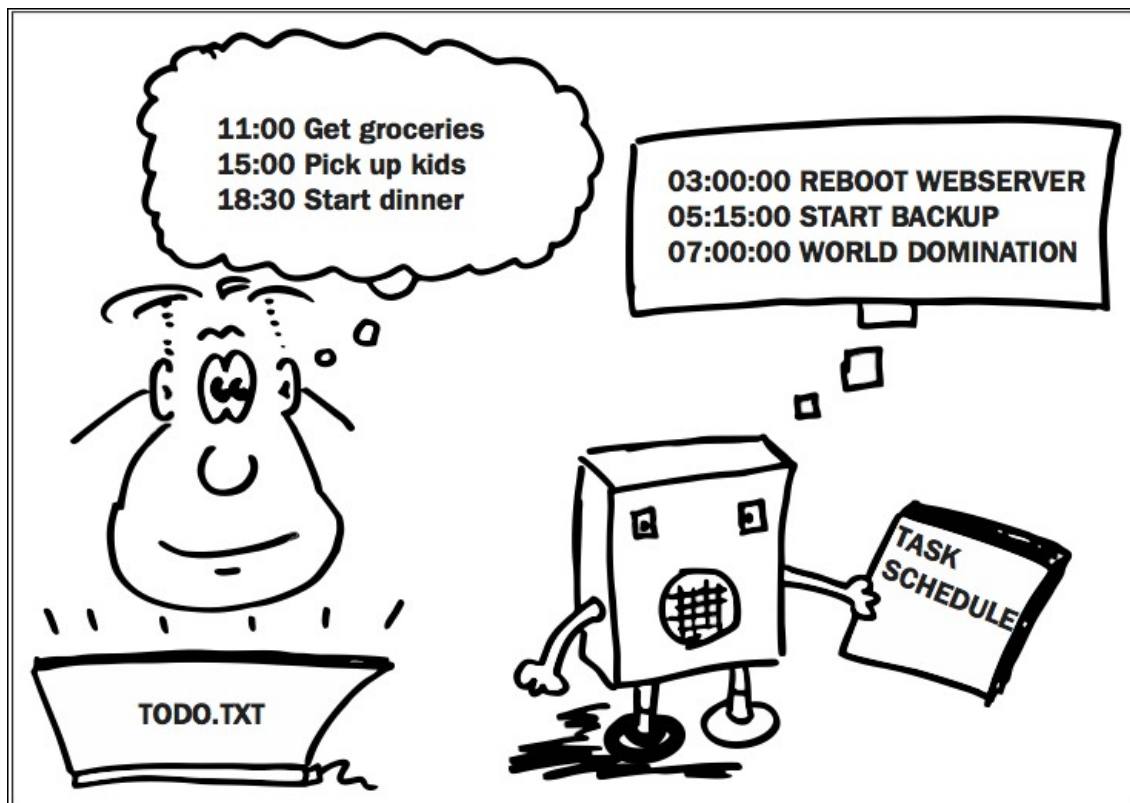


Lab 9. Managing files with templates



In this lab, we'll learn about an important and powerful feature of Puppet: the **template**. We'll see how to use a simple template to interpolate the values of Puppet variables, facts, and Hiera data into a file, and we'll also introduce more complex templates using iteration and conditional statements to generate dynamic configuration files.



What are templates?

In previous labs, we've used Puppet to manage the **contents of files** on the node by various means, including setting the contents to a literal string using the `content` attribute, and copying a file from a Puppet module using the `source` attribute. While these methods are very useful, they are limited in one respect: they can only use **static text**, rather than building the contents of the file dynamically, based on Puppet data.

The dynamic data problem

To see why this is a problem, consider a common Puppet file management task such as a backup script. There are a number of site- and node-specific things the backup script needs to know: the local directories to back up, the destination to copy them to, and any credentials needed to access the backup storage. While we could insert these into the script as literal values, this is rather inflexible. We might have to maintain several versions of the script, each identical to the others except for a backup location, for example. This is clearly less than satisfactory.

Consider a configuration file for an application where some of the settings depend on specific information about the node: the available memory, perhaps. Obviously, we don't want to have to maintain multiple versions of an almost identical config file, each containing a suitable value for all the different sizes of memory we may come across. We have a way of obtaining that information directly in Puppet.

Puppet template syntax

Puppet's **template** mechanism is one way to achieve this. A template is simply an ordinary text file, containing special placeholder markers which Puppet will replace with the relevant data values. The following example shows what these markers look like (`aws_credentials.epp`):

```
aws_access_key_id = <%= $aws_access_key %>
```

Everything outside the `<%=` and `%>` delimiters is literal text and will be rendered as-is by Puppet.

The text inside the delimiters, however, is interpreted as a Puppet expression (in this case, just the variable `$aws_access_key`), which will be evaluated when the template is compiled, and the result will be interpolated into the text.

For example, if the variable `$aws_access_key` has the value `AKIAIAF7V6N2PTOIZVA2` , then when the template is processed by Puppet the resulting output text will look like the following:

```
aws_access_key_id = AKIAIAF7V6N2PTOIZVA2
```

You can have as many of these delimited expressions (called **tags**) in the template as you like, and they will all be evaluated and interpolated when the template is used.

Puppet's template mechanism is called **EPP** (for **Embedded Puppet**), and template files have the extension `.epp` .

Using templates in your manifests

Since the end result of a template is a file, you won't be surprised that we use Puppet's `file` resource to work with templates. In fact, we use an attribute of the `file` resource that you've seen before: the `content` attribute.

Referencing template files

Recall from [Lab 2], [*Creating your first manifests*], that you can use the `content` attribute to set a file's contents to a literal string:

```
file { '/tmp/hello.txt':  
  content => "hello, world\n",  
}
```

And, of course, you can interpolate the value of Puppet expressions into that string:

```
file { "/usr/local/bin/${task}":  
  content => "echo I am ${task}\n",  
  mode    => '0755',  
}
```

So far, so familiar, but we can take one further step and replace the literal string with a call to the `epp()` function (`file_epp.pp`):

```
file { '/usr/local/bin/backup':  
  content => epp('/examples/backup.sh.epp',  
    {  
      'data_dir' => '/examples',  
    }  
  )  
}
```

```

),
mode    => '0755',
}

```

Puppet will compile the template file referenced by `backup.sh.epp`, replacing any tags with the value of their expressions, and the resulting text will be written to the file `/usr/local/bin/backup`. The template file might look something like the following (`backup.sh.epp`):

```

<%- | String $data_dir | -%>
#!/bin/bash
mkdir -p /backup
tar cvzf /backup/backup.tar.gz <%= $data_dir %>

```

You can use the `epp()` function anywhere a string is expected, but it's most common to use it to manage a file, as shown in the example.

To reference a template file from within a module (for example, in our NTP module from [Lab 7], [Mastering modules]), put the file in the `modules/pbg_ntp/templates/` directory, and prefix the filename with `pbg_ntp/`, as in the following example:

```

file { '/etc/ntp.conf':
  content => epp('pbg_ntp/ntp.conf.epp'),
}

```

Note

Remember

Don't include `templates/` as part of the path. Puppet knows it's a template, so it will automatically look in the `templates/` directory of the named module.

Inline templates

Your template text need not be in a separate file: if it's a short template, you can put it in a literal string in your Puppet manifest and use the `inline_epp()` function to compile it (`file_inline_epp.pp`):

```

$web_root = '/var/www'
$backup_dir = '/backup/www'

file { '/usr/local/bin/backup':
  content => inline_epp('rsync -a <%= $web_root %>/ <%= $backup_dir %>/'),
  mode    => '0755',
}

```

Note that we used a **single-quoted string** to specify the inline template text. If we'd used a double-quoted string, Puppet would have interpolated the values of `$web_root` and `$backup_dir` [before] processing the template, which is not what we want.

In general, though, it's better and more readable to use a separate template file for all but the simplest templates.

Template tags

The tag we've been using in the examples so far in this lab is known as an **expression-printing tag**:

```
<%= $aws_access_key %>
```

Puppet expects the contents of this tag to have a value, which will then be inserted into the template in place of the tag.

A **non-printing tag** is very similar, but will not generate any output. It has no `=` sign in the opening delimiter:

```
<% notice("This has no effect on the template output") %>
```

You can also use a **comment tag** to add text which will be removed when Puppet compiles the template:

```
<%# This is a comment, and it will not appear in the output of the template %>
```

Computations in templates

So far, we've simply interpolated the value of a variable into our template, but we can do more. Template tags can contain any valid Puppet expression.

It's very common for certain values in config files to be **computed** from other values, such as the amount of physical memory on the node. We saw an example of this in Lab 5, where we computed a config value based on the value of `$facts['memory']['system']['total_bytes']`.

Naturally, whatever we can do in Puppet code, we can also do in a template, so here's the same computation in template form (`template_compute.epp`):

```
innodb_buffer_pool_size=<%= $facts['memory']['system']['total_bytes'] * 3/4 %>
```

The generated output (on my lab environment) is as follows:

```
puppet epp render --environment pbj /examples/template_compute.epp
innodb_buffer_pool_size=780257280
```

You're not restricted to numerical computations; you can do anything a Puppet expression can do, including string manipulation, array and hash lookups, fact references, function calls, and so on.

Conditional statements in templates

You might not be very impressed with templates so far, pointing out that you can already interpolate the values of Puppet expressions in strings, and hence files, without using a template. That said, templates allow you to interpolate data into much bigger files than it would be practical or desirable to create with a literal string in your Puppet manifest.

Templates also allow you to do something else very useful: **include or exclude sections of text** based on the result of some Puppet conditional expression.

We've already met conditional statements in manifests in [Lab 5], [*Variables, expressions, and facts*], where we used them to conditionally include sets of Puppet resources (`if.pp`):

```
if $install_perl {
    ...
} else {
    ...
}
```

Since the content of template tags is just Puppet code, you can use an `if` statement in a template too. Here's a similar example to the previous one, but this time controlling inclusion of a block of configuration in a template (`template_if.epp`):

```
<% if $ssl_enabled { -%>
  ## SSL directives
  SSLEngine on
  SSLCertificateFile      "<%= $ssl_cert %>"
  SSLCertificateKeyFile   "<%= $ssl_key %>"
  ...
<% } -%>
```

This looks a little more complicated, but it's actually exactly the same logic as in the previous example. We have an `if` statement which tests the value of a Boolean variable, `$ssl_enabled`, and depending on the result, the following block is either included or excluded.

You can see that the `if` statement and the closing `}` are enclosed in non-printing tags, so they generate no output themselves, and as Puppet compiles the template, it will execute the Puppet code within the tags and that will determine the output. If `$ssl_enabled` is true, the file generated by the template will contain the following:

```
## SSL directives
SSLEngine on
SSLCertificateFile      "<%= $ssl_cert %>"
SSLCertificateKeyFile   "<%= $ssl_key %>"
...
```

Otherwise, this part of the template will be omitted. This is a very useful way of conditionally including blocks in a configuration file.

Just as with `if` statements in manifest files, you can also use `else` to include an alternative block instead, if the conditional statement is false.

Iteration in templates

Iterating over Facter data

Our first example generates part of the config file for an application which captures network packet data. To tell it which interfaces to listen on, we need to generate a list of all the live network interfaces on the node.

How can we generate this output? We know Facter can give us a list of all the network interfaces available, with `$facts['networking']['interfaces']`. This is actually a hash, where the key is the name of the interface, and the value is a hash of the interface's attributes, such as the IP address and netmask.

You may recall from [Lab 5], *[Variables, expressions, and facts]* that in order to iterate over a hash, we use a syntax like the following:

```
HASH.each | KEY, VALUE | {
  BLOCK
}
```

So let's apply this pattern to the Facter data and see what the output looks like (`template_iterate.epp`):

```
<% $facts['networking']['interfaces'].each |String $interface, Hash $attrs| { -%>
interface <%= $interface %>;
```

```
<% } -%>
```

Each time round the loop, the values of `$interface` and `$attrs` will be set to the next key and value of the hash returned by `$facts['networking']['interfaces']`. As it happens, we will not be using the value of `$attrs`, but we still need to declare it as part of the loop syntax.

Each time round the loop, the value of `$interface` is set to the name of the next interface in the list, and a new output line like the following is generated:

```
interface em1;
```

At the end of the loop, we have generated as many output lines as we have interfaces, which is the desired result. Here's the final output, on a node with lots of network interfaces:

```
interface em1;
interface em2;
interface em3;
interface em4;
interface em5;
interface lo;
```

Iterating over structured facts

The next configuration data required for our application is a list of IP addresses associated with the node, which we can generate in a similar way to the previous example.

We can use more or less the same Puppet code as in the previous example, only this time we will be using each interface's `$attrs` hash to get the IP address of the associated interface.

The following example shows how this works (`template_iterate2.epp`):

```
<% $facts['networking']['interfaces'].each |String $interface, Hash $attrs| { -%>
  local_address <%= $attrs['bindings'][0]['address'] %>;
<% } -%>
```

The loop is the same as in the previous example, but this time each output line contains, not the value of `$interface`, but the value of `$attrs['bindings'][0]['address']`, which contains the IP address of each interface.

Here's the final output:

```
local_address 10.170.81.11;
local_address 75.76.222.21;
local_address 204.152.248.213;
local_address 66.32.100.81;
local_address 189.183.255.6;
local_address 127.0.0.1;
```

Iterating over Hieradata

In [Lab 6], *[Managing data with Hieradata]* we used a Hieradata array of users to generate Puppet resources for each user. Let's use the same Hieradata data now to build a dynamic configuration file using iteration in a template.

The SSH daemon `ssh` can be configured to allow SSH access only by a list of named users (with the `AllowUsers` directive), and, indeed, it's good practice to do this.

Note

Security tip

Most servers accessible from the public Internet regularly receive brute-force login attempts for random usernames, and dealing with these can use up a lot of resources. If `ssh` is configured to allow only specified users, it can quickly reject any users not in this list, without having to process the request further.

If our users are listed in Hiera, then it's easy to use a template to generate this `AllowUsers` list for the `sshd_config` file.

Just as we did when generating Puppet `user` resources, we will make a call to `lookup()` to get the array of users, and iterate over this using `each`. The following example shows what this looks like in the template (`template_hiera.epp`):

```
AllowUsers<% lookup('users').each | $user | { -%>
  <%= $user -%>
<% } %>
```

Note the leading space on the second line, which results in the usernames in the output being space-separated. Note also the use of the leading hyphen to the closing tag (`-%>`) which, as we saw earlier in the lab, will suppress any trailing whitespace on the line.

Here's the result:

```
AllowUsers katy lark bridget hsing-hui charles
```

Passing parameters to templates

To declare parameters for a template, list them between pipe characters (`|`) inside a non-printing tag, as shown in the following example (`template_params.epp`):

```
<% | String[1] $aws_access_key,
      String[1] $aws_secret_key,
| -%>
aws_access_key_id = <%= $aws_access_key %>
aws_secret_access_key = <%= $aws_secret_key %>
```

When you declare parameters in a template, you must pass those parameters explicitly, in hash form, as the second argument to the `epp()` function call. The following example shows how to do this (`epp_params.pp`):

```
file { ['/root/aws_credentials']:
  content => epp('/examples/template_params.epp',
    {
      'aws_access_key' => 'AKIAIAF7V6N2PTOIZVA2',
      'aws_secret_key' => '7IBpXjoYRVbJ/rCTVLaAMyud+i4col1lVt1Df1vt',
    }
  ),
}
```

This form of the `epp()` function call takes two parameters: the path to the template file, and a hash containing all the required template parameters. The keys to the hash are the parameter names, and the values are the values. (These need not be literal values; they could be Hieradata lookups, for example.)

It's very likely that you will be using Hieradata in templates, and although in our previous `AllowUsers` example we called `lookup()` directly from the template to look up the data, this isn't really the best way to do it. Now that we know how to declare and pass parameters to templates, we should do the same thing with Hieradata.

Here is an updated version of the `AllowUsers` example where we do the Hieradata lookup in the manifest, as part of the `epp()` call. First, we need to declare a `$users` parameter in the template (`template_hiera_params.epp`):

```
<% | Array[String] $users | -%>
AllowUsers<% $users.each | $user | { -%>
  <%= $user -%>
<% } %>
```

Then, when we compile the template with `epp()`, we pass in the Hieradata by calling `lookup()` in the parameters hash (`epp_hiera.pp`):

```
file { ['/tmp/sshd_config_example':
  content => epp('/examples/template_hiera_params.epp',
    {
      'users' => lookup('users'),
    }
  ),
}
```

If you have declared a parameter list in the template, you must pass it exactly those parameters in the `epp()` call, and no others. EPP templates declare parameters in the same way as classes do: parameters can be given default values, and any parameter without a default value is mandatory.

Best practices

Use EPP templates for dynamically-generated files, declare typed parameters in the template, and pass those parameters as a hash to the `epp()` function. To make your template code easier to understand and maintain, always pass data explicitly to the template. If the template needs to look up Hieradata, do the lookup in your Puppet manifest and have the template declare a parameter to receive the data.

Validating template syntax

We've seen in this lab that templates can contain complex logic and iteration that can generate almost any output required. The downside of this power and flexibility is that it can be difficult to read and debug template code.

Fortunately, Puppet includes a tool to check and validate your templates on the command line: `puppet epp validate`. To use it, run the following command against your template file:

```
puppet epp validate /examples/template_params.epp
```

If there is no output, the template is valid. If the template contains an error, you will see an error message, something like the following:

```
Error: Syntax error at '%' at /examples/template_params.epp:3:4
Error: Errors while validating epp
```



```
Error: Try 'puppet help epp validate' for usage
```

Rendering templates on the command line

As any programmer knows, even programs with valid syntax don't necessarily produce the correct results. It can be very useful to see exactly what output the template is going to produce, and Puppet also provides a tool to do this:

```
puppet epp render .
```

To use it, run the following command:

```
puppet epp render --values "{ 'aws_access_key' => 'foo', 'aws_secret_key' => 'bar' }"
/examples/template_params.epp
aws_access_key_id = foo
aws_secret_access_key = bar
```

The `--values` argument allows you to pass in a hash of parameter-value pairs, just as you would when calling the `epp()` function in your Puppet manifest.

Alternatively, you can use the `--values_file` argument to reference a Puppet manifest file containing the hash of parameters:

```
echo "{ 'aws_access_key' => 'foo', 'aws_secret_key' => 'bar' }" >params.pp
puppet epp render --values_file params.pp /examples/template_params.epp
aws_access_key_id = foo
aws_secret_access_key = bar
```

You can pass parameters both on the command line with `--values`, and from a file with `--values_file`, simultaneously. Parameters given on the command line will take priority over those from the file:

```
puppet epp render --values_file params.pp --values "{ 'aws_access_key' => 'override'
}" /examples/template_params.epp
aws_access_key_id = override
aws_secret_access_key = bar
```

You can also use `puppet epp render` to test inline template code, using the `-e` switch to pass in a literal template string:

```
puppet epp render --values "{ 'name' => 'Dave' }" -e 'Hello, <%= $name %>'
Hello, Dave
```

Just as when testing your manifests, you can also use `puppet apply` to test your templates directly, using a command similar to the following:

```
puppet apply -e "file { ['/tmp/result': content =>
epp('/examples/template_iterate.epp')}"
```

One advantage of this approach is that all Puppet variables, facts, and Hieradata will be available to your template.

Legacy ERB templates

You'll probably come across references to a different type of Puppet template in older code and documentation: the **ERB template**. ERB (for Embedded Ruby) was the only template mechanism provided in Puppet up until version 3.5, when EPP support was added, and EPP has now replaced ERB as Puppet's default template format.

ERB template syntax looks quite similar to EPP. The following example is a snippet from an ERB template:

```
AllowUsers <%= @users.join(' ') %><%= scope['::ubuntu'] == 'yes' ? ',ubuntu' : '' %>
```

The difference is that the template language inside the tags, is Ruby, not Puppet. Early versions of Puppet were rather limited in language features (for example, there was no `each` function to iterate over variables), so it was common to use Ruby code embedded in templates to work around this.

This required some complicated plumbing to manage the interface between Puppet and Ruby; for example, accessing variables in non-local scope in ERB templates requires the use of the `scope` hash, as in the previous example. Similarly, in order to access Puppet functions such as `strftime()`, you have to call:

```
scope.call_function('strftime', ...)
```

ERB templates also do not support declared parameters or type checking. It is recommend to use only EPP templates in your own code.

Summary

In this lab we've looked at one of the most powerful tools in Puppet's toolbox, the template file. We've examined the EPP tag syntax and seen the different kinds of tags available, including printing and non-printing tags.

We've seen how to check the syntax of templates using `puppet epp validate`, and how to render the output of a template using `puppet epp render`, passing in canned values for the template parameters using `--values` and `--values_file`, or using `puppet apply` to render the template directly.

Finally, we've touched on legacy ERB templates, where they come from, how they compare to EPP templates, and why, although you may still encounter ERB templates in the wild, you should only use EPP in your own code.