Welcome

Quest Objectives

In this quest you will be introduced to Elvium, the Learning Virtual Machine environment.

Welcome to Elvium

> *Any sufficiently advanced technology is indistinguishable from magic.*
> *-Arthur C. Clarke*

Welcome to Elvium, user. Have a look around, if you like, but take note: what you see now is only the surface. The real channels of power in Elvium lie deeper. You will learn of these things. You see, you are not an ordinary user, for you have come into the Elvium with a user id of '0', the mark of the **Superuser**.

If you choose to follow the path set forth in this Quest Guide, you will learn to channel your powers by using the art of Puppet. With only a few words, shape the environment around as you see fit. Inscribe your wishes into potent Puppet manifests to ensure that your will continues to be done when journey beyond Elvium to bring other systems under your dominion.

Getting Started

Your arrival has been foretold, user, and all the necessary arrangements were made to ensure that this Quest Guide would fall into your hands. By entering the command `puppet apply setup/guide.pp`, you activated a Puppet **manifest** we had prepared in anticipation of your arrival.

Quest Navigation

In order to learn more about Puppet, and to work your way through Elvium, you will have to complete a series of Quests. Each Quest has Tasks that need to be

completed. Transparent information is essential on Elvium. To monitor your status with the quests on Elvium we've created custom commands for you.

The following command will help you with using the quest tool:

```
quest --help
```

The `quest --help` command provides you with an understanding of several quest tools such as the following:

```
quest --progress    # Display details of tasks completed (default: true)
quest --brief       # Display number of tasks compelted
quest --name        # Name of the quest to track
quest --completed   # Display completed quests
quest --showall     # Show all available quests
quest --start       # Provide name of the quest to start tracking
```

We'll explain the commands needed as we go.

Facter

You are a curious being. Would you like to further investigate Elvium? To do this on Elvium we use the tool `facter` to obtains `facts` about the system. Here are a few for you to try out:

```
facter ipaddress
facter facterversion
facter memorysize
facter operatingsystem
facter osfamily
facter puppetversion
```

Tip:

You can see all the facts by running facter -p

Resources

Quest Objectives

In this quest you will be introduced to the fundamental applications of Puppet resources and using them to inspect the state of your Virtual Machine. The tasks we will accomplish in this quest will help you learn more about Puppet resources. If you're ready to get started, type the following command:

```
quest --start resources
```

Puppet Resources

Real power, whether a spoken spell or a terminal command, is never as simple as it seems. Wizards turn to ancient tomes and grimoirs to tell of the bonds between the syllables of a spell and the elements under its influence. As an aspirant in the mystical arts of Puppet, you must learn of the connections between the syntax of Puppet and the environment around you.

We'll begin with **Resources**, the basic units that Puppet uses to describe an environment.

Anatomy of a Resource

Know thyself, user, for you too are a resource. Use the following command to see how you look to Puppet:

```
puppet resource user root
```

The output will look like this:

```
user { 'root':
    ensure           => 'present',
    comment          => 'root',
    gid              => '0',
    home             => '/root',
    password         => '$1$jrm5tnjw$h8JJ9mCZLmJvIxvDLjw1M/',
    password_max_age => '99999',
    password_min_age => '0',
    shell            => '/bin/bash',
    uid              => '0',
}
```

This block of code that describes a resource is called a **resource declaration**. It's a little abstract, but a nice portrait, don't you think?

Resource Type

Look at the first line of the resource declaration. The word you see before the curly brace is the **resource type**, in this case, `user`. Just as any individual cat or dog is a member of its species (*Felis catus* and *Canus lupis familiaris* to be precise) any instance of a resource must be a member of a **resource type**. Think of this type as a framework that defines the range of characteristics an individual resource can have.

Though Puppet allows you to describe and manipulate a great variety of resource types, there are some core resource types you will encounter most often:

- `user` A user
- `group` A user group
- `file` A specific file
- `package` A software package
- `service` A running service
- `cron` A scheduled cron job
- `exec` An external command
- `host` A host entry

Resource Title

After the resource type comes a curly brace and a single-quoted `title` of the resource: in your case, 'root'. (Be proud to have such a noble title!) Because the title of a resource is used to identify it, it must be unique. No two resources of the same type can share the same title.

Attribute Value Pairs

After the colon, comes a list of **attributes** and their corresponding **values**. Each line consists of an attribute name, a `=>` (hash rocket), a value, and a final comma. For example, the attribute value pair `home => '/root',` indicates that your home is set to the directory `/root`.

Puppet DSL

In general terms, a resource declaration will match the following pattern:

```
type {'title':
    attribute => 'value',
}
```

The syntax you see here is an example of Puppet's Domain-Specific Language (DSL), which is built on the Ruby programming language. Because the Puppet DSL is a **declarative** language rather than a **procedural** one, the descriptions themselves have the power to change the state of the environment. Use the DSL to paint a picture of what you want to see, and Puppet's providers will make it so.

> Tip:
>
> Though a comma isn't strictly necessary at the end of the final attribute value pair, it is best practice to include it for the sake of consistency.

Tasks

The first step in mastering Puppet is to learn to understand the world around you as a collection of **resources**. You will not be using resource declarations to shape your environment just yet. Instead you will exercise your power by hand and use Puppet only to inspect the consequences of your actions.

? Task 1 :

TASK #1: The path to greatness is a lonely one. Fortunately, your superuser status gives you the ability to create an assistant for yourself:

```
useradd ralph
```

Awesome! Did you notice that your 'completed tasks' increased to 1/6? Check on your progress to see how you're doing.

```
quest --progress
```

This shows your progress by displaying tasks that you have completed and tasks that still need completeing.

TASK #2: Now take a look at your creation:

```
puppet resource user ralph
```

Potent stuff. Note that Ralph's password attribute is set to `'!!'`. This isn't a proper password at all! In fact, it's a special value indicating Ralph has no password whatsoever. If he had a soul, it would be locked out of his body.

TASK #3: Rectify the situation. Set Ralph's password to *puppetlabs*.

```
passwd ralph
```

If you take another look at `puppet resource user ralph`, the value for his password attribute should now be set to a SHA1 hash of his password, something a little like: `'$1$hNahKZqJ$9ul/RR2U.9ITZlKcMbOqJ.'`

TASK #4: Now have a look at Ralph's home directory. When you created him, it was set to `'/home/ralph'` by default. His home is a `directory`, which is really just a special kind of the resource type `file`. The `title` of any file is the same as the path to that file. Take a look at Ralph's home directory. Enter the command:

```
puppet resource file /home/ralph/spells
```

TASK #5: What? `ensure => 'absent',`? Values of the `ensure` attribute indicate the basic state of a resource. A value of absent means it doesn't exist at all. When you created Ralph, you automatically assigned him an address, but neglected to put anything there. Do this now:

```
mkdir /home/ralph/spells
```

Now have another look:

```
puppet resource file /home/ralph/spells
```

You're on a roll! So far you have completed 5/6 tasks. Let's take a look at what you have completed so far.

```
quest --brief
```

Almost there to officially completing your first quest!

TASK #6: Just one more thing. Ralph does not want his spells to be seen by anyone else! Let's make it so:

```
chmod 700 /home/ralph/spells
```

..and inspect the result one more time:

```
puppet resource file /home/ralph/spells
```

The Resource Abstraction Layer

If you completed this quest, you will be familiar with the basics of resources. A great part of the utility of resources, however, is in their power to abstract away the partocularities of a system while still providing a full description of your environment.

Our sages have long known that Elvium operates according to the rules of **CentOS**, which they call its **Operating System**. We know of distant continents, however, where the fabric of the world has a different weave; that is, there is a different Operating System.

If you find yourself on the shores of Ubuntu and croak out a `useradd`, you will be laughed right off the beach for getting it backwards; as any Ubuntu native could tell you, `adduser` is the right way to say it there. And attempting to install a package with `yum` on a system where `apt-get` is appropriate is a *faux pas* indeed.

If you aspire to extend your influence across these differing systems, it will be wise to learn a method of applying your power consistently, and resources are a key part of this puzzle.

Puppet takes the descriptions expressed by resource declarations and uses providers to implement the system-specific processes to realize them. These Providers abstract away the complexity of managing diverse implementations of resource types on different systems. As a whole, this system of resource types and the providers that implement them is called the **Resource Abstraction Layer**, or **RAL**. If you want to create a user, for instance, Puppet's RAL will abstract away the `useradd` and `adduser`, giving you a single way to do things across systems. Similarly, when you wish to install a package, you can stand back and let Puppet's providers decide whether to use `yum` or `apt-get` for package management.

By harnessing the power of the RAL, you can be confident of the potency of your Puppet skills wherever your journey takes you.

Manifests

Prerequisites

- Resources Quest

Quest Objectives

In this quest you will gain a better understanding of resource declarations, the resource abstraction layer and using puppet apply for applying manifests. The tasks we will accompish in this quest will help you learn more about Puppet manifests. If you're ready to get started, type the following command:

```
quest --start manifest
```

Puppet Manifests

As you saw in the Resources quest, Puppet's **resource declarations** can be used to keep track of just about anything here in Elvium. So far, you have made changes to the world without using Puppet. You looked at resource declarations only in order to keep track of the effects of what you did. In this quest, you will learn to craft your own resource declarations and inscribe them in a special document called a **manifest**.

Once you've created a manifest, you will use the `puppet apply` tool to implement it. Puppet will check each resource in your environment against the resource declaration in the manifest. Puppet's **providers** will then do everything neccessary to bring the state of those resources in line with the resource declarations in your manifest.

Manifests, like the resource declarations they contain, are written in Puppet's Domain Specific Language (DSL). In addition to resource declarations, a manifest will often contain **classes**, which organize resource declarations into functional sets, and **logic** to allow you to manage resources according to variables in the Elvium environment. You will learn more about these aspects of manifest creation in a later quest.

Tasks

Before you get started lets take a look at quests you have completed

```
quest --completed
```

> Tip:
>
> You can use `puppet describe user` and `puppet resource user` for help using and understanding the user resource.

Great! It shows you have completed the Resources Quest. Feel free to use this tool anytime to checkout which quests you've completed on your jounrey. Now lets get back to learning about Puppet manifests.

To establish the manifest you want to place it in your home directory (`/root`). Make sure you're in this directory before creating your manifest.

```
cd /root
```

TASK #1: Now we are ready to create a manifest to manage the user Ralph. The `.pp` extension identifies a file as a manifest.

```
nano ralph.pp
```

> **On Editor Wars**
>
> For the sake of simplicity and consistency, we use the text editor nano in our instructions. Feel free to use vim if you prefer. Emacs isn not installed.

TASK #2: Type the following deadly incantation into your manifest:

```
user { 'ralph':
    ensure => 'absent',
}
```

Save the file and exit your text editor.

Puppet Parser

`puppet parser` is like Puppet's version of spellcheck. This action validates Puppet's DSL syntax without compiling a catalog or syncing any resources. If no manifest files are provided, Puppet will validate the default site manifest. If there are no syntax errors, Puppet will return nothing, otherwise Puppet will display the first syntax error encountered.

Tasks

TASK #3: Enter the following command in the command line:

```
puppet parser validate ralph.pp
```

If the parser returns nothing, continue on. If not, make the necessary changes with nano and re-validate until the syntax checks out.

> ### Warning:
>
> The puppet parser can only ensure that the syntax of a manifest is well-formed. It can't guarantee that your variables are correctly named, your logic is valid, or, for that matter, that your manifest does what you want it to.

Puppet Apply

This is a very handy tool for learning to write code in Puppet's DSL and is a standalone puppet execution tool. However, in reality, you'll probably use this tool to describe the configurations of an entire system in a single file. This description will be constructed as a list of all resources you want to manage on your system, but as you can imagine, that might end up being a *really* long file! You will see how this can be a more manageable process when you come across **Classes** during your journey.

Tasks

TASK #4: Simulate the change in the system without actually enforcing the `ralph.pp` manifest

```
puppet apply --noop ralph.pp
```

TASK #5: Using `puppet apply` apply the simulated manifest `ralph.pp`

```
puppet apply ralph.pp
```

TASK #6: Check in on Ralph. How is his health?

```
puppet resource user ralph
```

If you like, you can check the old fashioned way as well. If the user doesn't exist, it will return nothing.

```
egrep -i "^ralph" /etc/passwd
```

He is not too well, it appears! Clearly Ralph was no match for your skills in magic.

Check on your progress. How are you doing so far?

TASK #7: With manifests you can create as well as destroy. Lets create a new assistant by adding Jack to the Elvium system.

```
nano jack.pp
```

TASK #8: Write the following code to your manifest:

```
user { 'jack':
    ensure => 'present',
    gid => '501',
    home => '/home/jack',
    password => '!!',
    uid => '501',
}
```

TASK #9: Check your syntax:

```
puppet parser validate jack.pp
```

TASK #10: Simulate your changes:

```
puppet apply --noop jack.pp
```

TASK #11: And apply:

```
puppet apply jack.pp
```

TASK #12: Now check in on Jack.

```
puppet resource user jack
```

You have an extordinary power to add and remove whomever and whatever you may like. Use your powers wisely.

Now that you have an understanding of how Puppet manifests work, let's get started with simplifying the process with the Classes Quest!

Variables and Variability

In this quest you will gain a better understanding of using variable and facts in your manifest to make operating more scalable.

> *The green reed which bends in the wind is stronger than the mighty oak which breaks in a storm.*
> *--Confuscius*

The tasks we will accompish in this quest will help you learn more about manipulating your Puppet manifests to accomplish specific tasks. If you're ready to get started, type the following command:

```
quest --start variables
```

What you should already know

- Resources Quest
- Mainfest Quest
- Classes Quest

Variables

What you have learned so far about manifests gives you the means to achieve a great deal. However, you need not travel far down your destined path of Puppet mastery before manifests like the ones you have written thus far will seem stiff and brittle; you will need something more supple, able to shift to meet changing conditions as you travel the roads of Elvium and beyond.

You've almost definitely used variables before in some other programming or scripting language, but `$variables` always start with a dollar sign and you assign them with the `=` operator. You can use variables as the value for any resource attribute, or as the title of a resource.

```
$longthing = "Really long SSH key"

file {'authorized_keys':
  path    => '/root/.ssh/authorized_keys',
  content => $longthing,
}
```

What about the `$::variable`

People who write manifests to share with the public often adopt the habit of always using the $::variable notation when referring to facts. The double-colon prefix specifies that a given variable should be found at top scope. This isn't actually necessary, since variable lookup will always reach top scope anyway.

Lets make our manifests more adaptable by using variables.

Tasks

1. To see an example of a variable doing its duty, first navigate to your workshop directory and create a new manifest called `fickle.pp`
2. Enter the following to name and assign a variable:

```
$variable_name = "variable value!\n"
```

1. Now lets have that variable do something productive for us. Can you...

Facts

> *Get your facts first, then distort them as you please.*
> *--Mark Twain*

Puppet has a bunch of built-in, pre-assigned variables that you can use. Remember the Puppet tool `facter` when you first started? Just a quick refresher, `facter` discovers system information, normalizes it into a set of variables, and then passes them off to Puppet. Puppet's compiler accesses those facts when it's reading a manifest.

Remember running `facter ipaddress`? Puppet told you your IP address without you having to do anything. What if I wanted to turn `facter ipaddress` into a variable? You guessed it. It would look like this `${ipaddress}`

How does this make our manifests more flexible? If I entered the following variables in a manifest:

```
file {'motd':
  ensure  => file,
  path    => '/etc/motd',
  mode    => 0644,
  content => "This Learning Puppet VM's IP address is ${ipaddress}. It is running
```

```
${operatingsystem} ${operatingsystemrelease} and Puppet ${puppetversion}.

Web console login:
  URL: https://${ipaddress_eth0}
  User: puppet@example.com
  Password: learningpuppet
  ",
  }
```

and then applied the manifest, this is what would be returned:

```
file {'motd':
  ensure  => file,
  path    => '/etc/motd',
  mode    => 0644,
  content => This Learning Puppet VM's IP address is 172.16.52.135. It is running
CentOS 5.7 and Puppet 3.0.1.

Web console login:
  URL: https://172.16.52.135
  User: puppet@example.com
  Password: learningpuppet
  }
```

It's as simple as that without hardly doing anything on your end. Feeling confident? Lets combine our knowledge of variables and facts in our manifest

Tasks

1.

Our manifests are becoming more flexible, with pretty much no real work on our part.

Resource Ordering

The tasks we will accompish in this quest will help you learn more about specifying the order in which Puppet should manage the resources in a manifest. If you're ready to get started, type the following command:

```
quest --start ordering
```

What you should already know

- Resources Quest
- Mainfest Quest
- Classes Quest
- Varibales Quest
- Conditions Quest

Explicit Ordering

You are likely used to reading instructions from top to bottom. If you wish to master the creation of Puppet manifests, however, you must learn think of ordering in a different way. Rather than processessing instructions from top to bottom, Puppet interprets the resource declarations in a manifest in whatever order is most effecient.

Often, however, you will need to ensure that once resource declaration is applied before another. For instance, if you wish to declare in a manifest that a service should be running, you need to ensure that the package for that service is already installed and configured.

If you require that a group of resources be managed in a specific order, you must explicitly state the dependency relationships between these resources within the resource declarations.

Relationship Metaparameters

There are four metaparameter attributes that you can include in your resource declaration to establish order relationships among resources. The value of any relationship metaparameter should be the title or titles (in an array) of one or more target resources.

> Fact:
>
> In Puppet's DSL, a resource metaparamter is a variable that doesn't refer directly to the state of a resource, but rather tells Puppet how to process the resource declaration itself.

- `before` - causes a resource to be applied **before** a target resource.
- `require` - causes a resource to be applied **after** a target resouroce.
- `notify` - causes a resource to be applied **before** the target resource. The target resource will refresh if the notifying resource changes.
- `subscribe` - causes a resource to be applied **after** the target resource. The subscribing resource will refresh if the target resource changes.

These parameters are included in a resource declaration just like any other attribute value pair. For instance, refer to the following example:

```
service { 'sshd':
  ensure    => running,
  enable    => true,
  subscribe => File['/etc/ssh/sshd_config'],
}
```

Here, the service resource with the title 'sshd' will be applied **after** the file resource with the title '/etc/ssh/sshd_config'. Furthermore, if any other changes are made to the targeted file resource, the service will refresh.

Tasks

1.

Let's do a Quick Review

Up until this point you've been on a journey towards learning Puppet. To be successful at doing that, it is imperative you understand the fundatmental components of using Puppet: Resources and Manifests. Before we progress any further, it is important that you reflect on your understanding of these components. Feel free to revisit those quests should you not fully grasp the information.

Furthermore, Puppet Manifests are highly scable components to stablizing and maximizing your infrastructure. To customize that stablization, we examined Classes, Variables, Facts, Conditional Statements and Resource Ordering. It is important that you understand when and how these components to Puppet Manifests are used. Should you be in doubt of your understanding, please revisit those quests.

You're powers are immense and you're quickly learning how to control them. You are about to embark on a very important journey through the Modules Quest!