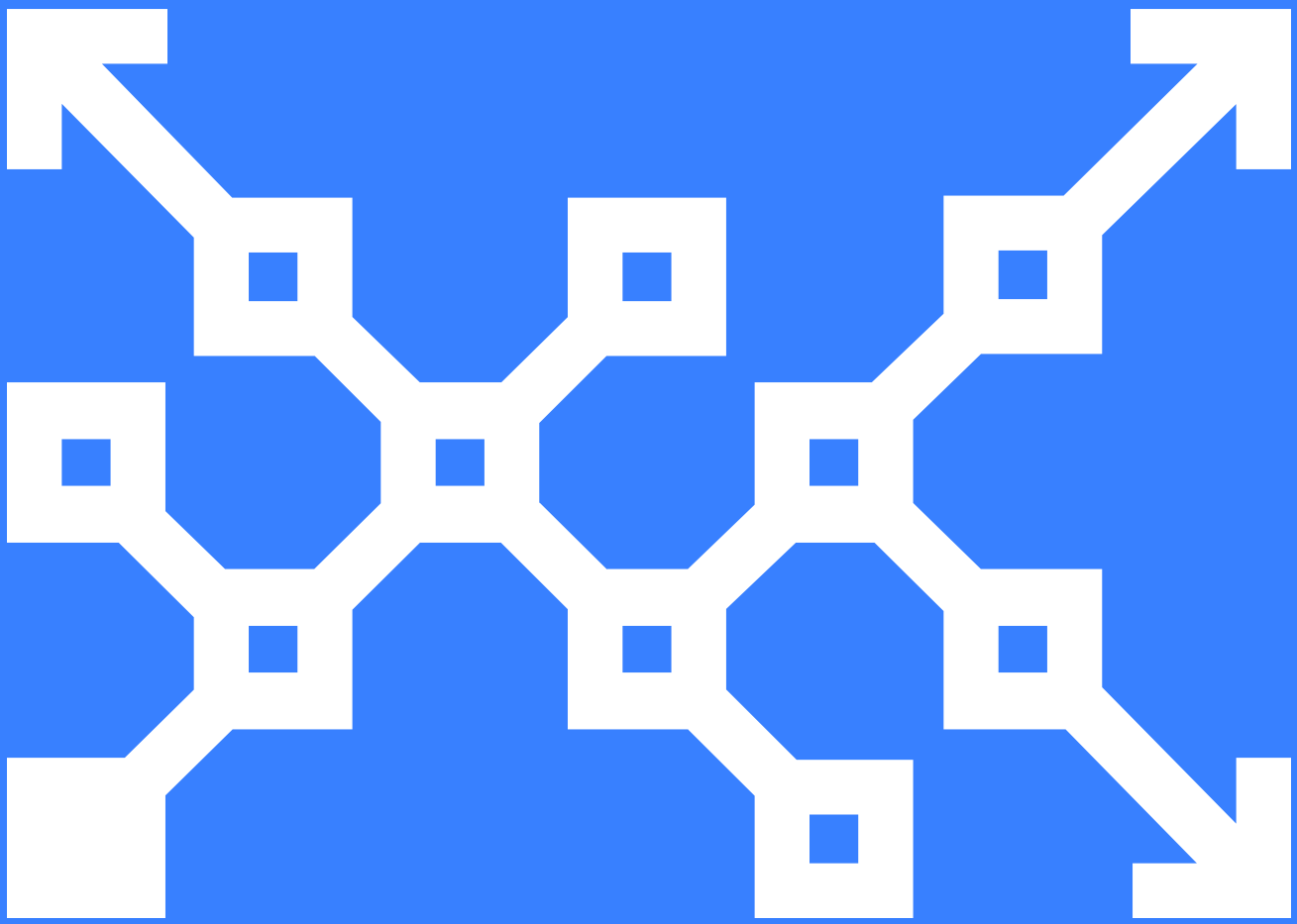
WHITE PAPER

The Top 5 Things to Automate with Puppet Right Now



puppet

Contents

You've just installed Puppet Enterprise and have a nice shiny console waiting to manage your infrastructure. Now what? Now we get to the fun part! We're going to walk you through our recommendations for what to manage first.

How to spend your time

Before we go through the list, let's talk about how to invest your time. When getting started with automation, it's important not slay your dragons out of the gate. Sure, it would be great to automate that incredibly painful deployment process on the AS/400 that's somehow still running critical infrastructure. However, that project will likely take a long time and give you only nominal returns in terms of net new time gains. It's better to tackle the low-hanging fruit — the tasks you do very often, but that probably don't cause that much pain. The net new time gained will free you up to focus your attention on improving your existing processes.

Initial setup

As you manage more and more with Puppet and have more individual contributors to the Puppet code base, you'll want to be well positioned to manage scale and complexity.

Below are instructions to take advantage of a Puppet control repository. The control repository holds all the information needed to construct a Puppet environment. We've combined all the hard-won best practices for managing Puppet infrastructures into the example control repository below.

The following instructions will guide you to take advantage of these best practices on Day One, so your Puppet Enterprise deployment can scale as rapidly as you need it to.

If you've already set up a control repository for your installation, you can safely skip this step.

1. Fork the **Puppet example Controlrepo** into your own GitHub account.
2. **Create a Puppet Enterprise master**.
3. **Create a user for yourself**.
4. **Set your password**.
5. **Add your new user** to the **Administrators** group.
6. **Enable code manager** and point the `r10k_remote` at the repository that you forked in Step 1. Remember, if you are using https to connect to Git, you don't need a private key.
7. Run Puppet at the command line using `puppet agent -t`.
8. Get a token from the RBAC service so we can run commands that require authentication by running `puppet-access login [username] --service-url https://<HOSTNAME OF PUPPET ENTERPRISE CONSOLE>:4433/rbac-api --lifetime 1y`
9. Synchronize your code by running `puppet code deploy production --wait`.

Deploy your code

In order to make changes to your Puppet code you will need need to do the following things:

- Make the changes locally
- Push those changes to GitHub
- Pull those changes down to the Puppet Master

Although you can circumvent this by editing code directly on the Puppet Master, it's much easier to collaborate if you are using this process.

Once you have made changes to your Puppet code you should do the following to get it onto the Master:

1. Add your changes to git using `git add`
2. Push those changes to GitHub using `git push`
3. Pull the new code down onto the Master using `puppet code deploy production --wait` (on the Puppet Master)

If you think this is tedious, then just **automate it**.

Task 1: Standardize service accounts

Many IT organizations manage local service accounts on servers. Not only do you likely have a handful of service accounts distributed across your various OSes, they probably don't have consistent user identifications (UIDs), group memberships, SSH keys, etc.

Luckily, Puppet built a module to solve that exact problem — the **accounts module**. The accounts module provides a simple way to standardize service accounts across all of your *nix operating systems.

Add the following to your puppetfile:

```
mod 'puppetlabs/stdlib',    '4.12.0'
mod 'puppetlabs/accounts', '1.0.0'
```

Create `site/profile/manifests/base.pp` if it does not already exist. This will be a profile that we apply to all servers as part of their base configuration. You can think of this as the standard operating environment (SOE). This profile will be included as part of our roles, which we will discuss later on.

A **profile** is a reusable block that you can use to build different types of servers. The place we pull those blocks together is called a **role**. Each server gets one role, and that role consists of many profiles (i.e., building blocks) to create that server.

Create the base shell of your profile:

```
class profile::base {
  #the base profile should include component modules that will be on all
nodes
}
```

Let's now create a role for our Puppet master. Create `site/role/manifests/master.pp` if it does not already exist and include our first class:

```
class role::master {
  # Here we include all of the profiles that make up a Puppet master
  # for now it's just the base (the SOE) but we will probably add more later
  include profile::base
}
```

Now deploy your code to the Puppet master.

The last thing we need to do with our role is assign it to the Puppet master. We will do this by going into the Puppet Enterprise console, clicking **Nodes > Classification > Add Group...** This will allow us to add a group for our master and assign our new role to it.

Create a group called **Puppet masters**. Click the newly created group from the list and, under **Pin specific nodes to the group,** open it up, add your Puppet master and click **Pin**. Now go to the Classes tab and add your new role. If it is not available yet, you may need to click the refresh button in the middle right portion of the screen. Once you have added the master role, click **Commit Changes**.

Now that we have done this, we are almost ready to test it out. Let's create a test account called **monitoring** in our base profile:

```
class profile::base {
  accounts::user { 'monitoring':
    managehome => false,
  }
}
```

This will create a monitoring user (but not a home directory) on all the machines with the base profile. Add, commit and push your code, then deploy it to the master.

Run `puppet agent -t`.

We should see something like this in our logs:

```
Notice: /Stage[main]/Profile::Base/Accounts::User[monitoring]/
Group[monitoring]/ensure: created
Notice: /Stage[main]/Profile::Base/Accounts::User[monitoring]/
User[monitoring]/ensure: created
```

We can also view the status of this run in the console by clicking **view graph** on the Puppet master. This will show us everything that Puppet is managing; the resources relating to our new `monitoring` user will come up as blue dots since they were changed in the last run.

We have now created a user as part of our SOE, and any machines with roles that include the `profile::base` class will also have this user (currently only the Puppet master).

However, this is not a very good service account. We are not managing the UID (user identifier) or GID (group identifier), meaning that they will most likely be different across machines. Plus, the user is allowed to log in, which is probably not what we want. Let's go ahead and fix that:

```
class profile::base {
  accounts::user { 'monitoring':
    ensure     => present,
    uid        => '450',
    gid        => '450',
    managehome => false,
    locked     => true,
  }
}
```

Now deploy your code and run Puppet using `puppet agent -t`.

You should see output that looks like this:

```
Notice: /Stage[main]/Profile::Base/Accounts::User[monitoring]/
Group[monitoring]/gid: gid changed '1003' to '450'
Notice: /Stage[main]/Profile::Base/Accounts::User[monitoring]/
User[monitoring]/uid: uid changed '1003' to '450'
Notice: /Stage[main]/Profile::Base/Accounts::User[monitoring]/
User[monitoring]/shell: shell changed '/bin/bash' to '/sbin/nologin'
```

We can see that Puppet has now amended the GID and UID for our `monitoring` user and disallowed the user from logging in by changing the shell from `/bin/bash` to `/sbin/nologin`.

All we need to do now is apply this to all our servers, and we will have a standardized `monitoring` service account with exactly the same GID and UID across all servers.

One neat benefit of the accounts module is the ability to get rid of passwords. Since the module provides a central point of management for SSH keys, passwords can be disabled on the accounts and SSH keys used instead. This makes it trivial (and auditable) to revoke access on a per-key basis, instead of having to globally change a password every time an employee who knows the password leaves the company.

A good use of this is to create a backup account that can be used to log into servers if LDAP (lightweight directory access protocol) is unavailable. Here we will add one such user to our base profile:

```
class profile::base {
  accounts::user { 'monitoring':
    ensure     => present,
    uid        => '450',
    gid        => '450',
    managehome => false,
    locked     => true,
  }
  accounts::user { 'emergency':
    ensure     => present,
    uid        => '460',
    gid        => '460',
    shell      => '/bin/bash',
    password   => '!!', # No password
    sshkeys    => [
      "ssh-rsa AAAA...",
      "ssh-rsa BBBB...",
      "ssh-rsa CCCC...",
      ],
  }
}
```

Under the `sshkeys` parameter, we have added the SSH keys of all sysadmins who should be able to log into that emergency account. If one of these sysadmins were to leave, all we would have to do is remove that key from the list, and their access would be revoked without having to log into each machine.

Task 2: Cron jobs and scheduled tasks

Many IT organizations use Cron to run regularly scheduled tasks. The trouble is, it's easy to forget what Cron jobs exist where, and why. Forgotten Cron jobs can wreak havoc on systems, and it's often difficult to track down the root cause — the forgotten Cron task itself.

Puppet Enterprise includes a way to manage Cron resources so you can see all the Cron jobs under management from a central place, and know exactly what they do and when they do it.

In this example, instead of creating a new Cron job or scheduled task, we are going to take one that we know is correct and make sure that it is the same across all our systems.

Let's start by checking what Cron jobs and scheduled tasks we have already:

Cron jobs

```
puppet resource cron
```

Scheduled tasks on Windows

```
puppet resource scheduled_task
```

On one of my database servers I get:

```
cron { 'unmanaged:su_-mysqlcheck_--databases_webportal_customer-1':
  ensure  => 'present',
  command => 'mysqlcheck --databases webportal_customer',
  hour    => ['0'],
  target  => 'root',
  user    => 'root',
}
```

This represents a Cron job I created manually that verifies my customer database from my `webportal` application, every night at midnight. I'm going to add this to my Puppet code so that all the database servers for my `webportal` application run the same cleanup at the same time. In order to do that, I am going to first create a role and profile for this to live in.

Now at this point, it might be tempting to create a profile called `profile::cron_jobs` and put all our cron jobs in there. However, we are going to create one called `profile::webportal_database` instead. This is because it makes more sense to group our Puppet resources by the particular component they are relevant to than by their type. Using the knowledge we acquired in Task 1, our profile will look like this:

```
# site/profile/manifests/webportal_database.pp
class profile::webportal_database {

  # This code is copy-pasted from the `puppet resource cron` command
  cron { 'vacuum_customer_database':
    ensure  => 'present',
    command => 'mysqlcheck --databases webportal_customer',
    hour    => ['0'],
    target  => 'root',
    user    => 'root',
  }
}
```

Note that I have changed the title of this resource to be more descriptive. This is just for readability.

Now that I have created a profile, I can create a role for my webportal database server:

```
# site/role/manifests/webportal_database
class role::webportal_database {
  include profile::base                # From Task 1
  include profile::webportal_database # The one we just created
}
```

Now deploy your code.

The last thing we need todo is assign our new role to our database servers. We will do this by going into the Puppet Enterprise console, clicking **Nodes > Classification > Add Group...** This will allow us to add a group for our database servers and assign our new role to it.

Create a group called **Database Servers**. Click the newly created group from the list, and under **Pin specific nodes to the group,** open it up, add your database servers, and click **Pin**. Now go to the **Classes** tab and add your new role. If it is not available yet, you may need to click the **Refresh** button in the middle right of the screen. Once you have added the `role::webportal_database role`, click **Commit Changes**.

Run Puppet on your database servers using `puppet agent -t` or by clicking **Run Puppet** in the console.

Task 3: Standardize NTP

NTP is usually configured when a server is installed and never touched again. As such, it's unlikely the NTP configurations are consistent across all the servers in the infrastructure. Time discrepancies between servers can cause strange failures, and it can be difficult to track down the root cause of these.

The **NTP module** takes care of that problem. Using the NTP module, all your servers will have consistent settings. If you need to change a time server, you simply update the Puppet code and you're done.

To standardize NTP across our entire environment is very easy with Puppet. Simply add the NTP module and its dependencies to your puppetfile:

```
mod 'puppetlabs/stdlib', '4.12.0'
mod 'puppetlabs/ntp',    '4.2.0'
```

Then use the NTP module inside your base SOE like so:

```
class profile::base {

  include ::ntp

  # This is left over from Task 1
  accounts::user { 'monitoring':
    ensure     => present,
    uid        => '450',
    gid        => '450',
    managehome => false,
    locked     => true,
  }
}
```

This will set up NTP on all your systems in its default state. If you would like to change the way it is configured, simply pass parameters to the **ntp** class. Here is a common example of managing NTP with custom servers:

```
class profile::base {

  class { '::ntp':
    servers => [ 'ntp1.corp.com', 'ntp2.corp.com' ],
  }

  # This is left over from Task 1
  accounts::user { 'monitoring':
    ensure     => present,
    uid        => '450',
    gid        => '450',
    managehome => false,
    locked     => true,
  }

}
```

Because this is part of the base profile that we created in Task 1, and every role includes the base profile, it will get applied to every server without us needing to change anything.

Task 4: Disable Windows shutdown event tracker on non-production systems

One thing we commonly find systems administrators do when setting up a Windows server is to disable the shutdown event tracker, particularly on non-production systems. It's a small thing that removes a big annoyance. Puppet can make sure it's always disabled on old and new Windows systems.

The Windows shutdown event tracker is enabled and disabled either through group policies or the registry. In this example, we are going to use the registry module to directly modify the registry and turn it off. First, add the registry module to your puppetfile:

```
mod 'puppetlabs/stdlib',   '4.12.0'

mod 'puppetlabs/registry', '1.1.3'

Then manage the setting in the registry using Puppet like so:

puppet
registry_value { 'HKLM\SOFTWARE\Policies\Microsoft\Windows NT\Reliability\
ShutdownReasonOn':
  ensure => present,
  data   => '0',
  type   => 'dword',
}
```

This code can be placed anywhere you like, but it would make sense to create a profile called `profile::non_prod_base` if you want to disable the Windows shutdown event tracker only on non-production systems, or a profile called `profile::base` if you want to disable it across the board. Note that the names of the profiles don't actually mean anything here — you could put it in a profile called `profile::im_so_hungry`, and as long as you included it in all your roles, it would end up on all your servers. Naming profiles descriptively just makes more sense, and aids communication with others on your team.

Remember, Puppet knows which OS it is working on, and we can access this for use in our code by using the `$::os['family']` fact.

Task 5: Standardize MySQL deployments

MySQL server configuration can be complex. As system administrators learn more about how to manage MySQL servers, or as new administrators take over a system, configurations of these servers change. Some MySQL servers likely accept connections from any system when they shouldn't. Others will still be hosting forgotten databases that no one has taken the time to delete.

Puppet Enterprise provides a way to standardize existing MySQL servers, set up new ones in a standard configuration, and purge unmanaged databases.

Installing MySQL using Puppet is very simple. To get started, let's first add the **MySQL module** and its dependencies to our puppetfile:

```
mod 'puppet/staging',    '2.0.0'
mod 'puppetlabs/stdlib', '4.12.0'

mod 'puppetlabs/mysql',  '3.8.0'
```

Then, to install MySQL server using the default options, simply include the `::mysql::server` class in a profile. Let's create a new one called `profile::mysql`:

```
# site/profile/manifests/mysql.pp
class profile::mysql {
  include ::mysql::server
}
```

For more advanced configuration of the MySQL server, we can pass a hash of options to this class. This is useful when standardizing existing MySQL installations. To do so, simply take the `/etc/my.cnf` config file and convert it into a Puppet hash like so:

```
ini
[section]
thing = X
```

Becomes:

```
$override_options = {
  'section' => {
    'thing' => 'x',
  }
}
```

It can then be used like so:

```
# site/profile/manifests/mysql.pp
class profile::mysql {
  $override_options = {
    'section' => {
      'thing' => 'x',
    }
  }

  class { '::mysql::server':
    override_options => $override_options,
  }
}
```

A common use of this is to ensure that MySQL servers are listening on the correct addresses. It is easy to set up MySQL to listen on `0.0.0.0`, but this means that the database server will accept connections on all its interfaces, even when it may only need to accept connections on a single interface. One way to solve this is to change the address that MySQL listens on to the address of the correct network interface — e.g., `192.168.0.167`. However when we are using Puppet, it is important that we make our code as reusable as possible, and try not to hardcode too many values. This is where Facter comes in.

We can use Facter to dynamically get the IP address of a given network interface and use that value in our Puppet code. To see what values are available to you from Facter, simply run `facter -p` from the command line. For this MySQL example, we will be using the **networking** fact like so:

```
# site/profile/manifests/mysql.pp
class profile::mysql {
  $override_options = {
    'mysqld' => {
      'bind_address' => $::networking['interfaces']['eth0']['ip'],
    }
  }

  class { '::mysql::server':
    override_options => $override_options,
  }
}
```

This means that MySQL will always be listening on the IP address of the **eth0** interface, regardless of what that value is. Now we can apply this code to as many servers as we like, without having to change our Puppet code each time.

Managing databases

Creating MySQL databases in Puppet is simple using the `mysql::db` type:

```
mysql::db { 'webportal_customer':
  ensure   => present,
  user     => 'webportal',
  password => 'hunter2',
}
```

This will set up not only the database itself, but also a user and grants for that user, so it can read and write to the database. If you want to set up a only single database and not the user, you can use the `mysql_database` type. A good way of seeing how this works is to run `puppet resource mysql_database`, which will list all the databases on the system in Puppet code.

Bringing the correct databases under Puppet control on your MySQL servers will go a long way to standardizing those servers. However, just because we are getting Puppet to create 10 databases on a server doesn't mean they will be the only 10 databases on that server. Databases may have been created for testing purposes and never cleaned up, and can easily cause issues going forward. We can deal with this in Puppet by purging all databases that we are not managing, using the resources `resource` like so:

```
# site/profile/manifests/mysql.pp
class profile::mysql {
  # Purge all unmanaged databases
  resources { 'mysql_database':
    purge => true,
  }

  $override_options = {
    'mysqld' => {
      'bind_address' => $::networking['interfaces']['eth0']['ip'],
    }
  }

  # Install MySQL
  class { '::mysql::server':
    override_options => $override_options,
  }

  # Default MySQL tables that need to exist
  mysql_database { 'information_schema':
    ensure  => 'present',
    charset => 'utf8',
    collate => 'utf8_general_ci',
  }
  mysql_database { 'mysql':
    ensure  => 'present',
    charset => 'latin1',
    collate => 'latin1_swedish_ci',
  }
  mysql_database { 'performance_schema':
    ensure  => 'present',
    charset => 'utf8',
    collate => 'utf8_general_ci',
  }

  # Our custom database table
  mysql::db { 'webportal_customer':
    ensure   => present,
    user     => 'webportal',
    password => 'hunter2',
  }
}
```

We could also purge any unused MySQL users by adding the following code snippet:

```
resources { 'mysql_user':
  purge => true,
}
```

It is important to note that Puppet will purge any resources that are not being explicitly managed by Puppet. This could include default MySQL users that the server needs. If you don't want to purge these users, simply use `puppet resource mysql_user` to list them, and add them to your Puppet code.

Congratulations, and a few helpful resources

Wow, you stuck through with us all the way to the end! Thanks for that, and we bet you're already feeling a lot more efficient.

Moving forward, there are quite a few great resources for learning more and doing more with Puppet. Below are a few places to visit on your Puppet learning journey.

- The **Puppet docs site**. You can find helpful documentation here for all things Puppet, including some great introductory material for yourself and your colleagues:
  - **Quick start guide for Windows**
  - **Quick start guide for Unix, Linux and other *nix operating systems**
  - **Puppet Enterprise installation guide**
  - And much, much more.
- The **Puppet Forge**. Here's where you'll find thousands of pre-written, pre-tested modules for automating your fundamental tasks as a system administrator, and modules for more advanced work, too — all created by Puppet employees and Puppet community members. Look for the **Puppet Enterprise supported modules** and **approved modules**, which have passed Puppet engineering tests. You'll also find community ratings for modules.
- **What Windows Admins Need from Configuration Management.** This white paper offers practical tips, resources and some hands-on tutorial help for Windows admins just getting started with Puppet.
- **Continuous Delivery: What it Is and How to Get Started**. This guide defines continuous delivery, and outlines the practices and tools that make it possible.