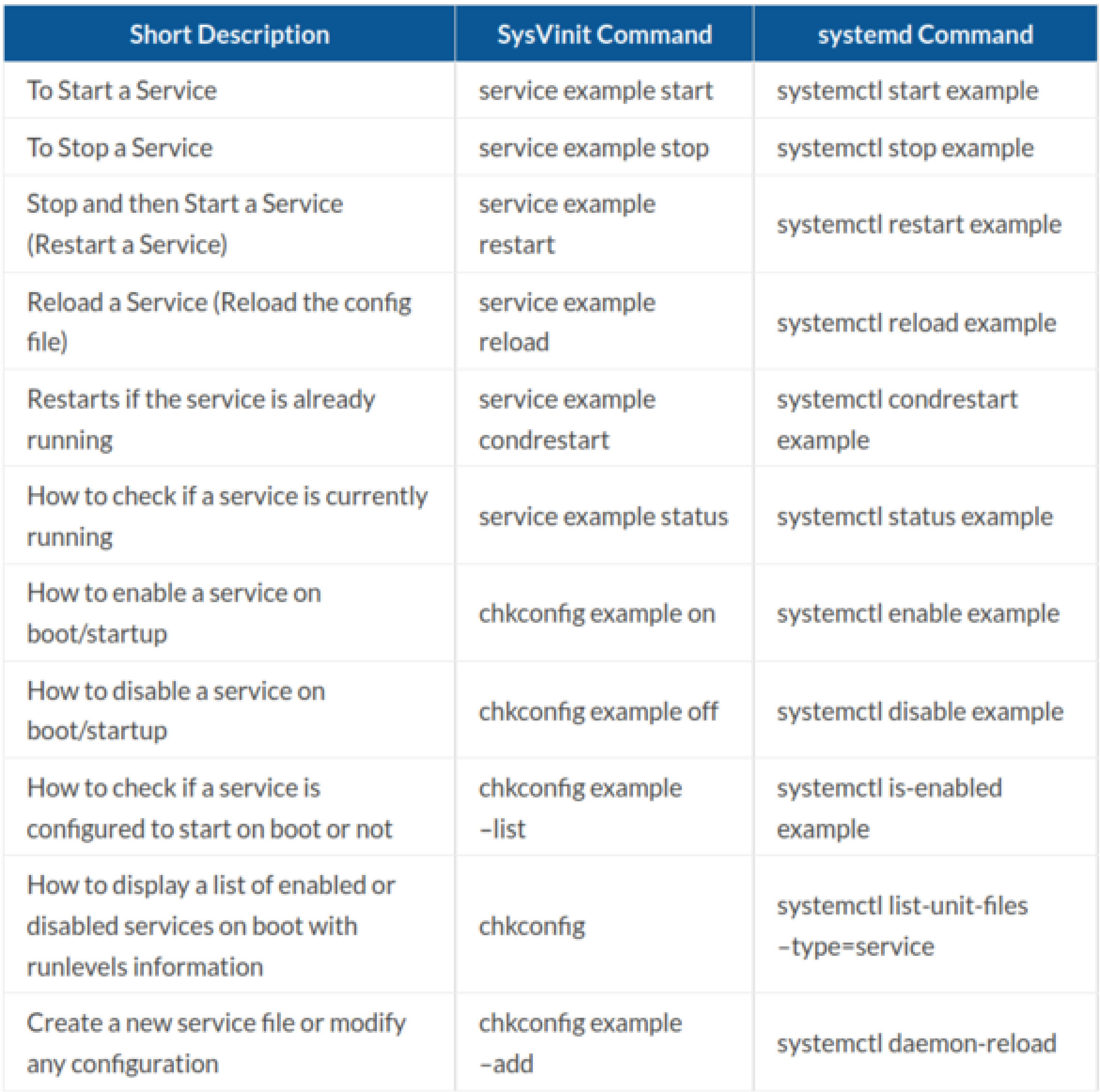
```
$systemctl is-active application.service
```

  - …

```
$systemctl is-enabled application.service
```

  - …

```
$systemctl is-failed application.service
```



Short Description	SysVinit Command	systemd Command
To Start a Service	service example start	systemctl start example
To Stop a Service	service example stop	systemctl stop example
Stop and then Start a Service (Restart a Service)	service example restart	systemctl restart example
Reload a Service (Reload the config file)	service example reload	systemctl reload example
Restarts if the service is already running	service example condrestart	systemctl condrestart example
How to check if a service is currently running	service example status	systemctl status example
How to enable a service on boot/startup	chkconfig example on	systemctl enable example
How to disable a service on boot/startup	chkconfig example off	systemctl disable example
How to check if a service is configured to start on boot or not	chkconfig example –list	systemctl is-enabled example
How to display a list of enabled or disabled services on boot with runlevels information	chkconfig	systemctl list-unit-files –type=service
Create a new service file or modify any configuration	chkconfig example –add	systemctl daemon-reload

- **Overview of the System State**
  - get all of the unit files that systemd has listed as “active”

    ```
    $ systemctl list-units
    ```

    - `list-units` can be even omitted

      ```
      $ systemctl
      ```

  - To list all of the units that systemd has loaded or attempted to load into memory, including those that are not currently active, add the –all switch

    ```
    $ systemctl list-units --all
    ```

  - To list all of the units installed on the system, including those that systemd has not tried to load into memory

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

- ## Viewing Log Information
  - By default, this will show you entries from the current and previous boots (if `journald` is configured to save previous boot records).
    - to enable this, either edit the `/etc/systemd/journald.conf` file and set the `Storage=` option to “`persistent`”, or create the persistent directory `sudo mkdir -p /var/log/journal`
  - journal entries from the current boot (add the -b flag)

```
$ journalctl -b
```

  - only kernel messages, such as those that are typically represented by `dmesg`, (use the -k flag)

```
$ journalctl -k
```

  - combine both of above

```
$ journalctl -k -b
```

- ## more logs
  - -b can be combined with boot offset

```
$ journalctl -b -1
```

  - you can also list available boot journals

```
$ journalctl --list-boots
```

  - Time ranges

```
$ journalctl --since "1 hour ago"
```

    - more precise

```
$ $ journalctl --since "2022-06-26 23:15:00" --until
"2022-06-26 23:20:00"
```

      - this is where having a proper sync on system time gets important
      - for even greater precisions see systemd.time specification .

- ### UTC timestamps

If you are about to combine/compare logs from multiple sources/timezones it is best to transform everything to UTC first to mitigate eventual confussions.

```
journalctl --utc
```

- ## Logs by systemd unit
  - messages logged by any systemd unit, use the `-u` switch

```
$ journalctl -u nginx.service
```

  - `-u` switch can be used multiple times to specify more than one unit source

```
$journalctl -u nginx.service -u mysql.service
```

  - Follow/Tail `-f`
  - specified number of most recent journal entries `-n`
  - journal entries in reverse chronological order, so the latest messages are printed first `-r`

- **Log output formats**
  - `-o` or `–output`

```
journalctl -u apache2.service -r -o json-pretty
```

    - `jsonwill` - show each journal entry in json format in one long line. This is useful when sending logs to a log centralization or analysis service, since it makes them easier to parse
    - `json-pretty`- will show each log entry in easy-to-read json format
    - `verbose` - will show very detailed information for each journal record with all fields listed
    - `cat` - shows messages in very short form, without any date/time or source server names
    - `short` - default output format, shows messages in syslog style
    - `short-monotonic` is similar to short, but the time stamp second value is shown with precision. This can be useful when you are looking at error messages generated from more than one source, which apparently are throwing error messages at the same time and you want to go to the granular level.

- **Logs by priority or by user**
  - priority from..to :

```
journalctl -b -1  -p "emerg".."crit"
```

```
    0: emerg
    1: alert
    2: crit
    3: err
    4: warning
    5: notice
    6: info
    7: debug
```

  - find all messages related to a particular user, use the UID

```
user@machine:~$ id postfix
uid=124(postfix) gid=129(postfix) skupiny=129(postfix)
user@machine:~$ journalctl _UID=124
-- Logs begin at Fri 2022-04-08 22:30:01 CEST, end at Sun
2022-10-30 20:32:29 CET. --
```

```
říj 19 18:42:51 machine postfix/pickup[34788]: CAA8048347D:
uid=1000 from=<user>
říj 19 18:42:51 machine postfix/cleanup[35617]: CAA8048347D:
message-id=<20221019164251.CAA8048347D@machine>
říj 19 18:42:51 machine postfix/qmgr[2151]: CAA8048347D:
from=<user@machine>, size=526, nrcpt=1 (queue acti>
říj 19 18:42:51 machine postfix/qmgr[2151]: CAA8048347D: removed
user@machine:~$
```

- ## Unit config files
  - Systemd is configured by text files - unit files.
    - `/lib/systemd/system/XXX.service`
  - full contents of a **unit file**

```
systemctl cat sshd.service
```

  - **dependency tree** of a unit (which units systemd will attempt to activate when starting the unit)

```
systemctl list-dependencies sshd.service
```

    - or add `-all` flag to expand all dependent units recursively
  - the **low-level details** of the unit’s settings on the system

```
systemctl show sshd.service
```

- ## Modifying Unit Files
  - To add an unit file snippet, which can be used to append or override settings in the default unit file, simply call the `edit` option

```
sudo systemctl edit sshd.service
```

  - To modify the entire content of the unit file instead of creating a snippet, pass the `–full` flag

```
sudo systemctl edit --full sshd.service
```

  - After modifying a unit file reload the systemd process to apply changes

```
sudo systemctl daemon-reload
```

- ## Masking & unmasking units
  - masking marks a unit as completely unstartable

```
sudo systemctl mask nginx.service
```

```
Output
```

```
. . .
kmod-static-nodes.service                    static
ldconfig.service                             static
mandb.service                                static
messagebus.service                           static
nginx.service                                masked
quotaon.service                              static
rc-local.service                             static
rdisc.service                                disabled
rescue.service                               static
. . .
```

  - …unmask

    ```
    sudo systemctl unmask nginx.service
    ```

- ## Using Targets

  In systemd “**targets**” is the closest equivalent of what was known as **runlevels** in the good old init system.

  - Targets are basically **synchronization points** that the server can use to get itself into a specific state.
  - Services and other unit files can be tied to a target and multiple targets can be active at the same time.
  - all the **available targets**

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

  - view the **default target** that systemd tries to reach at boot (which in turn starts all of the unit files that make up the dependency tree of that target)

    ```
    systemctl get-default
    ```

- ### targets continue

  - **change the default target**… **Warning**: this makes Ubuntu (or any other system) not boot to your beloved graphical.target.

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

  - see **what units are tied to a target**

    ```
    systemctl list-dependencies multi-user.target
    ```

  - **modify the system state** to transition between targets with the `isolate` option.

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

    This will stop any units that are not tied to the specified target. Be sure that the target you are isolating does not stop any essential services

Stopping or Rebooting

Shortcuts are available for some of the major states that a system can transition to.

- `sudo systemctl poweroff`
- `sudo systemctl reboot`
- `sudo systemctl rescue`

Most distributions include traditional aliases to these operations so that you can simply type `sudo poweroff` or `sudo reboot` without the systemctl.

Exercise - Creating own service

The program to run, our program will be simple http server which exposes its working(or given) directory content to the web. This is just a command to be run by the service

```
user@machine:~$ python3 -m http.server --help
usage: server.py [-h] [--cgi] [--bind ADDRESS] [--directory DIRECTORY]
[port]
positional arguments:
  port                  specify alternate port (default: 8000)
options:
  -h, --help            show this help message and exit
  --cgi                 run as CGI server
  --bind ADDRESS, -b ADDRESS
                        specify alternate bind address (default: all
                        interfaces)
  --directory DIRECTORY, -d DIRECTORY
                        specify alternate directory (default: current
                        directory)
user@machine:~$ python3 -m http.server --bind 192.168.21.130 8080
Serving HTTP on 192.168.21.130 port 8080 (http://192.168.21.130:8080/)
...
```

Turning a command into a service

Let's create a file called /etc/systemd/system/MyhttpServer.service

MyhttpServer.service

```
[Unit]
Description=Simple python based web server - demo service
After=network.target
StartLimitIntervalSec=0
```

```
[Service]
Type=simple
Restart=always
RestartSec=1
User=user
ExecStart=/usr/bin/python3 -m http.server --d /home/user/tmp --
bind 192.168.123.123 8080

[Install]
WantedBy=default.target
```

  - start the service

```
$ systemctl start MyhttpServer
```

  - enable the service

```
$ systemctl enable MyhttpServer
```

  - disable, mask, etc..
- Explanations:
  - `After=` - we want our service to be run only after the network is ready, you may want to add more like `After=mysqld.service` etc.
  - `Restart=` - By default, systemd does not restart the service if the program exits. We’re instructing it to always restart on exit. You could also use `on-failure` to only restart if the exit status is not `0`
  - `RestartSec=` - By default, systemd attempts a restart after 100ms. You can specify the number of seconds to wait before attempting a restart.
  - By default, when you configure `Restart=always`, systemd gives up restarting your service if it fails to start **more than 5 times within a 10 seconds** interval. This can be customized using

```
StartLimitBurst=5
StartLimitIntervalSec=10
```

  - Or by setting the`StartLimitIntervalSec=0` systemd will attempt to restart your service forever. Now it is good idea to set `RestartSec=` at least to 1 second no to put too much stress on the server if something goes wrong with the service.

- **Exercise continue**

  1. Go google how to make the previous MyhttpServer.service entirely controlled by a regular unprivileged user.
  2. transform your service to be triggered by `.socket` unit

From:
https://milixis.xyz/ - **Wiki on milixis.xyz**

Permanent link:
https://milixis.xyz/doku.php?id=nos:semminar6

Last update: **2024/11/12 08:50**