Components of a Puppet architecture

This tutorial has been taken from the second edition of Extending Puppet. Until January 9th you can pick it up for just $5. In fact - you can pick up any eBook or video from Packt for that price - so check their site out today.

With Puppet, you manage your systems using the catalog that the Puppet Master compiles for each node. This is the total sum of all the resources we have declared in our code, based on parameters and variables whose values reflect our logic and needs.

Most of the time, you also provide configuration files either as static files or viB templates, populated according to the variables you have set.
You can identify the following major tasks when we have to manage what we want to configure on our nodes:

- Definition of the classes to be included in each node
- Definition of the parameters to use for each node
- Definition of the configuration files provided to the nodes

These tasks can be provided by different, partly interchangeable, components:

- `site.pp`, the first file parsed by the Puppet Master and eventually all the files imported from there (import `nodes/*.pp` would import and parse all the code defined in the files with `.pp` suffix in the `/etc/puppet/manifests/nodes/` directory). Here we have code in Puppet language.
- An **External Node Classifier** (**ENC**) is an alternative source which can be used to define classes and parameters to apply to nodes. It's enabled with the following lines on the Puppet Master's `puppet.conf`:

```
[master]
node_terminus = exec
external_nodes = /etc/puppet/node.rb
```

What's referred by the `external_nodes` parameter can be any script that uses any backend; it's invoked with the client's `certname` as first argument (for example, `/etc/puppet/node.rb web01.example.com`) and should return a YAML-formatted output that defines the classes to include for that node, the parameters, and the Puppet environment to use.

Besides the well-known Puppet-specific ENCs, such as the **Foreman** and **Puppet Dashboard** (a former Puppet Labs project now maintained by community members), it's not uncommon to write custom ones that leverage on existing tools and infrastructure management solutions:

- **LDAP** can be used to store nodes information (classes, environment, and variables) as an alternative to the usage of an ENC. To enable LDAP integration, add the following to the Master's `puppet.conf`:

```
[master]
node_terminus = ldap
```

```
ldapserver = ldap.example.com
ldapbase = ou=Hosts,dc=example,dc=com
```

- Then we have to add Puppet's schema to our LDAP server. For more information and details, check: http://docs.puppetlabs.com/guides/ldap_nodes.html
- **Hiera** is the hierarchical key-value data store. It's embedded in Puppet from version 3 and is available as an add-on on previous versions. Here we can set parameters, but also include classes and eventually provide contents for files.
- **Public** modules can be retrieved from the Puppet Forge, GitHub, or other sources; they typically manage applications and system settings. Being public, they might not fit all our custom needs, but they are supposed to be reusable, support different OS, and adapt to different usage cases. We are supposed to use them without any modification, as if they were public libraries, committing back to the upstream repository our fixes and enhancements. A common, but less recommended alternative is to fork a public module and adapt it to our needs. This might seem a quicker solution, but doesn't definitively help the open source ecosystem and would prevent us from having benefits from updates on the original repository.
- **Site module(s)** are custom modules with local resources and files where we can place all the logic we need, or the resources we can't manage, with public modules. They may be one or more, and may be called `site` or have the name of our company, customer, project, or anything in general. Site modules have particular sense as companions to public modules when used without local modifications; on site modules, we can place local settings, files, custom logic, and resources.

Let's see how these components fit into different Puppet tasks.

Defining the classes to include in each node

This is typically done when, in Puppet, we talk about **nodes' classification**: the task that the Puppet Master accomplishes when it receives a client's request and determines the classes and parameters to use when compiling the relevant catalog.

Nodes' classification can be done in different ways:

- We can use the `node` object on `site.pp` and other manifests eventually imported from there. In this way we identify each node by `certname` and declare all the resources and classes we want for it:

```
node 'web01.example.com' {
  include ::general
  include ::apache
}
```

- Here we may even decide to follow a **node-less** layout, where we don't use the node object at all and rely on facts to manage what classes and parameters to assign to our nodes. An example of this approach is examined later.
- On an **ENC** where can be defined the classes (and parameters) that each node should have. The returned YAML for our simple case would be something like the following:

```
---
classes:
  - general:
  - apache:
parameters:
  dns_servers:
    - 8.8.8.8
    - 8.8.4.4
  smtp_server: smtp.example.com
environment: production
```

- Via LDAP, where we can have a hierarchical structure where a node can inherit the classes (referenced with the `puppetClass` attribute) set in a parent node (`parentNode`).
- On Hiera, using the `hiera_include` function; just add in `site.pp`:

```
hiera_include('classes')
```

- On **site module(s)** any custom logic can be placed as, for example, the classes and resources to include for all the nodes or for specific groups of nodes.

Defining the parameters to use for each node

This is another crucial part, as with parameters we can characterize our nodes and define the resources we want for them.

Generally, to **identify** and characterize a node in order to differentiate it from the others and provide the specific resources we want for it, we need very few key parameters, such as (names used here may be common, but are arbitrary and are not Puppet's internal ones):

- **role**, is almost a standard de facto name to identify the kind of server; a node is supposed to have just one role, which might be something like `webserver`, `app_be`, `db` or anything that identifies the function of the node. Note that web servers that serve different web applications should have different roles (that is `webserver_site`, `webserver_blog`). We can have one or more nodes with the same role.

- **env**, or any name that identifies the operational environment of the node (is it a `development`, `test`, `qa`, or `production` server?).
- **zone**, **site**, **datacenter**, **country**, or any parameter that might identify the network, country, availability zone, or datacenter where the node is placed. A node is supposed to belong to only one of these. We might not require this in our infrastructure.
- **tenant**, **component**, **application**, **project**, or cluster might be other kinds of variables that characterize our node. There's no real standard for their naming, and their usage and necessity strictly depend on the underlying infrastructure.

With parameters like these, any node can be fully identified and be served with any specific configuration. It makes sense to provide them, where possible, as facts.

The parameters and the variables we use in our manifests may have different natures, such as:

- Role/env/zone as defined before, to identify the nodes; they are typically used to determine the values of other parameters
- OS related parameters, like package names and file paths
- Variables that define services of our infrastructure (DNS servers, NTP servers...)
- Usernames and passwords, which should be reserved, used to manage credentials
- Parameters that express any further custom logic and classifying need (master, slave, host_number…)
- Parameters exposed by the parameterized classes or defines we use

Many times, the values of these variables and parameters have to change according to the values of other variables, and it's important to have a general idea, from the beginning, of what the variations involved and the possible exceptions are, as we will probably define our logic according to them. As a general rule we will most of the time use the **identifying parameters** (role/env/zone...) to define most of the other parameters, so we'll probably need to use them in our Hiera hierarchy or in Puppet selectors. This also means that we will probably need toset them as **top scope variables** (for example via an ENC) or **facts**.

As with classes to include, parameters may be set by various components; some of them are actually the same, since in Puppet, node classification involves both classes to include and parameters to apply:

- On `site.pp`, we can set variables. If they are outside nodes' definitions they are at top scope, and if they are inside they are at node scope. Top scope variables should be referenced with a `::` prefix, for example `$::role`. Node scope variables are available inside the node's classes with their plain name: `$role`.
- An ENC returns parameters, treated as top scope variables, alongside classes, and the logic of how they can be set depends entirely on their structure. Popular ENCs such as the Foreman, Puppet Dashboard, and Puppet Enterprise allow users to set variables for single nodes or for groups, often in a hierarchical fashion. The kind and amount of parameters set here depend on how much information we want to manage on the ENC and how much to manage somewhere else.
- LDAP, when used as node classifier, returns variables for each node as defined with the `puppetVar` attribute. They are all set at top scope.
- On **shared modules**, we typically deal with OS specific parameters. Modules should be considered as reusable components that know all about how to manage the homonymous application on different OS but nothing about custom logic: they should expose parameters and defines that allow users to determine their behavior and fit their own needs.

- On **site module(s)**, we can place infrastructural parameters or any custom logic more or less based on other variables.
- Finally, it's possible, and generally recommended, to create custom facts that identify the node directly from the client. A case of this approach is a totally facts-driven infrastructure, where all the nodes identifying variables, upon which all the other parameters are defined, are set as facts.

Defining the configuration files provided to the nodes

It's almost certain that we will need to manage configuration files with Puppet and that we need to store them somewhere, either as plain static files to serve via Puppet's fileserver functionality using the `source` argument of the `File` type, or via `.erb` templates.

While it's possible to configure custom fileserver shares for static files and absolute paths for templates, it's definitely recommended to rely on the modules' auto-loading conventions and place such files inside custom or shared modules, unless we decide to use Hiera for them.

Configuration files, therefore, are typically placed in the following:

- **Shared modules**: These may provide default templates that use variables exposed as parameters by the modules' classes and defines them. As users, we don't directly manage the module's template but the variables used inside it. A good and reusable module should allow us to override the default template with a custom one. In this case, our custom template should be placed in a site module. If we've forked a public shared module and maintain a custom version,

we might be tempted to place all our custom files and templates there. In doing so, we lose in reusability and gain, maybe, in short term usage simplicity.

- **Site module(s)**: These are, instead, a more correct place for custom files and templates if we want to maintain a setup based on public shared modules, which are not forked, and custom site ones where all our stuff stays confined in a single or few modules. This allows us to recreate similar setups just by copying and modifying our site modules, as all our logic, files, and resources are concentrated there.
- **Hiera**: Thanks to the smart `hiera-file` backend, this can be an interesting alternative place to store configuration files, both static ones and templates. We can benefit from the hierarchy logic that works for us and can manage any kind of file without touching modules.
- Custom **fileserver** mounts can be used to serve any kind of static files from any directory of the Puppet Master. They can be useful if we need to provide via Puppet files generated/managed by third party scripts or tools. An entry in `/etc/puppet/fileserver.conf` is like the following:

```
[data]
path /etc/puppet/static_files
allow *
```

- Allows us to serve a file such as `/etc/puppet/static_files/generated/file.txt` with the argument:

```
source => 'puppet:///data/generated/file.txt',
```

Defining custom resources and classes

We'll probably need to provide custom resources to our nodes that are not declared in the shared modules because they are too specific, and we'll probably want to create some grouping classes, for example, to manage the common baseline of resources and classes we want applied to all our nodes.

This is typically a bunch of custom code and logic that we have to place somewhere. The usual locations are as follows:

- **Shared modules**: These are forked and modified to include custom resources; as already outlined, this approach doesn't pay in the long term.
- **Site module(s)**: The preferred place-to-place custom stuff, including some classes where we can manage common baselines, role classes and other container classes.

- **Hiera**: Partially, if we are fond of the `create_resources` function fed by hashes provided in Hiera. In this case, in a site, shared module, or maybe even in `site.pp`, we have to place the `create_resources` statements somewhere.

Thanks for reading this tutorial from the second edition of Extending Puppet.