<https://www.digitalocean.com/community/tutorials/understanding-systemd-units-and-unit-files>

# Understanding Systemd Units and Unit Files

PostedFebruary 17, 2015 374.8k [LINUX BASICS](https://www.digitalocean.com/community/tags/linux-basics?type=tutorials) [SYSTEM TOOLS](https://www.digitalocean.com/community/tags/system-tools?type=tutorials) [CONCEPTUAL](https://www.digitalocean.com/community/tags/conceptual?type=tutorials)

views

### **Introduction**

Increasingly, Linux distributions are adopting or planning to adopt the systemd init system. This powerful suite of software can manage many aspects of your server, from services to mounted devices and system states.

In systemd, a unit refers to any resource that the system knows how to operate on and manage. This is the primary object that the systemd tools know how to deal with. These resources are defined using configuration files called unit files.

In this guide, we will introduce you to the different units that systemd can handle. We will also be covering some of the many directives that can be used in unit files in order to shape the way these resources are handled on your system.

## What do Systemd Units Give You?

Units are the objects that systemd knows how to manage. These are basically a standardized representation of system resources that can be managed by the suite of daemons and manipulated by the provided utilities.

Units in some ways can be said to similar to services or jobs in other init systems. However, a unit has a much broader definition, as these can be used to abstract services, network resources, devices, filesystem mounts, and isolated resource pools.

Ideas that in other init systems may be handled with one unified service definition can be broken out into component units according to their focus. This organizes by function and allows you to easily enable, disable, or extend functionality without modifying the core behavior of a unit.

Some features that units are able implement easily are:

* **socket-based activation**: Sockets associated with a service are best broken out of the daemon itself in order to be handled separately. This provides a number of advantages, such as delaying the start of a service until the associated socket is first accessed. This also allows the system to create all sockets early in the boot process, making it possible to boot the associated services in parallel.
* **bus-based activation**: Units can also be activated on the bus interface provided by D-Bus. A unit can be started when an associated bus is published.
* **path-based activation**: A unit can be started based on activity on or the availability of certain filesystem paths. This utilizes inotify.
* **device-based activation**: Units can also be started at the first availability of associated hardware by leveraging udev events.
* **implicit dependency mapping**: Most of the dependency tree for units can be built by systemditself. You can still add dependency and ordering information, but most of the heavy lifting is taken care of for you.
* **instances and templates**: Template unit files can be used to create multiple instances of the same general unit. This allows for slight variations or sibling units that all provide the same general function.
* **easy security hardening**: Units can implement some fairly good security features by adding simple directives. For example, you can specify no or read-only access to part of the filesystem, limit kernel capabilities, and assign private /tmp and network access.
* **drop-ins and snippets**: Units can easily be extended by providing snippets that will override parts of the system's unit file. This makes it easy to switch between vanilla and customized unit implementations.

There are many other advantages that systemd units have over other init systems' work items, but this should give you an idea of the power that can be leveraged using native configuration directives.

## Where are Systemd Unit Files Found?

The files that define how systemd will handle a unit can be found in many different locations, each of which have different priorities and implications.

The system's copy of unit files are generally kept in the /lib/systemd/system directory. When software installs unit files on the system, this is the location where they are placed by default.

Unit files stored here are able to be started and stopped on-demand during a session. This will be the generic, vanilla unit file, often written by the upstream project's maintainers that should work on any system that deploys systemd in its standard implementation. You should not edit files in this directory. Instead you should override the file, if necessary, using another unit file location which will supersede the file in this location.

If you wish to modify the way that a unit functions, the best location to do so is within the /etc/systemd/system directory. Unit files found in this directory location take precedence over any of the other locations on the filesystem. If you need to modify the system's copy of a unit file, putting a replacement in this directory is the safest and most flexible way to do this.

If you wish to override only specific directives from the system's unit file, you can actually provide unit file snippets within a subdirectory. These will append or modify the directives of the system's copy, allowing you to specify only the options you want to change.

The correct way to do this is to create a directory named after the unit file with .d appended on the end. So for a unit called example.service, a subdirectory called example.service.d could be created. Within this directory a file ending with .conf can be used to override or extend the attributes of the system's unit file.

There is also a location for run-time unit definitions at /run/systemd/system. Unit files found in this directory have a priority landing between those in /etc/systemd/system and /lib/systemd/system. Files in this location are given less weight than the former location, but more weight than the latter.

The systemd process itself uses this location for dynamically created unit files created at runtime. This directory can be used to change the system's unit behavior for the duration of the session. All changes made in this directory will be lost when the server is rebooted.

## Types of Units

Systemd categories units according to the type of resource they describe. The easiest way to determine the type of a unit is with its type suffix, which is appended to the end of the resource name. The following list describes the types of units available to systemd:

* **.service**: A service unit describes how to manage a service or application on the server. This will include how to start or stop the service, under which circumstances it should be automatically started, and the dependency and ordering information for related software.
* **.socket**: A socket unit file describes a network or IPC socket, or a FIFO buffer that systemd uses for socket-based activation. These always have an associated .service file that will be started when activity is seen on the socket that this unit defines.
* **.device**: A unit that describes a device that has been designated as needing systemdmanagement by udev or the sysfs filesystem. Not all devices will have .device files. Some scenarios where .device units may be necessary are for ordering, mounting, and accessing the devices.
* **.mount**: This unit defines a mountpoint on the system to be managed by systemd. These are named after the mount path, with slashes changed to dashes. Entries within /etc/fstab can have units created automatically.
* **.automount**: An .automount unit configures a mountpoint that will be automatically mounted. These must be named after the mount point they refer to and must have a matching .mount unit to define the specifics of the mount.
* **.swap**: This unit describes swap space on the system. The name of these units must reflect the device or file path of the space.
* **.target**: A target unit is used to provide synchronization points for other units when booting up or changing states. They also can be used to bring the system to a new state. Other units specify their relation to targets to become tied to the target's operations.
* **.path**: This unit defines a path that can be used for path-based activation. By default, a .serviceunit of the same base name will be started when the path reaches the specified state. This uses inotify to monitor the path for changes.
* **.timer**: A .timer unit defines a timer that will be managed by systemd, similar to a cron job for delayed or scheduled activation. A matching unit will be started when the timer is reached.
* **.snapshot**: A .snapshot unit is created automatically by the systemctl snapshot command. It allows you to reconstruct the current state of the system after making changes. Snapshots do not survive across sessions and are used to roll back temporary states.
* **.slice**: A .slice unit is associated with Linux Control Group nodes, allowing resources to be restricted or assigned to any processes associated with the slice. The name reflects its hierarchical position within the cgroup tree. Units are placed in certain slices by default depending on their type.
* **.scope**: Scope units are created automatically by systemd from information received from its bus interfaces. These are used to manage sets of system processes that are created externally.

As you can see, there are many different units that systemd knows how to manage. Many of the unit types work together to add functionality. For instance, some units are used to trigger other units and provide activation functionality.

We will mainly be focusing on .service units due to their utility and the consistency in which administrators need to managed these units.

## Anatomy of a Unit File

The internal structure of unit files are organized with sections. Sections are denoted by a pair of square brackets "[" and "]" with the section name enclosed within. Each section extends until the beginning of the subsequent section or until the end of the file.

### **General Characteristics of Unit Files**

Section names are well defined and case-sensitive. So, the section [Unit] will **not** be interpreted correctly if it is spelled like [UNIT]. If you need to add non-standard sections to be parsed by applications other than systemd, you can add a X- prefix to the section name.

Within these sections, unit behavior and metadata is defined through the use of simple directives using a key-value format with assignment indicated by an equal sign, like this:

[Section]

Directive1=value

Directive2=value

. . .

In the event of an override file (such as those contained in a unit.type.d directory), directives can be reset by assigning them to an empty string. For example, the system's copy of a unit file may contain a directive set to a value like this:

Directive1=default\_value

The default\_value can be eliminated in an override file by referencing Directive1 without a value, like this:

Directive1=

In general, systemd allows for easy and flexible configuration. For example, multiple boolean expressions are accepted (1, yes, on, and true for affirmative and 0, no off, and false for the opposite answer). Times can be intelligently parsed, with seconds assumed for unit-less values and combining multiple formats accomplished internally.

### **[Unit] Section Directives**

The first section found in most unit files is the [Unit] section. This is generally used for defining metadata for the unit and configuring the relationship of the unit to other units.

Although section order does not matter to systemd when parsing the file, this section is often placed at the top because it provides an overview of the unit. Some common directives that you will find in the [Unit] section are:

* **Description=**: This directive can be used to describe the name and basic functionality of the unit. It is returned by various systemd tools, so it is good to set this to something short, specific, and informative.
* **Documentation=**: This directive provides a location for a list of URIs for documentation. These can be either internally available man pages or web accessible URLs. The systemctl statuscommand will expose this information, allowing for easy discoverability.
* **Requires=**: This directive lists any units upon which this unit essentially depends. If the current unit is activated, the units listed here must successfully activate as well, else this unit will fail. These units are started in parallel with the current unit by default.
* **Wants=**: This directive is similar to Requires=, but less strict. Systemd will attempt to start any units listed here when this unit is activated. If these units are not found or fail to start, the current unit will continue to function. This is the recommended way to configure most dependency relationships. Again, this implies a parallel activation unless modified by other directives.
* **BindsTo=**: This directive is similar to Requires=, but also causes the current unit to stop when the associated unit terminates.
* **Before=**: The units listed in this directive will not be started until the current unit is marked as started if they are activated at the same time. This does not imply a dependency relationship and must be used in conjunction with one of the above directives if this is desired.
* **After=**: The units listed in this directive will be started before starting the current unit. This does not imply a dependency relationship and one must be established through the above directives if this is required.
* **Conflicts=**: This can be used to list units that cannot be run at the same time as the current unit. Starting a unit with this relationship will cause the other units to be stopped.
* **Condition...=**: There are a number of directives that start with Condition which allow the administrator to test certain conditions prior to starting the unit. This can be used to provide a generic unit file that will only be run when on appropriate systems. If the condition is not met, the unit is gracefully skipped.
* **Assert...=**: Similar to the directives that start with Condition, these directives check for different aspects of the running environment to decide whether the unit should activate. However, unlike the Condition directives, a negative result causes a failure with this directive.

Using these directives and a handful of others, general information about the unit and its relationship to other units and the operating system can be established.

### **[Install] Section Directives**

On the opposite side of unit file, the last section is often the [Install] section. This section is optional and is used to define the behavior or a unit if it is enabled or disabled. Enabling a unit marks it to be automatically started at boot. In essence, this is accomplished by latching the unit in question onto another unit that is somewhere in the line of units to be started at boot.

Because of this, only units that can be enabled will have this section. The directives within dictate what should happen when the unit is enabled:

* **WantedBy=**: The WantedBy= directive is the most common way to specify how a unit should be enabled. This directive allows you to specify a dependency relationship in a similar way to the Wants= directive does in the [Unit] section. The difference is that this directive is included in the ancillary unit allowing the primary unit listed to remain relatively clean. When a unit with this directive is enabled, a directory will be created within /etc/systemd/system named after the specified unit with .wants appended to the end. Within this, a symbolic link to the current unit will be created, creating the dependency. For instance, if the current unit has WantedBy=multi-user.target, a directory called multi-user.target.wants will be created within /etc/systemd/system (if not already available) and a symbolic link to the current unit will be placed within. Disabling this unit removes the link and removes the dependency relationship.
* **RequiredBy=**: This directive is very similar to the WantedBy= directive, but instead specifies a required dependency that will cause the activation to fail if not met. When enabled, a unit with this directive will create a directory ending with .requires.
* **Alias=**: This directive allows the unit to be enabled under another name as well. Among other uses, this allows multiple providers of a function to be available, so that related units can look for any provider of the common aliased name.
* **Also=**: This directive allows units to be enabled or disabled as a set. Supporting units that should always be available when this unit is active can be listed here. They will be managed as a group for installation tasks.
* **DefaultInstance=**: For template units (covered later) which can produce unit instances with unpredictable names, this can be used as a fallback value for the name if an appropriate name is not provided.

### **Unit-Specific Section Directives**

Sandwiched between the previous two sections, you will likely find unit type-specific sections. Most unit types offer directives that only apply to their specific type. These are available within sections named after their type. We will cover those briefly here.

The device, target, snapshot, and scope unit types have no unit-specific directives, and thus have no associated sections for their type.

#### The [Service] Section

The [Service] section is used to provide configuration that is only applicable for services.

One of the basic things that should be specified within the [Service] section is the Type= of the service. This categorizes services by their process and daemonizing behavior. This is important because it tells systemd how to correctly manage the servie and find out its state.

The Type= directive can be one of the following:

* **simple**: The main process of the service is specified in the start line. This is the default if the Type=and Busname= directives are not set, but the ExecStart= is set. Any communication should be handled outside of the unit through a second unit of the appropriate type (like through a .socketunit if this unit must communicate using sockets).
* **forking**: This service type is used when the service forks a child process, exiting the parent process almost immediately. This tells systemd that the process is still running even though the parent exited.
* **oneshot**: This type indicates that the process will be short-lived and that systemd should wait for the process to exit before continuing on with other units. This is the default Type= and ExecStart= are not set. It is used for one-off tasks.
* **dbus**: This indicates that unit will take a name on the D-Bus bus. When this happens, systemd will continue to process the next unit.
* **notify**: This indicates that the service will issue a notification when it has finished starting up. The systemd process will wait for this to happen before proceeding to other units.
* **idle**: This indicates that the service will not be run until all jobs are dispatched.

Some additional directives may be needed when using certain service types. For instance:

* **RemainAfterExit=**: This directive is commonly used with the oneshot type. It indicates that the service should be considered active even after the process exits.
* **PIDFile=**: If the service type is marked as "forking", this directive is used to set the path of the file that should contain the process ID number of the main child that should be monitored.
* **BusName=**: This directive should be set to the D-Bus bus name that the service will attempt to acquire when using the "dbus" service type.
* **NotifyAccess=**: This specifies access to the socket that should be used to listen for notifications when the "notify" service type is selected This can be "none", "main", or "all. The default, "none", ignores all status messages. The "main" option will listen to messages from the main process and the "all" option will cause all members of the service's control group to be processed.

So far, we have discussed some pre-requisite information, but we haven't actually defined how to manage our services. The directives to do this are:

* **ExecStart=**: This specifies the full path and the arguments of the command to be executed to start the process. This may only be specified once (except for "oneshot" services). If the path to the command is preceded by a dash "-" character, non-zero exit statuses will be accepted without marking the unit activation as failed.
* **ExecStartPre=**: This can be used to provide additional commands that should be executed before the main process is started. This can be used multiple times. Again, commands must specify a full path and they can be preceded by "-" to indicate that the failure of the command will be tolerated.
* **ExecStartPost=**: This has the same exact qualities as ExecStartPre= except that it specifies commands that will be run after the main process is started.
* **ExecReload=**: This optional directive indicates the command necessary to reload the configuration of the service if available.
* **ExecStop=**: This indicates the command needed to stop the service. If this is not given, the process will be killed immediately when the service is stopped.
* **ExecStopPost=**: This can be used to specify commands to execute following the stop command.
* **RestartSec=**: If automatically restarting the service is enabled, this specifies the amount of time to wait before attempting to restart the service.
* **Restart=**: This indicates the circumstances under which systemd will attempt to automatically restart the service. This can be set to values like "always", "on-success", "on-failure", "on-abnormal", "on-abort", or "on-watchdog". These will trigger a restart according to the way that the service was stopped.
* **TimeoutSec=**: This configures the amount of time that systemd will wait when stopping or stopping the service before marking it as failed or forcefully killing it. You can set separate timeouts with TimeoutStartSec= and TimeoutStopSec= as well.

#### The [Socket] Section

Socket units are very common in systemd configurations because many services implement socket-based activation to provide better parallelization and flexibility. Each socket unit must have a matching service unit that will be activated when the socket receives activity.

By breaking socket control outside of the service itself, sockets can be initialized early and the associated services can often be started in parallel. By default, the socket name will attempt to start the service of the same name upon receiving a connection. When the service is initialized, the socket will be passed to it, allowing it to begin processing any buffered requests.

To specify the actual socket, these directives are common:

* **ListenStream=**: This defines an address for a stream socket which supports sequential, reliable communication. Services that use TCP should use this socket type.
* **ListenDatagram=**: This defines an address for a datagram socket which supports fast, unreliable communication packets. Services that use UDP should set this socket type.
* **ListenSequentialPacket=**: This defines an address for sequential, reliable communication with max length datagrams that preserves message boundaries. This is found most often for Unix sockets.
* **ListenFIFO**: Along with the other listening types, you can also specify a FIFO buffer instead of a socket.

There are more types of listening directives, but the ones above are the most common.

Other characteristics of the sockets can be controlled through additional directives:

* **Accept=**: This determines whether an additional instance of the service will be started for each connection. If set to false (the default), one instance will handle all connections.
* **SocketUser=**: With a Unix socket, specifies the owner of the socket. This will be the root user if left unset.
* **SocketGroup=**: With a Unix socket, specifies the group owner of the socket. This will be the root group if neither this or the above are set. If only the SocketUser= is set, systemd will try to find a matching group.
* **SocketMode=**: For Unix sockets or FIFO buffers, this sets the permissions on the created entity.
* **Service=**: If the service name does not match the .socket name, the service can be specified with this directive.

#### The [Mount] Section

Mount units allow for mount point management from within systemd. Mount points are named after the directory that they control, with a translation algorithm applied.

For example, the leading slash is removed, all other slashes are translated into dashes "-", and all dashes and unprintable characters are replaced with C-style escape codes. The result of this translation is used as the mount unit name. Mount units will have an implicit dependency on other mounts above it in the hierarchy.

Mount units are often translated directly from /etc/fstab files during the boot process. For the unit definitions automatically created and those that you wish to define in a unit file, the following directives are useful:

* **What=**: The absolute path to the resource that needs to be mounted.
* **Where=**: The absolute path of the mount point where the resource should be mounted. This should be the same as the unit file name, except using conventional filesystem notation.
* **Type=**: The filesystem type of the mount.
* **Options=**: Any mount options that need to be applied. This is a comma-separated list.
* **SloppyOptions=**: A boolean that determines whether the mount will fail if there is an unrecognized mount option.
* **DirectoryMode=**: If parent directories need to be created for the mount point, this determines the permission mode of these directories.
* **TimeoutSec=**: Configures the amount of time the system will wait until the mount operation is marked as failed.

#### The [Automount] Section

This unit allows an associated .mount unit to be automatically mounted at boot. As with the .mount unit, these units must be named after the translated mount point's path.

The [Automount] section is pretty simple, with only the following two options allowed:

* **Where=**: The absolute path of the automount point on the filesystem. This will match the filename except that it uses conventional path notation instead of the translation.
* **DirectoryMode=**: If the automount point or any parent directories need to be created, this will determine the permissions settings of those path components.

#### The [Swap] Section

Swap units are used to configure swap space on the system. The units must be named after the swap file or the swap device, using the same filesystem translation that was discussed above.

Like the mount options, the swap units can be automatically created from /etc/fstab entries, or can be configured through a dedicated unit file.

The [Swap] section of a unit file can contain the following directives for configuration:

* **What=**: The absolute path to the location of the swap space, whether this is a file or a device.
* **Priority=**: This takes an integer that indicates the priority of the swap being configured.
* **Options=**: Any options that are typically set in the /etc/fstab file can be set with this directive instead. A comma-separated list is used.
* **TimeoutSec=**: The amount of time that systemd waits for the swap to be activated before marking the operation as a failure.

#### The [Path] Section

A path unit defines a filesystem path that systmed can monitor for changes. Another unit must exist that will be be activated when certain activity is detected at the path location. Path activity is determined thorugh inotify events.

The [Path] section of a unit file can contain the following directives:

* **PathExists=**: This directive is used to check whether the path in question exists. If it does, the associated unit is activated.
* **PathExistsGlob=**: This is the same as the above, but supports file glob expressions for determining path existence.
* **PathChanged=**: This watches the path location for changes. The associated unit is activated if a change is detected when the watched file is closed.
* **PathModified=**: This watches for changes like the above directive, but it activates on file writes as well as when the file is closed.
* **DirectoryNotEmpty=**: This directive allows systemd to activate the associated unit when the directory is no longer empty.
* **Unit=**: This specifies the unit to activate when the path conditions specified above are met. If this is omitted, systemd will look for a .service file that shares the same base unit name as this unit.
* **MakeDirectory=**: This determines if systemd will create the directory structure of the path in question prior to watching.
* **DirectoryMode=**: If the above is enabled, this will set the permission mode of any path components that must be created.

#### The [Timer] Section

Timer units are used to schedule tasks to operate at a specific time or after a certain delay. This unit type replaces or supplements some of the functionality of the cron and at daemons. An associated unit must be provided which will be activated when the timer is reached.

The [Timer] section of a unit file can contain some of the following directives:

* **OnActiveSec=**: This directive allows the associated unit to be activated relative to the .timerunit's activation.
* **OnBootSec=**: This directive is used to specify the amount of time after the system is booted when the associated unit should be activated.
* **OnStartupSec=**: This directive is similar to the above timer, but in relation to when the systemdprocess itself was started.
* **OnUnitActiveSec=**: This sets a timer according to when the associated unit was last activated.
* **OnUnitInactiveSec=**: This sets the timer in relation to when the associated unit was last marked as inactive.
* **OnCalendar=**: This allows you to activate the associated unit by specifying an absolute instead of relative to an event.
* **AccuracySec=**: This unit is used to set the level of accuracy with which the timer should be adhered to. By default, the associated unit will be activated within one minute of the timer being reached. The value of this directive will determine the upper bounds on the window in which systemd schedules the activation to occur.
* **Unit=**: This directive is used to specify the unit that should be activated when the timer elapses. If unset, systemd will look for a .service unit with a name that matches this unit.
* **Persistent=**: If this is set, systemd will trigger the associated unit when the timer becomes active if it would have been triggered during the period in which the timer was inactive.
* **WakeSystem=**: Setting this directive allows you to wake a system from suspend if the timer is reached when in that state.

#### The [Slice] Section

The [Slice] section of a unit file actually does not have any .slice unit specific configuration. Instead, it can contain some resource management directives that are actually available to a number of the units listed above.

Some common directives in the [Slice] section, which may also be used in other units can be found in the systemd.resource-control man page. These are valid in the following unit-specific sections:

* [Slice]
* [Scope]
* [Service]
* [Socket]
* [Mount]
* [Swap]

## Creating Instance Units from Template Unit Files

We mentioned earlier in this guide the idea of template unit files being used to create multiple instances of units. In this section, we can go over this concept in more detail.

Template unit files are, in most ways, no different than regular unit files. However, these provide flexibility in configuring units by allowing certain parts of the file to utilize dynamic information that will be available at runtime.

### **Template and Instance Unit Names**

Template unit files can be identified because they contain an @ symbol after the base unit name and before the unit type suffix. A template unit file name may look like this:

example@.service

When an instance is created from a template, an instance identifier is placed between the @ symbol and the period signifying the start of the unit type. For example, the above template unit file could be used to create an instance unit that looks like this:

example@instance1.service

An instance file is usually created as a symbolic link to the template file, with the link name including the instance identifier. In this way, multiple links with unique identifiers can point back to a single template file. When managing an instance unit, systemd will look for a file with the exact instance name you specify on the command line to use. If it cannot find one, it will look for an associated template file.

### **Template Specifiers**

The power of template unit files is mainly seen through its ability to dynamically substitute appropriate information within the unit definition according to the operating environment. This is done by setting the directives in the template file as normal, but replacing certain values or parts of values with variable specifiers.

The following are some of the more common specifiers will be replaced when an instance unit is interpreted with the relevant information:

* **%n**: Anywhere where this appears in a template file, the full resulting unit name will be inserted.
* **%N**: This is the same as the above, but any escaping, such as those present in file path patterns, will be reversed.
* **%p**: This references the unit name prefix. This is the portion of the unit name that comes before the @ symbol.
* **%P**: This is the same as above, but with any escaping reversed.
* **%i**: This references the instance name, which is the identifier following the @ in the instance unit. This is one of the most commonly used specifiers because it will be guaranteed to be dynamic. The use of this identifier encourages the use of configuration significant identifiers. For example, the port that the service will be run at can be used as the instance identifier and the template can use this specifier to set up the port specification.
* **%I**: This specifier is the same as the above, but with any escaping reversed.
* **%f**: This will be replaced with the unescaped instance name or the prefix name, prepended with a /.
* **%c**: This will indicate the control group of the unit, with the standard parent hierarchy of /sys/fs/cgroup/ssytemd/ removed.
* **%u**: The name of the user configured to run the unit.
* **%U**: The same as above, but as a numeric UID instead of name.
* **%H**: The host name of the system that is running the unit.
* **%%**: This is used to insert a literal percentage sign.

By using the above identifiers in a template file, systemd will fill in the correct values when interpreting the template to create an instance unit.

## Conclusion

When working with systemd, understanding units and unit files can make administration simple. Unlike many other init systems, you do not have to know a scripting language to interpret the init files used to boot services or the system. The unit files use a fairly simple declarative syntax that allows you to see at a glance the purpose and effects of a unit upon activation.

Breaking functionality such as activation logic into separate units not only allows the internal systemdprocesses to optimize parallel initialization, it also keeps the configuration rather simple and allows you to modify and restart some units without tearing down and rebuilding their associated connections. Leveraging these abilities can give you more flexibility and power during administration.