Hiera 1 Manual

(Generated on November 16, 2012, from git revision fed21940866615ff2ef0f2c8ed6b3159932ba595)

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: Complete Example

Coming Soon.

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.

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.

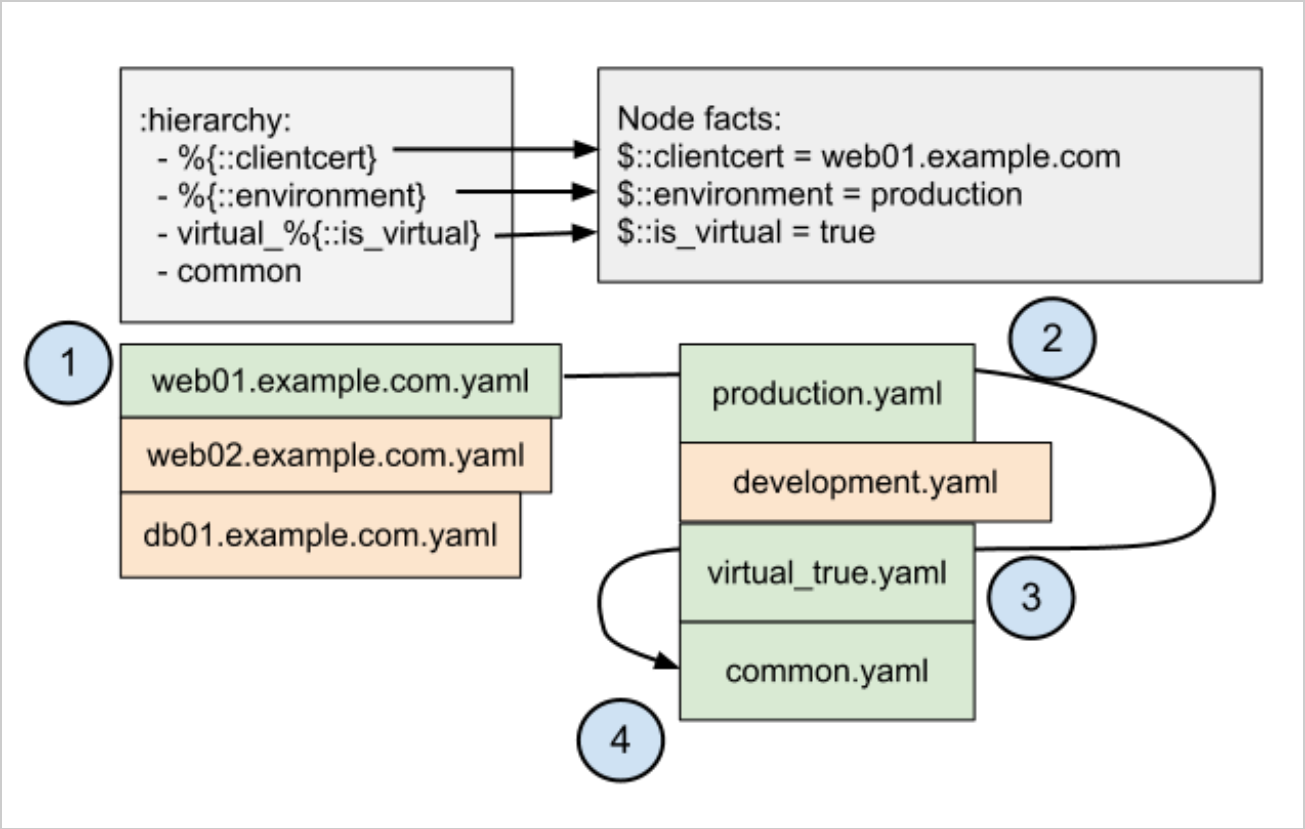
Given two different nodes with different Puppet variables, here are two ways the hierarchy could be interpreted:

web01.example.com

VARIABLES

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

DATA SOURCE RESOLUTION



FINAL HIERARCHY

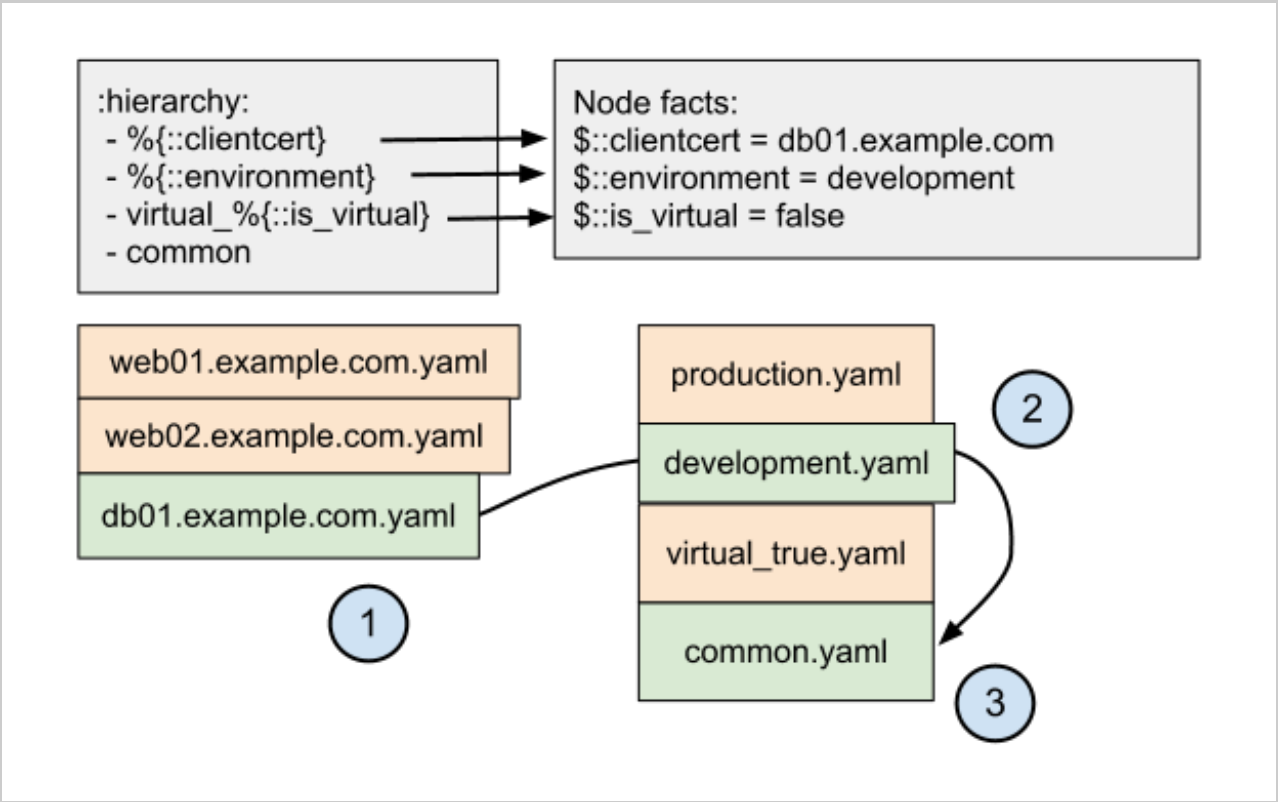
- web01.example.com.yaml
- production.yaml
- virtual_true.yaml
- common.yaml

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: 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` in addition to `true` and `false`)

Any string may include any number of variable interpolation tokens.

Example

Coming soon.

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

Coming soon.

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

Coming Soon

Hiera 1: Command Line Usage

Coming Soon

Hiera 1: Writing Custom Backends

Coming Soon

© 2010 Puppet Labs info@puppetlabs.com 411 NW Park Street / Portland, OR 97209 1-877-575-9775