INFRASTRUCTURE AS CODE

- Current state of industry
- Infrastructure as Code - Main concepts
- Configuration management with Puppet

CURRENT STATE OF INDUSTRY

(parts that affect you as a DevOps Engineer)

AGILE WORKFLOW

Agile (SCRUM, Kanban) workflow focuses on creating a product through fast iterations and short feedback loop

- Product will be changing often and may change completely over a course of iterations
- Every change to a product is a potential change to your infrastructure

APPLICATION ENVIRONMENTS

Each change may pass zero or more acceptance phases, for example:

- Automatic testing
- Automatic integration testing
- Manual testing
- Acceptance testing

Typically, following environments exist:

- Test (Automatic tests happen here, also used by devs for convenience)
- Stage (Manual and acceptance tests happen here)
- Production (Final product, end users interact with it)

What this means for you:

DEV/PROD PARITY

Maintaining a set of application *environments*, one for each such phase. They must be as close as possible to each other in terms of configuration.

CONTINUOUS DELIVERY

Continuous Delivery takes Agile approach further

Absolutely every change to a product is a potential Release Candidate and must be treated as such

What this means for you:

- From **1 test deployments per day** to **50+ test deployments per day**
- From **1 release per week** to **10+ releases per day**

APPLICATIONS AT SCALE

An established product will, most likely, grow beyond a few servers

What this *potentially* means for you:

- Managing from **10+** to **1000+** nodes
- Nodes will be, typically, divided into groups that share configuration (Database, Application, Load Balancing, etc.)

HIGH AVAILABILITY AND RESILIENCE

An established product must be resilient enough to handle unexpected disasters or a sudden increase in load

What this means for you:

- Building an infrastructure with redundancy in mind
- Be able to replace or add required amount of new nodes in reasonable time

SERVICE ORIENTED ARCHITECTURE

SOA has recently gained a lot of popularity. It's core concept is breaking a monolithic application into group of independent services which interact together, forming a product.

What this *potentially* means for you:

- Supporting **10+** separate services that interact with each other. Each service may potentially have different requirements to infrastructure.

CONCEPTS

3.1

IMPERATIVE VS DECLARATIVE

- Imperative: Install this
- Declarative: I need this to be installed

IMPERATIVE

Traditional bash automation is the imperative approach. It holds a set of actions that need to be executed:

```
#!/bin/bash

apt-get -y install httpd
```

DECLARATIVE

On the other hand, declarative approach describes the desired system state. It focuses on what needs to be done, rather than how to do it:

```
package { "httpd":
  ensure => present,
}
```

INFRASTRUCTURE AS CODE

Declarative approach together with tooling that understands it makes defining your infrastructure as code possible

Main points of IaC are:

- Automate
- Reduce and Reuse
- Version control all the changes
- Code is documentation, so focus on documenting important things

A few sidenotes about

VERSION CONTROL

As long as you stick to the declarative approach, you can use any Version Control System to track changes to your infrastructure.

With *Git* it will look like this

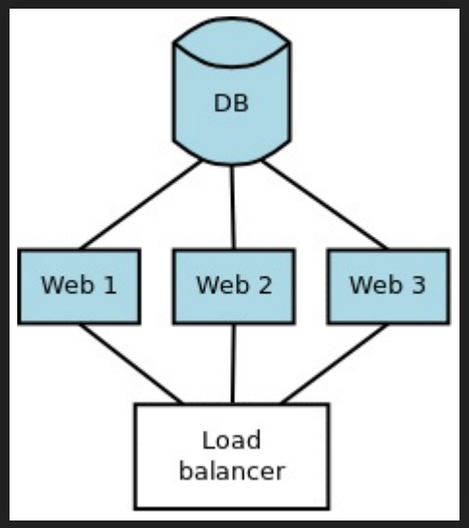
```
# Someone committed on 23.08.27
# Make sure httpd package is gone

@@ -1,2 +1,2 @@
package { "httpd":
-  ensure => present,
+  ensure => absent,
}
```

Since there are tools that understand and apply
declarative configurations across thousands of nodes,
a few approaches were borrowed from the SOA.

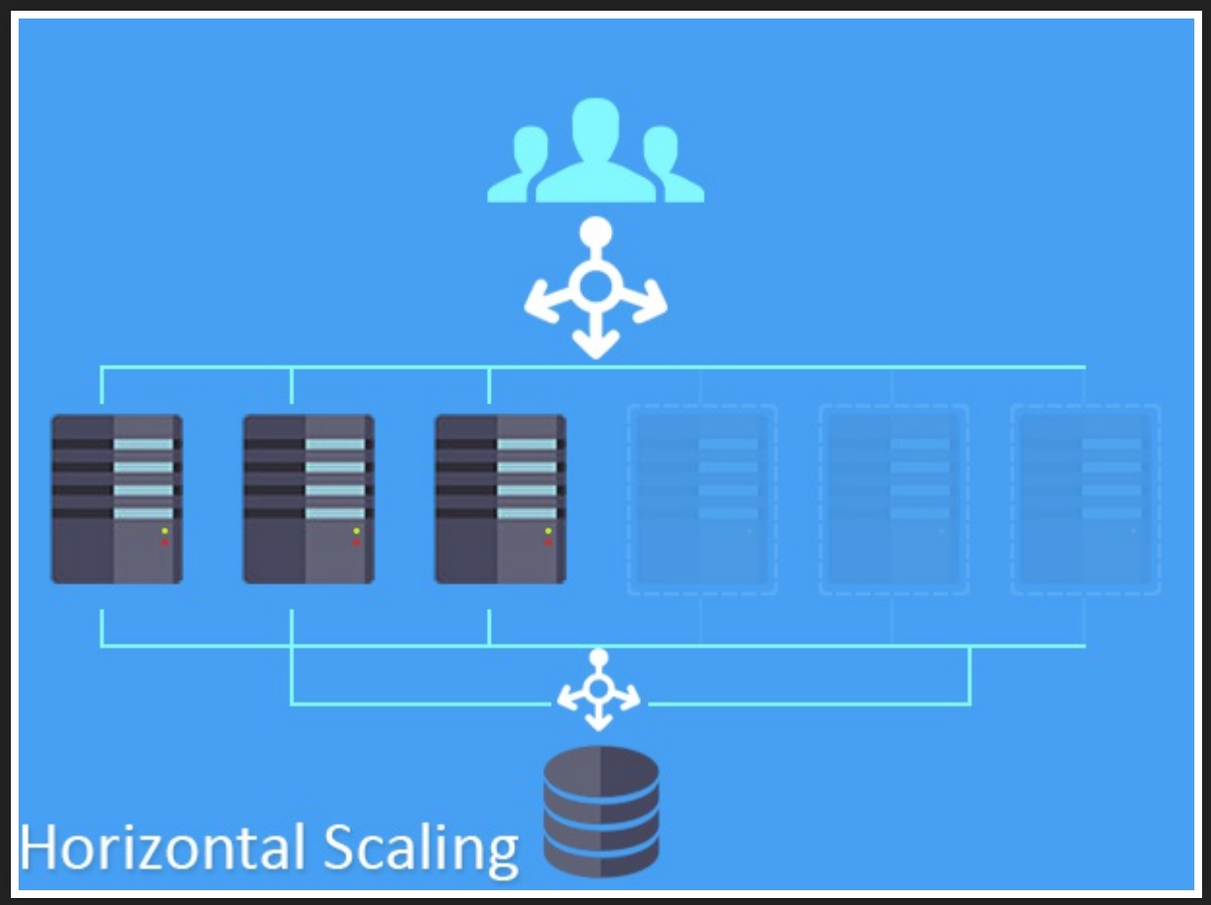
ORCHESTRATION

Means controlled series of changes to infrastructure.
For example, rolling deployments - each app node is taken off load balancing, updated, then put back on:



CHOREOGRAPHY

Means initiated by infrastructure itself series of changes. For example, automatic cloud scaling in response to increased CPU load on app nodes:



PUPPET

- A Configuration Management Tool
- A framework for Systems Automation
- A Declarative Domain Specific Language
- Works on Linux, Unix (Solaris, AIX, *BSD), MacOS, Windows
- Used by: Uber, Salesforce, ...

SOFTWARE RELATED TO PUPPET

- Facter - Complementary tool to retrieve system's data
- Hiera - a Key-Value storage
- PuppetDB - a general storage for data generated by Puppet
- Foreman - well-known third-party complementary tool

INSTALLATION

Debian, Ubuntu - Available by default

```
apt-get install puppet        # On clients (nodes)
apt-get install puppetmaster # On server (master)
```

RedHat, Centos, Fedora - Add EPEL repository or RHN Extra channel

```
yum install puppet         # On clients (nodes)
yum install puppet-server # On server (master)
```

Or use the official puppet repository for latest updates

PUPPET LANGUAGE AND CONCEPTS

- Introduction to Puppet language and terms
- Resource types
- Classes and defines
- Variables and parameters
- Nodes classification

PUPPET LANGUAGE

- A Declarative Domain Specific Language
- It defines *STATES* (Not procedures)
- Puppet code is written in *manifests* (files with .pp extension)
- In the code we declare *resources* that affect elements of the system (files, packages, services ...)
- Resources are grouped in *classes* which may expose parameters that affect their behavior.
- Classes and configuration files are organized in *modules*.

NODES CLASSIFICATION

When clients connect, the Puppet Master generates a catalog with the list of of the resources that clients have to apply locally. The Puppet Master has to classify nodes and define for each of them:

- The classes to include
- The parameters to pass
- The Puppet environment to use

The catalog is generated by the Master according to the logic of our Puppet code and data.

RESOURCE TYPES (TYPES)

Resource Types are single units of configuration composed by:

- A type (package, service, file, user, mount, exec ...)
- A title (how is called and referred)
- Zero or more arguments

```
type { 'title':
  argument  => value,
  other_arg => value,
}
```

For example, a file resource type:

```
file { 'motd':
  path    => '/etc/motd',
  content => 'Tomorrow is another day',
}
```

RESOURCE TYPES REFERENCE

Find a complete reference online

Or use `puppet` cli tool

```
# Describe specific resource type
puppet describe file

# Describe all resource types available
puppet describe --list
```

SIMPLE RESOURCE EXAMPLES

Installation of OpenSSH package

```
package { 'openssh':
  ensure => present,
}
```

Creation of /etc/motd file

```
file { 'motd':
  path => '/etc/motd',
}
```

Start of httpd service

```
service { 'httpd':
  ensure => running,
  enable => true,
}
```

MORE COMPLEX RESOURCE EXAMPLES

Management of nginx service with parameters defined in module's variables

```
service { 'nginx':
  ensure      => $::nginx::manage_service_ensure,
  name        => $::nginx::service_name,
  enable      => $::nginx::manage_service_enable,
}
```

Creation of nginx.conf with content retrieved from different sources (first found is served)

```
file { 'nginx.conf':
  ensure  => present,
  path    => '/etc/nginx/nginx.conf',
  source  => [
      "puppet:///modules/site/nginx.conf--${::fqdn}",
      "puppet:///modules/site/nginx.conf" ],
}
```

RESOURCE ABSTRACTION LAYER

Resources are abstracted from the underlying OS. Resource types have different providers for different OS. The package type is known for the great number of providers:

```
aix
appdmg
apple
apt
aptitude
aptrpm
blastwave
dnf
dpkg
fink
freebsd
gem
hpux
macports
nim
```

CLASSES - DEFINITION

Classes are containers of different resources. Example of a class definition:

```
class mysql (
  root_password => 'default_value',
  port          => '3306',
) {
  package { 'mysql-server':
    ensure => present,
  }
  service { 'mysql':
    ensure     => running,
  }
  [...]
}
```

Note that when we define a class we just describe what it does and what parameters it has, we don't actually add it and its resources to the catalog.

CLASSES - DECLARATION

When we have to use a class previously defined, we declare it.

Without parameters:

```
include mysql
```

With parameters:

```
class { 'mysql':
  root_password => 'my_value',
  port          => '3307',
}
```

DEFINES

Also called: Defined resource types or defined types.
Similar to parametrized classes but can be used
multiple times (with different titles).

Declaration:

```
define apache::virtualhost (
  $ensure   = present,
  $template = 'apache/virtualhost.conf.erb' ,
  [...] ) {

  file { "ApacheVirtualHost_${name}":
    ensure  => $ensure,
    content => template("${template}"),
  }
}
```

Definition:

```
apache::virtualhost { 'www.example.com':
  template => 'site/apache/www.example.com-erb'
}
```

VARIABLES

In our code we can define our variables and use other ones that may come from different sources:

- *Facts* generated directly by the client
- *Parameters* obtained from node's classification

FACTS

Facter runs on clients and collects facts that the server can use as variables:

```
$ facter

architecture => x86_64
fqdn => example.com
hostname => Macante
interfaces => lo0,eth0
ipaddress => 10.42.42.98
ipaddress_eth0 => 10.42.42.98
kernel => Linux
macaddress => 20:c9:d0:44:61:57
macaddress_eth0 => 20:c9:d0:44:61:57
memorytotal => 16.00 GB
netmask => 255.255.255.0
operatingsystem => Centos
operatingsystemrelease => 6.3
```

PARAMETERS

We can define custom variables in different ways:

- In Puppet manifests:

```
$package = $::operatingsystem ? {
/(?i:Ubuntu|Debian|Mint)/ => 'apache2',
default                   => 'httpd',
}
```

- In an External Node Classifier (ENC)
- In an Hiera backend:

```
$syslog_server = hiera(syslog_server)
```

BUILT-IN VARIABLES

Puppet provides some useful built in variables, they can be:

- `$environment` (default: production) - the Puppet environment where are placed modules and manifests.
- `$servername`, `$serverip` - the Puppet Master FQDN and IP
- `$serverversion` - the Puppet version on the server
- `$settings::<name>` - any configuration setting on the Master's puppet.conf

ENVIRONMENTS

Puppet environments allow isolation of Puppet code and data: for each environment we can have different paths manifest files, Hiera data and modules.

Puppet's environments DO NOT necessarily have to match the operational environments of our servers.

Environments are configured in `puppet.conf` as follows:

```
[main]
environmentpath = $configdir/
```

Then inside the `/etc/puppet/environments/$environment/` directory we have:

```
modules/    # Directory containing modules
manifests/ # Directory containing site.pp
environment.conf # Conf file for the environment
```

NODES - DEFAULT CLASSIFICATION

A node is identified by the PuppetMaster by its *certname*, which defaults to the node's *fqdn*

In the first manifest file parsed by the Master, site.pp, we can define nodes with a syntax like:

```
node 'web01' {
  include apache
}
```

We can also define a list of matching names:

```
node 'web01', 'web02', 'web03' {
  include apache
}
```

or use a regular expression:

```
node /^www\d+$/ {
  include apache
}
```

NODES CLASSIFICATION VIA AN ENC

Puppet can query an external source to retrieve the classes and the parameters to assign to a node. This source is called External Node Classifier (ENC) and can be anything that, when interrogated via a script with the clients' certname as first parameter, returns a yaml file with the list of classes and parameters.

Common ENCs are:

- Puppet DashBoard
- The Foreman
- Puppet Enterprise

To enable the usage of an ENC in `puppet.conf`:

```
# Enable the usage of a script to classify nodes
node_terminus = exec

# Path of the script to execute to classify nodes
external_nodes = /etc/puppet/node.rb
```

THE CATALOG

The catalog is the complete list of resources, and their relationships, that the Puppet Master generates for the client.

The client uses the RAL (Resource Abstraction Layer) to execute the actual system's commands that convert abstract resources like

```
package { 'openssh': }
```

to their actual fulfillment on the system

```
apt-get install openssh , yum install openssh ...
```

PUPPET CONFIGURATION AND BASIC USAGE - OVERVIEW

- Operational modes: apply / agent
- Configuration files and options
- Anatomy of a Puppet run

MASTERLESS - PUPPET APPLY:

- Puppet code (written in manifests) is applied directly on the target system.
- No need of a complete client-server infrastructure.
- Have to distribute manifests and modules to the managed nodes.
- Command used: `puppet apply` (generally as root)

Puppet manifests are deployed directly on nodes and applied locally:

```
puppet apply --modulepath /modules/ /manifests/file.pp
```

- More fine grained control on what goes in production for what nodes
- Ability to trigger multiple truly parallel Puppet runs
- No single point of failure, no Master performance issues
- Need to define a fitting deployment workflow

MASTER / CLIENT - PUPPET AGENT:

- We have clients, managed nodes, where Puppet client is installed.
- And we have one or more Masters where Puppet server runs as a service
- Client/Server communication is via https (port 8140)
- Clients certificates have to be accepted (signed) on the Master
- Command used on the client: `puppet agent` (generally as root)
- Command used on the server: `puppet master` (generally as puppet)

CERTIFICATES MANAGEMENT

On the Master use puppet cert to manage certificates

- List the (client) certificates to sign:

```
puppet cert list
```

- List all certificates: signed (+), revoked (-), to sign ( ):

```
puppet cert list --all
```

- Sign a client certificate:

```
puppet cert sign <certname>
```

- Remove a client certificate:

```
puppet cert clean <certname>
```

CERTIFICATES MANAGEMENT - FIRST AGENT RUN

By default the first Puppet run on a client fails:

```
node $ puppet agent

> Exiting; no certificate found and waitforcert is disabled
```

An optional `--waitforcert 60` parameter makes client wait 60 seconds before giving up.

The server has received the client's CSR which has to be manually signed:

```
master $ puppet cert sign <certname>
```

Once signed on the Master, the client can connect and receive its catalog.

PUPPET CONFIGURATION: PUPPET.CONF

Puppet main configuration file, generally found in:

```
/etc/puppet/puppet.conf
```

Configurations are divided in `[sections]` for different Puppet sub commands:

- Common for all commands: `[main]`
- For puppet agent (client): `[agent]`
- For puppet apply (client): `[user]`
- For puppet master (server): `[master]`

MAIN CONFIGURATION OPTIONS

To view all or a specific configuration setting:

```
puppet config print all
puppet config print modulepath
```

Important options in `[main]` section:

- `vardir`: Path where Puppet stores dynamic data.
- `ssldir`: Path where SSL certifications are stored.

In `[agent]` section:

- `server` : Host name of the PuppetMaster. (Default: puppet)
- `certname` : Certificate name used by the client. (Default is its fqdn)
- `runinterval`: Number of minutes between Puppet runs, when running as service. (Default: 30)
- `report` : If to send Puppet runs' reports to the report_server. (Default: true)

In `[master]` section:

- `autosign`: If new clients certificates are automatically signed. (Default: false)
- `reports`: How to manage clients' reports (Default: store)
- `storeconfigs`: If to enable store configs to support exported resources. (Default: false)

COMMON COMMAND-LINE PARAMETERS

All configuration options can be overridden by command-line options.

A very common option used when we want to see immediately the effect of a Puppet run:

```
puppet agent --test # Can be abbreviated to -t
```

Run puppet using an environment different from the default one, without actually applying changes:

```
puppet agent --test --noop --environment testing
```

ANATOMY OF A PUPPET RUN

PART 1: CATALOG COMPILATION

Puppet client executes on the node:

- The client runs `facter` and send its facts to the serve

```
Client output # Info: Loading facts in /var/lib/puppet/lib/fact
```

- The server looks for the client's hostname (or certnam different from the hostname) and looks into its nodes The server compiles the catalog for the client using al client's facts.

```
Server's logs # Compiled catalog in environment production in 8
```

PART 2: CATALOG APPLICATION

- The client receives the catalog

```
Client output # Info: Caching catalog
```

- and starts to apply it locally - all changes to the system are shown here

```
Client output # Info: Applying configuration version '13553
```

- Client sends to the server a report of what has been changed

```
Client output # Finished catalog run in 13.78 seconds
```

REUSING CODE

5.1

ERB TEMPLATES

Files provisioned by Puppet can be Ruby ERB templates.

In a template all the Puppet variables (facts or user assigned) can be used:

```
# File managed by Puppet on <%= @fqdn %>
search <%= @domain %>
```

Also, full ruby syntax may be utilized:

```
<% @dns_servers.each do |ns| %>
nameserver <%= ns %>
<% end %>
```

RESOURCE DEFAULTS

It's possible to set default argument values for a resource in order to reduce code duplication. The syntax is:

```
Type {
  argument => value,
}
```

Common examples:

```
Exec {
  path => '/sbin:/bin:/usr/sbin:/usr/bin',
}

File {
  mode  => 0644,
  owner => 'root',
  group => 'root',
}
```

Resource defaults can be overriden when declaring a specific resource of the same type.

For example, an environment may partially override default arguments:

```
# /etc/puppet/manifests/site.pp
File {
  mode  => 0644,
  owner => 'root',
  group => 'root',
}

# /etc/puppet/environments/stage/site.pp
File {
  owner => 'stage',
  group => 'stage',
}
```

Note that the "Area of Effect" of resource defaults might bring unexpected results. The general suggestion is:

- Place global resource defaults in `/etc/puppet/manifests/site.pp` outside any node definition.
- Place local resource defaults at the beginning of a class that uses them (mostly for clarity sake, as they are parse-order independent).

NODES INHERITANCE

It is possible to have an inheritance structure for nodes, so that resources defined for a node are automatically included in an inheriting node.

```
node 'general' { ... }

node 'www01' inherits general { ... }
```

CLASS INHERITANCE

Child class can override the arguments of a resource defined in the main class besides adding a new resource. Note the syntax used when referring to the existing resource `File['/etc/puppet/puppet.conf']`:

```
class puppet {
  file { '/etc/puppet/puppet.conf':
    content => template('puppet/client/puppet.conf'),
  }
}

class puppet::server inherits puppet {
  File['/etc/puppet/puppet.conf'] {
    content => template('puppet/server/puppet.conf'),
  }
}
```

METAPARAMETERS

Metaparameters are parameters available to any resource type, they can be used for different purposes:

- Manage dependencies (`before`, `require`, `subscribe`, `notify`, `stage`)
- Manage resources' application policies (`audit`, `noop`, `schedule`, `loglevel`)
- Add information to a resource (`alias`, `tag`)

MANAGING DEPENDENCIES

Puppet language is declarative and not procedural: it defines states, the order in which resources are written in manifests does not affect the order in which they are applied to the desired state.

To manage resources ordering, there are different methods, which can coexist:

- Use the metaparameters: before, require, notify, subscribe
- Use the Chaining arrows (compared to the above metaparamers: -> , <- , <~ , ~>)

MANAGING DEPENDENCIES - BEFORE | NOTIFY

In a typical Package/Service/Configuration file example we want the package to be installed first, configure it and then start the service, eventually managing its restart if the config file changes.

This can be expressed with metaparameters:

```
package { 'exim':
  before => File['exim.conf'],
}

file { 'exim.conf':
  notify => Service['exim'],
}

service { 'exim':
}
```

Which is equivalent to chaining arrows notation:

```
Package['exim'] -> File['exim.conf'] ~> Service['exim']
```

MANAGING DEPENDENCIES - REQUIRE | SUBSCRIBE

The previous example can be expressed using the alternative reverse metaparameters:

```
package { 'exim':
}

file { 'exim.conf':
  require => Package['exim'],
}

service { 'exim':
  subscribe => File['exim.conf'],
}
```

```
Service['exim'] <~ File['exim.conf'] <- Package['exim']
```

CONDITIONALS

Puppet provides different constructs to manage conditionals inside manifests:

- Selectors allows to set the value of a variable or an argument inside a resource declaration according to the value of another variable. Selectors therefore just returns values and are not used to manage conditionally entire blocks of code.
- Control flow statements (case, if/elseif/else, unless) are used to execute different blocks of code and can't be used inside resources declarations.

EXAMPLE: ASSIGN A VARIABLE VALUE

Selector:

```
$package_name = $facts['os']['name'] ? {
   'RedHat' => 'httpd',
   'Debian' => 'apache2',
   default  => undef,
 }
```

case:

```
case $facts['os']['name'] {
   'Debian': { $package_name = 'apache2' }
   'RedHat': { $package_name = 'httpd' }
   default: { notify { "Operating system $facts['os']['name']
 }
```

if elsif else:

```
if $facts['os']['name'] == 'Debian' {
   $package_name = 'apache2'
 } elsif $facts['os']['name'] == 'RedHat' {
   $package_name = 'httpd'
 } else {
   notify { "Operating system $facts['os']['name'] not support
 }
```

COMPARISON OPERATORS

Puppet supports some common comparison operators:

- ==
- !=
- <
- >
- <=
- >=

`in` operator

```
if '64' in $facts['architecture']

if $monitor_tool in [ 'nagios' , 'icinga' , 'sensu' ]
```

EXPRESSIONS COMBINATIONS

```
if ($::osfamily == 'RedHat')
and ($::operatingsystemrelease == '5') {
   [ ... ]
}

if ($::osfamily == 'Debian') or ($::osfamily == 'RedHat') {
   [ ...]
}
```

EXPORTED RESOURCES

When we need to provide to an host informations about resources present in another host, we need exported resources: resources declared in the catalog of a node (based on its facts and variables) but applied (collected) on another node.

Resources are declared with the special `@@` notation which marks them as exported so that they are not applied to the node where they are declared:

```
@@host { $::fqdn:
  ip  => $::ipaddress,
}

@@concat::fragment { "balance-fe-${::hostname}",
  target  => '/etc/haproxy/haproxy.cfg',
  content => "server ${::hostname} ${::ipaddress} maxconn 500(
  tag     => "balance-fe",
}
```

Once a catalog containing exported resources has been applied on a node and stored by the PuppetMaster (typically on PuppetDB), the exported resources can be collected with the spaceshift syntax (where is possible to specify search queries):

```
Host << || >>
Concat::Fragment <<| tag == "balance-fe" |>>
```

INTRODUCTION TO HIERA

Hiera is the key/value lookup tool of reference where to store Puppet user data.

Hiera is installed by default with Puppet version 3

6.1

It provides an highly customizable way to lookup for parameters values based on a custom hierarchy using many different backends for data storage.

Command line tool `hiera` can be used to query the Hiera directly

And functions can be used inside Puppet manifests:

- hiera()
- hiera_array()
- hiera_hash()
- hiera_include()

NODES CLASSIFICATION WITH HIERA

Hiera provides a hiera_include function that allows the inclusion of classes as defined on Hiera. This is an approach that can be useful when there's massive usage of Hiera as backend for Puppet data.

In `/etc/puppet/manifests/site.pp` just place:

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

and place, as an array, the classes to include in Hiera data source under the key `classes`.

CONFIGURATION

Hiera's configuration file is in yaml format and is called `hiera.yaml`

Here we define the hierarchy we want to use and the backends where data is placed, with backend specific settings.

```
/etc/puppet/hiera.yaml
```

DEFAULT CONFIGURATION

```
:backends: yaml
:yaml:
  :datadir: /var/lib/hiera
:hierarchy: common
:logger: console
```

HIERARCHIES

With the `:hierarchy` global setting we can define a string or an array of data sources which are checked in order, from top to bottom.

In hierarchies we can interpolate variables with the %{} notation (variables interpolation is possible also in other parts of hiera.yaml and in the same data sources):

```
:hierarchy:
  - "nodes/%{::clientcert}"
  - "roles/%{::role}"
  - "%{::osfamily}"
  - "%{::environment}"
  - common
```

USING HIERA IN PUPPET

The data stored in Hiera can be retrieved by the PuppetMaster while compiling the catalog using the `hiera*()` functions.

For example:

```
$dns_servers = hiera("dns_servers")

$openssh_settings = hiera_hash("openssh_settings")
```

DATA BINDINGS

Automatic Hiera lookup is done for each class' parameter using the key `$class::$argument` this functionality is called data bindings or automatic parameter lookup.

For example:

```
class openssh (
    template = undef,
  ) { . . . }
```

Puppet automatically looks for the Hiera key `openssh::template` if no value is explitely set when declaring the class.