Pid Eins

レナート
لينارت

Google+ systemd PulseAudio Avahi Repositories Imprint

POSTED ON MO 26 SEPTEMBER 2011

systemd for Administrators, Part X

Here's the tenth installment of my ongoing series on systemd for Administrators:

Instantiated Services

Most services on Linux/Unix are *singleton* services: there's usually only one instance of Syslog, Postfix, or Apache running on a specific system at the same time. On the other hand some select services may run in multiple instances on the same host. For example, an Internet service like the Dovecot IMAP service could run in multiple instances on different IP ports or different local IP addresses. A more common example that exists on all installations is *getty*, the mini service that runs once for each TTY and presents a login prompt on it. On most systems this service is instantiated once for each of the first six virtual consoles `tty1` to `tty6`. On some servers depending on administrator configuration or boot-time parameters an additional getty is instantiated for a serial or virtualizer console. Another common instantiated service in the systemd world is *fsck*, the file system checker that is instantiated once for each block device that needs to be checked. Finally, in systemd socket activated per-connection services (think classic inetd!) are also implemented via instantiated services: a new instance is created for each incoming connection. In this installment I hope to explain a bit how systemd implements instantiated services and how to take advantage of them as an administrator.

If you followed the previous episodes of this series you are probably aware that services in systemd are named according to the pattern *`foobar`*`.service`, where *foobar* is an identification string for the service, and `.service` simply a fixed suffix that is identical for all service units. The definition files for these services are searched for in `/etc/systemd/system` and `/lib/systemd/system` (and possibly other directories) under this name. For instantiated services this pattern is extended a bit: the service name becomes *`foobar@quux`*`.service` where *foobar* is the common service identifier, and *quux* the instance identifier. Example:

`serial-getty@ttyS2.service` is the serial getty service instantiated for `ttyS2`.

Service instances can be created dynamically as needed. Without further configuration you may easily start a new getty on a serial port simply by invoking a `systemctl start` command for the new instance:

```
# systemctl start serial-getty@ttyUSB0.service
```

If a command like the above is run systemd will first look for a unit configuration file by the exact name you requested. If this service file is not found (and usually it isn't if you use instantiated services like this) then the instance id is removed from the name and a unit configuration file by the resulting *template* name searched. In other words, in the above example, if the precise `serial-getty@ttyUSB0.service` unit file cannot be found, `serial-getty@.service` is loaded instead. This unit template file will hence be common for all instances of this service. For the serial getty we ship a template unit file in systemd (`/lib/systemd/system/serial-getty@.service`) that looks something like this:

```
[Unit]
Description=Serial Getty on %I
BindTo=dev-%i.device
After=dev-%i.device systemd-user-sessions.service

[Service]
ExecStart=-/sbin/agetty -s %I 115200,38400,9600
Restart=always
RestartSec=0
```

(Note that the unit template file we actually ship along with systemd for the serial gettys is a bit longer. If you are interested, have a look at the actual file which includes additional directives for compatibility with SysV, to clear the screen and remove previous users from the TTY device. To keep things simple I have shortened the unit file to the relevant lines here.)

This file looks mostly like any other unit file, with one distinction: the specifiers `%I` and `%i` are used at multiple locations. At unit load time `%I` and `%i` are replaced by systemd with the instance identifier of the service. In our example above, if a service is instantiated as `serial-getty@ttyUSB0.service` the specifiers `%I` and `%i` will be replaced by `ttyUSB0`. If you introspect the instanciated unit with `systemctl status serial-getty@ttyUSB0.service` you will see these replacements having taken place:

```
$ systemctl status serial-getty@ttyUSB0.service
serial-getty@ttyUSB0.service - Getty on ttyUSB0
          Loaded: loaded (/lib/systemd/system/serial-getty@.service; static
          Active: active (running) since Mon, 26 Sep 2011 04:20:44 +0200; 2
        Main PID: 5443 (agetty)
          CGroup: name=systemd:/system/getty@.service/ttyUSB0
                  └ 5443 /sbin/agetty -s ttyUSB0 115200,38400,9600
```

And that is already the core idea of instantiated services in systemd. As you can see systemd provides a very simple templating system, which can be used to dynamically instantiate services as needed. To make effective use of this, a few more notes:

You may instantiate these services *on-the-fly* in `.wants/` symbolic links in the file system. For example, to make sure the serial getty on `ttyUSB0` is started automatically at every boot, create a symlink like this:

```
# ln -s /lib/systemd/system/serial-getty@.service /etc/systemd/system/getty
```

systemd will instantiate the symlinked unit file with the instance name specified in the symlink name.

You cannot instantiate a unit template without specifying an instance identifier. In other words `systemctl start serial-getty@.service` will necessarily fail since the instance name was left unspecified.

Sometimes it is useful to *opt-out* of the generic template for one specific instance. For these cases make use of the fact that systemd always searches first for the full instance file name before falling back to the template file name: make sure to place a unit file under the fully instantiated name in `/etc/systemd/system` and it will override the generic templated version for this specific instance.

The unit file shown above uses `%i` at some places and `%I` at others. You may wonder what the difference between these specifiers are. `%i` is replaced by the exact characters of the instance identifier. For `%I` on the other hand the instance identifier is first passed through a simple unescaping algorithm. In the case of a simple instance identifier like `ttyUSB0` there is no effective difference. However, if the device name includes one or more slashes ("/") this cannot be part of a unit name (or Unix file name). Before such a device name can be used as instance identifier it needs to be escaped so that "/" becomes "-" and most other special characters (including "-") are replaced by "\xAB" where AB is the ASCII code of the character in hexadecimal notation[1]. Example: to refer to a USB serial port by its bus path we want to use a port name like `serial/by-path/pci-0000:00:1d.0-usb-0:1.4:1.1-port0`. The escaped version of this name is `serial-by\x2dpath-pci\x2d0000:00:1d.0\x2dusb\x2d0:1.4:1.1\x2dport0`. `%I` will then refer to former, `%i` to the latter. Effectively this means `%i` is useful wherever it is necessary to refer to other units, for example to express additional dependencies. On the other hand %I is useful for usage in command lines, or inclusion in pretty description strings. Let's check how this looks with the above unit file:

```
# systemctl start 'serial-getty@serial-by\x2dpath-pci\x2d0000:00:1d.0\x2dus
# systemctl status 'serial-getty@serial-by\x2dpath-pci\x2d0000:00:1d.0\x2du
serial-getty@serial-by\x2dpath-pci\x2d0000:00:1d.0\x2dusb\x2d0:1.4:1.1\x2dp
          Loaded: loaded (/lib/systemd/system/serial-getty@.service; static
          Active: active (running) since Mon, 26 Sep 2011 05:08:52 +0200; 1
        Main PID: 5788 (agetty)
          CGroup: name=systemd:/system/serial-getty@.service/serial-by\x2dp
                  └ 5788 /sbin/agetty -s serial/by-path/pci-0000:00:1d.0-us
```

As we can see the while the instance identifier is the escaped string the command line and the description string actually use the unescaped version, as expected.

(Side note: there are more specifiers available than just %i and %I, and many of them are actually available in all unit files, not just templates for service instances. For more details see the man page which includes a full list and terse explanations.)

And at this point this shall be all for now. Stay tuned for a follow-up article on how instantiated services are used for `inetd`-style socket activation.

Footnotes

[1] Yupp, this escaping algorithm doesn't really result in particularly pretty escaped strings, but then again, most escaping algorithms don't help readability. The algorithm we used here is inspired by what udev does in a similar case, with one change. In the end, we had to pick something. If you'll plan to comment on the escaping algorithm please also mention where you live so that I can come around and paint your bike shed yellow with blue stripes. Thanks!

Category: projects

← BACK TO INDEX