# « (/Advent-Of-Code-2016-Haskell/) Systemd For (Impatient) Sysadmins

*09 July 2017*

systemd (http://www.freedesktop.org/wiki/Software/systemd/): it's the init system that (some?) love to hate.

Full disclosure: I find systemd a little overbearing, although by no means would consider myself militantly anti-systemd. It has obvious advantages, and although I'm at philosophical odds with it at some levels, I see no reason why everybody shouldn't understand it a bit better - especially now that most people will need to deal with it on their favorite distros.

This post is a sort of formalized set of operational notes I've made frequent use of in my experience with systemd. I hope this post serves as another tool in your operations/sysadmin utility belt; when you need to get solutions quickly but are working with a system that relies on different tools than your normal sysv toolbox.

Note: The paths and examples here will be drawn from Arch Linux, as it's the distro I have easiest access to during the writing of this post. Most of it should carry over into other distributions.

## A Systemd Primer

If you're unfamiliar with The Beast:

- systemd replaces the bash scripts you're normally used to throwing in `/etc/init.d/*` (or simple upstart/etc. configs)
- in addition to system services, systemd manages the following as well:
  - `init` (PID 1)
  - runlevels (analagous to systemd "targets")
  - several types of common sockets
  - logging (through `journald` )

## Common Tasks

I have to do these things often with my normal init scripts, so how does one accomplish them in systemd?

### Service Parameter Overrides

Remember how well-behaved init scripts can be opened up to change their behavior? What if you'd like to override how a unit file operates?

For example, say we have the following systemd unit file (actual file pulled from the Arch Linux nginx package):

```
[Unit]
Description=A high performance web server and a reverse proxy server
After=network.target

[Service]
Type=forking
PIDFile=/run/nginx.pid
PrivateDevices=yes
SyslogLevel=err

ExecStart=/usr/bin/nginx -g 'pid /run/nginx.pid; error_log stderr;'
ExecReload=/usr/bin/kill -HUP $MAINPID
KillSignal=SIGQUIT
KillMode=mixed

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

What if you want to re-nice the daemon so that nginx gets priority under high load?

Typically this could be done under the classic init system by either munging with the init script or maybe using the auto nice daemon (http://and.sourceforge.net/). However, systemd exposes multiple settings (http://www.freedesktop.org/software/systemd/man/systemd.exec.html) to control these types of values. See "Nice=" on that page's documentation.

So, how does one add settings to an existing unit file? It'd be bad practice to edit the unit file shipped with the package in `/usr/lib/systemd/system` , we want to keep custom configuration where it belongs: `/etc` .

Add a file like this: `/etc/systemd/system/nginx.service.d/nice.conf`

```
[Service]
Nice=-10
```

This lets systemd know that you're overriding the unit file with a custom setting for the service - the rest of the options will be left alone, you're just appending the new value. This generally works for most settings in unit files, although overriding some options requires special treatment (for example, `ExecStart=` requires an empty option before defining a new one to avoid stringing together multiple stanzas).

As an added bonus, if you've already got sufficient privileges and your `EDITOR` environment variable setup, systemd can actually create the necessary directories and files for you - try using:

```
$ systemctl edit nginx.service
```

systemd will open your `EDITOR` and let you edit an override file, dropping into the appropriate path in `/etc` once you write out and close the buffer. Command-line sugar, but useful in some situations.

### Viewing Boot Services

One very valid point of comparison with traditional init scripts is the predictability of startup services. Whereas you may use a command like `chkconfig` to view which services will start at the default boot runlevel, there's a lot of different types of units systemd manages, and finding a command analagous to `chkconfig` isn't immediately evident.

The key here is to leverage targets. Target (https://www.freedesktop.org/software/systemd/man/systemd.target.html) units are (very roughly) comparable to runlevels; as systemd will usually attempt to reach one overarching target at boot-time (oftentimes `multi-user.target` ). Typically (as seen with the previous `nginx.service` example), services ask to be initialized by your distribution's main target entrypoint (in the `[Install]` section), and that gives us a window into what to expect at boot time.

The following command asks, "which units does `multi-user.target` ask to be started in order for itself to be considered up?"

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

The resulting output looks something like this:

```
multi-user.target
● ├─fstrim.timer
● ├─gmetad.service
● ├─gmond.service
● ├─basic.target
● │ ├─-.mount
● │ ├─snapper-cleanup.timer
● │ ├─snapper-timeline.timer
● │ ├─tmp.mount
● │ ├─paths.target
● │ ├─slices.target
● │ │ ├─-.slice
● │ │ └─system.slice
● │ ├─sockets.target
● │ │ ├─cockpit.socket
● │ │ └─uuidd.socket
● │ └─timers.target
● │   ├─logrotate.timer
● │   ├─shadow.timer
● │   └─systemd-tmpfiles-clean.timer
● └─zfs.target
●   ├─zfs-import-cache.service
●   ├─zfs-mount.service
●   ├─zfs-share.service
●   └─zfs-zed.service
```

(I've trimmed out a great deal of these for brevity). Note that in addition to visibility into plain old persistent daemons (like ganglia's `gmetad` and `gmond` ), the command also lists the dependency `sockets.target` and its associated dependent sockets, so we have a pretty good picture of not only the services that will come online at boot, but any sockets systemd will open that could potentially start other services via socket activation.

## Tricks With Timers

One systemd replacement that I wholeheartedly find more appealing than the traditional tooling is timers (https://www.freedesktop.org/software/systemd/man/systemd.timer.html) - output is captured more easily, environment variables are more predictable, and execution is more closely tracked. There are a few traits that timers expose that are particularly useful to be aware of.

### Last and Next Run

Because systemd tracks when timers execute, their last and next execution time can be easily discovered:

```
$ systemctl list-timers
```

You'll see something like this:

```
NEXT                          LEFT                LAST                          PASSED     UNIT
Fri 2017-07-07 21:00:00 EDT  34min left          Fri 2017-07-07 20:00:00 EDT  25min ago  snapper-timeline.timer
Fri 2017-07-07 22:56:43 EDT  2h 30min left       Thu 2017-07-06 22:56:43 EDT  21h ago    snapper-cleanup.timer
Fri 2017-07-07 23:01:44 EDT  2h 36min left       Thu 2017-07-06 23:01:44 EDT  21h ago    systemd-tmpfiles-clean.timer
Sat 2017-07-08 00:00:00 EDT  3h 34min left       Fri 2017-07-07 00:00:00 EDT  20h ago    shadow.timer
Sat 2017-07-08 03:00:00 EDT  6h left             Fri 2017-07-07 03:00:00 EDT  17h ago    logrotate.timer
Sat 2017-07-08 03:00:00 EDT  6h left             Fri 2017-07-07 03:00:00 EDT  17h ago    snazzer.timer
Sat 2017-07-08 03:30:00 EDT  7h left             Fri 2017-07-07 03:30:00 EDT  16h ago    man-db.timer
Mon 2017-07-10 03:00:00 EDT  2 days left         Mon 2017-07-03 03:00:00 EDT  4 days ago fstrim.timer
Tue 2017-08-01 05:00:00 EDT  3 weeks 3 days left Sat 2017-07-01 05:00:00 EDT  6 days ago reflector.timer
```

You can pretty easily see, for example, that logrotate will run in about six hours, and that my filesystem got trimmed four days ago on July third.

### Persistence

If you've ever written a script to run periodically, you may have questioned what happens if the machine is off when the execution time passes. Timers have a convenient option called `Persistent=` (https://www.freedesktop.org/software/systemd/man/systemd.timer.html#Persistent=) that can record when the timer was fired, and if the interval lapsed when the timer wasn't able to fire (if, for example, the host was offline), systemd will trigger the timer once it's able to.

One use case that I have for this is a job to renew Let's Encrypt certificates. I occasionally bring down my servers for kernel updates, and with a persistent timer, I never worry about an expiring certificate as those timers will trigger once they're able to if the host happened to be down during that time.

## User Instances

This is a profoundly userful feature that I've been using more and more often. If you've ever poked around your environment in an ssh login on a systemd-based system, you'll notice that you're in a session systemd has spawned for your user. This is a little slice of the system that gives your user some dedicated tools, including your own session to run user units in.

Note that, by default, most systems only spawn user sessions on login and kill them on logout. You can ensure users get more permanent sessions at boot that persist after logout with a command:

```
$ loginctl enable-linger tylerjl
```

At this point you can do most of the things you can do with normal systemd units, but entirely within userspace and without the need to prepend `sudo` to most `systemctl` commands.

As an example, I keep Kodi (https://kodi.tv/) backups on a ZFS dataset that I prune occasionally. My normal, non-root user has read/write permission on these backups, so why not do this as an unprivileged user (and dodge a possibly destructive `find` invocation)?

I have the following file at `$HOME/.config/systemd/user/kodi-backup-cleanup.timer` :

```
[Unit]
Description=regularly clean kodi backups

[Timer]
OnCalendar=Thursday *-*-* 02:00:00

[Install]
WantedBy=default.target
```

And an associated service at `$HOME/.config/systemd/user/kodi-backup-cleanup.service`

```
[Unit]
Description=clean up old kodi backups

[Service]
ExecStart=/usr/bin/find /srv/storage/backups/kodi -type f -name '*.zip' -mtime +40 -delete
```

Two important items of note:

- The units live at a specific path that the user session will recognize.
- I've found that the `default` target is brought up with the user session, providing the necessary hook to get units started at boot for users. Your distribution may vary.

All that's left is to enable and start the timer, with the `--user` flag to connect to the user's instance, not the system one:

```
$ systemctl --user enable --now kodi-backup-cleanup.timer
```

And my timer will fire the `find` command as my user on the specified schedule. What's more, you get all the associated timer goodness in the forementioned sections (like `list-timer` , but with `--user` ).

## Path Units

Path (https://www.freedesktop.org/software/systemd/man/systemd.path.html) unit are a somewhat lesser used type of unit that can make a system very event-driven and add dependencies in a more natural way. Simply, they provide a mechanism to start other services if files or directories change in some way. A good illustration here is some of the automation I have around my Let's Encrypt certificate renewals.

As mentioned previously, I have a timer and service that regularly renews my certificates. Suppose my certificate bundle ends up at `/etc/pki/certs/blog_tjll_net-bundle.pem` . I have a couple of services that consume this certificate; for example, `nginx` reads the cert to serve up HTTPS. My certificate renewal script is a nice, standalone command that just does one thing, and it would be nice to provide a decoupled way to trigger a `reload` for nginx if the bundle gets updated. In addition, it would be convenient for any other applications that consume the certificate (and I have a few) to watch for updates without needing to update my script all the time.

With that in mind, consider the following path unit, `cert-bundle.path` :

```
[Unit]
Description=watches the cert bundle

[Path]
PathModified=/etc/pki/certs/blog_tjll_net-bundle.pem
Unit=reload-nginx.service

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

When `enable` d and `start` ed, this unit expresses "when the cert file gets updated, start `reload-nginx.service` ". The associated `service` file looks like this:

```
[Unit]
Description=reload nginx

[Service]
Type=oneshot
ExecStart=/usr/bin/systemctl reload nginx
```

Pretty straightforward. The entire flow seems overcomplicated at first, but it's actually very convenient as any updates to the certificate file outside of my script (which, because I can write buggy scripts, happens sometimes) will always trigger a reload, making the entire certificate update process agnostic to the method that actually drops the certificate there.

Path units can activate any unit that isn't another path and can monitor most inotify-related events, so there's a great deal of flexibility in how they can be used.

## Grab Bag

Aside from those broader topics, there's a few little niceties that I've been using that may be useful in day-to-day operations with systemd:

- Start and enable a unit with one command with the `--now` flag: `systemctl enable --now foo.service`
- Want to quickly find problems? List failed units that need attention with `systemctl --state=failed` (I use this one a lot)
- The `cat` operation got some shade thrown at it for re-inventing unix commands, but it actually shows a unit file's contents in addition to any override files that may be in effect for it (for example `systemctl cat nginx.service` )
- I haven't dove into `journalctl` in this post, but I use filtering fairly regularly. For example, to live-tail logs for both nginx and ssh, try `journalctl -f _SYSTEMD_UNIT=nginx.service + _SYSTEMD_UNIT=sshd.service` . The plus is a logical or.

If you want more tips like this, I'd refer you to my post over at SysAdvent (http://sysadvent.blogspot.com/2015/12/day-17-grokking-systemd-for-fun-and.html) that dives more deeply into topics like journald and dbus. Happying `systemctl` ing!

Start Discussion 0 replies

« Advent Of Code 2016 in Haskell (/advent-of-code-2016-haskell/)

 