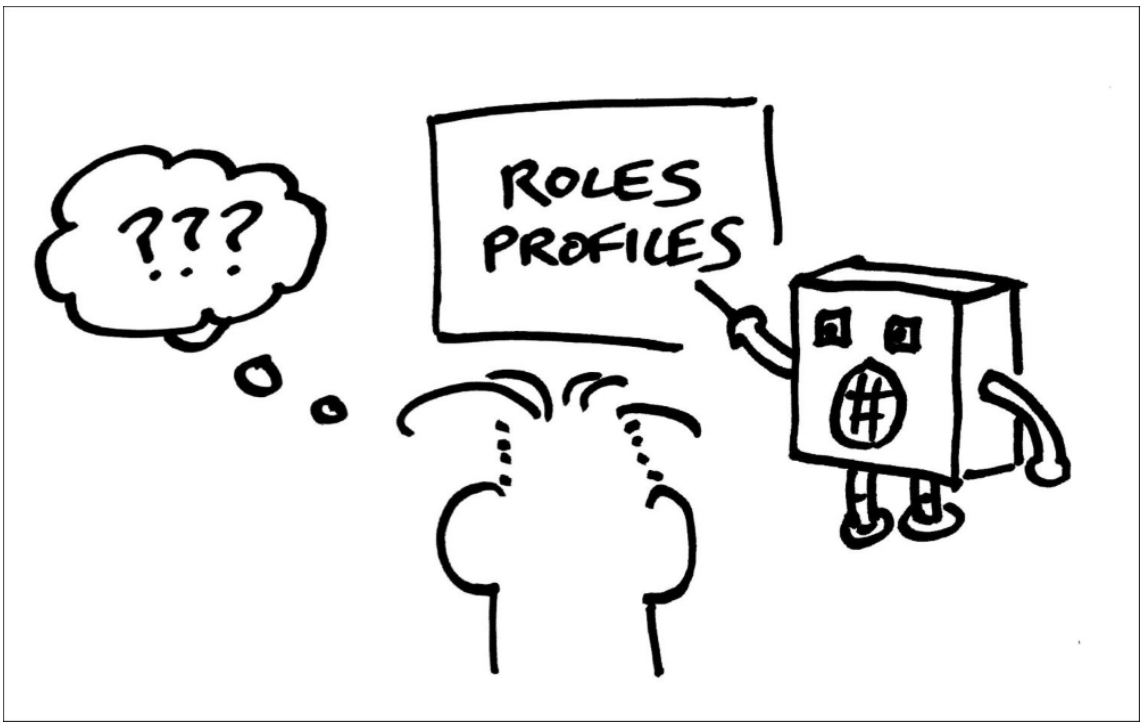
## Lab 8. Classes, roles, and profiles

In this lab you will explore the details of Puppet classes, the distinction between defining a class and including the class, how to supply parameters to classes, and how to declare classes with parameters and specify appropriate data types for them. You'll learn how to create defined resource types, and how they differ from classes. You'll also see how to organize your Puppet code using the concepts of nodes, roles, and profiles.




## Classes

We've come across the **class** concept a few times so far in this course, without really explaining it. Let's explore a little further now and see how to use this key Puppet language building block.

### The class keyword

You may have noticed that in the code for our example NTP module in Lab 7, we used the `class` keyword:

```
class pbg_ntp {
  ...
}
```

If you're wondering what the `class` keyword does, the surprising answer is nothing at all. Nothing, that is, except inform Puppet that the resources it contains should be grouped together and given a name ( `pbg_ntp` ), and that these resources should not be applied yet.

You can then use this name else where to tell Puppet to apply all the resources in the class together. We declared our example module by using the `include` keyword:

```
include ntp
```

The following example shows a class **definition**, which makes the class available to Puppet, but does not (yet) apply any of its contained resources:

```
class CLASS_NAME {
  ...
}
```

The following example shows a **declaration** of the `CLASS_NAME` class. A declaration tells Puppet to apply all the resources in that class (and the class must have already been defined):

```
include CLASS_NAME
```

### Declaring parameters to classes

The following example shows how to define a class that takes parameters. It's a simplified version of the `pbg_ntp` class we developed for our NTP module ( `class_params.pp` ):

```
# Manage NTP
class pbg_ntp_params (
  String $version = 'installed',
) {
  ensure_packages(['ntp'],
    {
      'ensure' => $version,
    }
  )
}
```

The important part to look at is in parentheses after the start of the class definition. This specifies the parameters that the class accepts:

```
String $version = 'installed',
```

If you don't supply a default value for a parameter, that makes the parameter **mandatory**, so Puppet will not let you declare the class without supplying a value for that parameter.

When you declare this class, you do it in exactly the same way that we did previously with the Puppet Forge modules, using the `include` keyword and the name of the class:

```
include pbg_ntp_params
```

There are no mandatory parameters for this class, so you need not supply any, but if you do, add a value like the following to your Hiera data, and Puppet will look it up automatically when the class is included:

```
pbg_ntp_params::version: 'latest'
```

Classes can take more than one parameter, of course, and the following (contrived) example shows how to declare multiple parameters of various types ( `class_params2.pp` ):

```
# Manage NTP
class pbg_ntp_params2 (
  Boolean $start_at_boot,
  String[1] $version                        = 'installed',
```

```
  Enum['running', 'stopped'] $service_state = 'running',
) {
  ensure_packages(['ntp'],
    {
      'ensure' => $version,
    }
  )

  service { 'ntp':
    ensure => $service_state,
    enable => $start_at_boot,
  }
}
```

To pass parameters to this class, add Hiera data like the following:

```
pbg_ntp_params2::start_at_boot: true
pbg_ntp_params2::version: 'latest'
pbg_ntp_params2::service_state: 'running'
```

Let's look closely at the parameter list:

```
  Boolean $start_at_boot,
  String[1] $version                       = 'installed',
  Enum['running', 'stopped'] $service_state = 'running',
```

The first parameter is of `Boolean` type and named `$start_at_boot`. There's no default value, so this parameter is mandatory.

The `$version` parameter we saw in the previous example, but now it's a `String[1]` instead of a `String`. What's the difference? A `String[1]` is a String with at least one character.

The final parameter, `$service_state` is of type `Enum`. With an **Enum parameter**, we can specify exactly the list of allowed values it can take.

If your class expects a String parameter which can only take one of a handful of values, you can list them all in an `Enum` parameter declaration, and Puppet will not allow any value to be passed to that parameter unless it is in that list. In our example, if you try to declare the `pbg_ntp_params2` class and pass the value `bogus` to the `$service_state` parameter, you'll get this error:

```
Error: Evaluation Error: Error while evaluating a Resource Statement,
Class[Pbg_ntp_params2]: parameter 'service_state' expects a match for Enum['running',
'stopped'], got String at /examples/class_params2.pp:22:1 on node ubuntu-xenial
```

Just like any other parameter, an `Enum` parameter can take a default value, as it does in our example.

### Automatic parameter lookup from Hiera data

We've seen in this lab, and the previous one that we can use Hiera data to pass parameters to classes. If we include a class named `ntp`, which accepts a parameter `version`, and a key exists in Hiera named `ntp::version`, its value will be passed to the `ntp` class as the value of `version`. For example, if the Hiera data looks like the following:

```
ntp::version: 'latest'
```

Puppet will automatically find this value and pass it to the `ntp` class when it's declared.

In general, Puppet determines parameter values in the following order of priority, highest first:

1. Literal parameters specified in a class declaration (you may see older code which does this)
2. Automatic parameter lookup from Hiera (the key must be named `CLASS_NAME::PARAMETER_NAME`)
3. Default values specified in a class definition

## Parameter data types

You should always specify types for your class parameters, as it makes it easier to catch errors where the wrong parameters or values are being supplied to the class. If you're using a String parameter, for example, if possible, make it an Enum parameter with an exact list of the values your class accepts.

## Defined resource types

Whereas a class lets you group together related resources, a **defined resource type** lets you create new kinds of resources and declare as many instances of them as you like. A defined resource type definition looks a lot like a class (`defined_resource_type.pp`):

```
# Manage user and SSH key together
define user_with_key(
  Enum[
    'ssh-dss',
    'dsa',
    'ssh-rsa',
    'rsa',
    'ecdsa-sha2-nistp256',
    'ecdsa-sha2-nistp384',
    'ecdsa-sha2-nistp521',
    'ssh-ed25519',
    'ed25519'
  ] $key_type,
  String $key,
) {
  user { $title:
    ensure     => present,
    managehome => true,
  }

  file { "/home/${title}/.ssh":
    ensure => directory,
    owner  => $title,
    group  => $title,
    mode   => '0700',
  }

  ssh_authorized_key { $title:
    user => $title,
```

```
    type => $key_type,
    key  => $key,
  }
}
```

You can see that instead of the `class` keyword, we use the `define` keyword. This tells Puppet that we are creating a defined resource type instead of a class. The type is called `user_with_key`, and once it's defined, we can declare as many instances of it as we want, just like any other Puppet resource:

```
user_with_key { 'john':
  key_type => 'ssh-rsa',
  key      => 'AAAA...AcZik=',
}
```

When we do this, Puppet applies all the resources inside `user_with_key`: a user, a `.ssh` directory for that user, and an `ssh_authorized_key` for the user, containing the specified key.

**Note**

Remember this rule of thumb when deciding whether to create a class or a defined resource type: if it's reasonable to have more than one instance on a given node, it should be a defined resource type, but if there will only ever be one instance, it should be a class.

## Type aliases

It's straightforward to define new **type aliases**, using the `type` keyword ( `type_alias.pp` ):

```
type ServiceState = Enum['running', 'stopped']

define myservice(ServiceState $state) {
  service { $name:
    ensure => $state,
  }
}

myservice { 'ntp':
  state => 'running',
}
```

Creating a type alias can be very useful when you want to ensure, for example, that parameter values match a complex pattern, which would be tiresome to duplicate. You can define the pattern in one place and declare multiple parameters of that type ( `type_alias_pattern.pp` ):

```
type IPAddress = Pattern[/\A([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])(\.([0-9]|
[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])){3}\z/]

define socket_server(
  IPAddress $listen_address,
  IPAddress $public_address,
) {
  # ...
}
```

```
socket_server { 'myserver':
  listen_address => '0.0.0.0',
  public_address => $facts['networking']['ip'],
}
```

When creating a type alias in a module, it should be in a file named after the type in the `types` subdirectory of the module. For example, a type named `IPAddress` should be defined in the file `types/ipaddress.pp`.

### Using include with lookup()

Previously, when including classes in our manifest, we've used the `include` keyword with a literal class name, as in the following example:

```
include postgresql
include apache
```

However, `include` can also be used as a function, which takes an array of class names to include:

```
include(['postgresql', 'apache'])
```

We already know that we can use Hiera to return different values for a query based on the node name (or anything else defined in the hierarchy), so let's define a suitable array in Hiera data, as in the following example:

```
classes:
- postgresql
- apache
```

Now we can simply use `lookup()` to get this Hiera value, and pass the result to the `include()` function:

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

In effect, this is your entire Puppet manifest. Every node will apply this manifest, and thus include the classes assigned to it by the Hiera data. Since the top-level manifest file is traditionally named `site.pp`, you can put this `include` line in `manifests/site.pp`, and the `papply` or `run-puppet` scripts will apply it because they apply everything in the `manifests/` directory.

### Common and per-node classes

We can specify a set of classes in `common.yaml` which will be applied to all nodes: things such as user accounts, SSH and `sudoers` config, time zone, NTP setup, and so on. The complete example repo outlined in [Lab 12,] [*Putting it all together*] has a typical set of such classes defined in `common.yaml`.

However, some classes will only be needed on particular nodes. Add these to the per-node Hiera data file. For example, our `pbg` environment on the lab environment contains the following in `hiera.yaml`:

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

So per-node data for a node named `node1` will live in the `nodes/node1.yaml` file under the `data/` directory.

Let's see a complete example. Suppose your `common.yaml` file contains the following:

```
classes:
- postgresql
- apache
```

And suppose your per-node file ( `nodes/node1.yaml` ) also contains:

```
classes:
- tomcat
- my_app
```

Now, what happens when you apply the following manifest in `manifests/site.pp` on `node1` ?

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

Which classes will be applied? You may recall from Lab 6, that the `unique` merge strategy finds all values for the given key throughout the hierarchy, merges them together, and returns them as a flattened array, with duplicates removed. So the result of this `lookup()` call will be the following array:

```
[apache, postgresql, tomcat, my_app]
```

## Roles and profiles

Now that we know how to include different sets of classes on a given node, depending on the job the node is supposed to do, let's think more about how to name those classes in the most helpful way. For example, consider the following list of included classes for a certain node:

```
classes:
- postgresql
- apache
- java
- tomcat
- my_app
```

The class names give some clues as to what this node might be doing. It looks like it's probably an app server running a Java app named `my_app` served by Tomcat behind Apache, and backed by a PostgreSQL database. That's a good start, but we can do even better than this, and we'll see how in the next section.

### Roles

To make it obvious that the node is an app server, why don't we create a class called `role::app_server` , which exists only to encapsulate the node's included classes? That class definition might look like this ( `role_app_server.pp` ):

```
# Be an app server
class role::app_server {
  include postgresql
  include apache
  include java
  include tomcat
  include my_app
}
```

We call this idea a **role class**. A role class could simply be a module in its own right, or to make it clear that this is a role class, we could organize it into a special `role` module. If you keep all your role classes in a single module, then they will all be named `role::something`, depending on the role they implement.

### Profiles

We can tidy up our manifest quite a bit by adopting the rule of thumb that, apart from common configuration in `common.yaml`, **nodes should only include role classes**. This makes the Hiera data more self-documenting, and our role classes are all neatly organized in the `role` module, each of them encapsulating all the functionality required for that role. It's a big improvement. But can we do even better?

Let's look at a role class such as `role::app_server`. It contains lots of lines including modules, like the following:

```
  include tomcat
```

If all you need to do is include a module and have the parameters automatically looked up from Hiera data, then there's no problem. This is the kind of simple, encouraging, unrealistic example you'll see in product documentation or on a conference slide.

We could, of course, create a custom module for each app, which hides away all that messy support code. However, it's a big overhead to create a new module just for a few lines of code, so it seems like there should be a niche for a small layer of code which bridges the gap between roles and modules.

We call this a **profile class**. A profile encapsulates some specific piece of software or functionality which is required for a role. In our example, the `app_server` role requires several pieces of software: PostgreSQL, Tomcat, Apache, and so on. Each of these can now have its own profile.

Let's rewrite the `app_server` role to include profiles, instead of modules ( `role_app_server_profiles.pp` ):

```
# Be an app server
class role::app_server {
  include profile::postgresql
  include profile::apache
  include profile::java
  include profile::tomcat
  include profile::my_app
}
```

What would be in these profile classes? The `profile::tomcat` class, for example, would set up the specific configuration of Tomcat required, along with any app-specific or site-specific resources required, such as firewall rules, `logrotate` config, file and directory permissions, and so on. The profile wraps the module, configures it, and provides everything the module does not, in order to support this particular application or site.

The `profile::tomcat` class might look something like the following example, adapted from a real production manifest ( `profile_tomcat.pp` ):

```
# Site-specific Tomcat configuration
class profile::tomcat {
  tomcat::install { '/usr/share/tomcat7':
    install_from_source => false,
    package_ensure      => present,
    package_name        => ['libtomcat7-java','tomcat7-common','tomcat7'],
  }
```

```
  exec { 'reload-tomcat':
    command     => '/usr/sbin/service tomcat7 restart',
    refreshonly => true,
  }

  lookup('tomcat_allowed_ips', Array[String[7]]).each |String $source_ip| {
    firewall { "100 Tomcat access from ${source_ip}":
      proto  => 'tcp',
      dport  => '8080',
      source => $source_ip,
      action => 'accept',
    }
  }

  file { '/usr/share/tomcat7/logs':
    ensure  => directory,
    owner   => 'tomcat7',
    require => Tomcat::Install['/usr/share/tomcat7'],
  }

  file { '/etc/logrotate.d/tomcat7':
    source => 'puppet:///site-modules/profile/tomcat/tomcat7.logrotate',
  }
}
```

## Summary

In this lab, we've looked at a range of different ways of organizing your Puppet code. We've covered classes in detail, explaining how to define them using the `class` keyword to define a new class, using the `include` keyword to declare the class, and using Hiera's automatic parameter lookup mechanism to supply parameters for included classes.

Declaring parameters involves specifying the allowable data types for parameters, and we've had a brief overview of Puppet's data types, including scalars, collections, content types and range parameters.

In the next lab we'll look at using Puppet to create files using templates, iteration, and Hiera data.