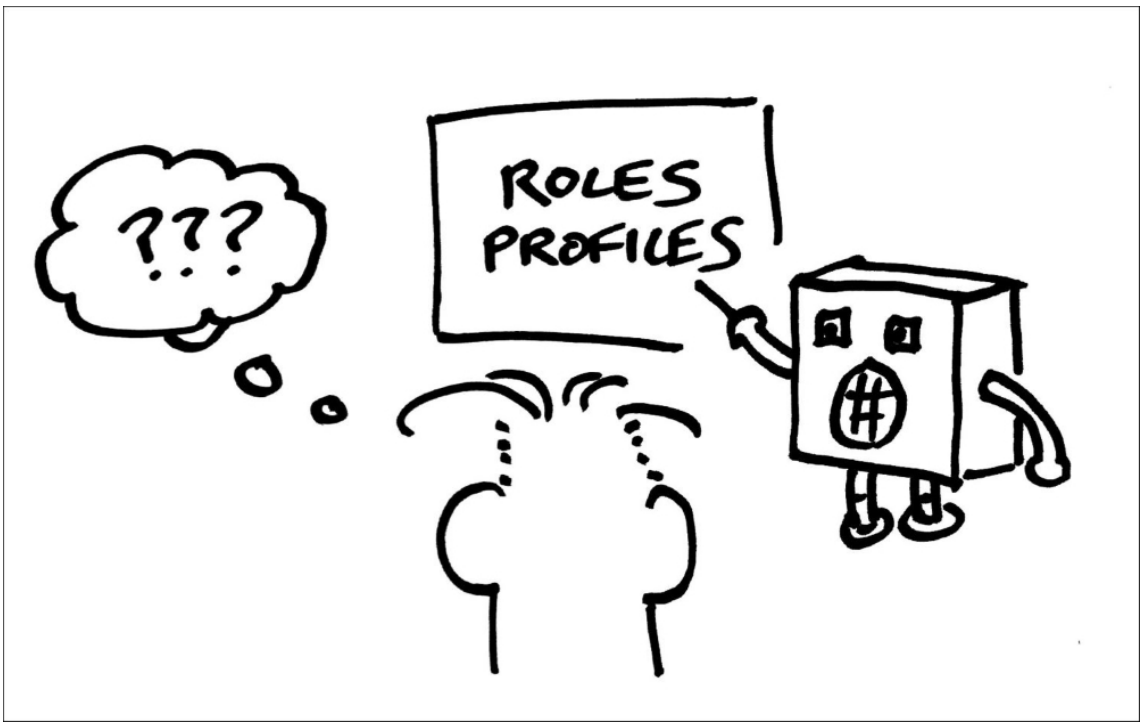
## Lab 8. Classes, roles, and profiles

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,] [*Mastering modules] (in the [Writing the module code*] section), 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 elsewhere 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
```

You may recall from [Lab 7,] [*Mastering modules*], that we used Hiera's automatic parameter lookup mechanism to supply parameters to classes. We'll find out more about this shortly, but first, how do we write a class that accepts parameters?

### Declaring parameters to classes

If all a class does is group together related resources, that's still useful, but a class becomes much more powerful if we can use **parameters**. Parameters are just like resource attributes: they let you pass data to the class to change how it's applied.

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',
```

`String` tells Puppet that we expect this value to be a String, and it will raise an error if we try to pass it anything else, such as an Integer. `$version` is the name of the parameter. Finally, the `'installed'` part specifies a **default value** for the parameter. If someone declares this class without supplying the `pbg_ntp_params::version` parameter, Puppet will fill it in automatically using this default value.

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. Mandatory parameters must be declared first, before any optional parameters (that is, parameters with a default value).

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. This means that you can't pass the empty string to such a parameter, for example. It's a good idea to specify a minimum length for String parameters, if appropriate, to catch the case where an empty string is accidentally passed to the class.

The final parameter, `$service_state` is of a new type, `Enum` , which we haven't come across before. 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. If you can't restrict it to a set of allowed values, specify a minimum length with `String[x]`. (If you need to specify a maximum length too, the syntax is `String[min, max]`.)

### Available data types

So far in this lab, we've encountered the data types String, Enum, and Boolean. Here are the others:

- Integer (whole numbers)
- Float (floating-point numbers, which have optional decimal fractions)
- Numeric (matches either integers or floats)
- Array
- Hash
- Regexp
- Undef (matches a variable or parameter which hasn't been assigned a value)

- Type (data type of literal values which represent Puppet data types, such as String, Integer, and Array)

There are also [*abstract*] data types, which are more general:

- Optional (matches a value which may be undefined, or not supplied)
- Pattern (matches Strings which conform to a specified regular expression)
- Scalar (matches Numeric, String, Boolean, or Regexp values, but not Array, Hash, or Undef)
- Data (matches Scalar values, but also Array, Hash, and Undef)
- Collection (matches Array or Hash)
- Variant (matches one of a specified list of data types)
- Any (matches any data type)

In general, you should use as specific a data type as possible. For example, if you know that a parameter will always be an integer number, use `Integer`. If it needs to accept floating-point values as well, use `Numeric`. If it could be a String as well as a Number, use `Scalar`.

### Content type parameters

Types which represent a collection of values, such as `Array` and `Hash` (or their parent type, `Collection`) can also take a parameter indicating the type of values they contain. For example, `Array[Integer]` matches an array of Integer values.

If you declare a content type parameter to a collection, then all the values in that collection must match the declared type. If you don't specify a content type, the default is `Data`, which matches (almost) any type of value. The content type parameter can itself take parameters: `Array[Integer[1]]` declares an array of positive Integers.

Hash takes two content type parameters, the first indicating the data type of its keys, the second the data type of its values. `Hash[String, Integer]` declares a hash whose keys are Strings, each of which is associated with an Integer value (this would match, for example, the hash `{'eggs' => 61}`).

### Range parameters

Most types can also accept parameters in square brackets, which make the type declaration more specific. For example, we've already seen that `String` can take a pair of parameters indicating the minimum and maximum length of the string.

Most types can take **range** ] **parameters**: `Integer[0]` matches any Integer greater than or equal to zero, while `Float[1.0, 2.0]` matches any Float between 1.0 and 2.0 inclusive.

If either range parameter is the special value `default`, the default minimum or maximum value for the type will be used. For example, `Integer[default, 100]` matches any Integer less than or equal to 100.

For arrays and hashes, the range parameters specify the minimum and maximum number of elements or keys: `Array[Any, 16]` specifies an array of no less than 16 elements of `Any` type. `Hash[Any, Any, 5, 5]` specifies a hash containing exactly five key-value pairs.

You can specify both range and content type parameters at once: `Array[String, 1, 10]` matches an array of between one and ten strings. `Hash[String, Hash, 1]` specifies a hash with String keys and Hash values, containing at least one key-value pair with String keys and values of type Hash.

### Flexible data types

If you don't know exactly what type the values may be, you can use one of Puppet's more flexible **abstract types**, such as `Variant`, which specifies a list of allowed types. For example, `Variant[String, Integer]` allows its value to be either a String or an Integer.

Similarly, `Array[Variant[Enum['true', 'false'], Boolean]]` declares an array of values which can be either the String values `'true'` or `'false'` or the Boolean values `true` and `false`.

The `Optional` type is very useful when a value may be undefined. For example, `Optional[String]` specifies a String parameter which may or may not be passed to the class. Normally, if a parameter is declared without a default value, Puppet will give an error when it is not supplied. If it is declared `Optional`, however, it may be omitted, or set to `Undef` (meaning that the identifier is defined, but has no value).

The `Pattern` type allows you to specify a regular expression. All Strings matching that regular expression will be allowed values for the parameter. For example, `Pattern[/a/]` will match any String which contains the lowercase letter a. In fact, you can specify as many regular expressions as you like. `Pattern[/a/, /[0-9]/]` matches any String which contains the letter `a`, or any string which contains a digit.

## 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**

Wait, we seem to be referring to a parameter called `$title` in the example code. Where does that come from? `$title` is a special parameter which is always available in classes and defined resource types, and its value is the title of this particular declaration of the class or type. In the example, that's `john`, because we gave the declaration of `user_with_key` the title `john`.

So what's the difference between defined resource types and classes? They look pretty much the same. They seem to act the same. Why would you use one rather than the other? The most important difference is that you can only have **one declaration** of a given class on a given node, whereas you can have as many different instances of a defined resource type as you like. The only restriction is that, like all Puppet resources, the title of each instance of the defined resource type must be unique.

Recall our example `ntp` class, which installs and runs the NTP daemon. Usually, you would only want one NTP service per node. There's very little point in running two. So we declare the class once, which is all we need.

Contrast this with the `user_with_key` defined resource type. It's quite likely that you'll want more than one `user_with_key` on a given node, perhaps several. In this case, a defined resource type is the right choice.

Defined resource types are ideal in modules when you want to make a resource available to users of the module. For example, in the `puppetlabs/apache` module, the `apache::vhost` resource is a defined resource type, provided by the `apache` class. You can think of a defined resource type as being a wrapper for a collection of multiple resources.

**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` .

## Managing classes with Hiera

In [Lab 3], [*Managing your Puppet code with Git*], we saw how to set up your Puppet repo on multiple nodes and auto-apply the manifest using a cron job and the `run-puppet` script. The `run-puppet` script runs the following commands:

```
cd /etc/puppetlabs/code/environments/production && git pull/opt/puppetlabs/bin/puppet
apply manifests/
```

You can see that everything in the `manifests/` directory will be applied on every node. Clearly, Puppet is much more useful when we can apply different manifests on each node; some nodes will be web servers, others database servers, and so on. In fact, we would like to include some classes on all nodes, for general administration, such as managing user accounts, and other classes only on specific nodes. So how do we do that?

### 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 Vagrant box 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,] [*Managing data with Hiera*] 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]
```

This is the complete list of classes that Puppet will apply to the node. Of course, you can add classes at any other level of the hierarchy, if you need to, but you will probably find the common and per-node levels to be the most useful for including classes.

Naturally, even though some nodes may include the same classes as others, they may need different configuration values for the classes. You can use Hiera in the same way to supply different parameters for the included classes, as described in the [*Automatic parameter lookup from Hiera data*] section earlier in this lab.

## 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.

#### Note

It's important to note that role classes are not special to Puppet in any way. They're just ordinary classes; we call them role classes only to remind ourselves that they are for expressing the roles assigned to a particular node.

The value of `classes` in Hiera is now reduced to just the following:

```
classes:
- role::app_server
```

Looking at the Hiera data, it's now very easy to see what the node's job is---what its [*role*] is---and all app servers now just need to include `role::app_server`. When or if the list of classes required for app servers changes, you don't need to find and update the Hiera `classes` value for every app server; you just need to edit the `role::app_server` class.

### 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.

Real-life Puppet code is often more complicated, however, with logic and conditionals and special cases, and extra resources that need to be added, and so forth. We don't want to duplicate all this code when we use Tomcat as part of another role (for example, serving another Tomcat-based app). How can we neatly encapsulate it at the right level of abstraction and avoid duplication?

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',
  }
}
```

The exact contents of this class don't really matter here, but the point you should take away is that this kind of site-specific 'glue' code, wrapping third-party modules and connecting them with particular applications, should live in a profile class.

In general, a profile class should include everything needed to make that particular software component or service work, including other profiles if necessary. For example, every profile which requires a specific configuration of Java should include that Java profile. You can include a profile from multiple other profiles without any conflicts.

Using profile classes in this way both makes your role classes neater, tidier, and easier to maintain, but it also allows you to reuse the profiles for different roles. The `app_server` role includes these profiles, and other roles can include them as well. This way, our code is organized to reduce duplication and encourage re-use. The second rule of thumb is, **roles should only include profiles**.

If you're still confused about the exact distinction between roles and profiles, don't worry: you're in good company. Let's try and define them as succinctly as possible:

- **Roles** identify a particular function for a node, such as being an app server or a database server. A role exists to document what a node is for. Roles should only include profiles, but they can include any number of profiles.
- **Profiles** identify a particular piece of software or functionality which contributes to a role; for example, the `tomcat` profile is required for the `app_server` role. Profiles generally install and configure a specific software component or service, its associated business logic, and any other Puppet resources needed. Profiles are the 'glue layer' which sits between roles and modules.

It's possible that your manifest may be so simple that you can organize it using only roles or only profiles. That's fine, but when things start getting more complex and you find yourself duplicating code, consider refactoring it to use the roles-and-profiles pattern in the way we've seen here.

## 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, abstract types, flexible types, and introduced creating your own type aliases. We've also introduced the defined resource type, and explained the difference between defined resource types and classes, and when you would use one or the other.

We've also looked at how to use the `classes` array in Hiera to include common classes on all nodes, and other classes only on particular nodes. We've introduced the idea of the role class, which encapsulates everything needed for a node to fulfil a particular role, such as an app server.

Finally, we've seen how to use profile classes to configure and support a particular software package or service, and how to compose several profile classes into a single role class. Between them, roles and profiles bridge the gap between the Hiera `classes` array, at the top level, and modules and configuration data (at the lowest level). We can summarize the rules by saying that [*nodes should only include roles, and roles should only include profiles*].

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