systemd.service

Name

systemd.service — Service unit configuration

Synopsis

service.service

Description

A unit configuration file whose name ends in ".service" encodes information about a process controlled and supervised by systemd.

This man page lists the configuration options specific to this unit type. See systemd.unit(5) for the common options of all unit configuration files. The common configuration items are configured in the generic [Unit] and [Install] sections. The service specific configuration options are configured in the [Service] section.

Additional options are listed in systemd.exec(5), which define the execution environment the commands are executed in, and in systemd.kill(5), which define the way the processes of the service are terminated, and in systemd.resource-control(5), which configure resource control settings for the processes of the service.

If SysV init compat is enabled, systemd automatically creates service units that wrap SysV init scripts (the service name is the same as the name of the script, with a ".service" suffix added); see systemd-sysv-generator(8).

The systemd-run(1) command allows creating .service and .scope units dynamically and transiently from the command line.

Service Templates

It is possible for **systemd** services to take a single argument via the "*service*@*argument*.service" syntax. Such services are called "instantiated" services, while the unit definition without the *argument* parameter is called a "template". An example could be a dhcpcd@.service service template which takes a network interface as a parameter to form an instantiated service. Within the service file, this parameter or "instance name" can be accessed with %-specifiers. See systemd.unit(5) for details.

Automatic Dependencies

Implicit Dependencies

The following dependencies are implicitly added:

- Services with Type=dbus set automatically acquire dependencies of type Requires= and After= on dbus.socket.
- Socket activated services are automatically ordered after their activating .socket units via an automatic After= dependency. Services also pull in all .socket units listed in Sockets= via automatic Wants= and After= dependencies.

Additional implicit dependencies may be added as result of execution and resource control parameters as documented in systemd.exec(5) and systemd.resource-control(5).

Default Dependencies

The following dependencies are added unless DefaultDependencies=no is set:

- Service units will have dependencies of type Requires= and After= on sysinit.target, a dependency of type After= on basic.target as well as dependencies of type Conflicts= and Before= on shutdown.target. These ensure that normal service units pull in basic system initialization, and are terminated cleanly prior to system shutdown. Only services involved with early boot or late system shutdown should disable this option.
- Instanced service units (i.e. service units with an "@" in their name) are assigned by default a per-template slice unit (see systemd.slice(5)), named after the template unit, containing all instances of the specific template. This slice is normally stopped at shutdown, together with all template instances. If that is not desired, set DefaultDependencies=no in the template unit, and either define your own per-template slice unit file that also sets DefaultDependencies=no, or set Slice=system.slice (or another suitable slice) in the template unit. Also see systemd.resource-control(5).

Command lines

This section describes command line parsing and variable and specifier substitutions for ExecStart=, ExecStartPre=, ExecStartPost=, ExecReload=, ExecStop=, and ExecStopPost= options.

Multiple command lines may be concatenated in a single directive by separating them with semicolons (these semicolons must be passed as separate words). Lone semicolons may be escaped as "`\;`".

Each command line is unquoted using the rules described in "Quoting" section in systemd.syntax(7). The first item becomes the command to execute, and the subsequent items the arguments.

This syntax is inspired by shell syntax, but only the meta-characters and expansions described in the following paragraphs are understood, and the expansion of variables is different. Specifically, redirection using "<", "<<", ">", and ">>", pipes using "|", running programs in the background using "&", and *other elements of shell syntax are not supported*.

The command to execute may contain spaces, but control characters are not allowed.

The command line accepts "%" specifiers as described in systemd.unit(5).

Basic environment variable substitution is supported. Use "`${FOO}`" as part of a word, or as a word of its own, on the command line, in which case it will be erased and replaced by the exact value of the environment variable (if any) including all whitespace it contains, always resulting in exactly a single argument. Use "`$FOO`" as a separate word on the command line, in which case it will be replaced by the value of the environment variable split at whitespace, resulting in zero or more arguments. For this type of expansion, quotes are respected when splitting into words, and afterwards removed.

If the command is not a full (absolute) path, it will be resolved to a full path using a fixed search path determined at compilation time. Searched directories include `/usr/local/bin/`, `/usr/bin/`, `/bin/` on systems using split `/usr/bin/` and `/bin/` directories, and their `sbin/` counterparts on systems using split `bin/` and `sbin/`. It is thus safe to use just the executable name in case of executables located in any of the "standard" directories, and an absolute path must be used in other cases. Using an absolute path is recommended to avoid ambiguity. Hint: this search path may be queried using **systemd-path search-binaries-default**.

Example:

```
Environment="ONE=one" 'TWO=two two'
ExecStart=echo $ONE $TWO ${TWO}
```

This will execute **/bin/echo** with four arguments: "`one`", "`two`", "`two`", and "`two two`".

Example:

```
Environment=ONE='one' "TWO='two two' too" THREE=
ExecStart=/bin/echo ${ONE} ${TWO} ${THREE}
ExecStart=/bin/echo $ONE $TWO $THREE
```

This results in `/bin/echo` being called twice, the first time with arguments "`'one'`", "`'two two' too`", "", and the second time with arguments "`one`", "`two two`", "`too`".

To pass a literal dollar sign, use "$$". Variables whose value is not known at expansion time are treated as empty strings. Note that the first argument (i.e. the program to execute) may not be a variable.

Variables to be used in this fashion may be defined through `Environment=` and `EnvironmentFile=`. In addition, variables listed in the section "Environment variables in spawned processes" in systemd.exec(5), which are considered "static configuration", may be used (this includes e.g. `$USER`, but not `$TERM`).

Note that shell command lines are not directly supported. If shell command lines are to be used, they need to be passed explicitly to a shell implementation of some kind. Example:

```
ExecStart=sh -c 'dmesg | tac'
```

Example:

```
ExecStart=echo one ; echo "two two"
```

This will execute **echo** two times, each time with one argument: "`one`" and "`two two`", respectively. Because two commands are specified, `Type=oneshot` must be used.

Example:

```
ExecStart=echo / >/dev/null & \; \
ls
```

This will execute **echo** with five arguments: "`/`", "`>/dev/null`", "`&`", "`;`", and "`ls`".

Examples

Example 2. Simple service

The following unit file creates a service that will execute `/usr/sbin/foo-daemon`. Since no `Type=` is specified, the default `Type=simple` will be assumed. systemd will assume the unit to be started immediately after the program has begun executing.

```
[Unit]
Description=Foo

[Service]
ExecStart=/usr/sbin/foo-daemon

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

Note that systemd assumes here that the process started by systemd will continue running until the service terminates. If the program daemonizes itself (i. e. forks), please use `Type=forking` instead.

Since no `ExecStop=` was specified, systemd will send SIGTERM to all processes started from this service, and after a timeout also SIGKILL. This behavior can be modified, see systemd.kill(5) for details.

Note that this unit type does not include any type of notification when a service has completed initialization. For this, you should use other unit types, such as `Type=notify` if the service understands systemd's notification protocol, `Type=forking` if the service can background itself or `Type=dbus` if the unit acquires a DBus name once initialization is complete. See below.

Example 3. Oneshot service

Sometimes, units should just execute an action without keeping active processes, such as a filesystem check or a cleanup action on boot. For this, `Type=oneshot` exists. Units of this type will wait until the process specified terminates and then fall back to being inactive. The following unit will perform a cleanup action:

```
[Unit]
Description=Cleanup old Foo data

[Service]
Type=oneshot
ExecStart=/usr/sbin/foo-cleanup

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

Note that systemd will consider the unit to be in the state "starting" until the program has terminated, so ordered dependencies will wait for the program to finish before starting themselves. The unit will revert to the "inactive" state after the execution is done, never reaching the "active" state. That means another request to start the unit will perform the action again.

`Type=oneshot` are the only service units that may have more than one `ExecStart=` specified. For units with multiple commands (`Type=oneshot`), all commands will be run again.

For `Type=oneshot`, `Restart=always` and `Restart=on-success` are *not* allowed.

Example 4. Stoppable oneshot service

Similarly to the oneshot services, there are sometimes units that need to execute a program to set up something and then execute another to shut it down, but no process remains active while they are considered "started". Network configuration can sometimes fall into this category. Another use case is if a oneshot service shall not be executed each time when they are pulled in as a dependency, but only the first time.

For this, systemd knows the setting `RemainAfterExit=yes`, which causes systemd to consider the unit to be active if the start action exited successfully. This directive can be used with all types, but is most useful with `Type=oneshot` and `Type=simple`. With `Type=oneshot`, systemd waits until the start action has completed before it considers the unit to be active, so dependencies start only after the start action has succeeded. With `Type=simple`, dependencies will start immediately after the start action has been dispatched. The following unit provides an example for a simple static firewall.

```
[Unit]
Description=Simple firewall

[Service]
Type=oneshot
RemainAfterExit=yes
ExecStart=/usr/local/sbin/simple-firewall-start
ExecStop=/usr/local/sbin/simple-firewall-stop

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

Since the unit is considered to be running after the start action has exited, invoking **systemctl start** on that unit again will cause no action to be taken.

Example 5. Traditional forking services

Many traditional daemons/services background (i.e. fork, daemonize) themselves when starting. Set `Type=forking` in the service's unit file to support this mode of operation. systemd will consider the service to be in the process of initialization while the original program is still running. Once it exits successfully and at least a process remains (and `RemainAfterExit=no`), the service is considered started.

Often, a traditional daemon only consists of one process. Therefore, if only one process is left after the original process terminates, systemd will consider that process the main process of the service. In that case, the `$MAINPID` variable will be available in `ExecReload=`, `ExecStop=`, etc.

In case more than one process remains, systemd will be unable to determine the main process, so it will not assume there is one. In that case, `$MAINPID` will not expand to anything. However, if the process decides to write a traditional PID file, systemd will be able to read the main PID from there. Please set `PIDFile=` accordingly. Note that the daemon should write that file before finishing with its initialization. Otherwise, systemd might try to read the file before it exists.

The following example shows a simple daemon that forks and just starts one process in the background:

```
[Unit]
Description=Some simple daemon

[Service]
Type=forking
ExecStart=/usr/sbin/my-simple-daemon -d

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

Please see systemd.kill(5) for details on how you can influence the way systemd terminates the service.

Example 6. DBus services

For services that acquire a name on the DBus system bus, use `Type=dbus` and set `BusName=` accordingly. The service should not fork (daemonize). systemd will consider the service to be initialized once the name has been acquired on the system bus. The following example shows a typical DBus service:

```
[Unit]
Description=Simple DBus service

[Service]
Type=dbus
BusName=org.example.simple-dbus-service
ExecStart=/usr/sbin/simple-dbus-service

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

For *bus-activatable* services, do not include a [Install] section in the systemd service file, but use the `SystemdService=` option in the corresponding DBus service file, for example (`/usr/share/dbus-1/system-services/org.example.simple-dbus-service.service`):

```
[D-BUS Service]
Name=org.example.simple-dbus-service
Exec=/usr/sbin/simple-dbus-service
User=root
SystemdService=simple-dbus-service.service
```

Please see systemd.kill(5) for details on how you can influence the way systemd terminates the service.

Example 7. Services that notify systemd about their initialization

`Type=simple` services are really easy to write, but have the major disadvantage of systemd not being able to tell when initialization of the given service is complete. For this reason, systemd supports a simple notification protocol that allows daemons to make systemd aware that they are done initializing. Use `Type=notify` for this. A typical service file for such a daemon would look like this:

```
[Unit]
Description=Simple notifying service

[Service]
Type=notify
ExecStart=/usr/sbin/simple-notifying-service

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

Note that the daemon has to support systemd's notification protocol, else systemd will think the service has not started yet and kill it after a timeout. For an example of how to update daemons to support this protocol transparently, take a look at sd_notify(3). systemd will consider the unit to be in the 'starting' state until a readiness notification has arrived.

Please see systemd.kill(5) for details on how you can influence the way systemd terminates the service.