Hiera 1 Manual

(Generated on July 01, 2013, from git revision 46784ac1656bd7b57fcfb51d0865ec7ff65533d9)

Hiera 1: Overview

Hiera is a key/value lookup tool for configuration data, built to make Puppet better and let you set node-specific data without repeating yourself. See "Why Hiera?" below for more information, or get started using it right away:

Getting Started With Hiera

To get started with Hiera, you'll need to do all of the following:

- Install Hiera, if it isn't already installed.
- Make a `hiera.yaml` config file.
- Arrange a hierarchy that fits your site and data.
- Write data sources.
- Use your Hiera data in Puppet (or any other tool).

After you have Hiera working, you can adjust your data and hierarchy whenever you need to. You can also test Hiera from the command line to make sure it's fetching the right data for each node.

Learning From Example

If you learn best from example code, start with this simple end-to-end Hiera and Puppet walkthrough. To learn more, you can go back and read the sections linked above.

Why Hiera?

Making Puppet Better

Hiera makes Puppet better by keeping site-specific data out of your manifests. Puppet classes can request whatever data they need, and your Hiera data will act like a site-wide config file.

This makes it:

- Easier to configure your own nodes: default data with multiple levels of overrides is finally easy.
- Easier to re-use public Puppet modules: don't edit the code, just put the necessary data in Hiera.
- Easier to publish your own modules for collaboration: no need to worry about cleaning out your data before showing it around, and no more clashing variable names.

Avoiding Repetition

With Hiera, you can:

- Write common data for most nodes

- Override some values for machines located at a particular facility…
- …and override some of those values for one or two unique nodes.

This way, you only have to write down the differences between nodes. When each node asks for a piece of data, it will get the specific value it needs.

To decide which data sources can override which, Hiera uses a configurable hierarchy. This ordered list can include both static data sources (with names like “common”) and dynamic ones (which can switch between data sources based on the node’s name, operating system, and more).

Hiera 1: Installing

> Note: If you are using Puppet 3 or later, you probably already have Hiera installed. You can skip these steps, and go directly to configuring Hiera.

Prerequisites

- Hiera works on *nix and Windows systems.
- It requires Ruby 1.8.5 or later.
- To work with Puppet, it requires Puppet 2.7.x or later.
- If used with Puppet 2.7.x, it also requires the additional `hiera-puppet` package; see below.

Installing Hiera

If you are using Hiera with Puppet, you should install it on your puppet master server(s); it is optional and unnecessary on agent nodes. (If you are using a standalone puppet apply site, every node should have Hiera.)

Step 1: Install the `hiera` Package or Gem

Install the `hiera` package, using Puppet or your system’s standard package tools.

> Note: You may need to enable the Puppet Labs package repos first.

```
$ sudo puppet resource package hiera ensure=installed
```

If your system does not have native Hiera packages available, you may need to install it as a Rubygem.

```
$ sudo gem install hiera
```

Step 2: Install the Puppet Functions

If you are using Hiera with Puppet 2.7.x, you must also install the `hiera-puppet` package on every puppet master.

```
$ sudo puppet resource package hiera-puppet ensure=installed
```

Or, on systems without native packages:

```
$ sudo gem install hiera-puppet
```

> Note: Puppet 3 does not need the `hiera-puppet` package, and may refuse to install if it is present. You can safely remove `hiera-puppet` in the process of upgrading to Puppet 3.

Next

That's it: Hiera is installed. Next, configure Hiera with its `hiera.yaml` config file.

Hiera 1: The hiera.yaml Config File

Hiera's config file is usually referred to as `hiera.yaml`. Use this file to configure the hierarchy, which backend(s) to use, and settings for each backend.

Hiera will fail with an error if the config file can't be found, although an empty config file is allowed.

Location

Hiera uses different config file locations depending on how it was invoked.

From Puppet

By default, the config file is `$confdir/hiera.yaml`, which is usually one of the following:

- `/etc/puppet/hiera.yaml` in *nix open source Puppet
- `/etc/puppetlabs/puppet/hiera.yaml` in *nix Puppet Enterprise
- `COMMON_APPDATA\PuppetLabs\puppet\etc\hiera.yaml` on Windows

In Puppet 3 or later, you can specify a different config file with the `hiera_config` setting in `puppet.conf`. In Puppet 2.x, you cannot specify a different config file, although you can make

`$confdir/hiera.yaml` a symlink to a different file.

From the Command Line

- `/etc/hiera.yaml` on *nix
- `COMMON_APPDATA\PuppetLabs\hiera\etc\hiera.yaml` on Windows

You can specify a different config file with the `-c` (`--config`) option.

From Ruby Code

- `/etc/hiera.yaml` on *nix
- `COMMON_APPDATA\PuppetLabs\hiera\etc\hiera.yaml` on Windows

You can specify a different config file or a hash of settings when calling `Hiera.new`.

Format

Hiera's config file must be a YAML hash. The file must be valid YAML, but may contain no data.

Each top-level key in the hash must be a Ruby symbol with a colon (`:`) prefix. The available settings are listed below, under "Global Settings" and "Backend-Specific Settings".

Example Config File

```
---
:backends:
  - yaml
  - json
:yaml:
  :datadir: /etc/puppet/hieradata
:json:
  :datadir: /etc/puppet/hieradata
:hierarchy:
  - "%{::clientcert}"
  - "%{::custom_location}"
  - common
```

Default Config Values

If the config file exists but has no data, the default settings will be equivalent to the following:

```
---
:backends: yaml
:yaml:
  :datadir: /var/lib/hiera
:hierarchy: common
:logger: console
```

Global Settings

The Hiera config file may contain any the following settings. If absent, they will have default values.□
Note that each setting must be a Ruby symbol with a colon (`:`) prefix.□

`:hierarchy`

Must be a string or an array of strings, where each string is the name of a static or dynamic data source. (A dynamic source is simply one that contains a `%{variable}` interpolation token. See "Creating Hierarchies" for more details.)

The data sources in the hierarchy are checked in order, top to bottom.

Default value: `"common"` (i.e. a single-element hierarchy whose only level is named "common.")

`:backends`

Must be a string or an array of strings, where each string is the name of an available Hiera backend. The built-in backends are `yaml` and `json`; an additional `puppet` backend is available when using Hiera with Puppet. Additional backends are available as add-ons.

> Custom backends: See "Writing Custom Backends" for details on writing new backend. Custom backends can interface with nearly any existing data store.

The list of backends is processed in order: in the example above, Hiera would check the entire hierarchy in the yaml backend before starting again with the json backend.

Default value: `"yaml"`

`:logger`

Must be the name of an available logger, as a string.

Loggers only control where warnings and debug messages are routed. You can use one of the built-in loggers, or write your own. The built-in loggers are:

- `console` (messages go directly to STDERR)
- `puppet` (messages go to Puppet's logging system)
- `noop` (messages are silenced)

> Custom loggers: You can make your own logger by providing a class called, e.g., `Hiera::Foo_logger` (in which case Hiera's internal name for the logger would be `foo`), and giving it class methods called `warn` and `debug`, each of which should accept a single string.

Default value: `"console"`; note that Puppet overrides this and sets it to `"puppet"`, regardless of what's in the config file.

`:merge_behavior`

Must be one of the following:

- `native` (default) — merge top-level keys only.
- `deep` — merge recursively; in the event of conflicting keys, allow lower priority values to win. You almost never want this.
- `deeper` — merge recursively; in the event of a conflict, allow higher priority values to win.

Anything but `native` requires the `deep_merge` Ruby gem to be installed.

Which merge strategy to use when doing a hash merge lookup. See "Deep Merging in Hiera ≥ 1.2.0" for more details.

Backend-Specific Settings

Any backend can define its own settings and read them from Hiera's config file. If present, the value of a given backend's key must be a hash, whose keys are the settings it uses.

The following settings are available for the built-in backends:

`:yaml` and `:json`

`:DATADIR`

The directory in which to find data source files.

You can interpolate variables into the datadir using `%{variable}` interpolation tokens. This allows you to, for example, point it at `/etc/puppet/hieradata/%{::environment}` to keep your production and development data entirely separate.

Default value: `/var/lib/hiera` on *nix, and `COMMON_APPDATA\PuppetLabs\Hiera\var` on Windows.

`:puppet`

`:DATASOURCE`

The Puppet class in which to look for data.

Default value: `data`

Hiera 1: Creating Hierarchies

Hiera uses an ordered hierarchy to look up data. This allows you to have a large amount of

common data and override smaller amounts of it wherever necessary.

Location and Syntax

Hiera loads the hierarchy from the hiera.yaml config file. The hierarchy must be the value of the top-level `:hierarchy` key.

The hierarchy should be an array. (Alternately, it may be a string; this will be treated like a one-element array.)

Each element in the hierarchy must be a string, which may or may not include variable interpolation tokens. Hiera will treat each element in the hierarchy as the name of a data source.

```
# /etc/hiera.yaml
---
:hierarchy:
  - %{::clientcert}
  - %{::environment}
  - virtual_%{::is_virtual}
  - common
```

Terminology:

We use these two terms within the Hiera docs and in various other places:

- Static data source — A hierarchy element without any interpolation tokens. A static data source will be the same for every node. In the example above, `common` is a static data source — a virtual machine named `web01` and a physical machine named `db01` would both use `common`.
- Dynamic data source — A hierarchy element with at least one interpolation token. If two nodes have different values for the variables it references, a dynamic data source will use two different data sources for those nodes. In the example above: the special `$::clientcert` Puppet variable has a unique value for every node. A machine named `web01` would have a data source named `web01` at the top of its hierarchy, while a machine named `db01` would have `db01` at the top.

Behavior

Ordering

Each element in the hierarchy resolves to the name of a data source. Hiera will check these data sources in order, starting with the first.

- If a data source in the hierarchy doesn't exist, Hiera will move on to the next data source.

- If a data source exists but does not have the piece of data Hiera is searching for, it will move on to the next data source. (Since the goal is to help you not repeat yourself, Hiera expects that most data sources will either not exist or not have the data.)
- If a value is found:
  - In a normal (priority) lookup, Hiera will stop at the first data source with the requested data and return that value.
  - In an array lookup, Hiera will continue, then return all of the discovered values as a flattened array. Values from higher in the hierarchy will be the first elements in the array, and values from lower will be later.
  - In a hash lookup, Hiera will continue, expecting every value to be a hash and throwing an error if any non-hash values are discovered. It will then merge all of the discovered hashes and return the result, allowing values from higher in the hierarchy to replace values from lower.
- If Hiera goes through the entire hierarchy without finding a value, it will use the default value if one was provided, or fail with an error if one wasn't.

Multiple Backends

You can specify multiple backends as an array in `hiera.yaml`. If you do, they function as a second hierarchy.

Hiera will give priority to the first backend, and will check every level of the hierarchy in it before moving on to the second backend. This means that, with the following `hiera.yaml`:

```
---
:backends:
  - yaml
  - json
:hierarchy:
  - one
  - two
  - three
```

…hiera would check the following data sources, in order:

- `one.yaml`
- `two.yaml`
- `three.yaml`
- `one.json`
- `two.json`
- `three.json`

Example

Assume the following hierarchy:

```
# /etc/hiera.yaml
---
:hierarchy:
  - %{::clientcert}
  - %{::environment}
  - virtual_%{::is_virtual}
  - common
```

…and the following set of data sources:

- `web01.example.com`
- `web02.example.com`
- `db01.example.com`
- `production.yaml`
- `development.yaml`
- `virtual_true.yaml`
- `common.yaml`

…and only the `yaml` backend.

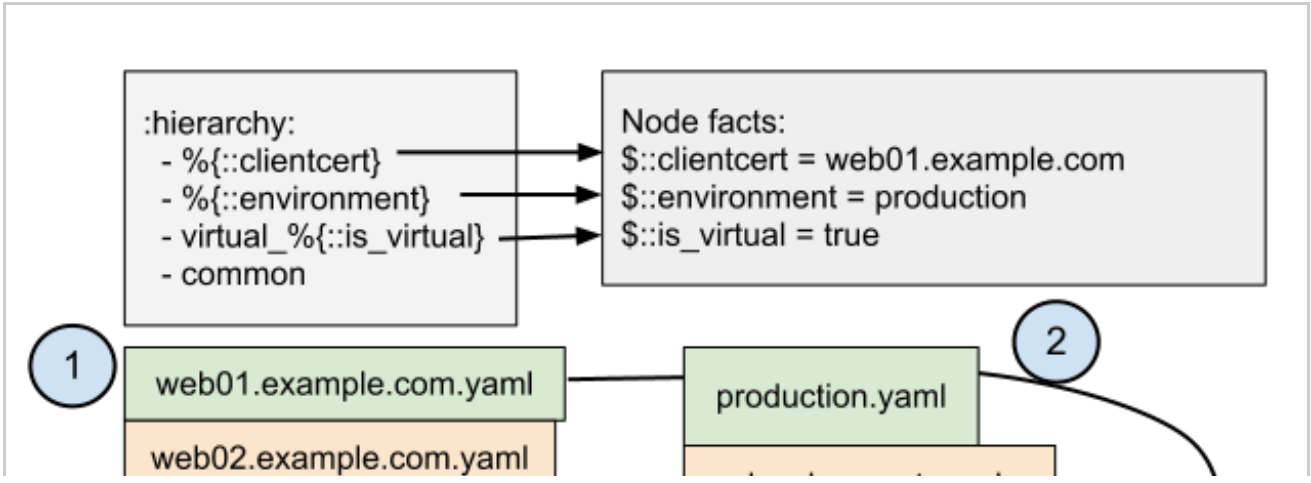
Given two different nodes with different Puppet variables, here are two ways the hierarchy could be interpreted:

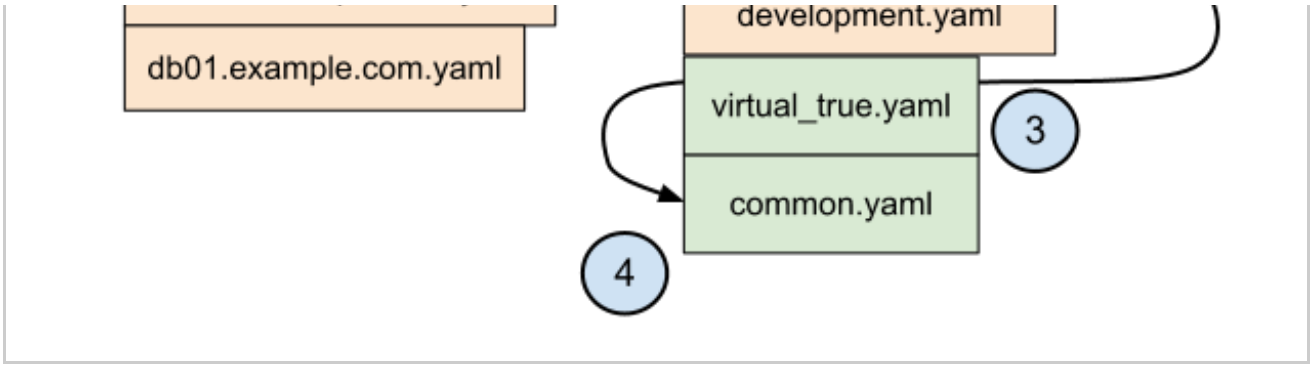
web01.example.com

VARIABLES

- `::clientcert` = `web01.example.com`
- `::environment` = `production`
- `::is_virtual` = `true`

DATA SOURCE RESOLUTION





FINAL HIERARCHY

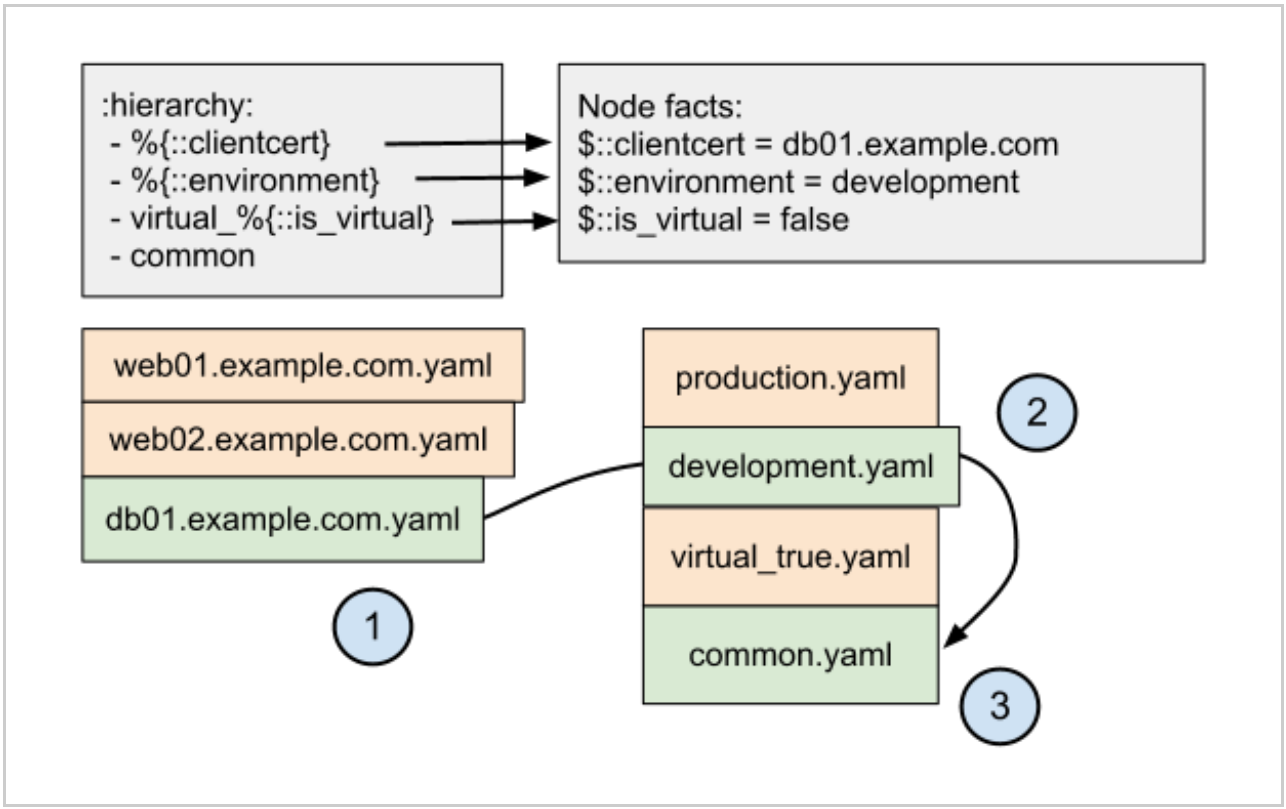
- web01.example.com.yaml
- production.yaml
- virtual_true.yaml
- common.yaml

db01.example.com

VARIABLES

- `::clientcert` = `db01.example.com`
- `::environment` = `development`
- `::is_virtual` = `false`

DATA SOURCE RESOLUTION



FINAL HIERARCHY

- db01.example.com.yaml

- development.yaml
- common.yaml

Note that, since `virtual_false.yaml` doesn't exist, it gets skipped entirely.

Hiera 1: Lookup Types

Hiera always takes a lookup key and returns a single value (of some simple or complex data type), but it has several methods for extracting/assembling that one value from the hierarchy. We refer to these as "lookup methods."

All of these lookup methods are available via Hiera's Puppet functions, command line interface, and Ruby API.

Priority (default)

A priority lookup gets a value from the most specific matching level of the hierarchy. Only one hierarchy level — the first one to match — is consulted.

Priority lookups can retrieve values of any data type (strings, arrays, hashes), but the entire value will come from only one hierarchy level.

This is Hiera's default lookup method.

Array Merge

An array merge lookup assembles a value from every matching level of the hierarchy. It retrieves all of the (string or array) values for a given key, then flattens them into a single array of unique values. If priority lookup can be thought of as a "default with overrides" pattern, array merge lookup can be though of as "default with additions."

For example, given a hierarchy of:

```
- web01.example.com
- common
```

...and the following data:

```
# web01.example.com.yaml
mykey: one

# common.yaml
mykey:
  - two
  - three
```

…an array merge lookup would return a value of `[one, two, three]`.

In this version of Hiera, array merge lookups will fail with an error if any of the values found in the data sources are hashes. It only works with strings, string-like scalar values (booleans, numbers), and arrays.

Hash Merge

A hash merge lookup assembles a value from every matching level of the hierarchy. It retrieves all of the (hash) values for a given key, then merges the hashes into a single hash.

In Hiera 1.x, hash merge lookups will fail with an error if any of the values found in the data sources are strings or arrays. It only works when every value found is a hash.

Native Merging

In Hiera 1.0 and 1.1, this is the only available kind of hash merging. In Hiera ≥ 1.2, deep merges are also available (see below).

In a native hash merge, Hiera merges only the top-level keys and values in each source hash. If the same key exists in both a lower priority source and a higher priority source, the higher priority value will be used.

For example, given a hierarchy of:

```
- web01.example.com
- common
```

…and the following data:

```
# web01.example.com.yaml
mykey:
  z: "local value"

# common.yaml
mykey:
  a: "common value"
  b: "other common value"
  z: "default local value"
```

…a native hash merge lookup would return a value of `{z => "local value", a => "common value", b => "other common value"}`. Note that in cases where two or more source hashes share some keys, higher priority data sources in the hierarchy will override lower ones.

Deep Merging in Hiera ≥ 1.2.0

In Hiera 1.2.0 and later, you can also configure hash merge lookups to recursively merge hash keys. (Implemented as Issue 16107.) This is intended for users who have moved complex data structures (such as `create_resources` hashes) into Hiera.

To configure deep merging, use the `:merge_behavior` setting, which can be set to `native`, `deep`, or `deeper`.

> Limitations:
>
> - This currently only works with the yaml and json backends.
> - You must install the `deep_merge` Ruby gem for deep merges to work. If it isn't available, Hiera will fall back to the default `native` merge behavior.
> - This configuration is global, not per-lookup.

MERGE BEHAVIORS

There are three merge behaviors available.

- The default `native` type is described above under "Native Merging," and matches what Hiera 1.0 and 1.1 do.
- The `deep` type is largely useless and should be avoided.
- The `deeper` type does a recursive merge, behaving as most users expect.

In a `deeper` hash merge, Hiera recursively merges keys and values in each source hash. For each key, if the value is:

- …only present in one source hash, it goes into the final hash.
- …a string/number/boolean and exists in two or more source hashes, the highest priority value goes into the final hash.
- …an array and exists in two or more source hashes, the values from each source are merged into a single array and de-duplicated (but not automatically flattened, as in an array merge lookup).
- …a hash and exists in two or more source hashes, the values from each source are recursively merged, as though they were source hashes.
- …mismatched between two or more source hashes, we haven't validated the behavior. It should act as described in the `deep_merge` gem documentation.

In a `deep` hash merge, Hiera behaves as above, except that when a string/number/boolean exists in two or more source hashes, the lowest priority value goes into the final hash. As mentioned above, this is generally useless.

EXAMPLE

A typical use case for hashes in Hiera is building a data structure which gets passed to the

`create_resources` function. This example implements business rules that say:

1. user 'jen' is defined only on the host 'deglitch'□
2. user 'bob' is defined everywhere, but has a different uid on 'deglitch' than he receives on other□ hosts.
3. user 'ash' is defined everywhere with a consistent uid and shell.□

In hiera.yaml, we set a two-level hierarchy:

```
# /etc/puppet/hiera.yaml
---
:backends:
  - yaml
:logger: puppet
:hierarchy:
  - "%{hostname}"
  - common
:yaml:
  :datadir: /etc/puppet/hieradata
# options are native, deep, deeper
:merge_behavior: deeper
```

In common.yaml, we set up default users for all nodes:

```
---
site_users:
  bob:
    uid: 501
    shell: /bin/bash
  ash:
    uid: 502
    shell: /bin/zsh
    group: common
```

In deglitch.yaml, we set up node-specific user details for the deglitch.example.com:□

```
---
site_users:
  jen:
    uid: 503
    shell: /bin/zsh
    group: deglitch
  bob:
    uid: 1000
    group: deglitch
```

With a standard `native` hash merge, we would end up with a hash like the following:

```
{
```

```
  "bob"=>{
    group=>"deglitch",
    uid=>1000,
  },
  "jen"=>{
    group=>"deglitch",
    uid=>503
    shell=>"/bin/zsh",
  },
  "ash"=>{
    group=>"common",
    uid=>502,
    shell=>"/bin/zsh"
  }
}
```

Notice that Bob is missing his shell — this is because the value of the top-level `bob` key from common.yaml was replaced entirely.

With a `deeper` hash merge, we would get a more intuitive behavior:

```
{
  "bob"=>{
    group=>"deglitch",
    uid=>1000,
    shell=>"/bin/bash"
  },
  "jen"=>{
    group=>"deglitch",
    uid=>503
    shell=>"/bin/zsh",
  },
  "ash"=>{
    group=>"common",
    uid=>502,
    shell=>"/bin/zsh"
  }
}
```

In this case, Bob's shell persists from common.yaml, but deglitch.yaml is allowed to override his uid and group, reducing the amount of data you have to duplicate between files.

With a `deep` merge, you would get:

```
{
  "bob"=>{
    group=>"deglitch",
    uid=>501,
    shell=>"/bin/bash"
  },
  "jen"=>{
    group=>"deglitch",
```

```
        shell=>"/bin/zsh",
        uid=>503
      },
      "ash"=>{
        group=>"common",
        uid=>502,
        shell=>"/bin/zsh"
      }
    }
```

In this case, deglitch.yaml was able to set the group because common.yaml didn't have a value for it, but where there was a conflict, like the uid, common won. Most users don't want this.

Unfortunately none of these merge behaviors work with data bindings for automatic parameter lookup, because there's no way to specify the lookup type. So instead of any of the above results, automatic data binding lookups will only see results from `deglitch.yaml`. See Bug #20199 to track progress on this.

Hiera 1: Writing Data Sources

Hiera can use several different data backends, including two built-in backends and various optional backends. Each backend uses a different document format for its data sources.

This page describes the built-in `yaml` and `json` backends, as well as the `puppet` backend included with Hiera's Puppet integration. For optional backends, see the backend's documentation.

YAML

Summary

The `yaml` backend looks for data sources on disk, in the directory specified in its `:datadir` setting. It expects each data source to be a text file containing valid YAML data, with a file extension of `.yaml`. No other file extension (e.g. `.yml`) is allowed.

Data Format

See yaml.org and the "YAML for Ruby" cookbook for a complete description of valid YAML.

The root object of each YAML data source must be a YAML mapping (hash). Hiera will treat its top level keys as the pieces of data available in the data source. The value for each key can be any of the data types below.

Hiera's data types map to the native YAML data types as follows:

Hiera	YAML
Hash	Mapping

Array	Sequence
String	Quoted scalar or non-boolean unquoted scalar
Number	Integer or float
Boolean	Boolean (note: includes `on` and `off`, `yes` and `no` in addition to `true` and `false`)

Any string may include any number of variable interpolation tokens.

> Important note: The "psych" YAML parser, which is used by many Ruby versions, requires that any strings containing a `%` be quoted.

Example

```
---
# array
apache-packages:
    - apache2
    - apache2-common
    - apache2-utils

# string
apache-service: apache2

# interpolated facter variable
hosts_entry: "sandbox.%{fqdn}"

# hash
sshd_settings:
    root_allowed: "no"
    password_allowed: "yes"

# alternate hash notation
sshd_settings: {root_allowed: "no", password_allowed: "yes"}

# to return "true" or "false"
sshd_settings: {root_allowed: no, password_allowed: yes}
```

JSON

Summary

The `json` backend looks for data sources on disk, in the directory specified in its `:datadir` setting. It expects each data source to be a text file containing valid JSON data, with a file extension of `.json`. No other file extension is allowed.

Data Format

See the JSON spec for a complete description of valid JSON.

The root object of each JSON data source must be a JSON object (hash). Hiera will treat its top level keys as the pieces of data available in the data source. The value for each key can be any of the data types below.

Hiera's data types map to the native JSON data types as follows:

Hiera	JSON
Hash	Object
Array	Array
String	String
Number	Number
Boolean	`true` / `false`

Any string may include any number of variable interpolation tokens.

Example

```
{
    "apache-packages" : [
    "apache2",
    "apache2-common",
    "apache2-utils"
    ],

    "hosts_entry" :  "sandbox.%{fqdn}",

    "sshd_settings" : {
                        "root_allowed" : "no",
                        "password_allowed" : "no"
                      }
}
```

Puppet

Coming soon.

Hiera 1: Variables and Interpolation

Hiera receives a set of variables whenever it is invoked, and the config file and data sources can insert these variables into settings and data. This lets you make dynamic data sources in the hierarchy, and avoid repeating yourself when writing data.

Inserting Variable Values

Interpolation tokens look like `%{variable}` — a percent sign (`%`) followed by a pair of curly braces (`{}`) containing a variable name.

If any setting in the config file or value in a data source contains an interpolation token, Hiera will replace the token with the value of the variable. Interpolation tokens can appear alone or as part of a string.

- Hiera can only interpolate variables whose values are strings. (Numbers from Puppet are also passed as strings and can be used safely.) You cannot interpolate variables whose values are booleans, numbers not from Puppet, arrays, hashes, resource references, or an explicit `undef` value.
- Additionally, Hiera cannot interpolate an individual element of any array or hash, even if that element's value is a string.

In Data Sources

The main use for interpolation is in the config file, where you can set dynamic data sources in the hierarchy:

```
---
:hierarchy:
  - %{::clientcert}
  - %{::custom_location}
  - virtual_%{::is_virtual}
  - %{::environment}
  - common
```

In this example, every data source except the final one will vary depending on the current values of the `::clientcert, ::custom_location, ::is_virtual,` and `::environment` variables.

In Other Settings

You can also interpolate variables into other settings, such as `:datadir` (in the YAML and JSON backends):

```
:yaml:
  :datadir: /etc/puppet/hieradata/%{::environment}
```

This example would let you use completely separate data directories for your production and development environments.

In Data

Within a data source, you can interpolate variables into any string, whether it's a standalone value or part of a hash or array value. This can be useful for values that should be different for every node, but which differ predictably:

```
# /var/lib/hiera/common.yaml
---
smtpserver: mail.%{::domain}
```

In this example, instead of creating a `%{::domain}` hierarchy level and a data source for each domain, you can get a similar result with one line in the `common` data source.

Passing Variables to Hiera

Hiera's variables can come from a variety of sources, depending on how Hiera is invoked.

From Puppet

When used with Puppet, Hiera automatically receives all of Puppet's current variables. This includes facts and built-in variables, as well as local variables from the current scope. Most users will almost exclusively interpolate facts and built-in variables in their Hiera configuration and data.

- Remove Puppet's `$` (dollar sign) prefix when using its variables in Hiera. (That is, a variable called `$::clientcert` in Puppet is called `::clientcert` in Hiera.)
- Puppet variables can be accessed by their short name or qualified name.

BEST PRACTICES

- Usually avoid referencing user-set local variables from Hiera. Instead, use facts, built-in variables, top-scope variables, node-scope variables, or variables from an ENC whenever possible.
- When possible, reference variables by their fully-qualified names (e.g. `%{::environment}` and `%{::clientcert}`) to make sure their values are not masked by local scopes.

These two guidelines will make Hiera more predictable, and can help protect you from accidentally mingling data and code in your Puppet manifests.

From the Command Line

When called from the command line, Hiera defaults to having no variables available. You can specify individual variables, or a file or service from which to obtain a complete "scope" of variables. See command line usage for more details.

From Ruby

When calling Hiera from Ruby code, you can pass in a complete "scope" of variables as the third argument to the `#lookup` method. The complete signature of `#lookup` is:

```
    hiera_object.lookup(key, default, scope, order_override=nil,
resolution_type=:priority)
```

Hiera 1: Using Hiera With Puppet

Puppet can use Hiera to look up data. This helps you disentangle site-specific data from Puppet code, for easier code re-use and easier management of data that needs to differ across your node population.

Enabling and Configuring Hiera for Puppet

Puppet 3 and Newer

Puppet 3.x and later ship with Hiera support already enabled. You don't need to do anything extra. Hiera data should live on the puppet master(s).

- Puppet expects to find the hiera.yaml file at `$confdir/hiera.yaml` (usually `/etc/puppet/hiera.yaml`); you can change this with the `hiera_config` setting.
- Remember to set the `:datadir` setting for any backends you are using. It's generally best to use something within the `/etc/puppet/` directory, so that the data is in the first place your fellow admins expect it.

Puppet 2.7

You must install both Hiera and the `hiera-puppet` package on your puppet master(s) before using Hiera with Puppet. Hiera data should live on the puppet master(s).

- Puppet expects to find the hiera.yaml file at `$confdir/hiera.yaml` (usually `/etc/puppet/hiera.yaml`). This is not configurable in 2.7.
- Remember to set the `:datadir` setting for any backends you are using. It's generally best to use something within the `/etc/puppet/` directory, so that the data is in the first place your fellow admins expect it.

Older Versions

Hiera is not supported with older versions, but you may be able to make it work similarly to Puppet 2.7.

Puppet Variables Passed to Hiera

Whenever a Hiera lookup is triggered from Puppet, Hiera receives a copy of all of the variables currently available to Puppet, including both top scope and local variables.

Hiera can then use any of these variables in the variable interpolation tokens scattered throughout its hierarchy and data sources. You can enable more flexible hierarchies by creating custom facts for things like datacenter location and server purpose.

Special Pseudo-Variables

When doing any Hiera lookup, with both automatic parameter lookup and the Hiera functions, Puppet sets two variables that aren't available in regular Puppet manifests:

- `calling_module` — The module in which the lookup is written. This has the same value as the Puppet `$module_name` variable.
- `calling_class` — The class in which the lookup is evaluated. If the lookup is written in a defined type, this is the class in which the current instance of the defined type is declared.

> Note: These variables were broken in some versions of Puppet.
>
> - Puppet 2.7.x: Good
> - Puppet 3.0.x and 3.1.x: Bad
> - Puppet 3.2.x and later: Good

Best Practices

Do not use local Puppet variables in Hiera's hierarchy or data sources. Only use facts and ENC-set top-scope variables. Use absolute top-scope notation (i.e. `%{::clientcert}` instead of `%{clientcert}`) in Hiera's config files to avoid accidentally using a local variable instead of a top-scope one.

This is not a hard and fast rule, and there are some exceptions, but you'll almost always regret breaking it. This is because the point of Hiera is to disentangle data from code, and enable node-specific hierarchical data that doesn't rely on Puppet's parse order.

If Hiera only relies on variables that are set before Puppet starts to parse its manifests, its data will be node-specific but static and reliable. This is good! It frees you from weird parse-order dependent bugs, makes it easier to test your data, and makes it easier to tell exactly what's going to happen by looking at a given section of Puppet code.

On the other hand, making Hiera rely on local variables set by Puppet's parser means you're still fundamentally embedding data in your code, and will still have all of the problems Hiera is meant to solve.

Again, there are sometimes exceptions, but tread carefully.

Automatic Parameter Lookup

Puppet will automatically retrieve class parameters from Hiera, using lookup keys like `myclass::parameter_one`.

> Note: This feature only exists in Puppet 3 and later.

Puppet classes can optionally include parameters in their definition. This lets the class ask for data to be passed in at the time that it's declared, and it can use that data as normal variables throughout its definition.

```
    # In this example, $parameter's value gets set when `myclass` is eventually
declared.
    # Class definition:
    class myclass ($parameter_one = "default text") {
      file {'/tmp/foo':
        ensure  => file,
        content => $parameter_one,
      }
    }
```

Parameters can be set several ways, and Puppet will try each of these ways in order when the class is declared or assigned by an ENC:

1. If it was a resource-like declaration/assignment, Puppet will use any parameters that were explicitly set. These always win if they exist.
2. Puppet will automatically look up parameters in Hiera, using `<CLASS NAME>::<PARAMETER NAME>` as the lookup key. (E.g. `myclass::parameter_one` in the example above.)
3. If 1 and 2 didn't result in a value, Puppet will use the default value from the class's definition. (E.g. `"default text"` in the example above.)
4. If 1 through 3 didn't result in a value, fail compilation with an error.

Step 2 interests us most here. Because Puppet will always look for parameters in Hiera, you can safely declare any class with `include`, even classes with parameters. (This wasn't the case in earlier Puppet versions.) Using the example above, you could have something like the following in your Hiera data:

```
    # /etc/puppet/hieradata/web01.example.com.yaml
    ---
    myclass::parameter_one: "This node is special, so we're overriding the
common configuration that the other nodes use."

    # /etc/puppet/hieradata/common.yaml
    ---
    myclass::parameter_one: "This node can use the standard configuration."
```

You could then say `include myclass` for every node, and each node would get its own appropriate data for the class.

Why

Automatic parameter lookup is good for writing reusable code because it is regular and

predictable. Anyone downloading your module can look at the first line of each manifest and easily see which keys they need to set in their own Hiera data. If you use the Hiera functions in the body of a class instead, you will need to clearly document which keys the user needs to set.

Limitations

PRIORITY ONLY

Automatic parameter lookup can only use the priority lookup method.

This means that, although it can receive any type of data from Hiera (strings, arrays, hashes), it cannot merge values from multiple hierarchy levels; you will only get the value from the most-specific hierarchy level.

If you need to merge arrays or merge hashes from multiple hierarchy levels, you will have to use the `hiera_array` or `hiera_hash` functions in the body of your classes.

NO PUPPET 2.7 SUPPORT

Relying on automatic parameter lookup means writing code for Puppet 3 and later only.

You can, however, mimic Puppet 3 behavior in 2.7 by combining parameter defaults and Hiera function calls:

```
class myclass (
  $parameter_one = hiera('myclass::parameter_one', 'default text')
) {
  # ...
}
```

- This pattern requires that 2.7 users have Hiera installed; it will fail compilation if the Hiera functions aren't present.
- Since all of your parameters will have defaults, your class will be safely declarable with `include`, even in 2.7.
- Puppet 2.7 will do Hiera lookups for the same keys that Puppet 3 automatically looks up.
- Note that this carries a performance penalty, since Puppet 3 will end up doing two Hiera calls for each parameter instead of one.

Hiera Lookup Functions

Puppet has three lookup functions for retrieving data from Hiera. All of these functions return a single value (though note that this value may be an arbitrarily complex nested data structure), and can be used anywhere that Puppet accepts values of that type. (Resource attributes, resource titles, the values of variables, etc.)

`hiera`

> Standard priority lookup. Gets the most specific value for a given key. This can retrieve values of any data type (strings, arrays, hashes) from Hiera.

`hiera_array`

> Uses an array merge lookup. Gets all of the string or array values in the hierarchy for a given key, then flattens them into a single array of unique values.

`hiera_hash`

> Uses a hash merge lookup. Expects every value in the hierarchy for a given key to be a hash, and merges the top-level keys in each hash into a single hash. Note that this does not do a deep-merge in the case of nested structures.

Each of these functions takes three arguments. In order:

1. Key (required): the key to look up in Hiera.
2. Default (optional): a fallback value to use if Hiera doesn't find anything for that key. If this isn't provided, a lookup failure will cause a compilation failure.
3. Override (optional): the name of an arbitrary hierarchy level to insert at the top of the hierarchy. This lets you use a temporary modified hierarchy for a single lookup. (E.g., instead of a hierarchy of `$clientcert -> $osfamily -> common`, a lookup would use `specialvalues -> $clientcert -> $osfamily -> common`; you would need to be sure to have `specialvalues.yaml` or whatever in your Hiera data.)

Using the Lookup Functions From Templates

In general, don't use the Hiera functions from templates. That pattern is too obscure, and will hurt your code's maintainability — if a co-author of your code needs to change the Hiera invocations and is searching `.pp` files for them, they might miss the extra invocations in the template. Even if only one person is maintaining this code, they're likely to make similar mistakes after a few months have passed.

It's much better to use the lookup functions in a Puppet manifest, assign their value to a local variable, and then reference the variable from the template. This keeps the function calls isolated in one layer of your code, where they'll be easy to find if you need to modify them later or document them for other users.

Nevertheless, you can, of course, use the `scope.function_` prefix to call any of the Hiera functions from a template.

Interacting With Structured Data from Hiera

The lookup functions and the automatic parameter lookup always return the values of top-level keys in your Hiera data — they cannot descend into deeply nested data structures and return only a portion of them. To do this, you need to first store the whole structure as a variable, then index into the structure from your Puppet code or template.

Example:

```
# /etc/puppet/hieradata/appservers.yaml
---
proxies:
  - hostname: lb01.example.com
    ipaddress: 192.168.22.21
  - hostname: lb02.example.com
    ipaddress: 192.168.22.28
```

Good:

```
# Get the structured data:
$proxies = hiera('proxies')
# Index into the structure:
$use_ip = $proxies[1]['ipaddress'] # will be 192.168.22.28
```

Bad:

```
# Try to skip a step, and give Hiera something it doesn't understand:
$use_ip = hiera( 'proxies'[1]['ipaddress'] ) # will explode
```

Assigning Classes to Nodes With Hiera (`hiera_include`)

You can use Hiera to assign classes to nodes with the special `hiera_include` function. This lets you assign classes in great detail without repeating yourself — it's essentially what people have traditionally tried and failed to use node inheritance for. It can get you the benefits of a rudimentary external node classifier without having to write an actual ENC.

1. Choose a key name to use for classes. Below, we assume you're just using `classes`.
2. In your `/etc/puppet/manifests/site.pp` file, write the line `hiera_include('classes')`. Put this outside any node definition, and below any top-scope variables that you might be relying on for Hiera lookups.
3. Create `classes` keys throughout your Hiera hierarchy.
   - The value of each `classes` key should be an array.
   - Each value in the array should be the name of a class.

Once you do these steps, Puppet will automatically assign classes from Hiera to every node. Note that the `hiera_include` function uses an array merge lookup to retrieve the `classes` array; this means every node will get every class from its hierarchy.

Example:

Assuming a hierarchy of:

```
:hierarchy:
  - '%{::clientcert}'
```

```
    - '%{::osfamily}'
    - 'common'
```

…and given Hiera data like the following:

```
# common.yaml
---
classes:
  - base
  - security
  - mcollective

# Debian.yaml
---
classes:
  - base::linux
  - localrepos::apt

# web01.example.com
---
classes:
  - apache
  - apache::passenger
```

…the Ubuntu node `web01.example.com` would get all of the following classes:

- apache
- apache::passenger
- base::linux
- localrepos::apt
- base
- security
- mcollective

In Puppet 3, each of these classes would then automatically look up any required parameters in Hiera.

Hiera 1: Complete Example

In this example, we'll use the popular Puppet Labs ntp module, an exemplar of the package/file/service pattern in common use among Puppet users. We'll start simply, using Hiera to provide the ntp module with parameter data based on particular nodes in our organization. Then we'll use Hiera to assign the `ntp` class provided by the module to specific nodes.

What Can We Do With Hiera?

Let's get started by looking at the ntp module. It does all of its work in a single `ntp` class, which lives in the `init.pp` manifest. The `ntp` class also evaluates some ERB templates stored in the module's template directory. So, what can we do with Hiera?

Express Organizational Information

The `ntp` class takes five parameters:

- servers
- restrict
- autoupdate
- enable
- template

Most of these parameters reflect decisions we have to make about each of the nodes to which we'd apply the `ntp` class: Can it act as a time server for other hosts? (`restrict`), which servers should it consult? (`servers`), or should we allow Puppet to automatically update the ntp package or not? (`autoupdate`).

Without Hiera, we might find ourselves adding organizational data to our module code as default parameter values, reducing how shareable it is. We might find ourselves repeating configuration data in our site manifests to cover minor differences in configuration between nodes.

With Hiera, we can move these decisions into a hierarchy built around the facts that drive these decisions, increase sharability, and repeat ourselves less.

Classify Nodes With Hiera

We can also use Hiera to assign classes to nodes using the hiera_include function, adding a single line to our `site.pp` manifest, then assigning classes to nodes within Hiera instead of within our site manifests. This can be a useful shortcut when we're explicitly assigning classes to specific nodes within Hiera, but it becomes very powerful when we implicitly assign classes based on a node's characteristics. In other words, we'll show you how you don't need to know the name of very VMWare guest in your organization to make sure they all have a current version of VMWare Tools installed.

Describing Our Environment

For purposes of this walkthrough, we'll assume a situation that looks something like this:

- We have two ntp servers in the organization that are allowed to talk to outside time servers. Other ntp servers get their time data from these two servers.
- One of our primary ntp servers is very cautiously configured — we can't afford outages, so it's

not allowed to automatically update its ntp server package without testing. The other server is more permissively configured.

- We have a number of other ntp servers that will use our two primary servers.
- We have a number of VMWare guest operating systems that need to have VMWare Tools installed.

Our Environment Before Hiera

How did things look before we decided to use Hiera? Classes are assigned to nodes via the puppet site manifest (`/etc/puppet/manifests/sites.pp` for Puppet open source), so here's how our site manifest might have looked:

```
 node "kermit.example.com" {
   class { "ntp":
  servers    => [ '0.us.pool.ntp.org iburst','1.us.pool.ntp.org
iburst','2.us.pool.ntp.org iburst','3.us.pool.ntp.org iburst'],
  autoupdate => false,
  restrict => false,
  enable => true,
   }
 }

 node "grover.example.com" {
   class { "ntp":
  servers    => [ 'kermit.example.com','0.us.pool.ntp.org
iburst','1.us.pool.ntp.org iburst','2.us.pool.ntp.org iburst'],
  autoupdate => true,
  restrict => false,
  enable => true,
   }
 }

 node "snuffie.example.com", "bigbird.example.com", "hooper.example.com" {
   class { "ntp":
  servers    => [ 'grover.example.com', 'kermit.example.com'],
  autoupdate => true,
  restrict => true,
  enable => true,
   }
 }
```

Configuring Hiera and Setting Up the Hierarchy

All Hiera configuration begins with `hiera.yaml`. You can read a full discussion of this file, including where you should put it depending on the version of Puppet you're using. Here's the one we'll be using for this walkthrough:

```
---
:backends:
  - json
:json:
```

```
    :datadir: /etc/puppet/hiera
  :hierarchy:
    - node/%{::fqdn}
    - common
```

Step-by-step:

`:backends:` tells Hiera what kind of data sources it should process. In this case, we'll be using JSON files.

`:json:` configures the JSON data backend, telling Hiera to look in `/etc/puppet/hiera` for JSON data sources.

`:hierarchy:` configures the data sources Hiera should consult. Puppet users commonly separate their hierarchies into directories to make it easier to get a quick top-level sense of how the hierarchy is put together. In this case, we're keeping it simple:

- A single `node/` directory will contain any number of files named after some node's `fqdn` (fully qualified domain name) fact. (E.g. `/etc/puppet/hiera/node/grover.example.com.json`) This lets us specifically configure any given node with Hiera. Not every node needs to have a file in `node/` — if it's not there, Hiera will just move onto the next hierarchy level.
- Next, the `common` data source (the `/etc/puppet/hiera/common.json` file) will provide any common or default values we want to use when Hiera can't find a match for a given key elsewhere in our hierarchy. In this case, we're going to use it to set common ntp servers and default configuration options for the ntp module.

> Hierarchy and facts note: When constructing a hierarchy, keep in mind that most of the useful Puppet variables are facts. Since facts are submitted by the agent node itself, they aren't necessarily trustworthy. We don't recommend using facts as the sole deciding factor for distributing sensitive credentials.
>
> In this example, we're using the `fqdn` fact to identify specific nodes; the special `clientcert` variable is another option, which the agent sets to the value of its `certname` setting. (This is usually the same as the fqdn, but not always.) Right now, Puppet doesn't provide any variables with trusted data about a given node; we're currently investigating the possibility of adding some.

> Puppet master note: If you modify `hiera.yaml` between agent runs, you'll have to restart your puppet master for your changes to take effect.

Configuring for the Command Line

The Hiera command line tool is useful when you're in the process of designing and testing your

hierarchy. You can use it to mock in facts for Hiera to look up without having to go through cumbersome trial-and-error puppet runs. Since the `hiera` command expects to find `hiera.yaml` at `/etc/hiera.yaml`, you should set a symbolic link from your `hiera.yaml` file to `/etc/hiera.yaml`:

```
$ ln -s /etc/puppet/hiera.yaml /etc/hiera.yaml
```

Writing the Data Sources

Now that we've got Hiera configured, we're ready to return to the ntp module and take a look at the `ntp` class's parameters.

> Learning About Hiera Data Sources: This example won't cover all the data types you might want to use, and we're only using one of two built-in data backends (JSON). For a more complete look at data sources, please see our guide to writing Hiera data sources, which includes more complete examples written in JSON and YAML.

Identifying Parameters

We need to start by figuring out the parameters required by the `ntp` class. So let's look at the ntp module's `init.pp` manifest, where we see five:

servers

> An array of time servers; `UNSET` by default. Conditional logic in `init.pp` provides a list of ntp servers maintained by the respective maintainers of our module's supported operating systems.

restrict

> Whether to restrict ntp daemons from allowing others to use as a server; `true` by default

autoupdate

> Whether to update the ntp package automatically or not; `false` by default

enable

> Whether to start the ntp daemon on boot; `true` by default

template

> The name of the template to use to configure the ntp service. This is `undef` by default, and it's configured within the `init.pp` manifest with some conditional logic.

- See the Puppet language reference for more about class parameters.

Making Decisions and Expressing Them in Hiera

Now that we know the parameters the `ntp` class expects, we can start making decisions about the nodes on our system, then expressing those decisions as Hiera data. Let's start with kermit and grover: The two nodes in our organization that we allow to talk to the outside world for purposes of timekeeping.

`KERMIT.EXAMPLE.COM.JSON`

We want one of these two nodes, `kermit.example.com`, to act as the primary organizational time server. We want it to consult outside time servers, we won't want it to update its ntp server package by default, and we definitely want it to launch the ntp service at boot. So let's write that out in JSON, making sure to express our variables as part of the `ntp` namespace to insure Hiera will pick them up as part of its automatic parameter lookup.

```
{
   "ntp::restrict" : false,
   "ntp::autoupdate" : false,
   "ntp::enable" : true,
   "ntp::servers" : [
    "0.us.pool.ntp.org iburst",
    "1.us.pool.ntp.org iburst",
    "2.us.pool.ntp.org iburst",
    "3.us.pool.ntp.org iburst"
    ]
}
```

Since we want to provide this data for a specific node, and since we're using the `fqdn` fact to identify unique nodes in our hierarchy, we need to save this data in the `/etc/puppet/hiera/node` directory as `kermit.example.com.json`.

Once you've saved that, let's do a quick test using the Hiera command line tool:

```
$ hiera ntp::servers ::fqdn=kermit.example.com
```

You should see this:

```
["0.us.pool.ntp.org iburst", "1.us.pool.ntp.org iburst", "2.us.pool.ntp.org
iburst", "3.us.pool.ntp.org iburst"]
```

That's just the array of outside ntp servers and options, which we expressed as a JSON array and which Hiera is converting to a Puppet-like array. The module will use this array when it generates configuration files from its templates.

Something Went Wrong? If, instead, you get `nil`, `false`, or something else completely, you should step back through your Hiera configuration making sure:

- Your `hiera.yaml` file matches the example we provided
- You've put a symlink to `hiera.yaml` where the command line tool expects to find it (`/etc/hiera.yaml`)
- You've saved your `kermit.example.com` data source file with a `.json` extension
- Your data source file's JSON is well formed. Missing or extraneous commas will cause the JSON parser to fail
- You restarted your puppet master if you modified `hiera.yaml`

Provided everything works and you get back that array of ntp servers, you're ready to configure another node.

GROVER.EXAMPLE.COM.JSON

Our next ntp node, `grover.example.com`, is a little less critical to our infrastructure than kermit, so we can be a little more permissive with its configuration: It's o.k. if grover's ntp packages are automatically updated. We also want grover to use kermit as its primary ntp server. Let's express that as JSON:

```
{
   "ntp::restrict" : false,
   "ntp::autoupdate" : true,
   "ntp::enable" : true,
   "ntp::servers" : [
    "kermit.example.com iburst",
    "0.us.pool.ntp.org iburst",
    "1.us.pool.ntp.org iburst",
    "2.us.pool.ntp.org iburst",
    ]
}
```

As with `kermit.example.com`, we want to save grover's Hiera data source in the `/etc/puppet/hiera/nodes` directory using the `fqdn` fact for the file name: `grover.example.com.json`. We can once again test it with the hiera command line tool:

```
$ hiera ntp::servers ::fqdn=grover.example.com
["kermit.example.com iburst", "0.us.pool.ntp.org iburst", "1.us.pool.ntp.org
iburst", "2.us.pool.ntp.org iburst"]
```

COMMON.JSON

So, now we've configured the two nodes in our organization that we'll allow to update from outside ntp servers. However, we still have a few nodes to account for that also provide ntp services. They

depend on kermit and grover to get the correct time, and we don't mind if they update themselves. Let's write that out in JSON:

```
{
   "ntp::restrict" : true,
   "ntp::autoupdate" : true,
   "ntp::enable" : true,
   "ntp::servers" : [
    "grover.example.com iburst",
    "kermit.example.com iburst"
   ]
}
```

Unlike kermit and grover, for which we had slightly different but node-specific configuration needs, we're comfortable letting any other node that uses the ntp class use this generic configuration data. Rather than creating a node-specific data source for every possible node on our network that might need to use the ntp module, we'll store this data in `/etc/puppet/hiera/common.json`. With our very simple hierarchy, which so far only looks for the `fqdn` facts, any node with a `fqdn` that doesn't match the nodes we have data sources for will get the data found in `common.json`. Let's test against one of those nodes:

```
$ hiera ntp::servers ::fqdn=snuffie.example.com
["kermit.example.com iburst", "grover.example.com iburst"]
```

MODIFYING OUR `SITE.PP` MANIFEST

Now that everything has tested out from the command line, it's time to get a little benefit from this work in production.

If you'll remember back to our pre-Hiera configuration, we were declaring a number of parameters for the `ntp` class in our `site.pp` manifest, like this:

```
 node "kermit.example.com" {
   class { "ntp":
  servers    => [ '0.us.pool.ntp.org iburst','1.us.pool.ntp.org
iburst','2.us.pool.ntp.org iburst','3.us.pool.ntp.org iburst'],
  autoupdate => false,
  restrict => false,
  enable => true,
   }
 }
```

In fact, we had three separate stanzas of that length. But now that we've moved all of that parameter data into Hiera, we can significantly pare down `site.pp`:

```
 node "kermit.example.com", "grover.example.com", "snuffie.example.com" {
```

```
        include ntp
        # or:
        # class { "ntp": }
    }
```

That's it.

Since Hiera is automatically providing the parameter data from the data sources in its hierarchy, we don't need to do anything besides assign the `ntp` class to the nodes and let Hiera's parameter lookups do the rest. In the future, as we change or add nodes that need to use the `ntp` class, we can:

- Quickly copy data source files to cover cases where a node needs a specialized configuration.
- If the new node can work with the generic configuration in `common.json`, we can say `include ntp` in our `site.pp` without writing any new Hiera data.
- Since Hiera looks up each parameter individually, we can also write a new JSON file that, for example, only changes `ntp::autoupdate` — Hiera will get the rest of the parameters from `common.json`.

If you're interested in taking things a step further, using the decision-making skills you picked up in this example to choose which nodes even get a particular class, let's keep going.

Assigning a Class to a Node With Hiera

In the first part of our example, we were concerned with how to use Hiera to provide data to a parameterized class, but we were assigning the classes to nodes in the traditional Puppet way: By making `class` declarations for each node in our `site.pp` manifest. Thanks to the `hiera_include` function, you can assign nodes to a class the same way you can assign values to class parameters: Picking a facter fact on which you want to base a decision, adding to the hierarchy in your `hiera.yaml` file, then writing data sources.

Using `hiera_include`

Where last we left off, our `site.pp` manifest was looking somewhat spare. With the `hiera_include` function, we can pare things down even further by picking a key to use for classes (we recommend `classes`), then declaring it in our `site.pp` manifest:

```
hiera_include('classes')
```

From this point on, you can add or modify an existing Hiera data source to add an array of classes you'd like to assign to matching nodes. In the simplest case, we can visit each of kermit, grover, and snuffie and add this to their JSON data sources in `/etc/puppet/hiera/node`:

```
"classes" : "ntp",
```

modifying kermit's data source, for instance, to look like this:

```
{
   "classes" : "ntp",
   "ntp::restrict" : false,
   "ntp::autoupdate" : false,
   "ntp::enable" : true,
   "ntp::servers" : [
    "0.us.pool.ntp.org iburst",
    "1.us.pool.ntp.org iburst",
    "2.us.pool.ntp.org iburst",
    "3.us.pool.ntp.org iburst"
    ]
}
```

`hiera_include` requires either a string with a single class, or an array of classes to apply to a given node. Take a look at the "classes" array at the top of our kermit data source to see how we might add three classes to kermit:

```
{
   "classes" : [
    "ntp",
    "apache",
    "postfix"
   ],
   "ntp::restrict" : false,
   "ntp::autoupdate" : false,
   "ntp::enable" : true,
   "ntp::servers" : [
    "0.us.pool.ntp.org iburst",
    "1.us.pool.ntp.org iburst",
    "2.us.pool.ntp.org iburst",
    "3.us.pool.ntp.org iburst"
    ]
}
```

We can test which classes we've assigned to a given node with the Hiera command line tool:

```
$ hiera classes ::fqdn=kermit.example.com
["ntp", "apache", "postfix"]
```

Note: The `hiera_include` function will do an array merge lookup, which can let more specific data sources add to common sources instead of replacing them. This helps you avoid repeating yourself.

That demonstrates a very simple case for `hiera_include`, where we knew that we wanted to assign a particular class to a specific host by name. But just as we used the `fqdn` fact to choose which of our nodes received specific parameter values, we can use that or any other fact to drive class assignments. In other words, you can assign classes to nodes based on characteristics that aren't as obvious as their names, creating the possibility of configuring nodes based on more complex characteristics.

Some organizations might choose to make sure all their Macs have Homebrew installed, or assign a `postfix` class to nodes that have a mail server role expressed through a custom fact, or assign a `vmware_tools` class that installs and configures VMWare Tools packages on every host that returns `vmware` as the value for its `virtual` fact.

In fact, let's use that last case for this part of the walkthrough: Installing VMWare Tools on a virtual guest. There's a puppet-vmwaretools module on the Puppet Forge that addresses just this need. It takes two parameters:

version

> The version of VMWare Tools we want to install

working_dir

> The directory into which we want to install VMWare

Two ways we might want to use Hiera to help us organize our use of the class this module provides include making sure it's applied to all our VMWare virtual hosts, and configuring where it's installed depending on the guest operating system for a given virtual host.

So let's take a look at our `hiera.yaml` file and make provisions for two new data sources. We'll create one based on the `virtual` fact, which will return `vmware` when a node is a VMWare-based guest. We'll create another based on the `osfamily` fact, which returns the general family to which a node's operating system belongs (e.g. "Debian" for Ubuntu and Debian systems, or "RedHat" for RHEL, CentOS, and Fedora systems):

```
---
:backends:
  - json
:json:
  :datadir: /etc/puppet/hieradata
:hierarchy:
  - node/%{::fqdn}
  - virtual/%{::virtual}
  - osfamily/%{osfamily}
  - common
```

Next, we'll need to create directories for our two new data sources:

```
`mkdir /etc/puppet/hiera/virtual; mkdir /etc/puppet/hiera/osfamily`
```

In our `virtual` directory, we'll want to create the file `vmware.json`. In this data source, we'll be assigning the `vmwaretools` class, so the file will need to look like this:

```
{
  "classes": "vmwaretools"
}
```

Next, we need to provide the data for the `vmwaretools` class parameters. We'll assume we have a mix of Red Hat and Debian VMs in use in our organization, and that we want to install VMWare Tools in `/opt/vmware` in our Red Hat VMs, and `/usr/local/vmware` for our Debian VMs. We'll need `RedHat.json` and `Debian.json` files in the `/etc/puppet/hiera/osfamily` directory.

`RedHat.json` should look like this:

```
{
  "vmwaretools::working_dir" : "/opt/vmware"
}
```

`Debian.json` should look like this:

```
{
 "vmwaretools::working_dir" : "/usr/local/vmware"
}
```

That leaves us with one parameter we haven't covered: the `version` parameter. Since we don't need to vary which version of VMWare Tools any of our VMs are using, we can put that in `common.json`, which should now look like this:

```
{
   "vmwaretools::version" : "8.6.5-621624",
   "ntp::restrict" : true,
   "ntp::autoupdate" : true,
   "ntp::enable" : true,
   "ntp::servers" : [
    "grover.example.com iburst",
    "kermit.example.com iburst"
   ]
}
```

Once you've got all that configured, go ahead and test with the Hiera command line tool:

```
$ hiera vmwaretools::working_dir osfamily=RedHat
/opt/vmware

$ hiera vmwaretools::working_dir osfamily=Debian
/usr/local/vmware

$ hiera vmwaretools::version
8.6.5-621624

$ hiera classes virtual=vmware
vmwaretools
```

If everything worked, great. If not, consult the checklist we provided earlier and give it another shot.

Exploring Hiera Further

We hope this walkthrough gave you a good feel for the things you can do with Hiera. There are a few things we didn't touch on, though:

- We didn't discuss how to use Hiera in module manifests, preferring to highlight its ability to provide data to parameterized classes. Hiera also provides a collection of functions that allow you to use Hiera within a module. Tread carefully here, though: Once you start requiring Hiera for your module to work at all, you're reducing its shareability and potentially closing it off to some users.
- We showed how to conduct priority lookups with Hiera; that is, retrieving data from the hierarchy based on the first match for a given key. This is the only way to use Hiera with parameterized classes, but Hiera's lookup functions include special hash and array lookups, allowing you to collect data from sources throughout your hierarchy, or to selectively override your hierarchy's normal precedence. This allows you to declare, for instance, certain base values for all nodes, then layer on additional values for nodes that match differing keys, receiving all the data back as an array or hash.

Hiera 1: Command Line Usage

Hiera provides a command line tool that's useful for verifying that your hierarchy is constructed correctly and that your data sources are returning the values you expect. You'll typically run the Hiera command line tool on a puppet master, mocking up the facts agents would would normally provide the puppet master using a variety of fact sources.

Invocation

The simplest Hiera command takes a single argument — the key to look up — and will look up the key's value using the static data sources in the hierarchy.

```
$ hiera ntp_server
```

A more standard invocation will provide a set of variables for Hiera to use, so that it can also use its dynamic data sources:

```
$ hiera ntp_server --yaml web01.example.com.yaml
```

Order of Arguments

Hiera is sensitive to the position of its command-line arguments:

- The first value is always the key to look up.
- The first argument after the key that does not include an equals sign (`=`) becomes the default value, which Hiera will return if no key is found. Without a default value and in the absence of a matching key from the hierarchy, Hiera returns `nil`.
- Remaining arguments should be `variable=value` pairs.
- Options may be placed anywhere.

Options

Hiera accepts the following command line options:

Argument	Use
`-V`, `--version`	Version information
`-c`, `--config FILE`	Specify an alternate configuration file location
`-d`, `--debug`	Show debugging information
`-a`, `--array`	Return all values as a flattened array of unique values
`-h`, `--hash`	Return all hash values as a merged hash
`-j`, `--json FILE`	JSON file to load scope from
`-y`, `--yaml FILE`	YAML file to load scope from
`-m`, `--mcollective IDENTITY`	Use facts from a node (via mcollective) as scope
`-i`, `--inventory_service IDENTITY`	Use facts from a node (via Puppet's inventory service) as scope

Configuration File Location

The Hiera command line tool looks for its configuration in `/etc/hiera.yaml`. You can use the `--config` argument to specify a different configuration file. See the documentation on Hiera's configuration file for notes on where to find this file depending on your Puppet version and operating system, and consider either reconfiguring Puppet to use `/etc/hiera.yaml` (Puppet 3) or set a symlink to `/etc/hiera.yaml` (Puppet 2.7).

Fact Sources

When used from Puppet, Hiera automatically receives all of the facts it needs. On the command line, you'll need to manually pass it those facts.

You'll typically run the Hiera command line tool on your puppet master node, where it will expect the facts to be either:

- Included on the command line as variables (e.g. `operatingsystem=Debian`)
- Given as a YAML or JSON scope file
- Retrieved on the fly from MCollective data
- Looked up from Puppet's inventory service

Descriptions of these choices are below.

Command Line Variables

Hiera accepts facts from the command line in the form of `variable=value` pairs, e.g. `hiera ntp_server osfamily=Debian clientcert="web01.example.com"`. Variable values must be strings and must be quoted if they contain spaces.

This is useful if the values you're testing only rely on a few facts. It can become unweildy if your hierarchy is large or you need to test values for many nodes at once. In these cases, you should use one of the other options below.

JSON and YAML Scopes

Rather than passing a list of variables to Hiera as command line arguments, you can use JSON and YAML files. You can construct these files yourself, or use a YAML file retrieved from Puppet's cache or generated with `facter --yaml`.

Given this command using command line variable assignments:

```
$ hiera ntp_server osfamily=Debian timezone=CST
```

The following YAML and JSON examples provide equivalent results:

EXAMPLE YAML SCOPE

```
$ hiera ntp_server -y facts.yaml
```

```
# facts.yaml
---
osfamily: Debian
timezone: CST
```

EXAMPLE JSON SCOPE

```
$ hiera ntp_server -j facts.json
```

```
# facts.json
{
  "osfamily" : "Debian",
  "timezone" : "CST"
}
```

MCollective

If you're using hiera on a machine that is allowed to issue MCollective commands, you can ask any node running MCollective to send you its facts. Hiera will then use those facts to drive the lookup.

Example coming soon.

Inventory Service

If you are using Puppet's inventory service, you can query the puppet master for any node's facts. Hiera will then use those facts to drive the lookup.

Example coming soon.

Lookup Types

By default, the Hiera command line tool will use a priority lookup, which returns a single value — the first value found in the hierarchy. There are two other lookup types available: array merge and hash merge.

Array Merge

An array merge lookup assembles a value by merging every value it finds in the hierarchy into a flattened array of unique values. See "Array Merge Lookup" for more details.

Use the `--array` option to do an array merge lookup.

If any of the values found in the data sources are hashes, the `--array` option will cause Hiera to return an error.

Hash

A hash merge lookup assembles a value by merging the top-level keys of each hash it finds in the hierarchy into a single hash. See "Hash Merge Lookup" for more details.

Use the `--hash` option to do a hash merge lookup.

If any of the values found in the data sources are strings or arrays, the `--hash` option will cause Hiera to return an error.

Hiera 1: Writing Custom Backends

Coming Soon