Tomloader

v0.2.2

This is the manual for tomloader v0.2.2, which is an utility designed to facilitate the creation of multiple systemd unit files.

Table of Contents

1 Overview

The primary purpose of Tomloader is to simplify the managment of multiple systemd unit files that share configuration fields.

Systemd already provides native support for configuration reuse through `.conf` files inside drop-in directories (`.d/`) which can be hierarchically organized (for example, unit `A-B-C.unit` inherits all drop-in configurations listed in `A-B-.unit.d/` and `A-.unit.d/`). While useful, it present some limitations:

- inheritance follows a single linear hierarchy, preventing composition from multiple unrelated sources;
- `.conf` files cannot be templated;
- the final result heavily depends on file ordering inside the drop-in directories, making outcomes sensitive to file naming;
- dependencies between `.conf` files cannot be explicitly expressed.

Although symlinks may be used the first issue, they do no address the lack of parametrization. Additionally, ordering constraints remain a source of potential misconfiguration even with naming conventions like numeric prefixes.

Tomloader addresses these limitations with the introduction of groups. A *group* is a collection of systemd configuration fields which may declare dependencies on other groups. Despite `.conf` files in drop-in directories:

- multiple groups are not allowed to modify the same field, such conflicts are detected and will prevent unit generation unless resolved;
- groups may define *parameters* that may change the values set by configuration fields;
- group can declare dependencies on other groups.

Groups are loaded during the generation of a systemd unit. When a group is loaded, all its systemd configuration fields are imported into the unit. A loaded group can still be unloaded and all its fields reverted to their original stated. A group can be unloaded because another group providing similar functionalities is loaded and imported in the generated unit, or just because it is loaded as a dependency of another group but it is not needed.

It is not important the order the groups are loaded in the generated systemd unit: the resulting systemd unit will be the same as long as the same groups (with the same parameters) are loaded into.

There are two kinds of dependencies for a group:

- *load dependencies*: automatically included whenever the parent group is used;
- *remove dependencies*: explicitly excluded whenever the parent group is used, even if references elsewhere.

2 The `tomloader` command

The `tomloader` executable generates systemd unit files from `.kdl` configuration files. A generic invocation of the `tomloader` utility has the following syntax:

```
tomloader subcommand [options]... other subcommands... [args]...
```

All the functionalities of `tomloader` are implemented through *subcommands*, and at least one of them must be specified. Backward-incompatible changes are introduced through new subcommands, existing subcommands remain backward-compatible, except where changes are required for security or deprecation reasons.

Without any subcommand, only the following options are understood by `tomloader`:

`-h`
`--help`
: Print a list of currently defined subcommands. Equivalent to invoking `tomloader help`.

`-V`
`--version`
: Print the current version of `tomloader`.

2.1 The `help` subcommand

The `help` subcommand just prints a list of subcommands currently implemented.

2.2 The `sd-v0.2` subcommand

The `sd-v0.2` subcommand generates one or more systemd unit files from their respective unit configuration files. This is the general syntax of the `sd-v0.2` subcommand:

```
tomloader sd-v0.2 [options] <unit config files>...
```

where <*unit config files*> is a whitespace-separated list of files or paths pointing to unit configuration files.

The following options are understood by `tomloader sd-v0.2`:

`-h`
`--help`
: Print a short help on standard output, then exit.

`-V`
`--version`
: Print the current version of `tomloader sd-v0.2`.

`-t` *outdir*
`--target-directory` *outdir*
: Save all generated systemd unit files in *outdir*. This option is **mandatory**.

`-d` *srcdir*
`--directory` *srcdir*
: In addition to unit configuration files listed as <*unit config files*>, search unit configuration files stored in *srcdir* directory. This option can be specified multiple times. Only regular files are checked, it does not traverse subdirectories or follow symlinks.

2.3 The `inspect-v0.2` subcommand

The `inspect-v0.2` just prints on `stdout` all the load and remove dependencies for selected unit configuration files or all the groups defined in `${XDG_CONFIG_HOME}/tomloader/groups.kdl`.

The following options are understood by `tomloader inspect-v0.2`:

`-h`
`--help` Print a short help on standard output, then exit.

`-V`
`--version`
Print the current version of `tomloader inspect-v0.2`.

`-u` *CONF_UNIT_PATH*
`--unit` *CONF_UNIT_PATH*
Prints on `stdout` a list of load and remove dependencies of the unit configuration file with path *CONF_UNIT_PATH*. This option can be specified multiple times, and if you do not provide this option then the dependencies or all groups defined in `groups.kdl`.

2.4 The `sd-v0.1` subcommand

The `sd-v0.1` subcommand, like `sd-v0.2`, generates one or more systemd unit files from their respective unit configuration files. The main difference between `sd-v0.1` and `sd-v0.2` is that `sd-v0.1` only works with the v0.1 syntax for group and unit configuration file format, whereas `sd-v0.2` only works with the v0.2 syntax.

See Section 2.2 [The sd-v0.2 subcommand], page 2, for documentation of the new command.

3 Group configuration

Groups are defined in the file `${XDG_CONFIG_HOME}/tomloader/groups.kdl` formatted as a KDL configuration file. Each group is declared by using a `def-group` node:

```
def-group Group1 {
    sd {
        (section)Unit {
            (add) After "dbus.service" "pipewire.service"
            (add) Requisite "dbus.service" "pipewire.service"
            (reset) JoinsNamespaceOf
            (set) Description "Group1"
        }
    }
}
def-group Group2 {
    sd {
        (section) Unit {
            (set) Documentation "man:group(2)"
        }
        (section) Service {
            (set) Type "exec"
        }
    }
}
```

Any valid KDL string can be used as group name, in particular you can embed whitespaces if you surround the whole name of the group between double quotes '`"`'.

Each group may contain zero or more `sd` nodes containing rules for Systemd fields. Each `sd` node contains nodes with type `section` (enclosed in parenthesis) representing systemd sections (e.g. `Unit`, `Service`, `Slice`). Within each section, the following operations are supported as node types:

`set` assigns one or more values to a field;

`reset` clears the field;

`add` adds one or more values to the field.

The names of the nodes with type `section`, `set`, `reset`, `add` is the section/field name represented by such node.

Field values are internally represented as a list of strings. Just before generating the systemd unit file each list of strings is merged into a single:

- *whitespace-separated* ('` `') string (default behaviour);
- *colon-separated* ('`:`') string for `ExecSearchPath`;
- *comma-separated* ('`,`') string for `RootImageOptions`.

Note: The order of values specified in `set` and `add` is not preserved, therefore in the generated unit they may appear in a different order (which will still be deterministic). If the order of those elements must be preserved, then a single `set` operation with a single

string containing the ordered values is sufficient (Tomloader never adds double quotes '"' to string values unless they are already present inside the value).

3.1 Dependencies

Groups declare their load and remove dependencies as child nodes inside `pull` and `replace` nodes respectively. Each node representing a dependency must have the respective group name as node name and may optionally have `group` as node type. For example:

```
def-group Group3 {}
def-group "Group 4" {
    pull {
        (group)Group3
    }
}
def-group Group5 {
    pull {
        (group) Group3
        "Group 4"
    }
}
def-group Group6 {
    pull {
        "Group 4"
    }
    replace {
        Group3
    }
}
```

A group listed as load dependency can still be prevented to be loaded if another group lists the same group as a remove dependency. On the contrary, groups listed as remove dependencies cannot be included by any mean because it is not possible to revert a remove dependency. The only way to load a group declared as remove dependency is to prevent the group that specifies it as a remove dependency to be loaded.

A group may appear in both `pull` and `replace`. In this case, the group itself is excluded while its dependencies remain included. Dependencies are *transitive* by default:

1. load dependencies propagate both load and remove dependencies;
2. remove dependencies propagate only remove dependencies.

Transitive behaviour can be disabled with the `inherit=#false` property in `group` nodes:

```
def-group Group7 {
    pull {
        // Group4 is loaded as a pull dependency
        // Group3 is loaded as a replace dependency
        Group6
    }
}
```

```
def-group Group8 {
    pull {
        // No further groups are loaded as dependency
        Group6 inherit=#false
    }
}
```

You can also assign the string values `"true"`, `"false"` to `inherit` property in place of boolean values `#true`, `#false` (double quotes `"` are mandatory for `"true"`, `"false"` as specified by KDL specifications). For remove dependencies, the `inherit` also accepts the string `pulls` as value. With this property, all the transitive pull dependencies are loaded as replace dependencies.

```
def-group Group9 {
    replace {
        // Group3 and Group4 are loaded as replace dependencies
        Group6 inherit=pulls
    }
}
```

Groups can declare dependencies also inside a `merge` node the same way you declare them in `pull` or `replace` node. Groups specified inside a `merge` are loaded as remove dependencies but with the following differences from usual remove dependencies:

- transitive load dependencies are transitively loaded as load dependencies, unless `inherit=#false` or `inherit=deep` are specified;
- all the fields specifications are loaded in the current group as their own specifications that can still be modified in its own `sd` node, for example

```
def-group GroupA {
    sd {
        (section) Unit {
            (set) StopWhenUnneeded #true
            (set) RefuseManualStart #true
        }
    }
}
def-group GroupB {
    merge {
        GroupA
    }
    sd {
        (section) Unit {
            (reset) RefuseManualStart
        }
    }
    // Now GroupA is listed as a remove dependency, GroupB sets the
    // StopWhenUnneeded field in [Unit] to true when loaded but resets the
    // RefuseManualStart field.
}
```

The `inherit` property of a dependency specified inside a `merge` node accepts the special value `deep` other than `#true` and `#false`. For each dependency inside `merge` with `inherit=deep`:

- all load transitive dependencies are imported as remove dependencies instead;
- all the fields specified in these load transitive dependencies are loaded in the current group as its own specifications.

Loading multiple groups inside `merge` node may generate conflits when one or more fields are modified by different groups. Section Section 4.1 [Conflicts], page 8, explains how to manage and resolve conflicts.

3.2 Parameters and Arguments

A group may define one or more *parameters* through an additional argument of the `def-group` node with the total number of parameters:

```
def-group GroupA 2 {
    sd {
        (section) Service {
            (set) PIDFile "/run/${0}-pid"
            (set) ExecStartPre "/usr/bin/pre-${1}"
        }
    }
}
```

Parameters can be accessed in string values or dependency arguments through `${0}`, `${1}`, ..., `${N-1}` specifiers, which are replaced with their respective values when the unit is instantiated. In the previous example, loading `GroupA` with arguments `"P1"` and `P2` will set the value of `PIDFile` field to `/run/P1-pid` and `ExecStartPre` field to `/usr/bin/pre-P2`. You cannot apply these specifiers to field names.

Additionally, the special specifier `${$}` expands into a literal `$`.

Arguments are provided through a child `args` node:

```
def-group GroupB {
    pull {
        GroupA {
            args "pidname" "exec-sh"
        }
    }
}
def-group GroupC 1 {
    pull {
        GroupA {
            args "${0}-A" "${0}-B"
        }
    }
}
```

4 Unit configuration

To generate a systemd unit named `unitname.ext`, a corresponding KDL-formatted *unit configuration file* named `unitname.ext.kdl` must be provided.

You can use the `pull`, `replace` and `sd` nodes inside each unit configuration file to describe your systemd unit, see Chapter 3 [Group configuration], page 4, and Section 3.1 [Dependencies], page 5, for a description of these nodes.

Example:

```
pull {
    (group) Group1
    Group2
}
sd {
    (section) Unit {
        (set) Description "Unit"
    }
    (section) Service {
        (set) ExecStart "/usr/bin/bash"
    }
}
```

Just like the respective nodes inside group specification, you can use the `pull` node for listing groups to include, and the `replace` node for listing groups to exclude.

4.1 Conflicts

When two or more groups try to modify the same field in an incompatible way then a *conflict* is generated that will prevent the systemd unif file to be generated. Modifying a field in an incompatible way means any sequence of modifications such that the final result will depend on the order in which these operations are performed.

These are some common situations that issue a conflict:

- a group `set` a field and another group perform any modification to the same field;
- a group `add` a field and another group `reset` the same field.

Notice that `add` operations on the same field do not generate conflicts because ordering is not preserved by `add` operations, in this way Tomloader can rearrange the values in order to make the final result independent on the order of the `add` operations.

The following table shows all the possible outcomes when two nodes try to modify the came field. The '`C`' outcome means a conflict, the '`V`' outcome means no conflict will be generated, with an explaination on how they interact.

	`section`	`set`	`reset`	`add`
`section`	V, childs are merged			
`set`	C	C		
`reset`	C	C	V	
`add`	C	C	C	V, all the values are added

To resolve a conflict, you should specify inside the `sd` node of the respective unit configuration file the final value for the conflicting fields. Only `set` and `reset` operations can be used to resolve a conflict.

```
// groups.kdl
def-group GroupA {
    sd {
        (section) Service {
            (set) Type simple
        }
    }
}
def-group GroupB {
    sd {
        (section) Service {
            (set) Type exec
        }
    }
}

// unit.service.kdl
pull {
    GroupA
    GroupB
    // A conflict is issued because both GroupA and GroupB try to set
    // the Type field.
    //
    // This conflict is solved in the next sd node.
}
sd {
    (section) Service {
        (set) Type oneshot
    }
}
```

Index