What you need to know about

Hiera

Authors

- Alessandro Franceschi
- Martin Alfke

Published by

example42

Contents

Hiera is Puppet's builtin key/value data lookup system, where we can store the data we use to configure our system. It has some peculiar characteristics:

- It's **hierarchical**: We can configure different hierarchies of data sources and these are traversed in order to find the value of the desired key, from the level at the top, to the one at the bottom. This is very useful to allow granular configurations of different settings for different groups of servers
- It has a **modular backend** system: data can be stored on different places, according to the used plugins, from simple `Yaml` or `Json` files, to `MongoDb`, `Mysql`, `PostgreSQL`, `Redis` and others to the popular Hiera eyaml backend which uses plain files for data storage and allows encryption of the values of selected keys (typically the ones which contains passwords or secrets)

This allows great flexibility in setting parameters that affect the nodes of our infrastructure.

Automatic parameters lookup

Hiera is important because it allows to assign values to the parameters of Puppet classes, so it's the best method to configure the classes we use according to our needs.

A parameter called `server` of a class called `ntp`, for example, can be evaluated via a lookup of the Hiera key `ntp::server`:

```
class ntp (
  String $server = 'ntp.pool.org'
) { ... }
```

Given the above class we can override the default value for the `server` parameter of the class `ntp` with an Hiera key called `ntp::server`, whatever is the backend used. In case the Yaml backend is used (it's the most popular and the default one) we would, in one of the `.yaml` files used by Hiera an entry as follows:

```
---
ntp::server: 'time.nist.gov'
```

This is useful to cleanly separate our `Puppet code`, where we declare, inside classes, the resources we want to apply to our nodes, from the `data` which defines how these resources should be (according to the parameters of the class where they are contained).

In `Puppet 4.9`, `Hiera version 5` has been introduced and this is the version used in modern Puppet setups.

Hiera configuration: hiera.yaml

`Hiera's` configuration file (`hiera.yaml`) has changed format in version 5, here's the default, which uses the core `Yaml` backend and has only a layer called common:

```
---
version: 5
hierarchy:
  - name: Common              # A level of the hierarchy. They can be more
    path: common.yaml         # The path of the file, under the datadir, w
defaults:
  data_hash: yaml_data        # Use the YAML backend
  datadir: data               # Yaml files are stored in the data dir of o
```

A real world `hiera.yaml` in a setup where the `hiera-eyaml` backend is used, may look as follows:

```
---
version: 5
defaults:
  datadir: data
  data_hash: yaml_data
hierarchy:
  - name: "Eyaml hierarchy"
    lookup_key: eyaml_lookup_key # eyaml backend
    paths:
      - "nodes/%{trusted.certname}.yaml"
      - "role/%{::role}-%{::env}.yaml"
      - "role/%{::role}.yaml"
      - "zone/%{::zone}.yaml"
      - "common.yaml"
    options:
      pkcs7_private_key: /etc/puppetlabs/puppet/keys/private_key.pkcs7.pem
      pkcs7_public_key:  /etc/puppetlabs/puppet/keys/public_key.pkcs7.pem
```

The key infos we can get from it are:

- The `eyaml-backend` is used (`lookup_key: eyaml_lookup_key`)
- It's `public` and `private keys` are stored in the directory `/etc/puppetlabs/puppet/keys/` . They have to be copied there wherever we run Puppet to compile a catalog which uses encrypted data: typically on the `Puppet Server` , on the `Vagrant VMs` where we use Puppet for local testing during development, and eventually also on any node used for CI tests. These keys should not be added to the control-repo git repository as they are used to encrypt and decrypt sensitive data.
- The `Yaml` files containing our Hiera data are placed in the directory data (`datadir: data`) in our `control-repo` / `Puppet environment` directory. So, for example, for production Puppet environment, all the `YAML` files are under `/etc/puppetlabs/code/environments/production/data`
- The hierarchy is based on several `paths` under the `datadir` . Variables are used there (`%{varname}`). Hierarchy goes from the most specific path: `nodes/%{::trusted.certname}.yaml` which refers to a specific node to the most generic (`common.yaml`) which is used as default value for keys which are not set at higher levels.
- The variables used in the hierarchy (here: $::role, $::env, $::zone) should be in the scope when Hiera is used. Commonly they are facts (native or custom) or top scope variables set in Puppet global scope, like the ones set by an External Node Classifier or defined in the main `manifests/site.pp` manifest.

For full reference on the format of `Hiera 5` configuration file, check the Official Documentation

Environment and module data

`Hiera 4` , used from Puppet versions 4.3 to 4.8, introduced the possibility of defining, inside a module, the default values of each class parameter using Hiera.

The actual user data, outside modules, was configured by a global `/etc/puppetlabs/puppet/hiera.yaml` file, which defines Hiera configurations for every `Puppet environment` .

With `Hiera 5` is possible to have environment specific configurations, so we can have a `hiera.yaml` inside a environment directory which may be different for each environment:

```
/etc/puppetlabs/code/environments/$environment_name/hiera.yaml
```

So, for the `production environment` :

```
/etc/puppetlabs/code/environments/production/hiera.yaml
```

This is useful to test hierarchies or backend changes before committing them to the `production environment` .

We can have also per module configurations, so in a `NTP module` , for example, we can have a:

```
$module_path/users/hiera.yaml
```

with the, now familiar, version 5 syntax:

```
---
version: 5

defaults:
  datadir: data
  data_hash: yaml_data

hierarchy:
  - name: "In module hierarchy"
    paths:
      - "%{facts.virtual}.yaml"
      - "%{facts.os.name}-%{facts.os.release.major}.yaml"
      - "%{facts.os.name}.yaml"
      - "%{facts.os.family}-%{facts.os.release.major}.yaml"
      - "%{facts.os.family}.yaml"
      - "common.yaml"
```

this refers `.yaml` files under the `data` directory of the module.

The interesting thing in this is that we have a uniform and common way to lookup for data, across the following three layers with each hierarchy of each layer is used to compose a "super hierarchy" which is traversed seamlessly:

- `global` layer, configured in `/etc/puppetlabs/puppet/hiera.yaml`
- `environment` layer, configured in `/etc/puppetlabs/code/environments/$environment/hiera.yaml`
- `module` layer, configured in `$MODULE_PATH/$module/hiera.yaml` , so, for example, `/etc/puppetlabs/code/environments/$environment/modules/$module/hiera.yaml`

Note that a t the module layer is possible to configure keys in the same module's namespace and usable only by classes of that module. For example, inside the Hiera data of a module called `apache` , we can configure only keys beginning with: `apache::`

Hiera Lookup Options

In the module data is also possible to define the kind of lookup to perform for each class parameter.

Previously the lookup was always a "normal" one: the value returned is the one of the key found the first time while traversing the hierarchy.

Now (actually since `Hiera 4`) it's possible to specify for some parameters alternative lookup methods (for example merging all the values found across the hierarchy for the requested key). This is done in the same data files where we specify our key values, so, for example, in our `$module_path/users/data/common.yaml` we can have:

```
lookup_options:
  # This lookup option applies to parameter 'local' of class 'users'
  users::local:
  # Merge the values found across hierarchies, instead of getting the firs
    merge:
  # Do a deep merge, useful when dealing with Hashes (to override single s
      strategy: deep
      merge_hash_arrays: true
  # This lookup option applies to parameter 'admins' of class 'users'
  users::admins:
    merge:
  # In this case we expect an array and will merge all the values found in
```

```
      strategy: unique
  # It's even possible to define a prefix (here --) to force the removal o
      knockout_prefix: "--"
```

Note that we can use regular expressions when defining specific lookup options for some keys:

```
lookup_options:
  "^profile::(.*)::(.*)_hash$":
    merge:
      strategy: deep
      knockout_prefix: "--"
  "^profile::(.*)::(.*)_list$":
    merge:
      strategy: unique
      knockout_prefix: "--"
```

The lookup command

It's possible to use the `puppet lookup` command to query Hiera for a given key.

If we run this on our `Puppet Server` we can easily find out the value of a given key for the specified node:

```
puppet lookup profiles --node git.lab # Looks for the profiles key on the
```

If we add the `--debug` option we will see a lot of useful information about where and how data is looked for.

We can also use the `lookup()` function inside our `Puppet code`, it replaces (and **deprecates**), the old `hiera()`, `hiera_array()`, `hiera_hash()` and `hiera_include()`.

The general syntax is:

```
lookup( <NAME>, [<VALUE TYPE>], [<MERGE BEHAVIOR>], [<DEFAULT VALUE>] )
```

or

```
lookup( [<NAME>], <OPTIONS HASH> )
```

Some examples follow.

Normal lookup. Same of `hiera('ntp::user')` :

```
lookup('ntp::user')
```

Normal lookup with default. Same of `hiera('ntp::user','root')` :

```
lookup('ntp::user','root')
```

Array lookup, same of `hiera_array('ntp_servers')` :

```
lookup('ntp_servers', Array, 'unique')
```

Deep merge lookup, same of `hiera_hash('users')` with `deep_merge` set to true:

```
lookup('users', Hash, 'deep')
```

Include classes found on Hiera, same of `hiera_include('classes')`

```
lookup('classes', Array[String], 'unique').include
```

All the above examples can be written in an expanded way. In the following example an array is merged across the hierarchies with the option to use the `"--"` prefix to **exclude** specific entries:

```
lookup({
  'name'  => 'ntp_servers',
  'merge' => {
    'strategy'        => 'unique',
    'knockout_prefix' => '--',
```

```
  },
})
```

Check the official reference for all the options available for the lookup function.

Encrypt data with Hiera Eyaml

Hiera-eyaml is a Hiera backend which can be used to encrypt single keys in Hiera `.yaml` files.

It's the most common method to manage `passwords`, `secrets` and `confidential data` in Puppet.

It's now included by default in `Hiera 5`, in earlier versions in can be installed as a gem:

```
gem install hiera-eyaml
```

On the `Puppet Server` we need to do that also in Puppet environment:

```
puppetserver gem install hiera-eyaml
```

To configure it we need to specify the backend in `hiera.yaml` and the location of the keys used to encrypt the data. Syntax for Hiera < 5 is something like:

```
---
:backends:
  - eyaml

:eyaml:
  :datadir: "/etc/puppetlabs/code/environments/%{environment}/hieradata"
  :pkcs7_private_key: /etc/puppetlabs/puppet/keys/private_key.pkcs7.pem
  :pkcs7_public_key:  /etc/puppetlabs/puppet/keys/public_key.pkcs7.pem
  :extension: 'yaml'
```

Syntax for Hiera version 5 is like:

```
---
version: 5
defaults:
  datadir: hieradata
  data_hash: yaml_data
hierarchy:
  - name: "Eyaml hierarchy"
    lookup_key: eyaml_lookup_key # eyaml backend
    paths:
      - "nodes/%{trusted.certname}.yaml"
      - "common.yaml"
    options:
        pkcs7_private_key: "/etc/puppetlabs/puppet/keys/private_key.pkcs7.
        pkcs7_public_key: "/etc/puppetlabs/puppet/keys/public_key.pkcs7.pe
```

The gem provides the `eyaml` command, which can be used to perform any Hiera `eyaml` related operation.

Before starting to encrypt data a pair of `public and private keys` has to be created:

```
eyaml createkeys
```

This creates in the `keys` directory (relative to the current working directory) the `private_key.pkcs7.pem` and `public_key.pkcs7.pem` files. The first one should never be shared and must be managed in a safe way, for this reason the keys (at least the private one) should not be added to the `control-repo` `Git repository` and must be readable by the user running the `Puppet Server` (`/etc/puppetlabs/puppet/keys` is a sane path, but could be anything).

Both of these files must be placed wherever `Hiera files` are evaluated: that means basically all the `Puppet Servers` but also, eventually, on `developers workstations` , if `Puppet code` is tested locally via `Vagrant` .

To avoid the need to share `private keys` to all developers, we recommend, anyway, to avoid to encrypt data in `Hiera files` used by machines running in `Vagrant` .

So for example, if we have a `Hiera layer` which represent a machine environment or tier, and for `Vagrant` nodes we use the `devel tier` , we can override eventually encrypted data in a general `common.yaml` with clear text entries in a `Vagrant` specific layer (like `"tier/devel.yaml"`). Just know that

we need the `private key` when encrypted data is looked for, if we manage to have no encrypted data for servers running under `Vagrant`, Hiera `eyaml` works flawlessly even if the public and private keys are not stored locally.

Creating encrypted Hiera values

We can generate the encrypted value of any Hiera key with the following command:

```
eyaml encrypt -l 'mysql::root_password' -s 'V3ryS3cr3T!'
```

This will print on stdout both the plain and encrypted string and a block of configuration that we can directly copy in our `.yaml` files as follows:

```
---
mysql::root_password: > ENC[PKCS7,MIIBeQYJKoZIhvcNAQcDoIIBajCCAWYCAQAxggEh
```

NOTE: the value is in the format `ENC[PKCS7,Encrypted_Value]`.

Since we have the password stored in plain text in our `bash history`, we should clean it using the following command:

```
history | grep encrypt
572  eyaml encrypt -l 'mysql::root_password' -s 'V3ryS3cr3T!'
history -d 572
```

Alternatively we can directly edit Hiera `.yaml` files with the following command:

```
eyaml edit hieradata/common.eyaml
```

Our editor of preference will open the file and decrypt the encrypted values eventually present so that we can edit our secrets in clear text and save the file again (of course, we can do this only on a machine where we have access to the private key).

To add a new encrypted key to a file we can open it with `eyaml edit filename.eyaml` and add a key with a syntax like this:

```
---
mysql::root_password: DEC::PKCS7[my_password]!
```

The string `my_password` (our password in clear text) will be encrypted once the file is saved.

To show the decrypted content of an `.eyaml` file, we can use the following command:

```
eyaml decrypt -f hieradata/common.eyaml
```

Since `hiera-eyaml` manages both `clear text` and `encrypted` values, we can use it as our only backend if we want to work only on `.yaml` files, so it's pointless to use both `yaml` and `eyaml` backends.

Migrating from Hiera 3 to Hiera 5

As previously described, with version 5 of Hiera three different layers of data are used: global data, environment data and module data each configured by a dedicated hiera.yaml file.

Global Hiera data are the same as they have been with the older Hiera version.

Data in environments allows to stage data and hiera config changes.

Data in modules are a replacement for params pattern and inheritance.

Global Hiera uses a global configuration file which can support the syntax both of version 3 and version 5 must be placed in the Puppet configuration directory (usually `/etc/puppetlabs/puppet/hiera.yaml`), in Hiera 5 this has the highest priority and the values of hierarchies defined here are evaluated before everything else.

The common approach for a migration then is:

- Convert the syntax of the existing global hiera.yaml from version 3 to version 5
- **Move** the global hiera.yaml inside the environment (in the root of the control-repo)

- Move coherently the relevant hiera data directories

Conversion of Hiera configuration file from v3 to v5

In older hiera configuration files backends and hierarchies were separated settings. First we provided an array of used backends and then we listed the hierarchies. Backends were searched in order of occurrence in the configuration file and then the hierarchies got queried for data.

The new hiera config allows to specify backends globally (as default) or on a per hierarchy level.

Let's assume the following existing Hiera config v3 file:

```
# Hiera config v3
:backends:
  - eyaml
  - yaml
:yaml:
  :datadir: "/etc/puppetlabs/code/environments/%{environment}/hieradata"
:eyaml:
  :datadir: "/etc/puppetlabs/code/environments/%{environment}/hieradata"
  :pkcs7_private_key: /etc/puppetlabs/puppet/eyaml/private_key.pkcs7.pem
  :pkcs7_public_key:  /etc/puppetlabs/puppet/eyaml/public_key.pkcs7.pem
:hierarchy:
  - "nodes/%{trusted.certname}"
  - "location/%{facts.whereami}/%{facts.group}"
  - "groups/%{facts.group}"
  - "os/%{facts.os.family}"
  - "common"
:logger: console
:merge_behavior: native
:deep_merge_options: {}
```

With hiera config v5 the `:logger:`, `:merge_behavior:` and `:deep_merge_options:` settings are no longer used and can be removed. Next we have the option to specify default lookup options like `datadir` and the data backend. Afterwards the hierarchies get listed. Single hierarchies can make use of different data backends.

```
# Hiera config v5
version: 5
defaults:
  datadir: data
```

```
  data_hash: yaml_data
hierarchy:
  - name: "Per-node data (yaml version)"
    path: "nodes/%{trusted.certname}.yaml" # Add file extension
    # Omitting datadir and data_hash to use defaults.

  - name: "Per-group secrets"
    path: "groups/%{facts.group}.eyaml"
    lookup_key: eyaml_lookup_key
    options:
      pkcs7_private_key: /etc/puppetlabs/puppet/eyaml/private_key.pkcs7.pe
      pkcs7_public_key:  /etc/puppetlabs/puppet/eyaml/public_key.pkcs7.pem

  - name: "Other YAML hierarchy levels"
    paths: # Can specify an array of paths instead of a single one.
      - "location/%{facts.whereami}/%{facts.group}.yaml"
      - "groups/%{facts.group}.yaml"
      - "os/%{facts.os.family}.yaml"
      - "common.yaml"
```

As we can see it is now possible to group hierarchies which use the same backend. In this special case it is also possible to completely remove the `yaml` backend and simplify the configuration file, as the eyaml backend is also capable of returning unencrypted values:

```
# Hiera config v5 - eyaml only
version: 5
defaults:
  datadir: data
  lookup_key: eyaml_lookup_key
    options:
      pkcs7_private_key: /etc/puppetlabs/puppet/eyaml/private_key.pkcs7.pe
      pkcs7_public_key:  /etc/puppetlabs/puppet/eyaml/public_key.pkcs7.pem

hierarchy:
  - name: "All hierarchies"
    paths:
      - "nodes/%{trusted.certname}.yaml" # Add file extension
      - "location/%{facts.whereami}/%{facts.group}.yaml"
      - "groups/%{facts.group}.yaml"
      - "os/%{facts.os.family}.yaml"
      - "common.yaml"
```

Migrating from hiera*() to lookup() function

The new `lookup` function provides a huge amount of possible usages, here is how it can replace the legacv hiera functions, which are now deprecated.

lookup type	hiera v3	hiera v5 with merge options hash	hiera v5 with Data type, default and merge option
single	hiera('key')	lookup('key')	lookup('key', DataType)
array	hiera_array('array')	lookup('array', {merge => unique})	lookup('array', Array, unique, [])
hash - first found values	hiera_hash('hash')	lookup('hash', {merge => hash})	lookup('hash', Hash, hash, {})
hash - merged values	hiera_hash('hash')	lookup('hash', {merge => deep})	lookup('hash', Hash, deep, {})
include	hiera_include('classes')	lookup('classes', {merge => unique}).include	lookup('classes', Array, unique, []).include

Hiera nested lookups

Hiera lets us run an additional hiera lookup inside of hiera data.

This allows us to specify shared information once only, instead of adding the same information several times:

An example:

```
profile::app::db::auth
  'app1':
    user: 'app1'
    password: '$\.fgeetd'

profile::app::web::auth:
  'app1':
    user: 'app1'
    password: '$\.fgeetd'
```

The "lookup" and "hiera" lookup

Instead of managing the same information on several places, we can ask hiera to run another lookup to fetch the required data:

```
profile::app::db::auth
  'app1':
    user: "%{lookup('app1_user')}"
    password: %{lookup('app1_user_pass')}"

profile::app::web::auth:
  'app1':
    user: "%{lookup('app1_user')}"
    password: %{lookup('app1_user_pass')}"

app1_user: 'app1'
app1_user_pass: '$\.fgeetd'
```

Instead of "lookup" we can also specify "hiera".

Please note that "lookup" and "hiera" nested lookups will only return string based values.

If we specify a non existing key, hiera will fail and return an error.

You can have multiple layers of nested lookups. That means that we run a lookup on a key, which can run another lookup.

Howeverm, we have to be aware that Hiera will fail, if we build loops. Hiera will detect these and return an error.

The "alias" lookup

What if we don't need string, but other data types like boolean, array or hash to be returned by a nested lookup?

In this case we can use the "alias" lookup:

```
profile::app::db::default_packages: "%{alias('default_packages')}"

profile::app::web::default_packages: "%{alias('default_packages')}"

default_packages:
```

```
  - 'tree'
  - 'net-utils'
```

Note, that we can not add additional data to an "alias" lookup.

The "literal" lookup

Consider the situation in which we don't like hiera to interpolate the percent (%) sign.

In this case we can use the "literal" lookup:

```
profile::app::web::server_name_string: "%{literal('%')}{SERVER_NAME}"
```

This will return the value `%{SERVER_NAME}` .

The "scope" lookup

The "scope" function interpolates variables.

The following two examples are identical:

```
profile::app::web:stage: "%{facts.app_stage}"

profile::app::web:stage: "%{scope('facts.app_stage')}"
```

As we also use the simple interpolation, the "scope" lookup is not really needed.

Advanced hierarchies: globs and mapped paths

The release of Hiera 5 has introduced several new features, one of them is the usage of the **glob** and **globs** keys in the hiera.yaml configuration.

Globs

Blogs behave like paths in Hiera configuration, but can be used to specify multiple files:

```
hierarchy:
  - name: "Hiera data"
    glob: "groups/*.yaml"
```

In this case the Hiera lookup is done for *each* yaml file present in the groups directory, parsed in **alphanumerical order**. Note that in the above example no variable interpolation is used, but that's still possible: any fact or variable in the scope can be used in the glob definition.

The Ruby glob method is used to map file paths, so the following rules apply:

- With one asterisk (*) we match any character for a single file.
- With two asterisks (**) any depth of nested directories is matched.
- A question mark (?) matches one character.
- Comma-separated lists in curly braces ({admins,dba}) match any option in the list.
- Sets of characters in square brackets ([abcd]) match any character in the set.
- A backslash () escapes special characters.

Using `globs` instead of `glob` allows us to specify an array of glob patterns, the same logic of `paths` and `path` .

This could be useful for cases where we want to have different users (or machines) to edit independently different files, or when we want to define the parameters for our classes in different places, for sake of order or simplicity.

For example it could be useful to split a file like this in different files, eventually one of each class / profile.

Mapped paths

Another interesting way to define hierarchies is by using **mapped paths** key.

An example:

```
- name: Applications
  mapped_paths: [apps, app, "apps/%{app}.yaml"]
```

The mapped_paths key must have an array as argument with three string elements, in the following order:

- A variable whose value is an array or hash (`apps` in the example)
- A temporary variable name to represent each element of the array or hash. This variable name, (`app` in the example), is used only in the path in this key.
- A path where that temporary variable can be used in interpolation expressions.

With the above example if we had a `$apps` variable containing an array like `['fe','be','db']` Hiera would lookup for data in the following files (relative to the defined `datadir`):

```
apps/fe.yaml
apps/be.yaml
apps/db.yaml
```

What's the use case for mapped paths?

One typical (?) case is when we want to assign to a node one or more *roles*. Usual practice is to have a role and only a role for each node, but in some cases the concept of role has slightly different nuances and we may want to be able to have more than one role (or equivalent concept) in a node.

With the good old `path` key we imply that for each variable used in the hierarchy there can be only one possible value at a time for a node, and that's the value used to identify the path of the file with our data.

Both globs and mapped_paths allow far more flexible hierarchies, with data which may be looked, in the same hierarchy, in different files according to values of variables (with `mapped_paths`) or more general wild card or regexp based matches (with `globs` and `glob`).