Service management with systemd
Lab manual

Michal Sekletár
msekleta@redhat.com

29 October 2017

Introduction

Welcome! This manual provides examples and guidance and it should help you solve lab exercises and following questions. The aim of the labs is to introduce systemd to novice users, but on numerous places we present material that should simplify day-to-day tasks performed by users at all levels of expertise.

Outline:

- Lab setup
- Components of systemd
- Units and unit files
- Dependencies
- Service management
- Using systemd-journald

1 Lab setup

There are several options how to get your hands on the system that is running systemd.

1. Your own laptop
   If you are running recent version of any popular Linux distribution then you most likely already have systemd on your system. Run following command to verify that `PID 1` is actually systemd[1],

   ```
   systemctl
   ```

   If you get some output back then you have systemd running on your system.

2. Virtual machine
   For Windows and MacOS users this is probably the easiest option. I'd advise to use pre-provisioned image for your hyper-visor (HyperV, VirtualBox) or use vagrant.

   ```
   vagrant init fedora/26-cloud-base
   vagrant up
   ```

3. systemd-nspawn OS container
   We can install entire base of the Linux distribution to some alternate root path and then boot this minimal system using systemd-nspawn. Instructions on how to install minimal package set are distribution specific. However, manual page for systemd-nspawn lists appropriate commands for most popular distributions in EXAMPLES section.

   ```
   systemd-nspawn -D /srv/f26
   echo lisa17 | passwd root --stdin
   systemd-nspawn -bD /srv/f26
   ```

1.1 Useful packages

systemd has superb shell completion support. Thus, if you do not have bash/zsh completion installed then you are missing out. bash and zsh completion is shipped in systemd package[2], but in order to make use of it you have to install generic bash completion package.

```
dnf install -y bash-completion
```

[1]Unless stated otherwise, all commands should be run as `root` user
[2]This is true on Fedora but maybe different on other Linux distributions

2 systemd in my distro

Aim of this lab is to get familiar with the way how systemd is packaged in your distribution of choice and to figure out what version of systemd you are running.

1. systemd is very actively developed, so more often that not, it is useful to know what package version[3] you are running,

   ```
   systemctl --version
   ```

2. Previous command showed us systemd's major version. However, distros tend to patch software they ship, thus we also need to figure out specific package version,

   ```
   rpm -q systemd
   ```

3. Go ahead and examine what is included in your main systemd package,

   ```
   rpm -ql systemd
   ```

   As you can see there is a lot. We won't be covering everything in detail, but we will learn about all the important components that comprise typical systemd installation.

4. Now we know package version and we know what is in it. As mentioned in the slides systemd is not monolithic, and the entire framework contains lot more components than `init` itself. Let's look at what other sub-packages are built from the same source package,

   ```
   srpm=$(rpm -qi systemd | grep Source)
   srpm=$(echo $srpm | awk '{ print $4 }')
   for i in $(rpm -qa); do
       rpm -qi $i | grep -q $srpm && echo $i;
   done
   ```

2.1 Components of systemd

In this part of the lab exercise we will discover main components of systemd framework. We will examine current system state, on disk configuration and we explain hierarchical nature of systemd's configuration.

[3]Command output also displays compile time configuration used in your build

2.1.1 Command line tools

systemd provides many command line client tools. Among them, the main tool that is used to interact with systemd is `systemctl`. For communication with systemd it uses DBus IPC mechanism. Note that `systemctl` can be also used in environments where DBus server isn't running. Fact that it uses DBus IPC doesn't imply running `dbus-daemon`.

systemd is very tightly integrated with logging service – `journald`. We believe that proper logging support is inherently necessary to provide useful service management capabilities to system administrators. Unlike PID 1, `journald` doesn't offer DBus API, but it offers command line client. `journalctl` is used to display system logs and it allows for very powerful filtering to quickly display desired parts of the system log.

Next very important component is `logind`. It is a system service that tracks sessions, users and implements multi-seat support. Associated command line tool is `loginctl`

systemd provides many more daemons (as you could see in the slides) and accompanying client utilities. However, we can't cover all of them here. Above three are usually enough to get an idea what state a system is in.

2.2 Introspecting system state

To get the very basic idea of current system state (as far as service manager is concerned) you can run,

```
systemctl status
```

Output is typically quite long because we also get detailed view of the process tree, but one of the most important pieces of information in the output is overall system state. In pretty much all cases you want to see, `State:  running`. Other, unfortunately very frequent state is `degraded`. It means that system is operational but some systemd unit is in the failed state. Listing of all services and their states can be obtained by running, `systemctl -type=service list-units` command.

Next information that we are maybe interested in is status of specific service. For example we can query systemd for status of `dbus.service`.

```
systemctl status dbus.service
```

Similarly to overall system status, service status also provides us with useful information about the service. Displayed information is much richer and useful than typical initscript output

`Running as PID 100`. Because systemd is tightly integrated with journald we also get up to ten log lines that were recently logged by the service.

A lot of useful information about systemd boot can be learned from output of the `systemd-analyze` tool.

```
systemd-analyze
systemd-analyze blame
```

But what runlevel (target) we actually booted into? Let's identify the default target,

```
systemctl get-default
```

Note that targets can be changed during runtime using,

```
systemctl isolate <TARGET>
```

Before we dive into more details on systemd, we will briefly look on other tools from the toolbox.

2.2.1 Quick look at systemd-journald

`systemd-journald` is an integral part of systemd framework. It is a central log aggregator, that stores lot of meta-data alongside log messages. Hence it allows for very powerful filtering. We will examine journald in-depth later in this tutorial. Now we just look at some often used capabilities provided by journald.

1. We can display log messages for any system service since the machine was booted[4]

   ```
   journalctl -b -u dbus.service
   ```

2. Second very useful feature is displaying log messages that are related to certain device (e.g. failing disk)

   ```
   journalctl -b /dev/sda
   ```

3. Because journald stores rich meta-data it allows you to write advanced filters. For example, I can display all error messages, logged by kernel since yesterday,

   ```
   journalctl --since yesterday --priority=err -k
   ```

Of course you could say that these capabilities were available before in some form and you'd be right. However, powerful filtering was previously available on log aggregation server and local users where usually left with grepping `/var/log/messages`.

[4]If you don't have persistent journal then some of the log messages are maybe not available any more

2.2.2 User session tracking

Of course we usually don't run servers just for the sake of running them. They either host services or serve users. Since user session tracking is again very important aspect of the system administration[5], systemd project introduced new system service `systemd-logind` designed for this purpose.

There is couple of reasons why logind exists, but main ones are,

- user tracking
- user session tracking
- session device management
- suspend and hibernation handling
- multi-seat support

1. In order to quickly see what users are currently logged on the system you can run,

   ```
   loginctl list-users
   ```

2. We can also list all sessions open by logged on users,

   ```
   loginctl list-sessions
   ```

3. Next we pick one session and we obtain more detailed status of that session,

   ```
   loginctl session-status <SESSION_ID>
   ```

   Above command displays a lot of detailed information about the session. We can see type of the session (X, Wayland or text), process tree associated with the session and more.

2.2.3 Questions

This concludes the lab exercise, and you should be able to answer following questions,

1. What version of systemd are you running?
2. Is SELinux support enabled in your build?
3. What exact version of systemd package is in your distribution, when it was built, by whom?
4. Are systemd components split to sub-packages? How many there are?

[5]We dare to say that it was previously very neglected. Anyone running ConsoleKit on RHEL/CentOS 6?

5. What is your system state?
6. Are there any processes on your system which are not tracked by systemd unit?
7. Do you have any failed services? If yes, can you tell from service log messages what is wrong?

3 Units and unit files

In this lab we will deepen our understanding of basic systemd concepts, units and unit files. After completing this lab you should be able to precisely articulate difference between them and know what types of units you can encounter while working with systemd and how to configure them via unit files.

3.1 Units

First lets examine what are *all* units that systemd is currently aware of.

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

Notice their load state, active state and sub-state respectively. Usually we are interested only is a subset of all units, for example, in all running services on the system.

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

Even more interesting is to only list units that are in the failed state.

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

If your system is in running state then you should see no failed units, however if system is running in degraded state then you will have some failed units.

In unit listing we can see there is a lot more unit types than just service. However, the workflow to obtain unit details is always the same. We `list-units` with appropriate `--type` argument and then we get unit `status`.

For each unit, systemd tracks a lot more information than is included in output of `systemctl status`. To get really detailed information you can call,

```
systemctl show dbus.service
```

To easily access unit file for given unit you can type,

```
systemctl cat dbus.service
```

Next we examine unit types that exist due an external event and do not have respective on-disk configuration. In most cases we encounter following unit types w/o unit files,

- device
- scope
- mount

Device units never have any on disk configuration. Scope units are externally created groups of processes that systemd treats as a unit. systemd also monitors mounts and filesystems that are mounted manually from command line and they will have a unit representation, but such units don't have any unit file.

3.2 Extending unit files

Now we know how to list-units and observe their state. From presentation we know that unit is an abstraction of a system entity and some units exists because systemd loaded their on disk configuration (e.g. services, sockets). Go ahead and create following test unit file, using your favorite text editor,

```
# /usr/lib/systemd/system/lisa17.service
[Service]
Type=simple
ExecStart=/bin/bash -c "echo hello LISA attendees; sleep 3600;"

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

By saving your editor buffer to disk you create unit file. However, systemd doesn't immediately create corresponding unit. We can convince ourselves about this,

```
  systemctl --all list-units  | grep lisa17
```

As you can see systemd really didn't create unit just yet. Any time we changed on disk configuration, either because we have added new unit files or edited old ones we need to explicitly tell systemd to load the new config,

```
  systemctl daemon-reload
```

After you have created unit file and reloaded systemd we can start the unit and examine its status.

```
  systemctl start lisa17.service
  systemctl status lisa17.service
```

Notice that status of the service contains path to corresponding unit file and state of the unit file. Since only thing we did was start of the service we still see that unit file state is `disabled`. Let's make sure our test service is enabled on next boot and run status once again,

```
systemctl enable lisa17.service
systemctl status lisa17.service
```

Before we proceed, try to answer for your self following questions,

- What information was displayed by `enable` command?
- Can you interpret meaning of the message?
- What has changed in status output?

In next part of the lab we will experiment with extending the configuration for our test unit with configuration drop-ins. First, we need to create directory where we place our drop-in config files,

```
mkdir -p /etc/systemd/system/lisa17.service.d
```

Now we can put our drop-ins in place and reload systemd. Note that some configuration will have effect immediately after systemd reloads while other requires you to restart the service.

```
cat > /etc/systemd/system/lisa17.service.d/dependency.conf << EOF
[Unit]
After=network-online.target
EOF

cat > /etc/systemd/system/lisa17.service.d/tmp.conf << EOF
[Service]
PrivateTmp=yes
EOF
```

To make sure our configuration takes effect we issue daemon reload[6]. Furthermore we query service status. As you will see, status also show drop-in configuration that we've just applied.

```
systemctl daemon-reload
systemctl restart lisa17.service
systemctl status lisa17.service
```

Let's look at unit details and verify that drop-in configuration is applied,

```
systemctl show lisa17.service
```

[6]Note that daemon reload is an expensive operation, hence we do it only once and not three times

Ouput of `systemctl show` is very useful while writing scripts. Never parse output of `systemctl status`! Also note that output of `systemctl show` is covered by Interface stability promise.

3.3 Dependencies

We have solid understanding of objects that systemd manages, how they are configured and we have some intuition about their lifetime. We continue our exploration of basic systemd concepts. Moving on to dependencies.

We will use following simple units to demonstrate how dependencies work in systemd (don't forget to reload systemd),

```
# /etc/systemd/system/foo.service
[Service]
ExecStart=/bin/bash -c "echo I am foo; sleep 2;"
```

```
# /etc/systemd/system/bar.service
[Service]
ExecStart=/bin/bash -c "echo I am bar; sleep 2;"
```

Right now we have two stand-alone units that only have default dependencies in their sets of dependencies. Very often, we need to start two related services together as a single unit. `Wants` dependency is ideal for this use-case. We now add `Wants=bar.service` to `foo.service`

```
  cat >> /etc/systemd/system/foo.service <<EOF
  [Unit]
  Wants=bar.service
  EOF

  systemctl daemon-reload
  systemctl start foo.service
```

Open up journal and observe what happened. As you can see start of `foo.service` caused also start of the other unit. Note that `Wants` dependency has no effect when we try to stop one of the units. We can stop either one and only that unit gets stopped. Also note that our current setup says nothing about ordering in which units are started. Now stop both services and proceed to next part of the manual.

We now introduce ordering between services. Previously configured `Wants` dependency stays in place.

```
cat >> /etc/systemd/system/foo.service <<EOF
[Unit]
After=bar.service
EOF

systemctl daemon-reload
systemctl start foo.service
```

Again, open up journal and try to verify that systemd applied your configuration and started services in the defined order.

Let's convince ourselves that systemd is really honoring dependencies as we defined them. To do that we introduce delay to start-up of `bar.service`. Please stop both services before proceeding further.

```
cat >> /etc/systemd/system/bar.service <<EOF
[Service]
ExecStartPre=/bin/sleep 10
EOF

systemctl daemon-reload
systemctl --no-block start foo.service
systemctl list-jobs
```

We started `foo.service` in a way that `systemctl` doesn't block and wait for completion of the start job because we want to have a look at systemd's job queue. As you can see systemd really waits 10 seconds until `bar.service` starts and only after that it starts `foo.service`

As last exercise we will experiment with other dependency types. First we replace our `Wants` dependency with `Requires` and then with `BindsTo`. As usual, stop both units before proceeding.

```
sed -i s/Wants/Requires/ /etc/systemd/system/foo.service

systemctl daemon-reload
systemctl start foo.service &
systemctl --signal=TERM kill bar.service
```

What happens when you kill `bar.service` when `foo.service` is started? What happens when you stop `bar.service`? In what respect does BindsTo behave differently?

3.4 Questions

Before you continue try to give answers to following questions,

1. What is a systemd unit?
2. What is a systemd unit file?
3. Which property of a unit contains path to corresponding unit file?
4. How you can list only timer units?
5. Is it possible to unmount filesystem using systemd?
6. Which directory systemd configuration directory should be edited by system administrators?
7. How is it possible that unit lisa17.service has much more dependencies that configured in the unit file?
8. Can you explain differences between Wants, Requires and BindsTo dependencies?
9. What happens when you don't specify any ordering dependency between related services?
10. Which systemctl sub-command you can use for listing dependencies?
11. How would you print all units ordered Before dbus.service?

4 Service management

In previous segments we worked with "synthetic" examples in order to don't be bothered by a lot of service details. Now we proceed to more concrete examples. Here we will be dealing with very classic situation, web server and database backend. We will be operating services, defining new ones and managing resources using cgroups. This example was kindly provided by Ben Breard.

4.1 Setup

First we need to install packages that we will be working with,

```
dnf -y install httpd mariadb-server mariadb php php-mysqlnd
   mariadb-bench
```

Let's make sure both services are enabled on boot and started,

```
systemctl --now enable httpd.service
systemctl --now enable mariadb.service
```

We also create two more services that will cause load to be applied to both apache and mariadb.

```
# /etc/systemd/system/httpd-bench.service
[Unit]
Wants=httpd.service
After=httpd.service

[Service]
ExecStart=/usr/bin/ab -c 100 -n 999999 http://localhost/index.html
```

```
# /etc/systemd/system/mariadb-bench.service
[Unit]
Wants=mariadb.service
After=mariadb.service

[Service]
ExecStart=/usr/share/sql-bench/run-all-tests --server=mysql
WorkingDirectory=/usr/share/sql-bench
```

We will further configure our web server so that it has lower nice value and also it is unlikely to get killed by OOM killer.

```
systemctl edit httpd.service
# Additional configuration
[Service]
Nice=-10
OOMScoreAdjust=-1000
```

4.2 Resource management

Let's put some load on apache, and monitor it. From the original terminal run,

```
systemctl start httpd-bench.service
```

Open up `systemd-cgtop` on other terminal and look at the output. Does it look as expected? Now stop httpd-bench.

```
systemctl stop httpd-bench.service
```

Output of `systemd-cgtop` didn't look as expected because we didn't enable any accounting for our services. Enable cgroup accounting for both httpd and mariadb by introducing new drop-in, e.g. `acct.conf`.

```
[Service]
CPUAccounting=1
MemoryAccounting=1
BlockIOAccounting=1
```

We need to enable accounting for the cgroup controllers that we want to tune or monitor. The available high-level controllers, as of recent versions of systemd, are CPUAccounting, Memory-Accounting, TaskAccounting and BlockIOAccounting.

Let's also generate some load in our user session,

```
dd if=/dev/zero of=/dev/null &
```

Now let's run our benchmark again on the system and monitor the output systemd-cgtop.

```
systemctl start httpd-bench.service
```

Notice that system.slice and user.slice are both using roughly the same amount of CPU. Why is that? Hint: look at the output of `systemd-cgls` and `top`. Note that the number of CPUs on the system will vary the percentage CPU split between the user and system slice. Why? Play around with enabling/disabling CPUs. You can disable CPU by running,

```
echo 0 > /sys/devices/system/cpu/cpu1/online
```

Now we're going to put the whole system under load. Run,

```
systemctl start httpd-bench.service mariadb-bench.service
dd if=/dev/zero of=/dev/null &
```

Monitor output of the `systemd-cgtop` terminal. You might want to also open a third terminal and leave `top` running.

With the system under load and cgroup accounting enabled, we can now start tweaking the cgroups. Let's lower the CPU shares for user.slice to allow the system slice, and hence mariadb and httpd, more CPU time. Then do the opposite,

```
systemctl set-property --runtime user.slice CPUShares=30
systemctl set-property --runtime user.slice CPUShares=3000
```

After watching the difference reset the CPUShares to 1024. Right now httpd and mariadb have equal CPU time under the system slice. Let's give httpd more.

```
systemctl set-property --runtime httpd.service CPUShares=3000
```

Notice that the user and system slices aren't altered by this, but httpd and mariadb are. When you're ready to move to the next step run,

```
systemctl stop httpd-bench.service mariadb-bench.service
kill %1
```

Note that we're making drastic changes to show how this works. It's recommended to make smaller, incremental changes when doing this in real life. Instead of jumping all the way to 3000, start with something closer to 1200 and monitor the environment. If you need more CPU time on the scheduler, move to 1400, etc.

Let's promote httpd to it's own slice. This is useful if you have an application or service that you want to guarantee gets significantly more scheduler time than other user or system processes/services. Modify httpd unit, add slice configuration and restart it.

```
# Move httpd to its own slice
[Service]
Slice=apache.slice
```

Next, run following commands,

```
systemctl start mariadb-bench.service httpd-bench.service
dd if=/dev/zero of=/dev/null &
systemctl set-property --runtime httpd.service CPUShares=3000
```

Did that do what you expected it would? If not, what is wrong, how to fix it? Once the settings are configured appropriately for your use case and workload, drop the –runtime option to persist the settings.

Note that all resource management options are described in detail in
`man 5 systemd.resource-control`.

4.3 Questions and exercises

- Explain concept of CPUShares.
- Configure memory limit of 2GB for Maria DB.
- Make sure that httpd can't read more than 50MB/s of data from `/var/www`
- Where does systemd store configuration applied by `set-property` sub-command?
- Is it necessary to restart service after dynamically adjusting cgroup configuration?
- Which cgroup controllers are currently not supported by systemd?

5 Using systemd-journald

`journalctl` command line tool is used most of the time when interacting with journald. I say we are interacting with journald, but this is a bit of a lie. Actually it is just reading journals (log files of the disk), i.e. it works even when journald isn't running. Let's look at some filtering capabilities.

Tail the journal (like `tail -f /var/log/messages`),

```
journalctl -f
```

Open journal in reverse order (newest logs first),

```
journalctl -r
```

Show logs only from current boot,

```
journalctl -b
```

Show logs from one boot before this one (note -1 argument),

```
journalctl -b-1
```

Filter based on unit name,

```
journalctl -u <UNIT>
```

Show logs from certain executable,

```
journalctl _EXE=<PATH>
```

Show logs from certain PID,

```
journalctl _PID=<PID>
```

Time based filtering,

```
    journalctl --since|until
```

Show only messages logged with priority error and higher,

```
journalctl -p err
```

Show only kernel logs,

```
journalctl -k
```

Show all log messages about `/dev/sda` (you can also use persistent device names),

```
journalctl /dev/sda
```

Of course it is possible to combine options together (logical AND), e.g. show all kernel errors from previous boot

```
journalctl -b-1 -k -p err
```

Using "+" operator it is possible to create logical OR matches. Show errors from previous boot of both dbus and NetworkManager,

```
journalctl -b-1 -p err _SYSTEMD_UNIT=dbus.service +
   _SYSTEMD_UNIT=NetworkManager.service}
```

Depending on your configuration, log files are stored in one of these locations,

- `/run/log/journal` – Volatile journal (deleted on reboot)
- `/var/log/journal` – Persistent configuration

Your default journald configuration depends heavily on distro. For example, RHEL-7 uses volatile configuration by default. On the side, there is rsyslog running that reads journal data and stores them in usual places (e.g. `/var/log/messages`).

- `journalctl --disk-usage` – Reports how much space is taken by journals
- `journalctl --header` – Show detailed information about present journal files
- `journalctl -o json-pretty` – Format journal data as json