EXPERIMENT NO: 11

Date of Performance	
Date of Submission	

AIM

To learn Software Configuration Management and provisioning using Puppet Blocks (Manifest, Modules, Classes, Function).

PROBLEM DEFINITION

Acquire knowledge in software configuration management and provisioning using Puppet components such as manifests, modules, classes, and functions.

THEORY

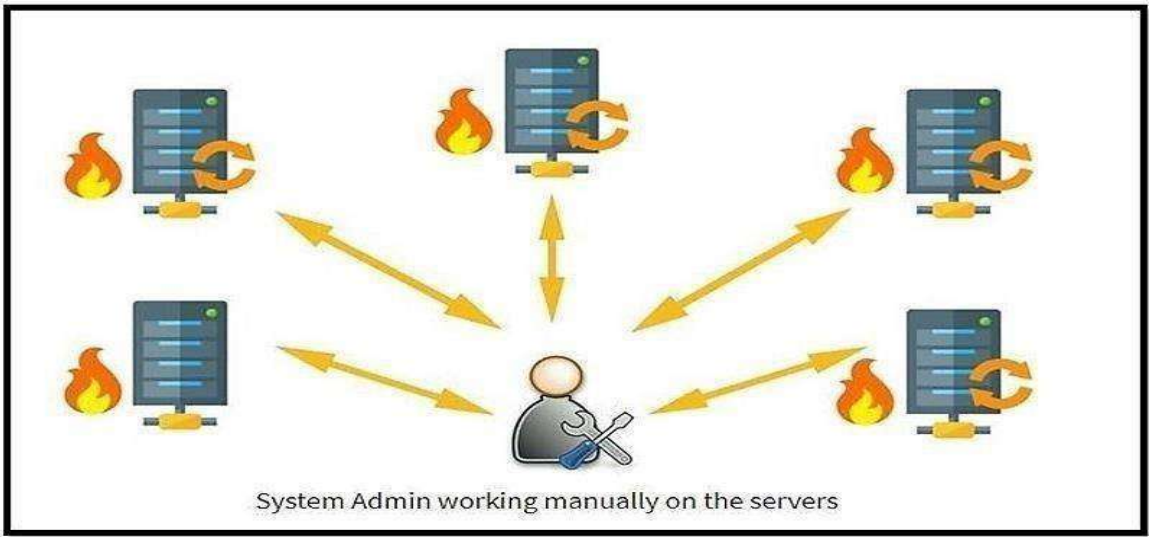
What is Configuration Management? Configuration management is the process of maintaining software and computer systems (e.g., servers, storage, networks) in a known, desired, and consistent state. It also allows access to an accurate historical record of system state for project management and audit purposes. System Administrators mostly perform repetitive tasks like installing servers, configuring those servers, etc. These professionals can automate these tasks by writing scripts. However, it is challenging when working on a massive infrastructure. Configuration management tools like Puppet were introduced to resolve such issues.

What is Puppet? Puppet is a system management tool for centralizing and automating the configuration management process. Puppet is also used as a software deployment tool. It is an open-source configuration management software widely used for server configuration, management, deployment, and orchestration of various applications and services across an organization's infrastructure. Puppet is specially designed to manage the configuration of Linux and Windows systems. It is written in Ruby and uses its unique Domain-Specific Language (DSL) to describe system configuration.

What are the Puppet versions? Puppet comes in two versions:

- **Open Source Puppet:** It is a basic version of the Puppet configuration management tool, also known as Open Source Puppet. It is available directly from Puppet's website and is licensed under the Apache 2.0 system.
  Puppet Enterprise: A commercial version that offers features like compliance reporting, orchestration, role-based access control, GUI, API, and command-line tools for effective management of nodes.

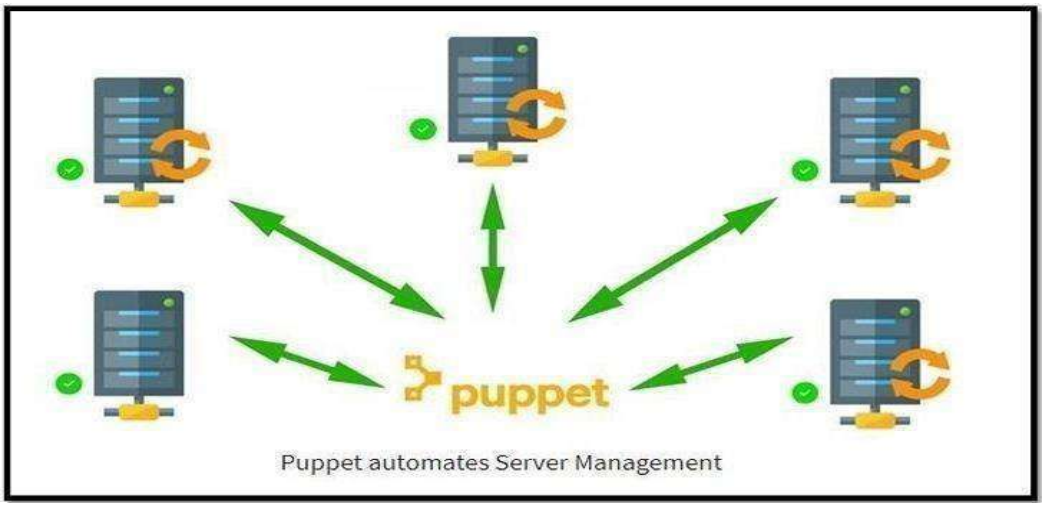
What Puppet can do: For example, if you have an infrastructure with about 100 servers, Puppet allows you to write simple code that can be deployed automatically on these servers. This reduces human effort and makes the development process faster and more effective.



System Admin working manually on the servers

Puppet performs the following functions:

- Puppet allows you to define distinct configurations for every host.
- The tool continuously monitors servers to confirm whether the required configuration exists or not and if it is not altered. If the configuration is changed, the Puppet tool will revert to the pre-defined configuration on the host.
- It also provides control over all the configured systems, so a centralized change gets automatically effected.
- It is also used as a deployment tool as it automatically deploys software to the system. It implements infrastructure as code because policies and configurations are written as code.



Puppet automates Server Management

Puppet DSL and Programming Paradigms: Before learning Puppet DSL, let's understand programming paradigms. A programming paradigm is a style used in computer programming. Four types of paradigms are:

1. Imperative
2. Declarative
3. Functional (a subset of the declarative paradigm)
4. Object-oriented

We will focus on Imperative and Declarative.

Imperative Paradigms: This programming paradigm expresses the logic of a computation (what to do) and describes its control flow (how to do it). Example: If you are going to your office, you book a cab and give step-by-step directions to the driver until you reach the office. Specifying what to do and how to do it is an imperative style.

Declarative Paradigms: This programming paradigm expresses the logic of a computation (what to do) without describing its control flow (how to do it). Example: If you are going to your office, you book an Uber cab and specify the final destination (office). Specifying what to do, not how to do it, is a declarative style. Puppet uses a declarative programming approach.

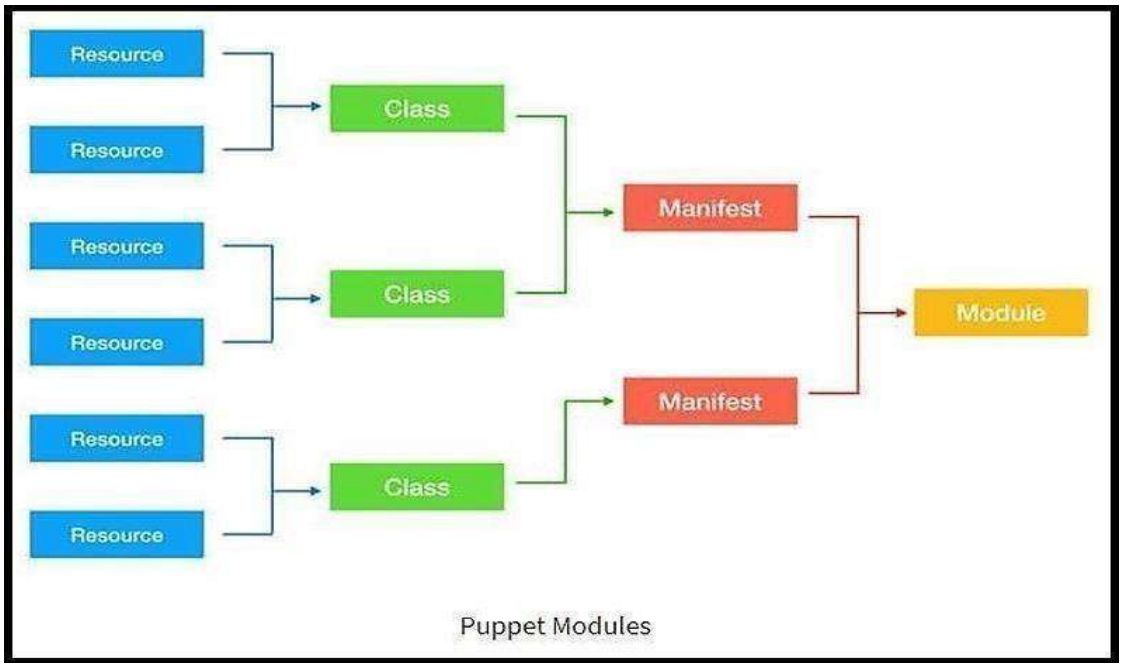
Example: Create a user on the system. It can be done using the imperative programming pattern by a shell script: Here, we specify how to create the user and what commands to use on the operating system. However, it can be done using the declarative programming pattern with only a few lines of Puppet code, Puppet DSL, and still achieve the same result.

Deployment models of configuration management tools: There are two deployment models for configuration management tools:

- **Push-based deployment model:** Initiated by a master node.
- **Pull-based deployment model:** Initiated by agents.

Push-based deployment model: In this deployment model, the master server pushes the configurations and software to the individual agents. After verifying a secure connection, the master runs commands remotely on the agents. For example, Ansible and Salt Stack.

Pull-based deployment model: In this deployment model, individual servers contact a master server, verify and establish a secure connection, download their configurations and software, and then configure themselves accordingly—for example, Puppet and Chef.



Puppet Modules

Types of Puppet resources

In general, a system consists of files, users, services, processes, packages, etc. In Puppet, these are called resources. Resources are the fundamental building blocks in Puppet. All the operations on puppet agents are performed with the help of puppet resources. Puppet resources are the readymade tools that are used to perform various tasks and operations on any supported platform. We can use a single puppet resource to performa specific task, or we can use multiple puppet resources together to perform some complex application configurations deployments.

Resources can have different types. Puppet uses resources and resource types in order to describe a system's configuration. There are three kinds of resource types:

1. Puppet core or built-in resource types.

2. Puppet defined resource types.

3. Puppet custom resource types.

Puppet core or built-in resource types:

Core or built-in resource types are the pre-built puppet resource types shipped with puppet software. All of the core or built-in Puppet resource types are written and maintained by Puppet team.

Puppet defined resource types: Defined resource types are lightweight resource types written in Puppet declarative language using a combination of existing resource types.

Puppet custom resource types:

Custom resource types are completely customized resource types written in Ruby. command to display a list of Puppet relevant subcommands:

In our case, we are interested in the subcommand "resource" which we will use to find the information about inbuilt puppet resource types. In the terminal, type any of the following commands to display a list of actions associated with the puppet subcommand "resource":

In this case, we have the resource as subcommand and –types as action. Puppet has 49 inbuilt core resource types. In the terminal, type the following command to display a list of available inbuilt puppet resource types:

Each type supports a list of attributes. These attributes provide a detailed description that Puppet uses to manage the resource. To find out all the attributes associated with the puppet resource type, use the following command:

puppet describe <resource type name>

Parameters will list all the available attributes for that resource type. puppet describe package. It's hard for a new person to understand and relate many unmanaged puppet code files. This is where we need some grouping to tie together operations. The aim is to solve a single problem, such as all operations required to configure ssh on a server or ntp service or a complete web server or database server from scratch.

What Are Puppet Classes?
Puppet classes are the collection of Puppet resources bundled together as a single unit. Puppet introduced classes to make the structure reusable and organized. First, we need to define a class using class definition syntax; classes must be unique and can be declared only once with the same name:

```
class <class-name> {
<Resource declarations>
}
```

Example:

```
class ntpconfig {

  file {
    "/etc/ntp.conf":
    ensure => "present",
    content => "server 0.centos.pool.ntp.org iburst\n",
  }
}
```

So far, we have only defined the class, but we have not used it anywhere. Meaning this code that we have written will never get executed unless we declare this class elsewhere.

Class Declaration
To use a defined class in code, use the include keyword.

```
class ntpconfig {
  file {
  "/etc/ntp.conf":
    ensure => "present",
    content => "server 0.centos.pool.ntp.org iburst\n",
  }
}

include ntpconfig
```

Demo Install NTP First, make sure the NTP package is not already present on the server; the following command will return nothing if the telnet is not present on the server:

As we can see, the NTP package is already present on the server. Let's remove the existing NTP package:

After removing the package, ensure that the ntp.conf file is not existing:

Verify the NTP service does not exist by running the following command:

Create a new .pp file to save the code. From the command line:

Change to insert mode by pressing i from the keyboard. Code to create a new file:

After done with Editing:Press Esc. To save the file, press :wq!.

Next step is to check whether the code has any syntax errors. Execute the following command:
puppet parser validate demontp.pp

Make sure that you switch to root to be able to complete the test without any error, by executing the command:
su root

Testing is the next step in the code creation process. Execute the following command to perform a smoke test:
puppet apply demontp.pp --noop

Last step is to run Puppet in real mode and verify the output. puppet apply demontp.pp

Puppet didn't perform anything because the demo class was just defined but not declared.

So, until you declare the Puppet class, the code will not get applied. Let's declare the demo class inside the same code using include class name at the end of the code:

Again check whether the code has any syntax errors. Execute the following command: puppet parser validate demontp.pp

Make sure that you switch to root to be able to complete the test without any error, by executing the command: su root

Testing is the next step in the code creation process. Execute the following command to perform a smoke test: puppet apply demontp.pp --noop

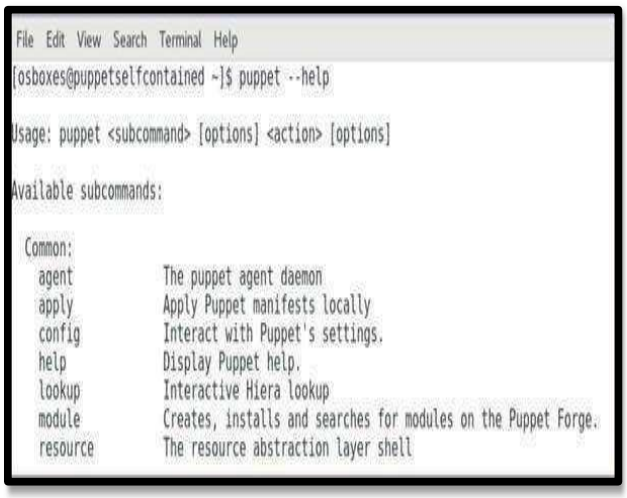
Last step is to run Puppet in real mode and verify the output. puppet apply demontp.pp

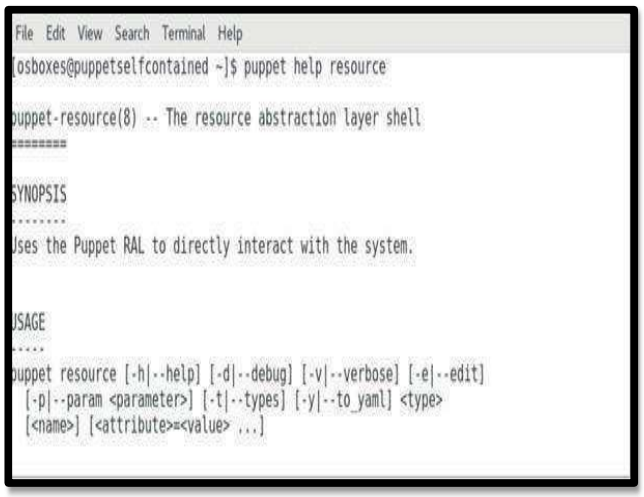
This time the code gets applied because the class was defined and then declared.

Ensure that ntp.conf is now existing: ls -lrt /etc/ntp.conf

Verify the NTP service has been started by running the following command: systemctl status ntpd

OUTPUT





```
File  Edit  View  Search  Terminal  Help
[osboxes@puppetselfcontained ~]$ puppet resource --types
augeas
cron
exec
file
filebucket
group
host
mount
notify
package
resources
schedule
scheduled_task
selboolean
selmodule
service
ssh_authorized_key
sshkey
stage
tidy
user
whit
yumrepo
zfs
```

```
File  Edit  View  Search  Terminal  Help
[osboxes@puppetselfcontained ~]$ puppet describe package

package
=======
Manage packages.  There is a basic dichotomy in package
support right now:  Some package types (such as yum and apt) can
retrieve their own package files, while others (such as rpm and sun)
cannot.  For those package formats that cannot retrieve their own files,
you can use the `source` parameter to point to the correct file.
Puppet will automatically guess the packaging format that you are
using based on the platform you are on, but you can override it
using the `provider` parameter; each provider defines what it
requires in order to function, and you must meet those requirements
to use a given provider.
You can declare multiple package resources with the same `name`, as long
as they specify different providers and have unique titles.
Note that you must use the _title_ to make a reference to a package
resource; `Package[<NAME>]` is not a synonym for `Package[<TITLE>]` like
it is for many other resource types.
**Autorequires:** If Puppet is managing the files specified as a
package's `adminfile`, `responsefile`, or `source`, the package
resource will autorequire those files.


Parameters
----------
```

```
File  Edit  View  Search  Terminal  Help


Parameters
----------

- **adminfile**
  A file containing package defaults for installing packages.
  This attribute is only used on Solaris. Its value should be a path to a
  local file stored on the target system. Solaris's package tools expect
  either an absolute file path or a relative path to a file in
  `/var/sadm/install/admin`.
  The value of `adminfile` will be passed directly to the `pkgadd` or
  `pkgrm` command with the `-a <ADMINFILE>` option.

- **allow_virtual**
  Specifies if virtual package names are allowed for install and
  uninstall.
  Valid values are `true`, `false`, `yes`, `no`.
  Requires features virtual_packages.

- **allowcdrom**
  Tells apt to allow cdrom sources in the sources.list file.
  Normally apt will bail if you try this.
  Valid values are `true`, `false`.

- **category**
```

```
File  Edit  View  Search  Terminal  Help
[root@puppetselfcontained etc]# rpm -qa | grep -i ntp
ntp-4.2.6p5-28.el7.centos.x86_64
python-ntplib-0.3.2-1.el7.noarch
fontpackages-filesystem-1.44-8.el7.noarch
ntpdate-4.2.6p5-28.el7.centos.x86_64
[root@puppetselfcontained etc]#
```

```
File  Edit  View  Search  Terminal  Help
[root@puppetselfcontained etc]# ls -lrt /etc/ntp.conf
ls: cannot access /etc/ntp.conf: No such file or directory
[root@puppetselfcontained etc]#
```

```
File  Edit  View  Search  Terminal  Help
[root@puppetselfcontained demo]# systemctl status ntpd
Unit ntpd.service could not be found.
[root@puppetselfcontained demo]#
```

Testing

CONCLUSION:

Thus, learning software configuration management and provisioning with Puppet Blocks (Manifest, Modules, Classes, Function) provided in-depth knowledge for effective configuration and automation.