NAME

systemd.unit - Unit configuration

SYNOPSIS

service.service, *socket*.socket, *device*.device, *mount*.mount, *automount*.automount, *swap*.swap, *target*.target, *path*.path, *timer*.timer, *slice*.slice, *scope*.scope

/etc/systemd/system/*
/run/systemd/system/*
/lib/systemd/system/*
...

~/.config/systemd/user/*
/etc/systemd/user/*
$XDG_RUNTIME_DIR/systemd/user/*
/run/systemd/user/*
~/.local/share/systemd/user/*
/usr/lib/systemd/user/*
...

DESCRIPTION

A unit configuration file encodes information about a service, a socket, a device, a mount point, an automount point, a swap file or partition, a start-up target, a watched file system path, a timer controlled and supervised by **systemd(1)**, a resource management slice or a group of externally created processes. The syntax is inspired by **XDG Desktop Entry Specification**[1] .desktop files, which are in turn inspired by Microsoft Windows .ini files.

This man page lists the common configuration options of all the unit types. These options need to be configured in the [Unit] or [Install] sections of the unit files.

In addition to the generic [Unit] and [Install] sections described here, each unit may have a type-specific section, e.g. [Service] for a service unit. See the respective man pages for more information: **systemd.service(5)**, **systemd.socket(5)**, **systemd.device(5)**, **systemd.mount(5)**, **systemd.automount(5)**, **systemd.swap(5)**, **systemd.target(5)**, **systemd.path(5)**, **systemd.timer(5)**, **systemd.slice(5)**, **systemd.scope(5)**.

Various settings are allowed to be specified more than once, in which case the interpretation depends on the setting. Often, multiple settings form a list, and setting to an empty value "resets", which means that previous assignments are ignored. When this is allowed, it is mentioned in the description of the setting. Note that using multiple assignments to the same value makes the unit file incompatible with parsers for the XDG .desktop file format.

Unit files are loaded from a set of paths determined during compilation, described in the next section.

Unit files may contain additional options on top of those listed here. If systemd encounters an unknown option, it will write a warning log message but continue loading the unit. If an option or section name is prefixed with **X-**, it is ignored completely by systemd. Options within an ignored section do not need the prefix. Applications may use this to include additional information in the unit files.

Boolean arguments used in unit files can be written in various formats. For positive settings the strings **1**, **yes**, **true** and **on** are equivalent. For negative settings, the strings **0**, **no**, **false** and **off** are equivalent.

Time span values encoded in unit files can be written in various formats. A stand-alone number specifies a time in seconds. If suffixed with a time unit, the unit is honored. A concatenation of multiple values with units is supported, in which case the values are added up. Example: "50" refers to 50 seconds; "2min 200ms" refers to 2 minutes and 200 milliseconds, i.e. 120200 ms. The following time units are understood: "s", "min", "h", "d", "w", "ms", "us". For details see **systemd.time(7)**.

Empty lines and lines starting with "#" or ";" are ignored. This may be used for commenting. Lines ending in a backslash are concatenated with the following line while reading and the backslash is replaced by a

space character. This may be used to wrap long lines.

Units can be aliased (have an alternative name), by creating a symlink from the new name to the existing name in one of the unit search paths. For example, systemd-networkd.service has the alias dbus-org.freedesktop.network1.service, created during installation as the symlink /lib/systemd/system/dbus-org.freedesktop.network1.service. In addition, unit files may specify aliases through the *Alias=* directive in the [Install] section; those aliases are only effective when the unit is enabled. When the unit is enabled, symlinks will be created for those names, and removed when the unit is disabled. For example, reboot.target specifies *Alias=ctrl-alt-del.target*, so when enabled it will be invoked whenever CTRL+ALT+DEL is pressed. Alias names may be used in commands like **enable**, **disable**, **start**, **stop**, **status**, ..., and in unit dependency directives *Wants=*, *Requires=*, *Before=*, *After=*, ..., with the limitation that aliases specified through *Alias=* are only effective when the unit is enabled. Aliases cannot be used with the **preset** command.

Along with a unit file foo.service, the directory foo.service.wants/ may exist. All unit files symlinked from such a directory are implicitly added as dependencies of type *Wants=* to the unit. This is useful to hook units into the start-up of other units, without having to modify their unit files. For details about the semantics of *Wants=*, see below. The preferred way to create symlinks in the .wants/ directory of a unit file is with the **enable** command of the **systemctl(1)** tool which reads information from the [Install] section of unit files (see below). A similar functionality exists for *Requires=* type dependencies as well, the directory suffix is .requires/ in this case.

Along with a unit file foo.service, a "drop-in" directory foo.service.d/ may exist. All files with the suffix ".conf" from this directory will be parsed after the file itself is parsed. This is useful to alter or add configuration settings for a unit, without having to modify unit files. Each drop-in file must have appropriate section headers. Note that for instantiated units, this logic will first look for the instance ".d/" subdirectory and read its ".conf" files, followed by the template ".d/" subdirectory and the ".conf" files there. Also note that settings from the "[Install]" section are not honored in drop-in unit files, and have no effect.

In addition to /etc/systemd/system, the drop-in ".d" directories for system services can be placed in /lib/systemd/system or /run/systemd/system directories. Drop-in files in /etc take precedence over those in /run which in turn take precedence over those in /lib. Drop-in files under any of these directories take precedence over unit files wherever located.

Some unit names reflect paths existing in the file system namespace. Example: a device unit dev-sda.device refers to a device with the device node /dev/sda in the file system namespace. If this applies, a special way to escape the path name is used, so that the result is usable as part of a filename. Basically, given a path, "/" is replaced by "-", and all other characters which are not ASCII alphanumerics are replaced by C-style "\x2d" escapes (except that "_" is never replaced and "." is only replaced when it would be the first character in the escaped path). The root directory "/" is encoded as single dash, while otherwise the initial and ending "/" are removed from all paths during transformation. This escaping is reversible. Properly escaped paths can be generated using the **systemd-escape(1)** command.

Optionally, units may be instantiated from a template file at runtime. This allows creation of multiple units from a single configuration file. If systemd looks for a unit configuration file, it will first search for the literal unit name in the file system. If that yields no success and the unit name contains an "@" character, systemd will look for a unit template that shares the same name but with the instance string (i.e. the part between the "@" character and the suffix) removed. Example: if a service getty@tty3.service is requested and no file by that name is found, systemd will look for getty@.service and instantiate a service from that configuration file if it is found.

To refer to the instance string from within the configuration file you may use the special "%i" specifier in many of the configuration options. See below for details.

If a unit file is empty (i.e. has the file size 0) or is symlinked to /dev/null, its configuration will not be loaded and it appears with a load state of "masked", and cannot be activated. Use this as an effective way to fully disable a unit, making it impossible to start it even manually.

The unit file format is covered by the **Interface Stability Promise**[2].

AUTOMATIC DEPENDENCIES

Note that while systemd offers a flexible dependency system between units it is recommended to use this functionality only sparingly and instead rely on techniques such as bus-based or socket-based activation which make dependencies implicit, resulting in a both simpler and more flexible system.

A number of unit dependencies are automatically established, depending on unit configuration. On top of that, for units with *DefaultDependencies=yes* (the default) a couple of additional dependencies are added. The precise effect of *DefaultDependencies=yes* depends on the unit type (see below).

If *DefaultDependencies=yes* is set, units that are referenced by other units of type .target via a *Wants=* or *Requires=* dependency might automatically gain an *Before=* dependency too. See **systemd.target(5)** for details.

UNIT FILE LOAD PATH

Unit files are loaded from a set of paths determined during compilation, described in the two tables below. Unit files found in directories listed earlier override files with the same name in directories lower in the list.

When the variable *$SYSTEMD_UNIT_PATH* is set, the contents of this variable overrides the unit load path. If *$SYSTEMD_UNIT_PATH* ends with an empty component (":"), the usual unit load path will be appended to the contents of the variable.

Table 1. Load path when running in system mode (--system).

Path	Description
/etc/systemd/system	Local configuration
/run/systemd/system	Runtime units
/lib/systemd/system	Units of installed packages

Table 2. Load path when running in user mode (--user).

Path	Description
$XDG_CONFIG_HOME/systemd/user	User configuration (only used when $XDG_CONFIG_HOME is set)
$HOME/.config/systemd/user	User configuration (only used when $XDG_CONFIG_HOME is not set)
/etc/systemd/user	Local configuration
$XDG_RUNTIME_DIR/systemd/user	Runtime units (only used when $XDG_RUNTIME_DIR is set)
/run/systemd/user	Runtime units
$XDG_DATA_HOME/systemd/user	Units of packages that have been installed in the home directory (only used when $XDG_DATA_HOME is set)
$HOME/.local/share/systemd/user	Units of packages that have been installed in the home directory (only used when $XDG_DATA_HOME is not set)
/usr/lib/systemd/user	Units of packages that have been installed system-wide

Additional units might be loaded into systemd ("linked") from directories not on the unit load path. See the **link** command for **systemctl(1)**. Also, some units are dynamically created via a **systemd.generator(7)**.

[UNIT] SECTION OPTIONS

The unit file may include a [Unit] section, which carries generic information about the unit that is not dependent on the type of unit:

Description=

A free-form string describing the unit. This is intended for use in UIs to show descriptive information along with the unit name. The description should contain a name that means something to the end user. "Apache2 Web Server" is a good example. Bad examples are "high-performance light-weight HTTP server" (too generic) or "Apache2" (too specific and meaningless for people who do not know Apache).

Documentation=
A space-separated list of URIs referencing documentation for this unit or its configuration. Accepted are only URIs of the types "http://", "https://", "file:", "info:", "man:". For more information about the syntax of these URIs, see **uri(7)**. The URIs should be listed in order of relevance, starting with the most relevant. It is a good idea to first reference documentation that explains what the unit's purpose is, followed by how it is configured, followed by any other related documentation. This option may be specified more than once, in which case the specified list of URIs is merged. If the empty string is assigned to this option, the list is reset and all prior assignments will have no effect.

Requires=
Configures requirement dependencies on other units. If this unit gets activated, the units listed here will be activated as well. If one of the other units gets deactivated or its activation fails, this unit will be deactivated. This option may be specified more than once or multiple space-separated units may be specified in one option in which case requirement dependencies for all listed names will be created. Note that requirement dependencies do not influence the order in which services are started or stopped. This has to be configured independently with the *After=* or *Before=* options. If a unit foo.service requires a unit bar.service as configured with *Requires=* and no ordering is configured with *After=* or *Before=*, then both units will be started simultaneously and without any delay between them if foo.service is activated. Often, it is a better choice to use *Wants=* instead of *Requires=* in order to achieve a system that is more robust when dealing with failing services.

Note that dependencies of this type may also be configured outside of the unit configuration file by adding a symlink to a .requires/ directory accompanying the unit file. For details, see above.

Requisite=
Similar to *Requires=*. However, if the units listed here are not started already, they will not be started and the transaction will fail immediately.

Wants=
A weaker version of *Requires=*. Units listed in this option will be started if the configuring unit is. However, if the listed units fail to start or cannot be added to the transaction, this has no impact on the validity of the transaction as a whole. This is the recommended way to hook start-up of one unit to the start-up of another unit.

Note that dependencies of this type may also be configured outside of the unit configuration file by adding symlinks to a .wants/ directory accompanying the unit file. For details, see above.

BindsTo=
Configures requirement dependencies, very similar in style to *Requires=*, however in addition to this behavior, it also declares that this unit is stopped when any of the units listed suddenly disappears. Units can suddenly, unexpectedly disappear if a service terminates on its own choice, a device is unplugged or a mount point unmounted without involvement of systemd.

PartOf=
Configures dependencies similar to *Requires=*, but limited to stopping and restarting of units. When systemd stops or restarts the units listed here, the action is propagated to this unit. Note that this is a one-way dependency — changes to this unit do not affect the listed units.

Conflicts=
A space-separated list of unit names. Configures negative requirement dependencies. If a unit has a *Conflicts=* setting on another unit, starting the former will stop the latter and vice versa. Note that this setting is independent of and orthogonal to the *After=* and *Before=* ordering dependencies.

If a unit A that conflicts with a unit B is scheduled to be started at the same time as B, the transaction

will either fail (in case both are required part of the transaction) or be modified to be fixed (in case one or both jobs are not a required part of the transaction). In the latter case, the job that is not the required will be removed, or in case both are not required, the unit that conflicts will be started and the unit that is conflicted is stopped.

Before=, *After=*
A space-separated list of unit names. Configures ordering dependencies between units. If a unit foo.service contains a setting **Before=bar.service** and both units are being started, bar.service's start-up is delayed until foo.service is started up. Note that this setting is independent of and orthogonal to the requirement dependencies as configured by *Requires=*. It is a common pattern to include a unit name in both the *After=* and *Requires=* option, in which case the unit listed will be started before the unit that is configured with these options. This option may be specified more than once, in which case ordering dependencies for all listed names are created. *After=* is the inverse of *Before=*, i.e. while *After=* ensures that the configured unit is started after the listed unit finished starting up, *Before=* ensures the opposite, i.e. that the configured unit is fully started up before the listed unit is started. Note that when two units with an ordering dependency between them are shut down, the inverse of the start-up order is applied. i.e. if a unit is configured with *After=* on another unit, the former is stopped before the latter if both are shut down. Given two units with any ordering dependency between them, if one unit is shut down and the other is started up, the shutdown is ordered before the start-up. It doesn't matter if the ordering dependency is *After=* or *Before=*. It also doesn't matter which of the two is shut down, as long as one is shut down and the other is started up. The shutdown is ordered before the start-up in all cases. If two units have no ordering dependencies between them, they are shut down or started up simultaneously, and no ordering takes place.

OnFailure=
A space-separated list of one or more units that are activated when this unit enters the "failed" state.

PropagatesReloadTo=, *ReloadPropagatedFrom=*
A space-separated list of one or more units where reload requests on this unit will be propagated to, or reload requests on the other unit will be propagated to this unit, respectively. Issuing a reload request on a unit will automatically also enqueue a reload request on all units that the reload request shall be propagated to via these two settings.

JoinsNamespaceOf=
For units that start processes (such as service units), lists one or more other units whose network and/or temporary file namespace to join. This only applies to unit types which support the *PrivateNetwork=* and *PrivateTmp=* directives (see **systemd.exec(5)** for details). If a unit that has this setting set is started, its processes will see the same /tmp, /var/tmp and network namespace as one listed unit that is started. If multiple listed units are already started, it is not defined which namespace is joined. Note that this setting only has an effect if *PrivateNetwork=* and/or *PrivateTmp=* is enabled for both the unit that joins the namespace and the unit whose namespace is joined.

RequiresMountsFor=
Takes a space-separated list of absolute paths. Automatically adds dependencies of type *Requires=* and *After=* for all mount units required to access the specified path.

Mount points marked with **noauto** are not mounted automatically and will be ignored for the purposes of this option. If such a mount should be a requirement for this unit, direct dependencies on the mount units may be added (*Requires=* and *After=* or some other combination).

OnFailureJobMode=
Takes a value of "fail", "replace", "replace-irreversibly", "isolate", "flush", "ignore-dependencies" or "ignore-requirements". Defaults to "replace". Specifies how the units listed in *OnFailure=* will be enqueued. See **systemctl(1)**'s **--job-mode=** option for details on the possible values. If this is set to "isolate", only a single unit may be listed in *OnFailure=*..

IgnoreOnIsolate=
Takes a boolean argument. If **true**, this unit will not be stopped when isolating another unit. Defaults to **false**.

StopWhenUnneeded=
Takes a boolean argument. If **true**, this unit will be stopped when it is no longer used. Note that, in order to minimize the work to be executed, systemd will not stop units by default unless they are conflicting with other units, or the user explicitly requested their shut down. If this option is set, a unit will be automatically cleaned up if no other active unit requires it. Defaults to **false**.

RefuseManualStart=, *RefuseManualStop=*
Takes a boolean argument. If **true**, this unit can only be activated or deactivated indirectly. In this case, explicit start-up or termination requested by the user is denied, however if it is started or stopped as a dependency of another unit, start-up or termination will succeed. This is mostly a safety feature to ensure that the user does not accidentally activate units that are not intended to be activated explicitly, and not accidentally deactivate units that are not intended to be deactivated. These options default to **false**.

AllowIsolate=
Takes a boolean argument. If **true**, this unit may be used with the **systemctl isolate** command. Otherwise, this will be refused. It probably is a good idea to leave this disabled except for target units that shall be used similar to runlevels in SysV init systems, just as a precaution to avoid unusable system states. This option defaults to **false**.

DefaultDependencies=
Takes a boolean argument. If **true**, (the default), a few default dependencies will implicitly be created for the unit. The actual dependencies created depend on the unit type. For example, for service units, these dependencies ensure that the service is started only after basic system initialization is completed and is properly terminated on system shutdown. See the respective man pages for details. Generally, only services involved with early boot or late shutdown should set this option to **false**. It is highly recommended to leave this option enabled for the majority of common units. If set to **false**, this option does not disable all implicit dependencies, just non-essential ones.

JobTimeoutSec=, *JobTimeoutAction=*, *JobTimeoutRebootArgument=*
When a job for this unit is queued, a time-out may be configured. If this time limit is reached, the job will be cancelled, the unit however will not change state or even enter the "failed" mode. This value defaults to "infinity" (job timeouts disabled), except for device units. NB: this timeout is independent from any unit-specific timeout (for example, the timeout set with *TimeoutStartSec=* in service units) as the job timeout has no effect on the unit itself, only on the job that might be pending for it. Or in other words: unit-specific timeouts are useful to abort unit state changes, and revert them. The job timeout set with this option however is useful to abort only the job waiting for the unit state to change.

JobTimeoutAction= optionally configures an additional action to take when the time-out is hit. It takes the same values as the per-service *StartLimitAction=* setting, see **systemd.service(5)** for details. Defaults to **none**. *JobTimeoutRebootArgument=* configures an optional reboot string to pass to the **reboot(2)** system call.

StartLimitIntervalSec=, *StartLimitBurst=*
Configure unit start rate limiting. By default, units which are started more than 5 times within 10 seconds are not permitted to start any more times until the 10 second interval ends. With these two options, this rate limiting may be modified. Use *StartLimitIntervalSec=* to configure the checking interval (defaults to *DefaultStartLimitIntervalSec=* in manager configuration file, set to 0 to disable any kind of rate limiting). Use *StartLimitBurst=* to configure how many starts per interval are allowed (defaults to *DefaultStartLimitBurst=* in manager configuration file). These configuration options are particularly useful in conjunction with the service setting *Restart=* (see **systemd.service(5)**); however, they apply to all kinds of starts (including manual), not just those triggered by the *Restart=* logic. Note that units which are configured for *Restart=* and which reach the start limit are not attempted to be restarted anymore; however, they may still be restarted manually at a later point, from which point on, the restart logic is again activated. Note that **systemctl reset-failed** will cause the restart rate counter for a service to be flushed, which is useful if the administrator wants to manually start a unit and the start limit interferes with that. Note that this rate-limiting is enforced after any unit condition checks are executed, and hence unit activations with failing conditions are not counted by this rate limiting.

Slice, target, device and scope units do not enforce this setting, as they are unit types whose activation may either never fail, or may succeed only a single time.

StartLimitAction=
Configure the action to take if the rate limit configured with *StartLimitIntervalSec=* and *StartLimitBurst=* is hit. Takes one of **none**, **reboot**, **reboot-force**, **reboot-immediate**, **poweroff**, **poweroff-force** or **poweroff-immediate**. If **none** is set, hitting the rate limit will trigger no action besides that the start will not be permitted. **reboot** causes a reboot following the normal shutdown procedure (i.e. equivalent to **systemctl reboot**). **reboot-force** causes a forced reboot which will terminate all processes forcibly but should cause no dirty file systems on reboot (i.e. equivalent to **systemctl reboot -f**) and **reboot-immediate** causes immediate execution of the **reboot(2)** system call, which might result in data loss. Similarly, **poweroff**, **poweroff-force**, **poweroff-immediate** have the effect of powering down the system with similar semantics. Defaults to **none**.

RebootArgument=
Configure the optional argument for the **reboot(2)** system call if *StartLimitAction=* or a service's *FailureAction=* is a reboot action. This works just like the optional argument to **systemctl reboot** command.

ConditionArchitecture=, *ConditionVirtualization=*, *ConditionHost=*, *ConditionKernelCommandLine=*, *ConditionSecurity=*, *ConditionCapability=*, *ConditionACPower=*, *ConditionNeedsUpdate=*, *ConditionFirstBoot=*, *ConditionPathExists=*, *ConditionPathExistsGlob=*, *ConditionPathIsDirectory=*, *ConditionPathIsSymbolicLink=*, *ConditionPathIsMountPoint=*, *ConditionPathIsReadWrite=*, *ConditionDirectoryNotEmpty=*, *ConditionFileNotEmpty=*, *ConditionFileIsExecutable=*
Before starting a unit, verify that the specified condition is true. If it is not true, the starting of the unit will be (mostly silently) skipped, however all ordering dependencies of it are still respected. A failing condition will not result in the unit being moved into a failure state. The condition is checked at the time the queued start job is to be executed. Use condition expressions in order to silently skip units that do not apply to the local running system, for example because the kernel or runtime environment doesn't require its functionality. Use the various *AssertArchitecture=*, *AssertVirtualization=*, ... options for a similar mechanism that puts the unit in a failure state and logs about the failed check (see below).

ConditionArchitecture= may be used to check whether the system is running on a specific architecture. Takes one of *x86*, *x86-64*, *ppc*, *ppc-le*, *ppc64*, *ppc64-le*, *ia64*, *parisc*, *parisc64*, *s390*, *s390x*, *sparc*, *sparc64*, *mips*, *mips-le*, *mips64*, *mips64-le*, *alpha*, *arm*, *arm-be*, *arm64*, *arm64-be*, *sh*, *sh64*, *m86k*, *tilegx*, *cris* to test against a specific architecture. The architecture is determined from the information returned by **uname(2)** and is thus subject to **personality(2)**. Note that a*Personality=* setting in the same unit file has no effect on this condition. A special architecture name *native* is mapped to the architecture the system manager itself is compiled for. The test may be negated by prepending an exclamation mark.

ConditionVirtualization= may be used to check whether the system is executed in a virtualized environment and optionally test whether it is a specific implementation. Takes either boolean value to check if being executed in any virtualized environment, or one of *vm* and *container* to test against a generic type of virtualization solution, or one of *qemu*, *kvm*, *zvm*, *vmware*, *microsoft*, *oracle*, *xen*, *bochs*, *uml*, *openvz*, *lxc*, *lxc-libvirt*, *systemd-nspawn*, *docker*, *rkt* to test against a specific implementation, or *private-users* to check whether we are running in a user namespace. See **systemd-detect-virt(1)** for a full list of known virtualization technologies and their identifiers. If multiple virtualization technologies are nested, only the innermost is considered. The test may be negated by prepending an exclamation mark.

ConditionHost= may be used to match against the hostname or machine ID of the host. This either takes a hostname string (optionally with shell style globs) which is tested against the locally set hostname as returned by **gethostname(2)**, or a machine ID formatted as string (see **machine-id(5)**). The test may be negated by prepending an exclamation mark.

ConditionKernelCommandLine= may be used to check whether a specific kernel command line option is set (or if prefixed with the exclamation mark unset). The argument must either be a single word, or

an assignment (i.e. two words, separated "="). In the former case the kernel command line is searched for the word appearing as is, or as left hand side of an assignment. In the latter case, the exact assignment is looked for with right and left hand side matching.

ConditionSecurity= may be used to check whether the given security module is enabled on the system. Currently, the recognized values are *selinux*, *apparmor*, *ima*, *smack* and *audit*. The test may be negated by prepending an exclamation mark.

ConditionCapability= may be used to check whether the given capability exists in the capability bounding set of the service manager (i.e. this does not check whether capability is actually available in the permitted or effective sets, see **capabilities(7)** for details). Pass a capability name such as "CAP_MKNOD", possibly prefixed with an exclamation mark to negate the check.

ConditionACPower= may be used to check whether the system has AC power, or is exclusively battery powered at the time of activation of the unit. This takes a boolean argument. If set to *true*, the condition will hold only if at least one AC connector of the system is connected to a power source, or if no AC connectors are known. Conversely, if set to *false*, the condition will hold only if there is at least one AC connector known and all AC connectors are disconnected from a power source.

ConditionNeedsUpdate= takes one of /var or /etc as argument, possibly prefixed with a "!" (for inverting the condition). This condition may be used to conditionalize units on whether the specified directory requires an update because /usr's modification time is newer than the stamp file .updated in the specified directory. This is useful to implement offline updates of the vendor operating system resources in /usr that require updating of /etc or /var on the next following boot. Units making use of this condition should order themselves before **systemd-update-done.service(8)**, to make sure they run before the stamp file's modification time gets reset indicating a completed update.

ConditionFirstBoot= takes a boolean argument. This condition may be used to conditionalize units on whether the system is booting up with an unpopulated /etc directory. This may be used to populate /etc on the first boot after factory reset, or when a new system instances boots up for the first time.

With *ConditionPathExists*= a file existence condition is checked before a unit is started. If the specified absolute path name does not exist, the condition will fail. If the absolute path name passed to *ConditionPathExists*= is prefixed with an exclamation mark ("!"), the test is negated, and the unit is only started if the path does not exist.

ConditionPathExistsGlob= is similar to *ConditionPathExists*=, but checks for the existence of at least one file or directory matching the specified globbing pattern.

ConditionPathIsDirectory= is similar to *ConditionPathExists*= but verifies whether a certain path exists and is a directory.

ConditionPathIsSymbolicLink= is similar to *ConditionPathExists*= but verifies whether a certain path exists and is a symbolic link.

ConditionPathIsMountPoint= is similar to *ConditionPathExists*= but verifies whether a certain path exists and is a mount point.

ConditionPathIsReadWrite= is similar to *ConditionPathExists*= but verifies whether the underlying file system is readable and writable (i.e. not mounted read-only).

ConditionDirectoryNotEmpty= is similar to *ConditionPathExists*= but verifies whether a certain path exists and is a non-empty directory.

ConditionFileNotEmpty= is similar to *ConditionPathExists*= but verifies whether a certain path exists and refers to a regular file with a non-zero size.

ConditionFileIsExecutable= is similar to *ConditionPathExists*= but verifies whether a certain path exists, is a regular file and marked executable.

If multiple conditions are specified, the unit will be executed if all of them apply (i.e. a logical AND is applied). Condition checks can be prefixed with a pipe symbol (|) in which case a condition becomes a

triggering condition. If at least one triggering condition is defined for a unit, then the unit will be executed if at least one of the triggering conditions apply and all of the non-triggering conditions. If you prefix an argument with the pipe symbol and an exclamation mark, the pipe symbol must be passed first, the exclamation second. Except for *ConditionPathIsSymbolicLink=*, all path checks follow symlinks. If any of these options is assigned the empty string, the list of conditions is reset completely, all previous condition settings (of any kind) will have no effect.

AssertArchitecture=, *AssertVirtualization=*, *AssertHost=*, *AssertKernelCommandLine=*, *AssertSecurity=*, *AssertCapability=*, *AssertACPower=*, *AssertNeedsUpdate=*, *AssertFirstBoot=*, *AssertPathExists=*, *AssertPathExistsGlob=*, *AssertPathIsDirectory=*, *AssertPathIsSymbolicLink=*, *AssertPathIsMountPoint=*, *AssertPathIsReadWrite=*, *AssertDirectoryNotEmpty=*, *AssertFileNotEmpty=*, *AssertFileIsExecutable=*
Similar to the *ConditionArchitecture=*, *ConditionVirtualization=*, ..., condition settings described above, these settings add assertion checks to the start-up of the unit. However, unlike the conditions settings, any assertion setting that is not met results in failure of the start job (which means this is logged loudly). Use assertion expressions for units that cannot operate when specific requirements are not met, and when this is something the administrator or user should look into.

SourcePath=
A path to a configuration file this unit has been generated from. This is primarily useful for implementation of generator tools that convert configuration from an external configuration file format into native unit files. This functionality should not be used in normal units.

[INSTALL] SECTION OPTIONS

Unit files may include an "[Install]" section, which carries installation information for the unit. This section is not interpreted by **systemd(1)** during runtime; it is used by the **enable** and **disable** commands of the **systemctl(1)** tool during installation of a unit. Note that settings in the "[Install]" section may not appear in .d/*.conf unit file drop-ins (see above).

Alias=
A space-separated list of additional names this unit shall be installed under. The names listed here must have the same suffix (i.e. type) as the unit file name. This option may be specified more than once, in which case all listed names are used. At installation time, **systemctl enable** will create symlinks from these names to the unit filename. Note that not all unit types support such alias names, and this setting is not supported for them. Specifically, mount, slice, swap, and automount units do not support aliasing.

WantedBy=, *RequiredBy=*
This option may be used more than once, or a space-separated list of unit names may be given. A symbolic link is created in the .wants/ or .requires/ directory of each of the listed units when this unit is installed by **systemctl enable**. This has the effect that a dependency of type *Wants=* or *Requires=* is added from the listed unit to the current unit. The primary result is that the current unit will be started when the listed unit is started. See the description of *Wants=* and *Requires=* in the [Unit] section for details.

WantedBy=foo.service in a service bar.service is mostly equivalent to **Alias=foo.service.wants/bar.service** in the same file. In case of template units, **systemctl enable** must be called with an instance name, and this instance will be added to the .wants/ or .requires/ list of the listed unit. E.g. **WantedBy=getty.target** in a service getty@.service will result in **systemctl enable getty@tty2.service** creating a getty.target.wants/getty@tty2.service link to getty@.service.

Also=
Additional units to install/deinstall when this unit is installed/deinstalled. If the user requests installation/deinstallation of a unit with this option configured, **systemctl enable** and **systemctl disable** will automatically install/uninstall units listed in this option as well.

This option may be used more than once, or a space-separated list of unit names may be given.

DefaultInstance=
In template unit files, this specifies for which instance the unit shall be enabled if the template is enabled without any explicitly set instance. This option has no effect in non-template unit files. The

specified string must be usable as instance identifier.

The following specifiers are interpreted in the Install section: %n, %N, %p, %i, %U, %u, %m, %H, %b, %v. For their meaning see the next section.

SPECIFIERS

Many settings resolve specifiers which may be used to write generic unit files referring to runtime or unit parameters that are replaced when the unit files are loaded. The following specifiers are understood:

Table 3. Specifiers available in unit files

EXAMPLES

Example 1. Allowing units to be enabled

The following snippet (highlighted) allows a unit (e.g. foo.service) to be enabled via **systemctl enable**:

```
[Unit]
Description=Foo

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

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

After running **systemctl enable**, a symlink /etc/systemd/system/multi-user.target.wants/foo.service linking to the actual unit will be created. It tells systemd to pull in the unit when starting multi-user.target. The inverse **systemctl disable** will remove that symlink again.

Example 2. Overriding vendor settings

There are two methods of overriding vendor settings in unit files: copying the unit file from /lib/systemd/system to /etc/systemd/system and modifying the chosen settings. Alternatively, one can create a directory named *unit*.d/ within /etc/systemd/system and place a drop-in file *name*.conf there that only changes the specific settings one is interested in. Note that multiple such drop-in files are read if present.

The advantage of the first method is that one easily overrides the complete unit, the vendor unit is not parsed at all anymore. It has the disadvantage that improvements to the unit file by the vendor are not automatically incorporated on updates.

The advantage of the second method is that one only overrides the settings one specifically wants, where updates to the unit by the vendor automatically apply. This has the disadvantage that some future updates by the vendor might be incompatible with the local changes.

Note that for drop-in files, if one wants to remove entries from a setting that is parsed as a list (and is not a dependency), such as *ConditionPathExists=* (or e.g. *ExecStart=* in service units), one needs to first clear the list before re-adding all entries except the one that is to be removed. See below for an example.

This also applies for user instances of systemd, but with different locations for the unit files. See the section on unit load paths for further details.

Suppose there is a vendor-supplied unit /lib/systemd/system/httpd.service with the following contents:

```
[Unit]
Description=Some HTTP server
After=remote-fs.target sqldb.service
Requires=sqldb.service
AssertPathExists=/srv/webserver

[Service]
Type=notify
ExecStart=/usr/sbin/some-fancy-httpd-server
Nice=5

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

Now one wants to change some settings as an administrator: firstly, in the local setup, /srv/webserver might not exist, because the HTTP server is configured to use /srv/www instead. Secondly, the local configuration makes the HTTP server also depend on a memory cache service, memcached.service, that should be pulled in (*Requires=*) and also be ordered appropriately (*After=*). Thirdly, in order to harden the service a bit more, the administrator would like to set the *PrivateTmp=* setting (see **systemd.service(5)** for details). And lastly, the administrator would like to reset the niceness of the service to its default value of 0.

The first possibility is to copy the unit file to /etc/systemd/system/httpd.service and change the chosen

settings:

[Unit]
Description=Some HTTP server
After=remote-fs.target sqldb.service *memcached.service*
Requires=sqldb.service *memcached.service*
AssertPathExists=*/srv/www*

[Service]
Type=notify
ExecStart=/usr/sbin/some-fancy-httpd-server
Nice=0
PrivateTmp=yes

[Install]
WantedBy=multi-user.target

Alternatively, the administrator could create a drop-in file /etc/systemd/system/httpd.service.d/local.conf with the following contents:

[Unit]
After=memcached.service
Requires=memcached.service
# Reset all assertions and then re-add the condition we want
AssertPathExists=
AssertPathExists=/srv/www

[Service]
Nice=0
PrivateTmp=yes

Note that dependencies (*After=*, etc.) cannot be reset to an empty list, so dependencies can only be added in drop-ins. If you want to remove dependencies, you have to override the entire unit.

SEE ALSO

systemd(1), **systemctl(1)**, **systemd.special(7)**, **systemd.service(5)**, **systemd.socket(5)**, **systemd.device(5)**, **systemd.mount(5)**, **systemd.automount(5)**, **systemd.swap(5)**, **systemd.target(5)**, **systemd.path(5)**, **systemd.timer(5)**, **systemd.scope(5)**, **systemd.slice(5)**, **systemd.time(7)**, **systemd-analyze(1)**, **capabilities(7)**, **systemd.directives(7)**, **uname(1)**

NOTES

1. XDG Desktop Entry Specification
   http://standards.freedesktop.org/desktop-entry-spec/latest/
2. Interface Stability Promise
   http://www.freedesktop.org/wiki/Software/systemd/InterfaceStabilityPromise