mgmt Documentation

James Shubin

Nov 12, 2022

Contents:

CHAPTER 1

General documentation

1.1 Overview

The `mgmt` tool is a next generation config management prototype. It's not yet ready for production, but we hope to get there soon. Get involved today!

1.2 Project Description

The mgmt tool is a distributed, event driven, config management tool, that supports parallel execution, and librarification to be used as the management foundation in and for, new and existing software.

For more information, you may like to read some blog posts from the author:

- Next generation config mgmt
- Automatic edges in mgmt
- Automatic grouping in mgmt
- Automatic clustering in mgmt
- Remote execution in mgmt
- Send/Recv in mgmt
- Metaparameters in mgmt

There is also an introductory video available. Older videos and other material is available.

1.3 Setup

You'll probably want to read the *quick start guide* to get going.

1.4 Features

This section details the numerous features of mgmt and some caveats you might need to be aware of.

1.4.1 Autoedges

Automatic edges, or AutoEdges, is the mechanism in mgmt by which it will automatically create dependencies for you between resources. For example, since mgmt can discover which files are installed by a package it will automatically ensure that any file resource you declare that matches a file installed by your package resource will only be processed after the package is installed.

Controlling autoedges

Though autoedges is likely to be very helpful and avoid you having to declare all dependencies explicitly, there are cases where this behaviour is undesirable.

Some distributions allow package installations to automatically start the service they ship. This can be problematic in the case of packages like MySQL as there are configuration options that need to be set before MySQL is ever started for the first time (or you'll need to wipe the data directory). In order to handle this situation you can disable autoedges per resource and explicitly declare that you want `my.cnf` to be written to disk before the installation of the `mysql-server` package.

You can disable autoedges for a resource by setting the `autoedge` key on the meta attributes of that resource to `false`.

Blog post

You can read the introductory blog post about this topic here: https://purpleidea.com/blog/2016/03/14/automatic-edges-in-mgmt/

1.4.2 Autogrouping

Automatic grouping or AutoGroup is the mechanism in mgmt by which it will automatically group multiple resource vertices into a single one. This is particularly useful for grouping multiple package resources into a single resource, since the multiple installations can happen together in a single transaction, which saves a lot of time because package resources typically have a large fixed cost to running (downloading and verifying the package repo) and if they are grouped they share this fixed cost. This grouping feature can be used for other use cases too.

You can disable autogrouping for a resource by setting the `autogroup` key on the meta attributes of that resource to `false`.

Blog post

You can read the introductory blog post about this topic here: https://purpleidea.com/blog/2016/03/30/automatic-grouping-in-mgmt/

1.4.3 Automatic clustering

Automatic clustering is a feature by which mgmt automatically builds, scales, and manages the embedded etcd cluster which is compiled into mgmt itself. It is quite helpful for rapidly bootstrapping clusters and avoiding the extra work to setup etcd.

If you prefer to avoid this feature. you can always opt to use an existing etcd cluster that is managed separately from mgmt by pointing your mgmt agents at it with the `--seeds` variable.

Blog post

You can read the introductory blog post about this topic here: https://purpleidea.com/blog/2016/06/20/automatic-clustering-in-mgmt/

1.4.4 Remote ("agent-less") mode

Remote mode is a special mode that lets you kick off mgmt runs on one or more remote machines which are only accessible via SSH. In this mode the initiating host connects over SSH, copies over the `mgmt` binary, opens an SSH tunnel, and runs the remote program while simultaneously passing the etcd traffic back through the tunnel so that the initiators etcd cluster can be used to exchange resource data.

The interesting benefit of this architecture is that multiple hosts which can't connect directly use the initiator to pass the important traffic through to each other. Once the cluster has converged all the remote programs can shutdown leaving no residual agent.

This mode can also be useful for bootstrapping a new host where you'd like to have the service run continuously and as part of an mgmt cluster normally.

In particular, when combined with the `--converged-timeout` parameter, the entire set of running mgmt agents will need to all simultaneously converge for the group to exit. This is particularly useful for bootstrapping new clusters which need to exchange information that is only available at run time.

This existed in earlier versions of mgmt as a `--remote` option, but it has been removed and is being ported to a more powerful variant where you can remote execute via a `remote` resource.

Blog post

You can read the introductory blog post about this topic here: https://purpleidea.com/blog/2016/10/07/remote-execution-in-mgmt/

1.4.5 Puppet support

You can supply a Puppet manifest instead of creating the (YAML) graph manually. Puppet must be installed and in `mgmt`'s search path. You also need the ffrank-mgmtgraph Puppet module.

Invoke `mgmt` with the `--puppet` switch, which supports 3 variants:

1. Request the configuration from the Puppet Master (like `puppet agent` does)

   ```
   mgmt run puppet --puppet agent
   ```

2. Compile a local manifest file (like `puppet apply`)

   ```
   mgmt run puppet --puppet /path/to/my/manifest.pp
   ```

3. Compile an ad hoc manifest from the commandline (like `puppet apply -e`)

```
mgmt run puppet --puppet 'file { "/etc/ntp.conf":  ensure => file }'
```

For more details and caveats see *puppet-guide.md*.

Blog post

An introductory post on the Puppet support is on Felix's blog.

1.5 Reference

Please note that there are a number of undocumented options. For more information on these options, please view the source at: https://github.com/purpleidea/mgmt/. If you feel that a well used option needs documenting here, please patch it!

1.5.1 Overview of reference

- *Meta parameters*: List of available resource meta parameters.
- *Lang metadata file*: Lang metadata file format.
- *Graph definition file*: Main graph definition file.
- *Command line*: Command line parameters.
- *Compilation options*: Compilation options.

1.5.2 Meta parameters

These meta parameters are special parameters (or properties) which can apply to any resource. The usefulness of doing so will depend on the particular meta parameter and resource combination.

AutoEdge

Boolean. Should we generate auto edges for this resource?

AutoGroup

Boolean. Should we attempt to automatically group this resource with others?

Noop

Boolean. Should the Apply portion of the CheckApply method of the resource make any changes? Noop is a concatenation of no-operation.

Retry

Integer. The number of times to retry running the resource on error. Use -1 for infinite. This currently applies for both the Watch operation (which can fail) and for the CheckApply operation. While they could have separate values, I've decided to use the same ones for both until there's a proper reason to want to do something differently for the Watch errors.

Delay

Integer. Number of milliseconds to wait between retries. The same value is shared between the Watch and CheckApply retries. This currently applies for both the Watch operation (which can fail) and for the CheckApply operation. While they could have separate values, I've decided to use the same ones for both until there's a proper reason to want to do something differently for the Watch errors.

Poll

Integer. Number of seconds to wait between `CheckApply` checks. If this is greater than zero, then the standard event based `Watch` mechanism for this resource is replaced with a simple polling mechanism. In general, this is not recommended, unless you have a very good reason for doing so.

Please keep in mind that if you have a resource which changes every `I` seconds, and you poll it every `J` seconds, and you've asked for a converged timeout of `K` seconds, and `I <= J <= K`, then your graph will likely never converge.

When polling, the system detects that a resource is not converged if its `CheckApply` method returns false. This allows a resource which changes every `I` seconds, and which is polled every `J` seconds, and with a converged timeout of `K` seconds to still converge when `J <= K`, as long as `I > J || I > K`, which is another way of saying that if the resource finally settles down to give the graph enough time, it can probably converge.

Limit

Float. Maximum rate of `CheckApply` runs started per second. Useful to limit an especially *eventful* process from causing excessive checks to run. This defaults to `+Infinity` which adds no limiting. If you change this value, you will also need to change the `Burst` value to a non-zero value. Please see the rate package for more information.

Burst

Integer. Burst is the maximum number of runs which can happen without invoking the rate limiter as designated by the `Limit` value. If the `Limit` is not set to `+Infinity`, this must be a non-zero value. Please see the rate package for more information.

Sema

List of string ids. Sema is a P/V style counting semaphore which can be used to limit parallelism during the CheckApply phase of resource execution. Each resource can have `N` different semaphores which share a graph global namespace. Each semaphore has a maximum count associated with it. The default value of the size is 1 (one) if size is unspecified. Each string id is the unique id of the semaphore. If the id contains a trailing colon (:) followed by a positive integer, then that value is the max size for that semaphore. Valid semaphore id's include: `some_id`, `hello:42`, `not:smart:4` and `:13`. It is expected that the last bare example be only used by the engine to add a global semaphore.

Rewatch

Boolean. Rewatch specifies whether we re-run the Watch worker during a graph swap if it has errored. When doing a graph compare to swap the graphs, if this is true, and this particular worker has errored, then we'll remove it and add it back as a new vertex, thus causing it to run again. This is different from the `Retry` metaparam which applies during the normal execution. It is only when this is exhausted that we're in permanent worker failure, and only then can we rely on this metaparam.

Realize

Boolean. Realize ensures that the resource is guaranteed to converge at least once before a potential graph swap removes or changes it. This guarantee is useful for fast changing graphs, to ensure that the brief creation of a resource is seen. This guarantee does not prevent against the engine quitting normally, and it can't guarantee it if the resource is blocked because of a failed pre-requisite resource. *XXX: This is currently not implemented!*

Reverse

Boolean. Reverse is a property that some resources can implement that specifies that some "reverse" operation should happen when that resource "disappears". A disappearance happens when a resource is defined in one instance of the graph, and is gone in the subsequent one. This disappearance can happen if it was previously in an if statement that then becomes false.

This is helpful for building robust programs with the engine. The engine adds a "reversed" resource to that subsequent graph to accomplish the desired "reverse" mechanics. The specifics of what this entails is a property of the particular resource that is being "reversed".

It might be wise to combine the use of this meta parameter with the use of the `realize` meta parameter to ensure that your reversed resource actually runs at least once, if there's a chance that it might be gone for a while.

1.5.3 Lang metadata file

Any module *must* have a metadata file in its root. It must be named `metadata.yaml`, even if it's empty. You can specify zero or more values in yaml format which can change how your module behaves, and where the `mcl` language looks for code and other files. The most important top level keys are: `main`, `path`, `files`, and `license`.

Main

The `main` key points to the default entry point of your code. It must be a relative path if specified. If it's empty it defaults to `main.mcl`. It should generally not be changed. It is sometimes set to `main/main.mcl` if you'd like your modules code out of the root and into a child directory for cases where you don't plan on having a lot deeper imports relative to `main.mcl` and all those files would clutter things up.

Path

The `path` key specifies the modules import search directory to use for this module. You can specify this if you'd like to vendor something for your module. In general, if you use it, please use the convention: `path/`. If it's not specified, you will default to the parent modules directory.

Files

The `files` key specifies some additional files that will get included in your deploy. It defaults to `files/`.

License

The `license` key allows you to specify a license for the module. Please specify one so that everyone can enjoy your code! Use a "short license identifier", like `LGPLv3+`, or `MIT`. The former is a safe choice if you're not sure what to use.

1.5.4 Graph definition file

graph.yaml is the compiled graph definition file. The format is currently undocumented, but by looking through the examples/ you can probably figure out most of it, as it's fairly intuitive. It's not recommended that you use this, since it's preferable to write code in the mcl language front-end.

1.5.5 Command line

The main interface to the `mgmt` tool is the command line. For the most recent documentation, please run `mgmt --help`.

`--converged-timeout <seconds>`

Exit if the machine has converged for approximately this many seconds.

`--max-runtime <seconds>`

Exit when the agent has run for approximately this many seconds. This is not generally recommended, but may be useful for users who know what they're doing.

`--noop`

Globally force all resources into no-op mode. This also disables the export to etcd functionality, but does not disable resource collection, however all resources that are collected will have their individual noop settings set.

`--sema <size>`

Globally add a counting semaphore of this size to each resource in the graph. The semaphore will get given an id of `:size`. In other words if you specify a size of 42, you can expect a semaphore if named: `:42`. It is expected that consumers of the semaphore metaparameter always include a prefix to avoid a collision with this globally defined semaphore. The size value must be greater than zero at this time. The traditional non-parallel execution found in config management tools such as `Puppet` can be obtained with `--sema 1`.

`--allow-interactive`

Allow interactive prompting for SSH passwords if there is no authentication method that works.

`--ssh-priv-id-rsa`

Specify the path for finding SSH keys. This defaults to `~/.ssh/id_rsa`. To never use this method of authentication, set this to the empty string.

--cconns

The maximum number of concurrent remote ssh connections to run. This defaults to `0`, which means unlimited.

--no-caching

Don't allow remote caching of the remote execution binary. This will require the binary to be copied over for every remote execution, but it limits the likelihood that there is leftover information from the configuration process.

--prefix <path>

Specify a path to a custom working directory prefix. This directory will get created if it does not exist. This usually defaults to `/var/lib/mgmt/`. This can't be combined with the `--tmp-prefix` option. It can be combined with the `--allow-tmp-prefix` option.

--tmp-prefix

If this option is specified, a temporary prefix will be used instead of the default prefix. This can't be combined with the `--prefix` option.

--allow-tmp-prefix

If this option is specified, we will attempt to fall back to a temporary prefix if the primary prefix couldn't be created. This is useful for avoiding failures in environments where the primary prefix may or may not be available, but you'd like to try. The canonical example is when running `mgmt` with remote execution there might be a cached copy of the binary in the primary prefix, but if there's no binary available continue working in a temporary directory to avoid failure.

1.5.6 Compilation options

You can control some compilation variables by using environment variables.

Disable libvirt support

If you wish to compile mgmt without libvirt, you can use the following command:

```
GOTAGS=novirt make build
```

Disable augeas support

If you wish to compile mgmt without augeas support, you can use the following command:

```
GOTAGS=noaugeas make build
```

Disable docker support

If you wish to compile mgmt without docker support, you can use the following command:

```
GOTAGS=nodocker make build
```

Combining compile-time flags

You can combine multiple tags by using a space-separated list:

```
GOTAGS="noaugeas novirt nodocker" make build
```

1.6 Examples

For example configurations, please consult the examples/ directory in the git source repository. It is available from:

https://github.com/purpleidea/mgmt/tree/master/examples

1.6.1 Systemd:

See `misc/mgmt.service` for a sample systemd unit file. This unit file is part of the RPM.

To specify your custom options for `mgmt` on a systemd distro:

```
sudo mkdir -p /etc/systemd/system/mgmt.service.d/

cat > /etc/systemd/system/mgmt.service.d/env.conf <<EOF
# Environment variables:
MGMT_SEEDS=http://127.0.0.1:2379
MGMT_CONVERGED_TIMEOUT=-1
MGMT_MAX_RUNTIME=0

# Other CLI options if necessary.
#OPTS="--max-runtime=0"
EOF

sudo systemctl daemon-reload
```

1.7 Development

This is a project that I started in my free time in 2013. Development is driven by all of our collective patches! Dive right in, and start hacking! Please contact me if you'd like to invite me to speak about this at your event.

You can follow along on my technical blog.

To report any bugs, please file a ticket at: https://github.com/purpleidea/mgmt/issues.

1.8 Authors

Please see the AUTHORS file for more information.

- github
- @purpleidea
- https://purpleidea.com/

CHAPTER 2

Quick start guide

2.1 Introduction

This guide is intended for users and developers. If you're brand new to `mgmt`, it's probably a good idea to start by reading an introductory article about the engine and an introductory article about the language. There are other articles and videos available if you'd like to learn more or prefer different formats. Once you're familiar with the general idea, or if you prefer a hands-on approach, please start hacking...

2.2 Getting mgmt

You can either build `mgmt` from source, or you can download a pre-built release. There are also some distro repositories available, but they may not be up to date. A pre-built release is the fastest option if there's one that's available for your platform. If you are developing or testing a new patch to `mgmt`, or there is not a release available for your platform, then you'll have to build your own.

2.2.1 Downloading a pre-built release:

The latest releases can be found here. An alternate mirror is available here.

Make sure to verify the signatures of all packages before you use them. The signing key can be downloaded from https://purpleidea.com/contact/#pgp-key to verify the release.

If you've decided to install a pre-build release, you can skip to the *Running mgmt* section below!

2.2.2 Building a release:

You'll need some dependencies, including `golang`, and some associated tools.

Installing golang

- You need golang version 1.18 or greater installed.
  - To install on rpm style systems: `sudo dnf install golang`
  - To install on apt style systems: `sudo apt install golang`
  - To install on macOS systems install Homebrew and run: `brew install go`
- You can run `go version` to check the golang version.
- If your distro is too old, you may need to download a newer golang version.

Setting up golang

- You can skip this step, as your installation will default to using `~/go/`, but if you do not have a `GOPATH` yet and want one in a custom location, create one and export it:

```
mkdir $HOME/gopath
export GOPATH=$HOME/gopath
```

- You might also want to add the GOPATH to your `~/.bashrc` or `~/.profile`.
- For more information you can read the GOPATH documentation.

Getting the mgmt code and associated dependencies

- Download the `mgmt` code and switch to that directory:

```
git clone --recursive https://github.com/purpleidea/mgmt/ ~/mgmt/
cd ~/mgmt/
```

- Add `$GOPATH/bin` to `$PATH`

```
export PATH=$PATH:$GOPATH/bin
```

- Run `make deps` to install system and golang dependencies. Take a look at `misc/make-deps.sh` if you want to see the details of what it does.

Building mgmt

- Now run `make` to get a freshly built `mgmt` binary. If this succeeds, you can proceed to the *Running mgmt* section below!

2.2.3 Installing a distro release

Installation of `mgmt` from distribution packages currently needs improvement. They are not always up-to-date with git master and as such are not recommended. At the moment we have:

- COPR (currently dead)
- Arch (currently stale)

Please contribute more and help improve these! We'd especially like to see a Debian package!

2.3 Running mgmt

- Run `mgmt run --tmp-prefix lang examples/lang/hello0.mcl` to try out a very simple example! If you built it from source, you'll need to use `./mgmt` from the project directory.
- Look in that example file that you ran to see if you can figure out what it did! You can press `^C` to exit `mgmt`.
- Have fun hacking on our future technology and get involved to shape the project!

2.4 Examples

Please look in the examples/lang/ folder for some more examples!

CHAPTER 3

Resource guide

3.1 Overview

The `mgmt` tool has built-in resource primitives which make up the building blocks of any configuration. Each instance of a resource is mapped to a single vertex in the resource graph. This guide is meant to instruct developers on how to write a brand new resource. Since `mgmt` and the core resources are written in golang, some prior golang knowledge is assumed.

3.2 Theory

Resources in `mgmt` are similar to resources in other systems in that they are idempotent. Our resources are uniquely different in that they can detect when their state has changed, and as a result can run to revert or repair this change instantly. For some background on this design, please read the original article on the subject.

3.3 Resource Prerequisites

3.3.1 Imports

You'll need to import a few packages to make writing your resource easier. Here is the list:

```
        "github.com/purpleidea/mgmt/engine"
        "github.com/purpleidea/mgmt/engine/traits"
```

The `engine` package contains most of the interfaces and helper functions that you'll need to use. The `traits` package contains some base functionality which you can use to easily add functionality to your resource without needing to implement it from scratch.

3.3.2 Resource struct

Each resource will implement methods as pointer receivers on a resource struct. The naming convention for resources is that they end with a `Res` suffix.

The resource struct should include an anonymous reference to the `Base` trait. Other `traits` can be added to the resource to add additional functionality. They are discussed below.

You'll most likely want to store a reference to the `*Init` struct type as defined by the engine. This is data that the engine will provide to your resource on Init.

Lastly you should define the public fields that make up your resource API, as well as any private fields that you might want to use throughout your resource. Do *not* depend on global variables, since multiple copies of your resource could get instantiated.

You'll want to add struct tags based on the different frontends that you want your resources to be able to use. Some frontends can infer this information if it is not specified, but others cannot, and some might poorly infer if the struct name is ambiguous.

If you'd like your resource to be accessible by the `YAML` graph API (GAPI), then you'll need to include the appropriate YAML fields as shown below. This is used by the `Puppet` compiler as well, so make sure you include these struct tags if you want existing `Puppet` code to be able to run using the `mgmt` engine.

Example

```
type FooRes struct {
        traits.Base // add the base methods without re-implementation
        traits.Groupable
        traits.Refreshable

        init *engine.Init

        Whatever string `lang:"whatever" yaml:"whatever"` // you pick!
        Baz      bool   `lang:"baz" yaml:"baz"`           // something else

        something string // some private field
}
```

3.4 Resource API

To implement a resource in `mgmt` it must satisfy the `Res` interface. What follows are each of the method signatures and a description of each.

3.4.1 Default

```
Default() engine.Res
```

This returns a populated resource struct as a `Res`. It shouldn't populate any values which already get a good default as the respective golang zero value. In general it is preferable if the zero values make for the correct defaults. (This is to say, resources are designed to behave safely and intuitively when parameters take a zero value, whenever this is possible.)

Example

```
// Default returns some sensible defaults for this resource.
func (obj *FooRes) Default() engine.Res {
        return &FooRes{
                Answer: 42, // sometimes, defaults shouldn't be the zero value
        }
}
```

3.4.2 Validate

```
Validate() error
```

This method is used to validate if the populated resource struct is a valid representation of the resource kind. If it does not conform to the resource specifications, it should return an error. If you notice that this method is quite large, it might be an indication that you should reconsider the parameter list and interface to this resource. This method is called by the engine *before* `Init`. It can also be called occasionally after a Send/Recv operation to verify that the newly populated parameters are valid. Remember not to expect access to the outside world when using this.

Example

```
// Validate reports any problems with the struct definition.
func (obj *FooRes) Validate() error {
        if obj.Answer != 42 { // validate whatever you want
                return fmt.Errorf("expected an answer of 42")
        }
        return nil
}
```

3.4.3 Init

```
Init() error
```

This is called to initialize the resource. If something goes wrong, it should return an error. It should do any resource specific work such as initializing channels, sync primitives, or anything else that is relevant to your resource. If it is not need throughout, it might be preferable to do some initialization and tear down locally in either the Watch method or CheckApply method. The choice depends on your particular resource and making the best decision requires some experience with mgmt. If you are unsure, feel free to ask an existing `mgmt` contributor. During `Init`, the engine will pass your resource a struct containing some useful data and pointers. You should save a copy of this pointer since you will need to use it in other parts of your resource.

Example

```
// Init initializes the Foo resource.
func (obj *FooRes) Init(init *engine.Init) error
        obj.init = init // save for later

        // run the resource specific initialization, and error if anything fails
        if some_error {
```

(continues on next page)

(continued from previous page)

```
                return err // something went wrong!
        }
        return nil
}
```

This method is always called after `Validate` has run successfully, with the exception that we can't prevent a malicious or buggy `libmgmt` user to not run this. In other words, you should expect `Validate` to have run first, but you shouldn't allow `Init` to dangerously `rm -rf /$the_world` if your code only checks `$the_world` in `Validate`. Remember to always program safely!

3.4.4 Close

```
Close() error
```

This is called to cleanup after the resource. It is usually not necessary, but can be useful if you'd like to properly close a persistent connection that you opened in the `Init` method and were using throughout the resource. It is *not* the shutdown signal that tells the resource to exit. That happens in the Watch loop.

Example

```
// Close runs some cleanup code for this resource.
func (obj *FooRes) Close() error {
        err := obj.conn.Close() // close some internal connection
        obj.someMap = nil       // free up some large data structure from memory
        return err
}
```

You should probably check the return errors of your internal methods, and pass on an error if something went wrong.

3.4.5 CheckApply

```
CheckApply(apply bool) (checkOK bool, err error)
```

`CheckApply` is where the real *work* is done. Under normal circumstances, this function should check if the state of this resource is correct, and if so, it should return: `(true, nil)`. If the `apply` variable is set to `true`, then this means that we should then proceed to run the changes required to bring the resource into the correct state. If the `apply` variable is set to `false`, then the resource is operating in *noop* mode and *no operational changes* should be made!

After having executed the necessary operations to bring the resource back into the desired state, or after having detected that the state was incorrect, but that changes can't be made because `apply` is `false`, you should then return `(false, nil)`.

You must cause the resource to converge during a single execution of this function. If you cannot, then you must return an error! The exception to this rule is that if an external force changes the state of the resource while it is being remedied, it is possible to return from this function even though the resource isn't now converged. This is not a bug, as the resources `Watch` facility will detect the new change, ultimately resulting in a subsequent call to `CheckApply`.

Example

```
// CheckApply does the idempotent work of checking and applying resource state.
func (obj *FooRes) CheckApply(apply bool) (bool, error) {
        // check the state
        if state_is_okay { return true, nil } // done early! :)

        // state was bad

        if !apply { return false, nil } // don't apply, we're in noop mode

        if any_error { return false, err } // anytime there's an err!

        // do the apply!
        return false, nil // after success applying
}
```

The `CheckApply` function is called by the `mgmt` engine when it believes a call is necessary. Under certain conditions when a `Watch` call does not invalidate the state of the resource, and no refresh call was sent, its execution might be skipped. This is an engine optimization, and not a bug. It is mentioned here in the documentation in case you are confused as to why a debug message you've added to the code isn't always printed.

Paired execution

For many resources it is not uncommon to see `CheckApply` run twice in rapid succession. This is usually not a pathological occurrence, but rather a healthy pattern which is a consequence of the event system. When the state of the resource is incorrect, `CheckApply` will run to remedy the state. In response to having just changed the state, it is usually the case that this repair will trigger the `Watch` code! In response, a second `CheckApply` is triggered, which will likely find the state to now be correct.

Summary

- Anytime an error occurs during `CheckApply`, you should return `(false, err)`.
- If the state is correct and no changes are needed, return `(true, nil)`.
- You should only make changes to the system if `apply` is set to `true`.
- After checking the state and possibly applying the fix, return `(false, nil)`.
- Returning `(true, err)` is a programming error and can have a negative effect.

3.4.6 Watch

```
Watch() error
```

`Watch` is a main loop that runs and sends messages when it detects that the state of the resource might have changed. To send a message you should write to the input event channel using the `Event` helper method. The Watch function should run continuously until a shutdown message is received. If at any time something goes wrong, you should return an error, and the `mgmt` engine will handle possibly restarting the main loop based on the `retry` meta parameter.

It is better to send an event notification which turns out to be spurious, than to miss a possible event. Resources which can miss events are incorrect and need to be re-engineered so that this isn't the case. If you have an idea for a resource which would fit this criteria, but you can't find a solution, please contact the `mgmt` maintainers so that this problem can be investigated and a possible system level engineering fix can be found.

You may have trouble deciding how much resource state checking should happen in the `Watch` loop versus deferring it all to the `CheckApply` method. You may want to put some simple fast path checking in `Watch` to avoid generating obviously spurious events, but in general it's best to keep the `Watch` method as simple as possible. Contact the `mgmt` maintainers if you're not sure.

If the resource is activated in `polling` mode, the `Watch` method will not get executed. As a result, the resource must still work even if the main loop is not running.

Select

The lifetime of most resources `Watch` method should be spent in an infinite loop that is bounded by a `select` call. The `select` call is the point where our method hands back control to the engine (and the kernel) so that we can sleep until something of interest wakes us up. In this loop we must wait until we get a shutdown event from the engine via the `<-obj.init.Done` channel, which closes when we'd like to shut everything down. At this point you should cleanup, and let `Watch` close.

Events

If the `<-obj.init.Done` channel closes, we should shutdown our resource. When When we want to send an event, we use the `Event` helper function. This automatically marks the resource state as `dirty`. If you're unsure, it's not harmful to send the event. This will ultimately cause `CheckApply` to run. This method can block if the resource is being paused.

Startup

Once the `Watch` function has finished starting up successfully, it is important to generate one event to notify the `mgmt` engine that we're now listening successfully, so that it can run an initial `CheckApply` to ensure we're safely tracking a healthy state and that we didn't miss anything when `Watch` was down or from before `mgmt` was running. You must do this by calling the `obj.init.Running` method.

Converged

The engine might be asked to shutdown when the entire state of the system has not seen any changes for some duration of time. The engine can determine this automatically, but each resource can block this if it is absolutely necessary. If you need this functionality, please contact one of the maintainers and ask about adding this feature and improving these docs right here.

This particular facility is most likely not required for most resources. It may prove to be useful if a resource wants to start off a long operation, but avoid sending out erroneous `Event` messages to keep things alive until it finishes.

Example

```
// Watch is the listener and main loop for this resource.
func (obj *FooRes) Watch() error {
        // setup the Foo resource
        var err error
        if err, obj.foo = OpenFoo(); err != nil {
                return err // we couldn't startup
        }
        defer obj.whatever.CloseFoo() // shutdown our Foo
```

(continues on next page)

(continued from previous page)

```
        // notify engine that we're running
        obj.init.Running() // when started, notify engine that we're running

        var send = false // send event?
        for {
                select {
                // the actual events!
                case event := <-obj.foo.Events:
                        if is_an_event {
                                send = true
                        }

                // event errors
                case err := <-obj.foo.Errors:
                        return err // will cause a retry or permanent failure

                case <-obj.init.Done: // signal for shutdown request
                        return nil
                }

                // do all our event sending all together to avoid duplicate msgs
                if send {
                        send = false
                        obj.init.Event()
                }
        }
}
```

Summary

- Remember to call `Running` when the `Watch` is running successfully.
- Remember to process internal events and shutdown promptly if asked to.
- Ensure the design of your resource is well thought out.
- Have a look at the existing resources for a rough idea of how this all works.

3.4.7 Cmp

```
Cmp(engine.Res) error
```

Each resource must have a `Cmp` method. It is an abbreviation for `Compare`. It takes as input another resource and must return whether they are identical or not. This is used for identifying if an existing resource can be used in place of a new one with a similar set of parameters. In particular, when switching from one graph to a new (possibly identical) graph, this avoids recomputing the state for resources which don't change or that are sufficiently similar that they don't need to be swapped out.

In general if all the resource properties are identical, then they usually don't need to be changed. On occasion, not all of them need to be compared, in particular if they store some generated state, or if they aren't significant in some way.

If the resource is identical, then you should return `nil`. If it is not, then you should return a short error message which gives the reason it differs.

Example

```
// Cmp compares two resources and returns if they are equivalent.
func (obj *FooRes) Cmp(r engine.Res) error {
        // we can only compare FooRes to others of the same resource kind
        res, ok := r.(*FooRes)
        if !ok {
                return fmt.Errorf("not a %s", obj.Kind())
        }

        if obj.Whatever != res.Whatever {
                return fmt.Errorf("the Whatever param differs")
        }
        if obj.Flag != res.Flag {
                return fmt.Errorf("the Flag param differs")
        }

        return nil // they must match!
}
```

3.5 Traits

Resources can have different `traits`, which means they can be extended to have additional functionality or special properties. Those special properties are usually added by extending your resource so that it is compatible with additional interface that contain the `Res` interface. Each of these interfaces represents the additional functionality. Since in most cases this requires some common boilerplate, you can usually get some or most of the functionality by embedding the correct trait struct anonymously in your struct. This is shown in the struct example above. You'll always want to include the `Base` trait in all resources. This provides some basics which you'll always need.

What follows are a list of available traits.

3.5.1 Refreshable

Some resources may choose to support receiving refresh notifications. In general these should be avoided if possible, but nevertheless, they do make sense in certain situations. Resources that support these need to verify if one was sent during the CheckApply phase of execution. This is accomplished by calling the `obj.init.Refresh() bool` method, and inspecting the return value. This is only necessary if you plan to perform a refresh action. Refresh actions should still respect the `apply` variable, and no system changes should be made if it is `false`. Refresh notifications are generated by any resource when an action is applied by that resource and are transmitted through graph edges which have enabled their propagation. Resources that currently perform some refresh action include `svc`, `timer`, and `password`.

It is very important that you include the `traits.Refreshable` struct in your resource. If you do not include this, then calling `obj.init.Refresh` may trigger a panic. This is programmer error.

3.5.2 Edgeable

Edgeable is a trait that allows your resource to automatically connect itself to other resources that use this trait to add edge dependencies between the two. An older blog post on this topic is available.

After you've included this trait, you'll need to implement two methods on your resource.

UIDs

```
UIDs() []engine.ResUID
```

The `UIDs` method returns a list of `ResUID` interfaces that represent the particular resource uniquely. This is used with the AutoEdges API to determine if another resource can match a dependency to this one.

AutoEdges

```
AutoEdges() (engine.AutoEdge, error)
```

This returns a struct that implements the `AutoEdge` interface. This struct is used to match other resources that might be relevant dependencies for this resource.

3.5.3 Groupable

Groupable is a trait that can allow your resource automatically group itself to other resources. Doing so can reduce the resource or runtime burden on the engine, and improve performance in some scenarios. An older blog post on this topic is available.

3.5.4 Sendable

Sendable is a trait that allows your resource to send values through the graph edges to another resource. These values are produced during `CheckApply`. They can be sent to any resource that has an appropriate parameter and that has the `Recvable` trait. You can read more about this in the Send/Recv section below.

3.5.5 Recvable

Recvable is a trait that allows your resource to receive values through the graph edges from another resource. These values are consumed during the `CheckApply` phase, and can be detected there as well. They can be received from any resource that has an appropriate value and that has the `Sendable` trait. You can read more about this in the Send/Recv section below.

3.5.6 Collectable

This is currently a stub and will be updated once the DSL is further along.

3.6 Resource Initialization

During the resource initialization in `Init`, the engine will pass in a struct containing a bunch of data and methods. What follows is a description of each one and how it is used.

3.6.1 Program

Program is a string containing the name of the program. Very few resources need this.

3.6.2 Hostname

Hostname is the uuid for the host. It will be occasionally useful in some resources. It is preferable if you can avoid depending on this. It is possible that in the future this will be a channel which changes if the local hostname changes.

3.6.3 Running

Running must be called after your watches are all started and ready. It is only called from within `Watch`. It is used to notify the engine that you're now ready to detect changes.

3.6.4 Event

Event sends an event notifying the engine of a possible state change. It is only called from within `Watch`.

3.6.5 Done

Done is a channel that closes when the engine wants us to shutdown. It is only called from within `Watch`.

3.6.6 Refresh

Refresh returns whether the resource received a notification. This flag can be used to tell a `svc` to reload, or to perform some state change that wouldn't otherwise be noticed by inspection alone. You must implement the `Refreshable` trait for this to work. It is only called from within `CheckApply`.

3.6.7 Send

Send exposes some variables you wish to send via the `Send/Recv` mechanism. You must implement the `Sendable` trait for this to work. It is only called from within `CheckApply`.

3.6.8 Recv

Recv provides a map of variables which were sent to this resource via the `Send/Recv` mechanism. You must implement the `Recvable` trait for this to work. It is only called from within `CheckApply`.

3.6.9 World

World provides a connection to the outside world. This is most often used for communicating with the distributed database. It can be used in `Init`, `CheckApply` and `Watch`. Use with discretion and understanding of the internals if needed in `Close`.

3.6.10 VarDir

VarDir is a facility for local storage. It is used to return a path to a directory which may be used for temporary storage. It should be cleaned up on resource `Close` if the resource would like to delete the contents. The resource should not assume that the initial directory is empty, and it should be cleaned on `Init` if that is a requirement.

3.6.11 Debug

Debug signals whether we are running in debugging mode. In this case, we might want to log additional messages.

3.6.12 Logf

Logf is a logging facility which will correctly namespace any messages which you wish to pass on. You should use this instead of the log package directly for production quality resources.

3.7 Further considerations

There is some additional information that any resource writer will need to know. Each issue is listed separately below!

3.7.1 Resource registration

All resources must be registered with the engine so that they can be found. This also ensures they can be encoded and decoded. Make sure to include the following code snippet for this to work.

```
func init() { // special golang method that runs once
        // set your resource kind and struct here (the kind must be lower case)
        engine.RegisterResource("foo", func() engine.Res { return &FooRes{} })
}
```

3.7.2 YAML Unmarshalling

To support YAML unmarshalling for your resource, you must implement an additional method. It is recommended if you want to use your resource with the `Puppet` compiler.

```
UnmarshalYAML(unmarshal func(interface{}) error) error // optional
```

This is optional, but recommended for any resource that will have a YAML accessible struct. It is not required because to do so would mean that third-party or custom resources (such as those someone writes to use with `libmgmt`) would have to implement this needlessly.

The signature intentionally matches what is required to satisfy the `go-yaml` Unmarshaler interface.

Example

```
// UnmarshalYAML is the custom unmarshal handler for this struct. It is
// primarily useful for setting the defaults.
func (obj *FooRes) UnmarshalYAML(unmarshal func(interface{}) error) error {
        type rawRes FooRes // indirection to avoid infinite recursion

        def := obj.Default()     // get the default
        res, ok := def.(*FooRes) // put in the right format
        if !ok {
                return fmt.Errorf("could not convert to FooRes")
        }
        raw := rawRes(*res) // convert; the defaults go here
```

(continues on next page)

(continued from previous page)

```
        if err := unmarshal(&raw); err != nil {
                return err
        }

        *obj = FooRes(raw) // restore from indirection with type conversion!
        return nil
}
```

3.8 Send/Recv

In `mgmt` there is a novel concept called *Send/Recv*. For some background, please read the introductory article. When using this feature, the engine will automatically send the user specified value to the intended destination without requiring much resource specific code. Any time that one of the destination values is changed, the engine automatically marks the resource state as `dirty`. To detect if a particular value was received, and if it changed (during this invocation of `CheckApply`) from the previous value, you can query the `obj.init.Recv()` method. It will contain a `map` of all the keys which can be received on, and the value has a `Changed` property which will indicate whether the value was updated on this particular `CheckApply` invocation. The type of the sending key must match that of the receiving one. This can *only* be done inside of the `CheckApply` function!

```
// inside CheckApply, probably near the top
if val, exists := obj.init.Recv()["SomeKey"]; exists {
        obj.init.Logf("the SomeKey param was sent to us from: %s.%s", val.Res, val.
↪Key)
        if val.Changed {
                obj.init.Logf("the SomeKey param was just updated!")
                // you may want to invalidate some local cache
        }
}
```

The specifics of resource sending are not currently documented. Please send a patch here!

3.9 Composite resources

Composite resources are resources which embed one or more existing resources. This is useful to prevent code duplication in higher level resource scenarios. The best example of this technique can be seen in the `nspawn` resource which can be seen to partially embed a `svc` resource, but without its `Watch`. Unfortunately no further documentation about this subject has been written. To expand this section, please send a patch! Please contact us if you'd like to work on a resource that uses this feature, or to add it to an existing one!

3.10 Frequently asked questions

(Send your questions as a patch to this FAQ! I'll review it, merge it, and respond by commit with the answer.)

3.10.1 Can I write resources in a different language?

Currently `golang` is the only supported language for built-in resources. We might consider allowing external resources to be imported in the future. This will likely require a language that can expose a C-like API, such as `python`

or `ruby`. Custom `golang` resources are already possible when using mgmt as a lib. Higher level resource collections will be possible once the `mgmt` DSL is ready.

3.10.2 Why does the resource API have `CheckApply` instead of two separate methods?

In an early version we actually had both "parts" as separate methods, namely: `StateOK` (Check) and `Apply`, but the decision was made to merge the two into a single method. There are two reasons for this:

1. Many situations would involve the engine running both `Check` and `Apply`. If the resource needed to share some state (for efficiency purposes) between the two calls, this is much more difficult. A common example is that a resource might want to open a connection to `dbus` or `http` to do resource state testing and applying. If the methods are combined, there's no need to open and close them twice. A counter argument might be that you could open the connection in `Init`, and close it in `Close`, however you might not want that open for the full lifetime of the resource if you only change state occasionally.
2. Suppose you came up with a really good reason why you wanted the two methods to be separate. It turns out that the current `CheckApply` can wrap this easily. It would look approximately like this:

```
func (obj *FooRes) CheckApply(apply bool) (bool, error) {
        // my private split implementation of check and apply
        if c, err := obj.check(); err != nil {
                return false, err // we errored
        } else if c {
                return true, nil // state was good!
        }

        if !apply {
                return false, nil // state needs fixing, but apply is false
        }

        err := obj.apply() // errors if failure or unable to apply

        return false, err // always return false, with an optional error
}
```

Feel free to use this pattern if you're convinced it's necessary. Alternatively, if you think I got the `Res` API wrong and you have an improvement, please let us know!

3.10.3 Why do resources have both a `Cmp` method and an `IFF` (on the UID) method?

The `Cmp()` methods are for determining if two resources are effectively the same, which is used to make graph change delta's efficient. This is when we want to change from the current running graph to a new graph, but preserve the common vertices. Since we want to make this process efficient, we only update the parts that are different, and leave everything else alone. This `Cmp()` method can tell us if two resources are the same. In case it is not obvious, `cmp` is an abbrev. for compare.

The `IFF()` method is part of the whole UID system, which is for discerning if a resource meets the requirements another expects for an automatic edge. This is because the automatic edge system assumes a unified UID pattern to test for equality. In the future it might be helpful or sane to merge the two similar comparison functions although for now they are separate because they are actually answer different questions.

3.10.4 What new resource primitives need writing?

There are still many ideas for new resources that haven't been written yet. If you'd like to contribute one, please contact us and tell us about your idea!

3.10.5 Is the resource API stable? Does it ever change?

Since we are pre 1.0, the resource API is not guaranteed to be stable, however it is not expected to change significantly. The last major change kept the core functionality nearly identical, simplified the implementation of all the resources, and took about five to ten minutes to port each resource to the new API. The fundamental logic and behaviour behind the resource API has not changed since it was initially introduced.

3.10.6 Where can I find more information about mgmt?

Additional blog posts, videos and other material is available!.

3.11 Suggestions

If you have any ideas for API changes or other improvements to resource writing, please let us know! We're still pre 1.0 and pre 0.1 and happy to break API in order to get it right!

CHAPTER 4

Prometheus support

Mgmt comes with a built-in prometheus support. It is disabled by default, and can be enabled with the `--prometheus` command line switch.

By default, the prometheus instance will listen on `127.0.0.1:9233`. You can change this setting by using the `--prometheus-listen` cli option:

To have mgmt prometheus bind interface on 0.0.0.0:45001, use: `./mgmt r --prometheus --prometheus-listen :45001`

4.1 Metrics

Mgmt exposes three kinds of resources: *go* metrics, *etcd* metrics and *mgmt* metrics.

4.1.1 go metrics

We use the prometheus go_collector to expose go metrics. Those metrics are mainly useful for debugging and perf testing.

4.1.2 etcd metrics

mgmt exposes etcd metrics. Read more in the upstream documentation

4.1.3 mgmt metrics

Here is a list of the metrics we provide:

- `mgmt_resources_total`: The number of resources that mgmt is managing
- `mgmt_checkapply_total`: The number of CheckApply's that mgmt has run

- `mgmt_failures_total`: The number of resources that have failed
- `mgmt_failures`: The number of resources that have failed
- `mgmt_graph_start_time_seconds`: Start time of the current graph since unix epoch in seconds

For each metric, you will get some extra labels:

- `kind`: The kind of mgmt resource

For `mgmt_checkapply_total`, those extra labels are set:

- `eventful`: "true" or "false", if the CheckApply triggered some changes
- `errorful`: "true" or "false", if the CheckApply reported an error
- `apply`: "true" or "false", if the CheckApply ran in apply or noop mode

4.2 Alerting

You can use prometheus to alert you upon changes or failures. We do not provide such templates yet, but we plan to provide some examples in this repository. Patches welcome!

4.3 Grafana

We do not have grafana dashboards yet. Patches welcome!

4.4 External resources

- prometheus website
- prometheus documentation
- prometheus best practices regarding metrics naming
- grafana website

CHAPTER 5

Puppet guide

mgmt can use Puppet as its source for the configuration graph. This document goes into detail on how this works, and lists some pitfalls and limitations.

For basic instructions on how to use the Puppet support, see the main documentation.

5.1 Prerequisites

You need Puppet installed in your system. It is not important how you get it. On the most common Linux distributions, you can use packages from the OS maintainer, or upstream Puppet repositories. An alternative that will also work on OSX is the puppet Ruby gem. It also has the advantage that you can install any desired version in your home directory or any other location.

Any release of Puppet's 3.x and 4.x series should be suitable for use with mgmt. Most importantly, make sure to install the ffrank-mgmtgraph Puppet module (referred to below as "the translator module").

```
puppet module install ffrank-mgmtgraph
```

Please note that the module is not required on your Puppet master (if you use a master/agent setup). It's needed on the machine that runs mgmt. You can install the module on the master anyway, so that it gets distributed to your agents through Puppet's pluginsync mechanism.

5.1.1 Testing the Puppet side

The following command should run successfully and print a YAML hash on your terminal:

```
puppet mgmtgraph print --code 'file { "/tmp/mgmt-test": ensure => present }'
```

You can use this CLI to test any manifests before handing them straight to mgmt.

5.2 Writing a suitable manifest

5.2.1 Unsupported attributes

mgmt inherited its resource module from Puppet, so by and large, it's quite possible to express mgmt graphs in terms of Puppet manifests. However, there isn't (and likely never will be) full feature parity between the respective resource types. In consequence, a manifest can have semantics that cannot be transferred to mgmt.

For example, at the time of writing this, the file type in mgmt had no notion of permissions (the file mode) yet. This lead to the following warning (among others that will be discussed below):

```
$ puppet mgmtgraph print --code 'file { "/tmp/foo": mode => "0600" }'
Warning: cannot translate: File[/tmp/foo] { mode => "600" } (attribute is ignored)
```

This is a heads-up for the user, because the resulting mgmt graph will in fact not pass this information to the /tmp/foo file resource, and mgmt will ignore this file's permissions. Including such attributes in manifests that are written expressly for mgmt is not sensible and should be avoided.

5.2.2 Unsupported resources

Puppet has a fairly large number of built-in types, and countless more are available through modules. It's unlikely that all of them will eventually receive native counterparts in mgmt.

When encountering an unknown resource, the translator module will replace it with an exec resource in its output. This resource will run the equivalent of a puppet resource command to make Puppet apply the original resource itself. This has quite abysmal performance, because processing such a resource requires the forking of at least one Puppet process (two if it is found to be out of sync). This comes with considerable overhead. On most systems, starting up any Puppet command takes several seconds. Compared to the split second that the actual work usually takes, this overhead can amount to several orders of magnitude.

Avoid Puppet types that mgmt does not implement (yet).

5.2.3 Avoiding common warnings

Many resource parameters in Puppet take default values. For the most part, the translator module just ignores them. However, there are cases in which Puppet will default to convenient behavior that mgmt cannot quite replicate. For example, translating a plain file resource will lead to a warning message:

```
$ puppet mgmtgraph print --code 'file { "/tmp/mgmt-test": }'
Warning: File[/tmp/mgmt-test] uses the 'puppet' file bucket, which mgmt cannot do.␣
↪There will be no backup copies!
```

The reason is that per default, Puppet assumes the following parameter value (among others)

```
file { "/tmp/mgmt-test":
        backup => 'puppet',
}
```

To avoid this, specify the parameter explicitly:

```
puppet mgmtgraph print --code 'file { "/tmp/mgmt-test": backup => false }'
```

This is tedious in a more complex manifest. A good simplification is the following resource default anywhere on the top scope of your manifest:

```
File { backup => false }
```

If you encounter similar warnings from other types and/or parameters, use the same approach to silence them if possible.

5.3 Configuring Puppet

Since `mgmt` uses an actual Puppet CLI behind the scenes, you might need to tweak some of Puppet's runtime options in order to make it do what you want. Reasons for this could be among the following:

- You use the `--puppet agent` variant and need to configure `servername`, `certname` and other master/agent-related options.
- You don't want runtime information to end up in the `vardir` that is used by your regular `puppet agent`.
- You install specific Puppet modules for `mgmt` in a non-standard location.

`mgmt` exposes only one Puppet option in order to allow you to control all of them, through its `--puppet-conf` option. It allows you to specify which `puppet.conf` file should be used during translation.

```
mgmt run puppet --puppet /opt/my-manifest.pp --puppet-conf /etc/mgmt/puppet.conf
```

Within this file, you can just specify any needed options in the `[main]` section:

```
[main]
server=mgmt-master.example.net
vardir=/var/lib/mgmt/puppet
```

5.4 Caveats

Please see the README of the translator module for the current state of supported and unsupported language features.

You should probably make sure to always use the latest release of both `ffrank-mgmtgraph` and `ffrank-yamlresource` (the latter is getting pulled in as a dependency of the former).

5.5 Using Puppet in conjunction with the mcl lang

The graph that Puppet generates for `mgmt` can be united with a graph that is created from native `mgmt` code in its mcl language. This is useful when you are in the process of replacing Puppet with mgmt. You can translate your custom modules into mgmt's language one by one, and let mgmt run the current mix.

Instead of the usual `--puppet-conf` flag and argv for `puppet` and `mcl` input, you need to use alternative flags to make this work:

- `--lp-lang` to specify the mcl input
- `--lp-puppet` to specify the puppet input
- `--lp-puppet-conf` to point to the optional puppet.conf file

`mgmt` will derive a graph that contains all edges and vertices from both inputs. You essentially get two unrelated subgraphs that run in parallel. To form edges between these subgraphs, you have to define special vertices that will be merged. This works through a hard-coded naming scheme.

5.5.1 Mixed graph example 1 - No merges

```
# lang
file "/tmp/mgmt_dir/" { state => "present" }
file "/tmp/mgmt_dir/a" { state => "present" }
```

```
# puppet
file { "/tmp/puppet_dir": ensure => "directory" }
file { "/tmp/puppet_dir/a": ensure => "file" }
```

These very simple inputs (including implicit edges from directory to respective file) result in two subgraphs that do not relate.

```
File[/tmp/mgmt_dir/] -> File[/tmp/mgmt_dir/a]

File[/tmp/puppet_dir] -> File[/tmp/puppet_dir/a]
```

5.5.2 Mixed graph example 2 - Merged vertex

In order to have merged vertices in the resulting graph, you will need to include special resources and classes in the respective input code.

- On the lang side, add `noop` resources with names starting in `puppet_`.
- On the Puppet side, add **empty** classes with names starting in `mgmt_`.

```
# lang
noop "puppet_handover_to_mgmt" {}
file "/tmp/mgmt_dir/" { state => "present" }
file "/tmp/mgmt_dir/a" { state => "present" }

Noop["puppet_handover_to_mgmt"] -> File["/tmp/mgmt_dir/"]
```

```
# puppet
class mgmt_handover_to_mgmt {}
include mgmt_handover_to_mgmt

file { "/tmp/puppet_dir": ensure => "directory" }
file { "/tmp/puppet_dir/a": ensure => "file" }

File["/tmp/puppet_dir/a"] -> Class["mgmt_handover_to_mgmt"]
```

The new `noop` resource is merged with the new class, resulting in the following graph:

```
File[/tmp/puppet_dir] -> File[/tmp/puppet_dir/a]
                                    |
                                    V
                 Noop[handover_to_mgmt]
                            |
                            V
         File[/tmp/mgmt_dir/] -> File[/tmp/mgmt_dir/a]
```

You put all your ducks in a row, and the resources from the Puppet input run before those from the mcl input.

Note: The names of the `noop` and the class must be identical after the respective prefix. The common part (here, `handover_to_mgmt`) becomes the name of the merged resource.

5.6 Mixed graph example 3 - Multiple merges

In most scenarios, it will not be possible to define a single handover point like in the previous example. For example, if some Puppet resources need to run in between two stages of native resources, you need at least two merged vertices:

```
# lang
noop "puppet_handover" {}
noop "puppet_handback" {}
file "/tmp/mgmt_dir/" { state => "present" }
file "/tmp/mgmt_dir/a" { state => "present" }
file "/tmp/mgmt_dir/puppet_subtree/state-file" { state => "present" }

File["/tmp/mgmt_dir/"] -> Noop["puppet_handover"]
Noop["puppet_handback"] -> File["/tmp/mgmt_dir/puppet_subtree/state-file"]
```

```
# puppet
class mgmt_handover {}
class mgmt_handback {}

include mgmt_handover, mgmt_handback

class important_stuff {
        file { "/tmp/mgmt_dir/puppet_subtree":
                ensure => "directory"
        }
        # ...
}

Class["mgmt_handover"] -> Class["important_stuff"] -> Class["mgmt_handback"]
```

The resulting graph looks roughly like this:

```
File[/tmp/mgmt_dir/] -> File[/tmp/mgmt_dir/a]
        |
        V
Noop[handover] -> ( class important_stuff resources )
                          |
                          V
                 Noop[handback]
                          |
                          V
File[/tmp/mgmt_dir/puppet_subtree/state-file]
```

You can add arbitrary numbers of merge pairs to your code bases, with relationships as needed. From our limited experience, code readability suffers quite a lot from these, however. We advise to keep these structures simple.