Lab 6. Managing data with Hiera

In this lab, you will learn why it's useful to separate your data and code. You will see how to set up Puppet's built-in Hiera mechanism, how to use it to store and query configuration data, including encrypted secrets such as passwords, and how to use Hiera data to create Puppet resources.

The Hiera way

Hiera lets you store your config data in simple text files (actually, YAML, JSON, or HOCON files, which use popular structured text formats), and it looks like the following example:

```
---
  test: 'This is a test'
  consul_node: true
  apache_worker_factor: 100
  apparmor_enabled: true
  ...
```

In your manifest, you query the database using the `lookup()` function, as in the following example (`lookup.pp`):

```
file { lookup('backup_path', String):
  ensure => directory,
}
```

The arguments to `lookup` are the name of the Hiera key you want to retrieve (for example `backup_path`), and the expected data type (for example `String`).

Setting up Hiera

Hiera needs to know one or two things before you can start using it, which are specified in the Hiera configuration file, named `hiera.yaml` (not to be confused this with Hiera data files, which are also YAML files, and we'll find about those later in this lab.) Each Puppet environment has its own local Hiera config file, located at the root of the environment directory (for example, for the `production` environment, the local Hiera config file would be `/etc/puppetlabs/code/environments/production/hiera.yaml`).

Note

Hiera can also use a global config file located at `/etc/puppetlabs/puppet/hiera.yaml` , which takes precedence over the per-environment file, but the Puppet documentation recommends you only use this config layer for certain exceptional purposes, such as temporary overrides; all your normal Hiera data and configuration should live at the environment layer.

The following example shows a minimal `hiera.yaml` file (`hiera_minimal.config.yaml`):

```
---
version: 5

defaults:
  datadir: data
  data_hash: yaml_data

hierarchy:
```

```
  - name: "Common defaults"
    path: "common.yaml"
```

YAML files begin with three dashes and a newline (`---`). This is part of the YAML format, not a Hiera feature; it's the syntax indicating the start of a new YAML document.

The most important setting in the `defaults` section is `datadir`. This tells Hiera in which directory to look for its data files. Conventionally, this is in a `data/` subdirectory of the Puppet manifest directory, but you can change this if you need to.

Adding Hiera data to your Puppet repo

Your VM is already set up with a suitable Hiera config and the sample data file, in the `/etc/puppetlabs/code/environments/pbg` directory. Try it now:

Run the following commands:

```
puppet lookup --environment pbg test
--- This is a test
```

Note

We haven't seen the `--environment` switch before, so it's time to briefly introduce Puppet environments. A Puppet **environment** is a directory containing a Hiera config file, Hiera data, a set of Puppet manifests --- in other words, a complete, self-contained Puppet setup. Each environment lives in a named directory under `/etc/puppetlabs/code/environments`. The default environment is `production`, but you can use any environment you like by giving the `--environment` switch to the `puppet lookup` command. In the example, we are telling Puppet to use the `/etc/puppetlabs/code/environments/pbg` directory.

When you come to add Hiera data to your own Puppet environment, you can use the example `hiera.yaml` and data files as a starting point.

Troubleshooting Hiera

If you don't get the result `This is a test`, your Hiera setup is not working properly. If you see the warning `Config file not found, using Hiera defaults`, check that your lab environment has an `/etc/puppetlabs/code/environments/pbg` directory.

If you see an error like the following, it generally indicates a problem with the Hiera data file syntax:

```
Error: Evaluation Error: Error while evaluating a Function Call,
(/etc/puppetlabs/code/environments/pbg/hiera.yaml): did not find expected key while
parsing a block mapping at line 11 column 5  at line 1:8 on node ubuntu-xenial
```

If this is the case, check the syntax of your Hiera data files.

Querying Hiera

In Puppet manifests, you can use the `lookup()` function to query Hiera for the specified key (you can think of Hiera as a key-value database, where the keys are strings, and values can be any type).

In general, you can use a call to `lookup()` anywhere in your Puppet manifests you might otherwise use a literal value. The following code shows some examples of this (`lookup2.pp`):

```
notice("Apache is set to use ${lookup('apache_worker_factor', Integer)} workers")

unless lookup('apparmor_enabled', Boolean) {
  exec { 'apt-get -y remove apparmor': }
}

notice('dns_allow_query enabled: ', lookup('dns_allow_query', Boolean))
```

To apply this manifest in the example environment, run the following command:

```
puppet apply --environment pbg /examples/lookup2.pp
Notice: Scope(Class[main]): Apache is set to use 100 workers
Notice: Scope(Class[main]): dns_allow_query enabled:  true
```

Typed lookups

As we've seen, `lookup()` takes a second parameter which specifies the expected type of the value to be retrieved. Although this is optional, you should always specify it, to help catch errors. If you accidentally look up the wrong key, or mistype the value in the data file, you'll get an error like this:

```
Error: Evaluation Error: Error while evaluating a Function Call, Found value has wrong
type, expects a Boolean value, got String at /examples/lookup_type.pp:1:8 on node
ubuntu-xenial
```

Types of Hiera data

As we've seen, Hiera data is stored in text files, structured using the format called **YAML Ain't Markup Language**, which is a common way of organizing data. Here's another snippet from our sample Hiera data file, which you'll find at `/etc/puppetlabs/code/environments/pbg/data/common.yaml` on the VM:

```
  syslog_server: '10.170.81.32'
  monitor_ips:
    - '10.179.203.46'
    - '212.100.235.160'
    - '10.181.120.77'
    - '94.236.56.148'
  cobbler_config:
    manage_dhcp: true
    pxe_just_once: true
```

There are actually three different kinds of Hiera data structures present: **single values**, **arrays**, and **hashes**. We'll examine these in detail in a moment.

Single values

Most Hiera data consists of a key associated with a single value, as in the previous example:

```
syslog_server: '10.170.81.32'
```

The value can be any legal Puppet value, such as a String, as in this case, or it can be an Integer:

```
apache_worker_factor: 100
```

Boolean values

You should specify Boolean values in Hiera as either `true` or `false`, without surrounding quotes. However, Hiera is fairly liberal in what it interprets as Boolean values: any of `true`, `on`, or `yes` (with or without quotes) are interpreted as a true value, and `false`, `off`, or `no` are interpreted as a false value. For clarity, though, stick to the following format:

```
consul_node: true
```

When you use `lookup()` to return a Boolean value in your Puppet code, you can use it as the conditional expression in, for example, an `if` statement:

```
if lookup('is_production', Boolean) {
  ...
```

Arrays

Usefully, Hiera can also store an array of values associated with a single key:

```
monitor_ips:
  - '10.179.203.46'
  - '212.100.235.160'
  - '10.181.120.77'
  - '94.236.56.148'
```

The key (`monitor_ips`) is followed by a list of values, each on its own line and preceded by a hyphen (`-`). When you call `lookup('monitor_ips', Array)` in your code, the values will be returned as a Puppet array.

Hashes

A hash (also called a **dictionary** in some programming languages) is like an array where each value has an identifying name (called the **key**), as in the following example:

```
cobbler_config:
  manage_dhcp: true
  pxe_just_once: true
```

Each key-value pair in the hash is listed, indented on its own line. The `cobbler_config` hash has two keys, `manage_dhcp` and `pxe_just_once`. The value associated with each of those keys is `true`.

When you call `lookup('cobbler_config', Hash)` in a manifest, the data will be returned as a Puppet hash, and you can reference individual values in it using the normal Puppet hash syntax, as we saw in Lab 5 (`lookup_hash.pp`):

```
$cobbler_config = lookup('cobbler_config', Hash)
$manage_dhcp = $cobbler_config['manage_dhcp']
$pxe_just_once = $cobbler_config['pxe_just_once']
if $pxe_just_once {
  notice('pxe_just_once is enabled')
} else {
  notice('pxe_just_once is disabled')
}
```

Since it's very common for Hiera data to be a hash of hashes, you can retrieve values from several levels down in a hash by using the following "dot notation" (`lookup_hash_dot.pp`):

```
$web_root = lookup('cms_parameters.static.web_root', String)
notice("web_root is ${web_root}")
```

Interpolation in Hiera data

Hiera data is not restricted to literal values; it can also include the value of Facter facts or Puppet variables, as in the following example:

```
 backup_path: "/backup/%{facts.hostname}"
```

Anything within the `%{}` delimiters inside a quoted string is evaluated and interpolated by Hiera. Here, we're using the dot notation to reference a value inside the `$facts` hash.

Using lookup()

Helpfully, you can also interpolate Hiera data in Hiera data, by using the `lookup()` function as part of the value. This can save you repeating the same value many times, and can make your data more readable, as in the following example (also from `hiera_sample.yaml`):

```
ips:
  home: '130.190.0.1'
  office1: '74.12.203.14'
  office2: '95.170.0.75'
firewall_allow_list:
  - "%{lookup('ips.home')}"
  - "%{lookup('ips.office1')}"
  - "%{lookup('ips.office2')}"
```

This is much more readable than simply listing a set of IP addresses with no indication of what they represent, and it prevents you accidentally introducing errors by updating a value in one place but not another. Use Hiera interpolation to make your data self-documenting.

Using alias()

When you use the `lookup()` function in a Hiera string value, the result is always a string. This is fine if you're working with string data, or if you want to interpolate a Hiera value into a string containing other text. However, if you're working with arrays, hashes, or Boolean values, you need to use the `alias()` function instead. This lets you re-use any Hiera data structure within Hiera, just by referencing its name:

```
firewall_allow_list:
  - "%{lookup('ips.home')}"
  - "%{lookup('ips.office1')}"
  - "%{lookup('ips.office2')}"
vpn_allow_list: "%{alias('firewall_allow_list')}"
```

Don't be fooled by the surrounding quotes: it may look as though `vpn_allow_list` will be a string value, but because we are using `alias()` , it will actually be an array, just like the value it is aliasing (`firewall_allow_list`).

Using literal()

Because the percent character (`%`) tells Hiera to interpolate a value, you might be wondering how to specify a literal percent sign in data. For example, Apache uses the percent sign in its configuration to refer to variable names like `%{HTTP_HOST}` . To write values like these in Hiera data, we need to use the `literal()` function, which exists only to refer to a literal percent character. For example, to write the value `%{HTTP_HOST}` as Hiera data, we would need to write:

```
%{literal('%')}{HTTP_HOST}
```

You can see a more complicated example in the sample Hiera data file:

```
force_www_rewrite:
  comment: "Force WWW"
  rewrite_cond: "%{literal('%')}{HTTP_HOST} !^www\\. [NC]"
  rewrite_rule: "^(.*)$ https://www.%{literal('%')}{HTTP_HOST}%{literal('%')}
{REQUEST_URI} [R=301,L]"
```

The hierarchy

So far, we've only used a single Hiera data source (`common.yaml`). Actually, you can have as many data sources as you like. Each usually corresponds to a YAML file, and they are listed in the `hierarchy` section of the `hiera.yaml` file, with the highest-priority source first and the lowest last:

```
hierarchy:
  ...
  - name: "Host-specific data"
    path: "nodes/%{facts.hostname}.yaml"
  - name: "OS release-specific data"
    path: "os/%{facts.os.release.major}.yaml"
  - name: "OS distro-specific data"
    path: "os/%{facts.os.distro.codename}.yaml"
  - name: "Common defaults"
    path: "common.yaml"
```

In general, though, you should keep as much data as possible in the `common.yaml` file, simply because it's easier to find and maintain data if it's in one place, rather than scattered through several files.

Dealing with multiple values

You may be wondering what happens if the same key is listed in more than one Hiera data source. For example, imagine the first source contains the following:

```
consul_node: false
```

Also, assume that `common.yaml` contains:

```
consul_node: true
```

What happens when you call `lookup('consul_node', Boolean)` with this data? There are two different values for `consul_node` in two different files, so which one does Hiera return?

The answer is that Hiera searches data sources in the order they are listed in the `hierarchy` section; that is to say, in priority order. It returns the first value found, so if there are multiple values, only the value from the first---that is, highest-priority--- data source will be returned (that's the "hierarchy" part).

Merge behaviors

We said in the previous section that if there is more than one value matching the specified key, the first matching data source takes priority over the others. This is the default behavior, and this is what you'll usually want. However, sometimes you may want `lookup()` to return the union of all the matching values found, throughout the hierarchy. Hiera allows you to specify which of these strategies it should use when multiple values match your lookup.

This is called a **merge behavior**, and you can specify which merge behavior you want as the third argument to `lookup()`, after the key and data type (`lookup_merge.pp`):

```
notice(lookup('firewall_allow_list', Array, 'unique'))
```

The default merge behavior is called `first`, and it returns only one value, the first found value. By contrast, the `unique` merge behavior returns all the values found, as a flattened array, with duplicates removed (hence `unique`).

Data sources based on facts

The hierarchy mechanism lets you set common default values for all situations (usually in `common.yaml`), but override them in specific circumstances. For example, you can set a data source in the hierarchy based on the value of a Puppet fact, such as the hostname:

```
  - name: "Host-specific data"
    path: "nodes/%{facts.hostname}.yaml"
```

Hiera will look up the value of the specified fact and search for a data file with that name in the `nodes/` directory. In the previous example, if the node's hostname is `web1`, Hiera will look for the data file `nodes/web1.yaml` in the Hiera data directory. If this file exists and contains the specified Hiera key, the `web1` node will receive that value for its lookup, while other nodes will get the default value from `common`.

Note

Note that you can organize your Hiera data files in subdirectories under the main `data/` directory if you like, such as `data/nodes/`.

Another useful fact to reference in the hierarchy is the operating system major version or codename. This is very useful when you need your manifest to work on more than one release of the operating system. If you have more than a handful of nodes, migrating to the latest OS release is usually a gradual process, upgrading one node at a time. If something has changed from one version to the next that affects your Puppet manifest, you can use the `os.distro.codename` fact to select the appropriate Hiera data, as in the following example:

```
  - name: "OS-specific data"
    path: "os/%{facts.os.distro.codename}.yaml"
```

Alternatively, you can use the `os.release.major` fact:

```
  - name: "OS-specific data"
    path: "os/%{facts.os.release.major}.yaml"
```

For example, if your node is running Ubuntu 16.04 Xenial, Hiera will look for a data file named `os/xenial.yaml` (if you're using `os.distro.codename`) or `os/16.04.yaml` (if you're using `os.release.major`) in the Hiera data directory.

Creating resources with Hiera data

When we started working with Puppet, we created resources directly in the manifest using literal attribute values. In this lab, we've seen how to use Hiera data to fill in the title and attributes of resources in the manifest. We can now take this idea one step further and create resources **directly from Hiera** queries. The advantage of this method is that we can create any number of resources of any type, based purely on data.

Building resources from Hiera arrays

In Lab 5, we learned how to use Puppet's `each` function to iterate over an array or hash, creating resources as we go. Let's apply this technique to some Hiera data. In our first example, we'll create some user resources from a Hiera array.

Run the following command:

```
puppet apply --environment pbg /examples/hiera_users.pp
Notice: /Stage[main]/Main/User[katy]/ensure: created
Notice: /Stage[main]/Main/User[lark]/ensure: created
Notice: /Stage[main]/Main/User[bridget]/ensure: created
Notice: /Stage[main]/Main/User[hsing-hui]/ensure: created
Notice: /Stage[main]/Main/User[charles]/ensure: created
```

Here's the data we're using (from the `/etc/puppetlabs/code/environments/pbg/data/common.yaml` file):

```
users:
  - 'katy'
  - 'lark'
  - 'bridget'
  - 'hsing-hui'
  - 'charles'
```

And here's the code which reads it and creates the corresponding user instances (`hiera_users.pp`):

```
lookup('users', Array[String]).each | String $username | {
  user { $username:
    ensure => present,
  }
}
```

Combining Hiera data with resource iteration is a powerful idea. This short manifest could manage all the users in your infrastructure, without you ever having to edit the Puppet code to make changes. To add new users, you need only edit the Hiera data.

Building resources from Hiera hashes

Of course, real life is never quite as simple as a programming language example. If you were really managing users with Hiera data in this way, you'd need to include more data than just their names: you'd need to be able to manage shells, UIDs, and so on, and you'd also need to be able to remove the users if necessary. To do that, we will need to add some structure to the Hiera data.

Run the following command:

```
puppet apply --environment pbg /examples/hiera_users2.pp
Notice: Compiled catalog for ubuntu-xenial in environment pbg in 0.05 seconds
Notice: /Stage[main]/Main/User[katy]/uid: uid changed 1001 to 1900
Notice: /Stage[main]/Main/User[katy]/shell: shell changed '' to '/bin/bash'
Notice: /Stage[main]/Main/User[lark]/uid: uid changed 1002 to 1901
Notice: /Stage[main]/Main/User[lark]/shell: shell changed '' to '/bin/sh'
Notice: /Stage[main]/Main/User[bridget]/uid: uid changed 1003 to 1902
Notice: /Stage[main]/Main/User[bridget]/shell: shell changed '' to '/bin/bash'
Notice: /Stage[main]/Main/User[hsing-hui]/uid: uid changed 1004 to 1903
Notice: /Stage[main]/Main/User[hsing-hui]/shell: shell changed '' to '/bin/sh'
Notice: /Stage[main]/Main/User[charles]/uid: uid changed 1005 to 1904
Notice: /Stage[main]/Main/User[charles]/shell: shell changed '' to '/bin/bash'
Notice: Applied catalog in 0.17 seconds
```

The first difference from the previous example is that instead of the data being a simple array, it's a hash of hashes:

```
users2:
  'katy':
    ensure: present
    uid: 1900
    shell: '/bin/bash'
  'lark':
    ensure: present
    uid: 1901
    shell: '/bin/sh'
  'bridget':
    ensure: present
    uid: 1902
    shell: '/bin/bash'
  'hsing-hui':
    ensure: present
    uid: 1903
    shell: '/bin/sh'
  'charles':
    ensure: present
    uid: 1904
    shell: '/bin/bash'
```

Here's the code which processes that data (`hiera_users2.pp`):

```
lookup('users2', Hash, 'hash').each | String $username, Hash $attrs | {
  user { $username:
    * => $attrs,
  }
}
```

Each of the keys in the `users2` hash is a username, and each value is a hash of user attributes such as `uid` and `shell` .

When we call `each` on this hash, we specify two parameters to the loop instead of one:

```
| String $username, Hash $attrs |
```

As we saw in Lab 5, when iterating over a hash, these two parameters receive the hash key and its value, respectively.

Inside the loop, we create a user resource for each element of the hash:

```
user { $username:
  * => $attrs,
}
```

You may recall from the previous lab that the `*` operator (the attribute splat operator) tells Puppet to treat `$attrs` as a hash of attribute-value pairs. So the first time round the loop, with user `katy`, Puppet will create a user resource equivalent to the following manifest:

```
user { 'katy':
  ensure => present,
  uid    => 1900,
  shell  => '/bin/bash',
}
```

Every time we go round the loop with the next element of `users`, Puppet will create another user resource with the specified attributes.

Summary

In this lab we've outlined some of the problems with maintaining configuration data in Puppet manifests, and introduced Hiera as a powerful solution. We've seen how to configure Puppet to use the Hiera data store, and how to query Hiera keys in Puppet manifests using `lookup()`.

We've explored using Hiera data to create resources, using an `each` loop over an array or hash.

In the next lab, we'll look at how to find and use public modules from Puppet Forge; how to use public modules to manage software including Apache, MySQL, and archive files; how to use the `r10k` tool to deploy and manage third-party modules; and how to write and structure your own modules.