Puppet
Gestion de Configuration

Aurélien Degrémont
aurelien.degremont@cea.fr

La gestion de configuration

La gestion de configuration ?

- *Automatiser* les processus d’installation
  - Installer des logiciels
  - Copier des fichiers
  - Configurer des services
  - Et bien plus
- Ne pas confondre avec la gestion de parc
  - Ce n’est pas la gestion du matériel et des configurations de ses serveurs et stations

Automatisation

- Gagner du temps
- Passer à l’échelle
- Améliorer la *reproductibilité*
- Formaliser
  - Ne plus utiliser des scripts
  - Partager un langage commun
- Auditer

Différents logiciels

- CFEngine
  - Un des pionniers
- Ansible
  - Approche simple, le grand concurrent
- Puppet
  - Le révolutionnaire, très déployé
- Chef, Salt, …

Les scripts

- Les scripts sont largement utilisés pour automatiser des tâches, mais avec beaucoup de défauts :
  - Chacun sa technique
  - Réinventer la roue
  - Difficulté à faire des tâches complexes (parallélisme, …)
  - Langage impératif

DSL – Domain Specific Language

- Structurer les tâches d’installation dans un langage dédié
- Tous les utilisateurs parlent la même langue
- Mutualisation des efforts
- Spécialisé pour les besoins

Approche déclarative

- Description de l’état souhaité du système
- Le logiciel détermine seul les actions pour y parvenir
- Il détecte l’écart entre l’état courant et l’état souhaité
- Il n'effectue que les changements nécessaires
- L'exécution est donc *idempotente*

Puppet

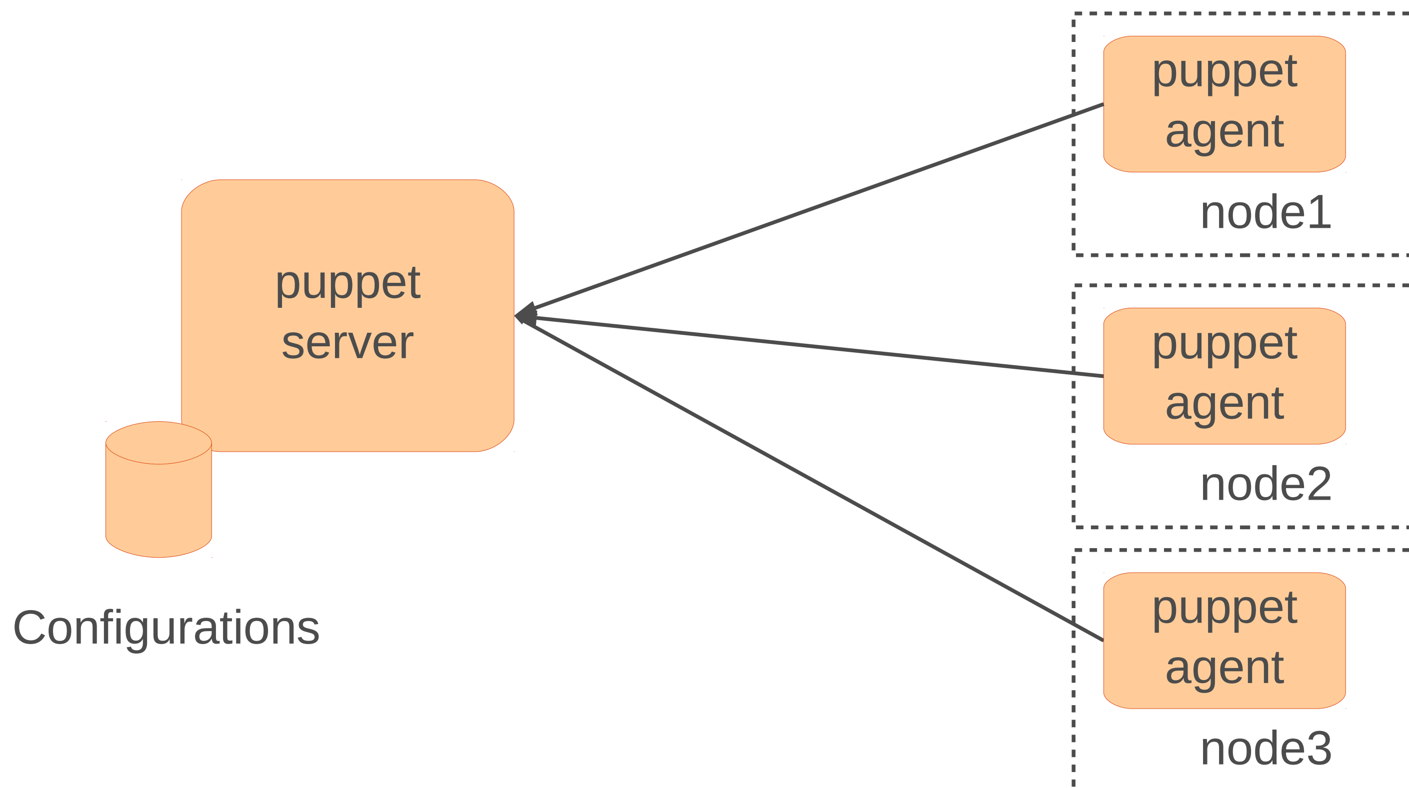
Présentation

- Logiciel open-source créé en 2005
- Version 5.3 (au 3 février 2018)
- A l’origine en Ruby, avec du Clojure et C++ actuellement
- Architecture client – serveur
- Langage de configuration déclaratif
  - Il décrit ce que l'on souhaite, pas *comment* y parvenir

Architecture Puppet

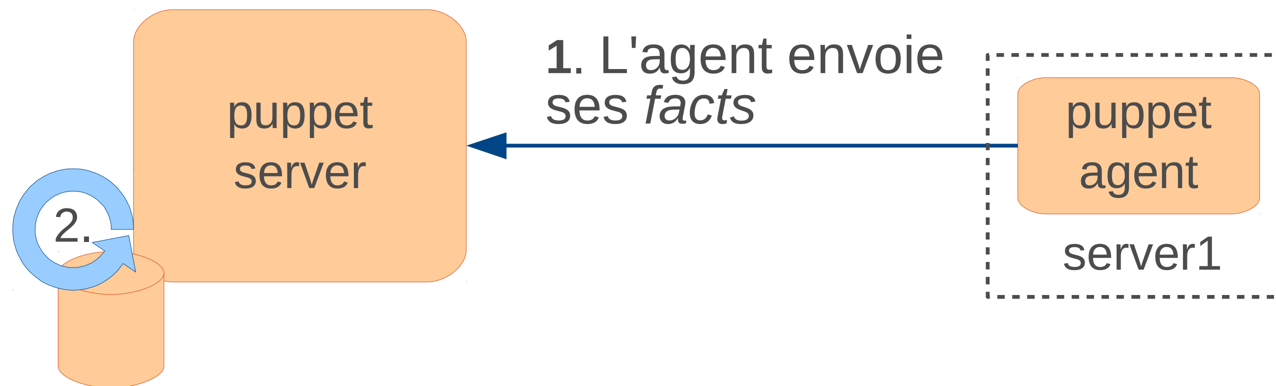
Architecture

- Modèle client – serveur
  - Un processus agent sur chaque serveur
  - Un serveur central qui connaît toutes les configurations à appliquer



Requêtes

- Séquence
  1. Le client envoie sa description (facts)
     - facts : os, architecture, nom, mémoire, etc...
  2. Le serveur lit sa configuration (modules et manifests) et les *facts* reçus.



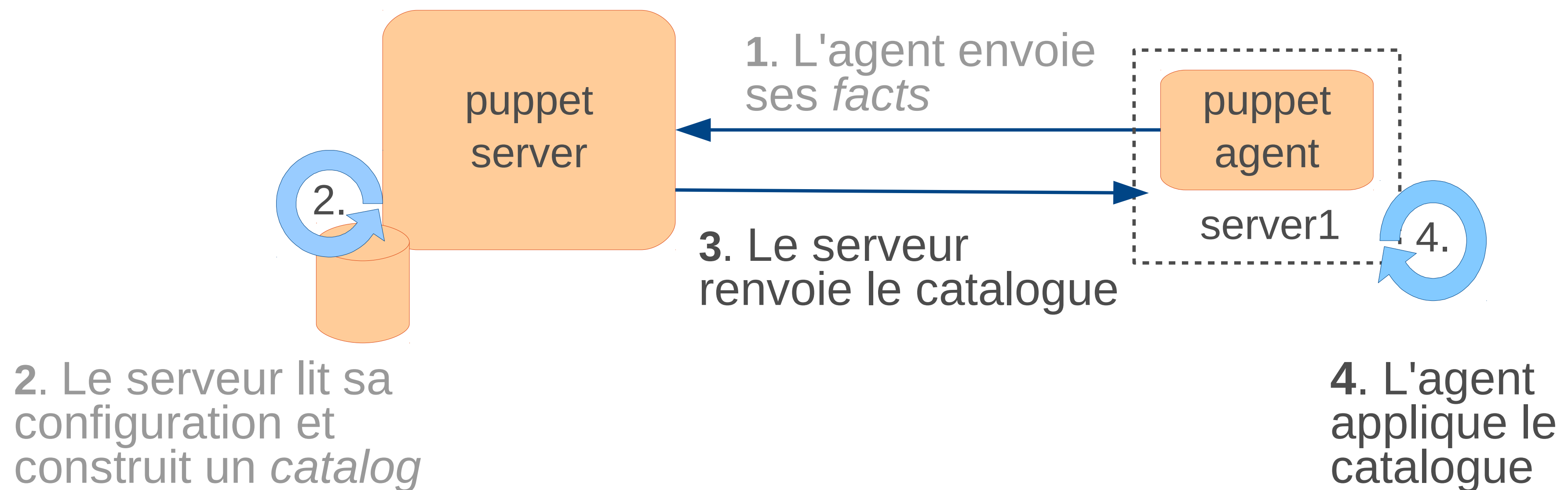
2. Le serveur lit sa configuration et construit un *catalog*

Requêtes

- Séquence

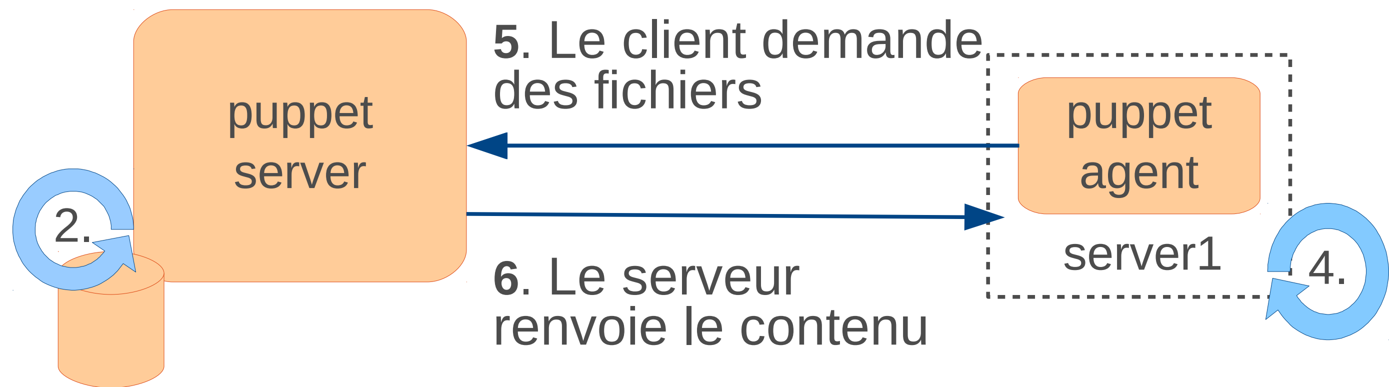
  3. Le serveur construit une configuration spécifique pour ce client (catalog)

  4. Le client récupère le catalogue et l'applique



Requêtes

- Séquence
  5. Le client demande des fichiers si nécessaire
  6. Le serveur renvoie le contenu demandé

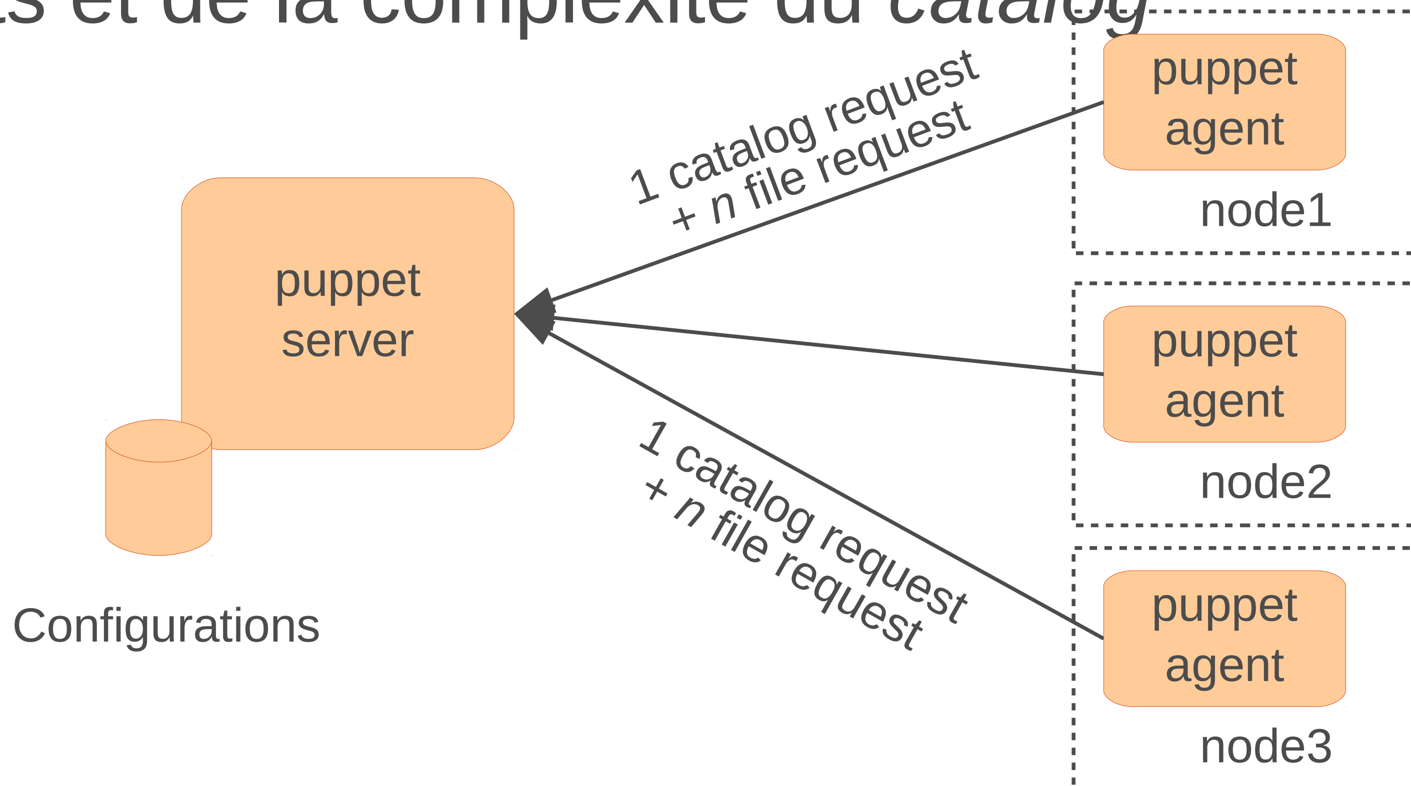


2. Le serveur lit sa configuration et construit un *catalog*

4. L'agent applique le catalogue

Passage à l'échelle

- Chaque client envoie :
  - 1 requête pour son catalogue
  - 1 requête / fichiers de configuration
- La charge du serveur dépend du nombre de clients et de la complexité du *catalog*



Chiffrement des communications

- Les échanges entre les agent et le master sont chiffrés
- Le serveur intègre une *autorité de certification* (CA)
- L'agent génère une requête de certificat (CSR) et la fait signer par le serveur maître
- C'est un préalable à tout autre type de communication
- Lister et signer les certificats :
  - `server# puppet cert list`
  - `server# puppet cert sign <HOSTNAME>`

Langage Puppet

Le langage Puppet

- Puppet utilise son propre langage permettant de décrire les ressources à configurer
- Il existe plusieurs dizaines de ressources, comme les `packages`, les `files` et les `services`
- Ces ressources ont des *noms* et des *propriétés*
- Documentation détaillée :
  - *https://puppet.com/docs/puppet/5.3/type.html*

Le langage Puppet

- Exemple de déclaration d'un package
  - Ici, on indique que le *package* `rsyslog` doit être installé
  - Et toujours à la version la plus récente (*latest*)

Type de la ressource

Nom de la ressource

```
package {
    "rsyslog":
        ensure => "latest";
}
```

Propriété

Ressources

```
package {
    "rsyslog":
        ensure => "latest";
    "chrony":
        ensure => "absent";
}
```

- On indique l'état que l'on veut atteindre, pas les commandes pour y arriver.
- Puppet analyse le système et décide d'installer le package s'il n'est pas présent, ou le mettre à jour si ce n'est pas le cas.

Ressources

```
package {
    "rsyslog":
        ensure => "latest";
}
```

- Vérifie s'il est présent et sa version
  - `rpm -q rsyslog`
- Installe si nécessaire
  - `yum -y` **`install`** `rsyslog`
- Ou met à jour
  - `yum -y` **`update`** `rsyslog`

Ressources

- Il existe d'autres ressources comme `service`

```
package {
    "rsyslog":
        ensure => "latest";
}

service {
    "rsyslog":
        enable  => true,
        ensure  => "running",
        require => Package["rsyslog"];
}
```

Différentes propriétés

Dépendance (ordre)

Attention à la majuscule

Ressources

- Fichiers avec droits et permissions
  - Le contenu peut être un fichier sur le serveur
    - `source => "puppet:///modules/rsyslog/rsyslog.conf"`
    - correspond à
      - /etc/puppetlabs/code/environment/production/modules/rsyslog/**files**/rsyslog.conf
  - Le contenu peut correspondre à un template
    - `content => template("rsyslog/rsyslog.conf.erb")`
  - Ou alors directement dans une chaîne de caractères :

```
file {
   "/etc/rsyslog.conf":
      owner   => "root",
      group   => "root",
      content => "Some content\n...";
}
```

Classes

- Les ressources sont groupées dans des classes.
- Chaque classe n'existe qu'en un seul exemplaire
- Il existe d'autres types ressources (yumrepo, augeas, ...)

```
class rsyslog {

  package {
      ...
  }

  file {
      ...
  }

  service {
      ...
  }

}
```

Classes paramétriques

- Les classes peuvent avoir des paramètres
  - avec un type (optionnel)
  - et des valeurs par défaut
    - La valeur "undef" peut être utilisé pour rendre un paramètre optionnel.
- Ces paramètres sont utilisés comme variables :

```
class rsyslog(
  String $service_ensure = "latest",
) {
  package {
      "rsyslog":
        ensure => $service_ensure;
  }
}
```

Define

- Pour déclarer plusieurs ressources similaires, il faut utiliser `define`
- C'est un nouveau type, avec un nom : $name

```
define config(
  $option1,
) {
  file {
     "/etc/${name}":
       owner  => "root",
       group  => "root",
       source => ...;
  }
}
```

Define

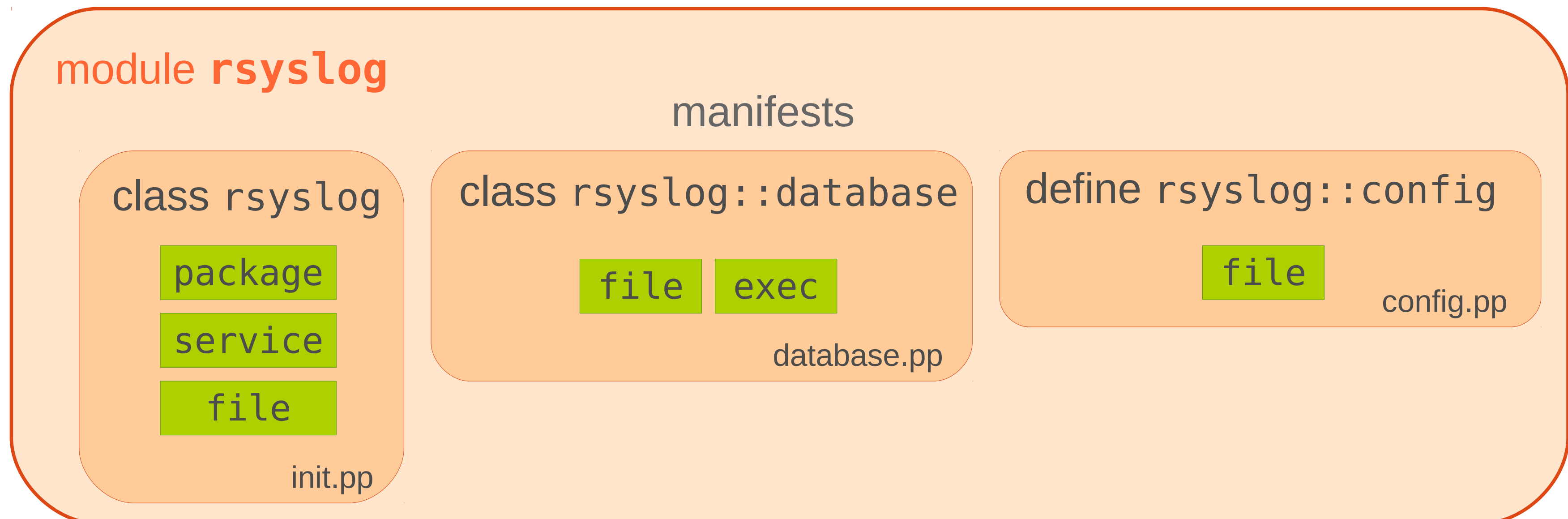
- Un `define` se déclare comme une ressource de même nom.
- On peut en déclarer autant que souhaitée

```
config {
      "foo":
         option1 => "value";
      "bar":
         option1 => "value";
}
```

Modularité Puppet

Modules

- Un module est composé de classes et de defines qui sont eux-mêmes composés de ressources.
- Ces objets composent les **manifests du module**



Modules

- Les classes sont regroupées dans des *modules*
  - `/etc/puppetlabs/code/environments/production/modules`
- La configuration Puppet comprend une liste de modules, chacun ayant son propre répertoire
  - `.../modules/rsyslog/`
  - `.../modules/ntp/`
  - etc.
- Chaque répertoire de module contient une liste de sous-répertoires :
  - `modules/rsyslog/manifests/*.pp` Définitions puppet
  - `modules/rsyslog/files/` Fichiers de config
  - `modules/rsyslog/templates/` Template Puppet
  - etc..

Modules

- C'est dans les manifests qu'on place les définitions de *classes* et de *define*.
- Chacun dans son propre fichier `.pp`
- Chaque nom est de la forme :
  - nom_du_module
  - nom_du_module::objet
  - nom_du_module::sous_element::sous_element
- Les conventions de nommage sont les suivantes :
  - classe `rsyslog` → ./`init.pp`
  - classe `rsyslog::variante` → ./`variante.pp`
  - define `rsyslog::sub::config` → `./sub/config.pp`

Manifests

- La configuration des agents se base sur les fichiers *manifests*
  - *différents de ceux des modules*
- Ces fichiers sont dans le répertoire `manifests/` au même niveau que le répertoire `modules/`

Manifests

- Ces fichiers sont dans le répertoire `manifests`
  - `/etc/puppetlabs/code/environments/production/manifests`
- Exemple de contenu :
    - `client.pp`
    - `server.pp`
  - Ici, chaque fichier correspond à un type de machine
- On peut créer autant de fichiers que nécessaire
- *puppetserver* concatène tous ces fichiers et les traite comme un unique fichier manifest.

Manifests

- Un manifest peut contenir de simple directive :

```
package { "emacs": ensure => "latest"; }
```

- Ou inclure des modules et leurs classes :
  - Inclure une simple classe, sans paramètre

    ```
    include ldap
    ```

  - Inclure une classe avec ou sans paramètres

    ```
    class { "ldap": }
    class { "ldap": param => "foo"; }
    ```

Manifests

- Filtrer les définitions par nœud :

```
node "vm1.pcocc" {
        include ldap
}
```

- Avec une expression régulière :

```
node /^vm2\./ {
        include ldap
        include bind
}
```

Appliquer la configuration

- Localement
  - `puppet apply /mon/fichier/config.pp`
- En mode client-serveur
  - `puppet agent -t`
- En mode test avec l'option --noop
  - `puppet agent -t --noop`

Astuces

- Vérifier la syntaxe d'un fichier Puppet `.pp`
  - `puppet parser validate monfichier.pp`
- La ressource `notify` permet d'afficher des messages

```
notify { "Je suis le noeud ${hostname}": }
```

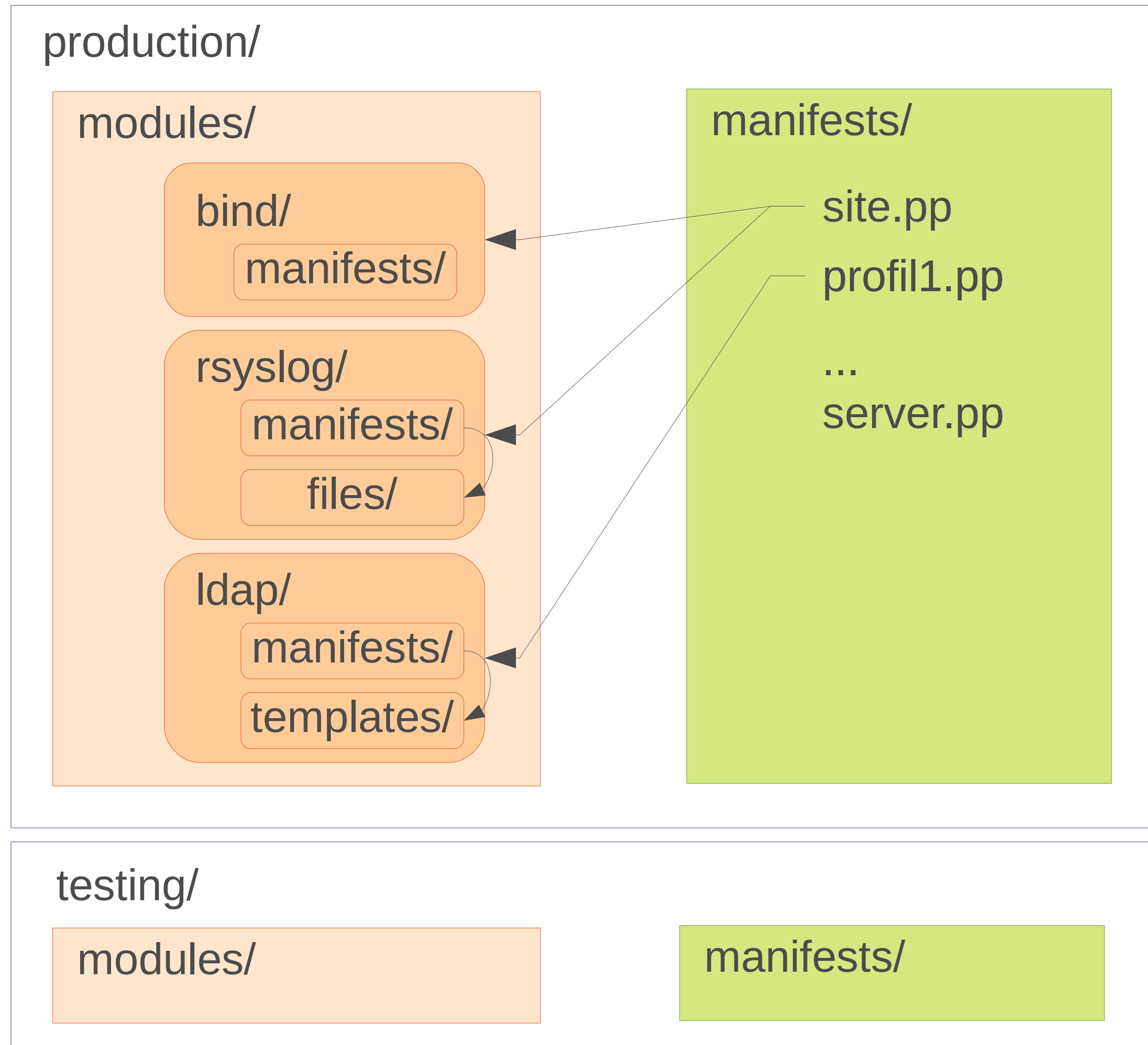
- Antisèche pour les types de bases :
  - *https://puppet.com/docs/puppet/5.3/cheatsheet_core_types.html*
- Doc complète
  - *https://puppet.com/docs/puppet/5.3/*

Environnements

- L'ensemble des configurations Puppet (modules, manifests, hiera, etc.) sont regroupés dans un environnement
  - `/etc/puppetlabs/code/environments/`
  - L'environnement par défaut s'appelle : **production**
- Il est possible de choisir l'environnement utilisé pour chaque machine
- Permet d'implémenter des niveaux de production par exemple :
  - production, testing, development
- Ou de servir des configs différentes ou incompatibles

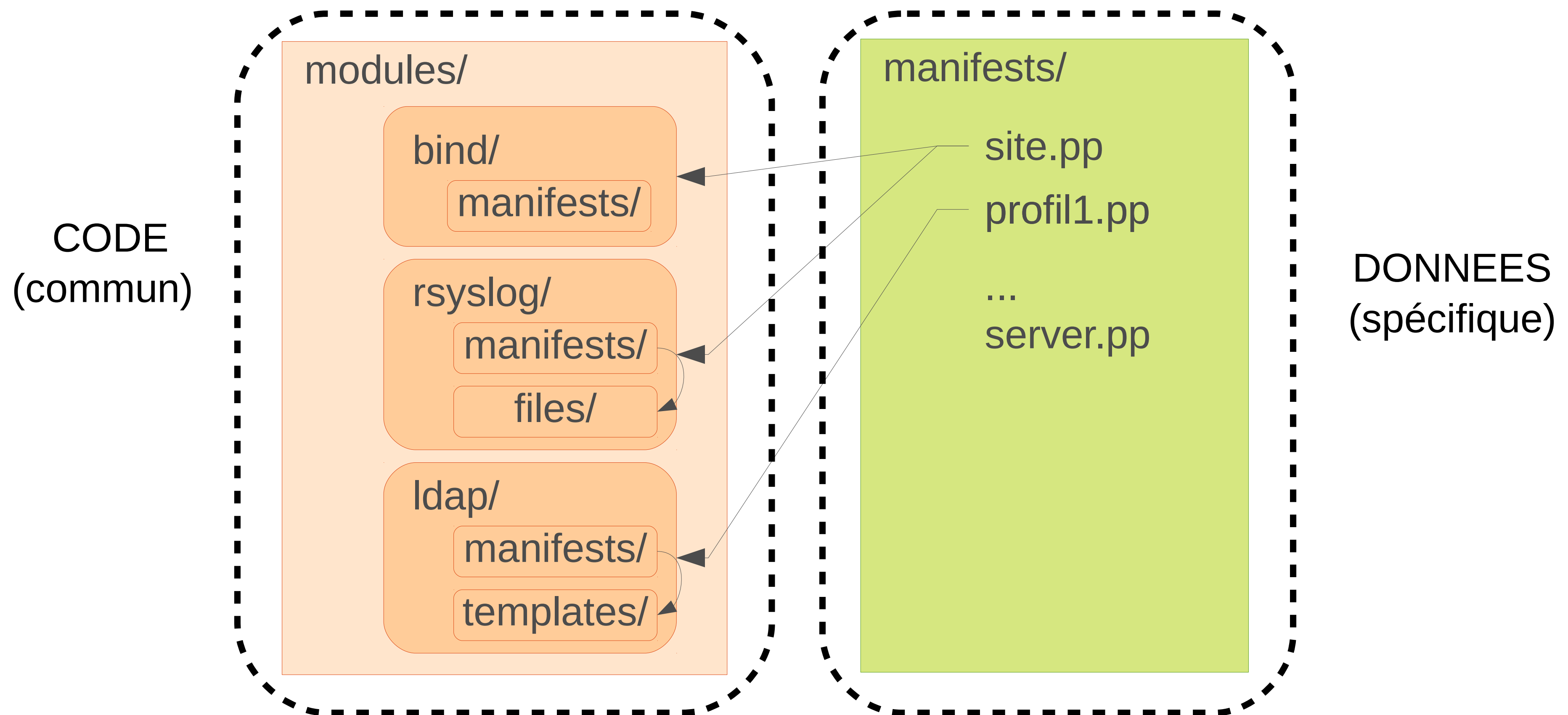
Environments

/etc/puppetlabs/code/environments



Hiera

- Principe de séparation du code et des données
  - Modules puppet : code
  - Paramètres des classes : données



Hiera

- Les modules sont communs à toutes les installations
- Les manifests décrivent la configuration spécifique
- Lorsque la configuration devient trop complexe, les fichiers dans les manifests peuvent être trop limités
- Hiera est un système de paires clés/valeurs hiérarchique
  - Stocker les données dans un système séparé
  - Avec une organisation hiérarchique
- La hiérarchie est configurable, exemple :
  - Serveur
  - Cluster
  - Commun

Hiera

- Puppet lit les valeurs des variables dans Hiera
- Hiera les cherche dans l'ordre de sa hiérarchie, dans des fichiers .yaml ou .json
- Les noms de fichiers peuvent contenir des variables. Exemple :

```
hierarchy:
  - name: "Per-node data"
    path: "nodes/%{trusted.certname}.yaml"

  - name: "Per-OS defaults"
    path: "os/%{facts.os.family}.yaml"

  - name: "Common data"
    path: "common.yaml"
```

Hiera

- Puppet résout les paramètres des classes dans l'ordre suivant :
  - Valeur dans le manifest

    ```
    class { 'foo': param => 3; }
    ```

  - Valeur définie dans Hiera

    ```
    include foo  # paramètres dans hiera
    ```

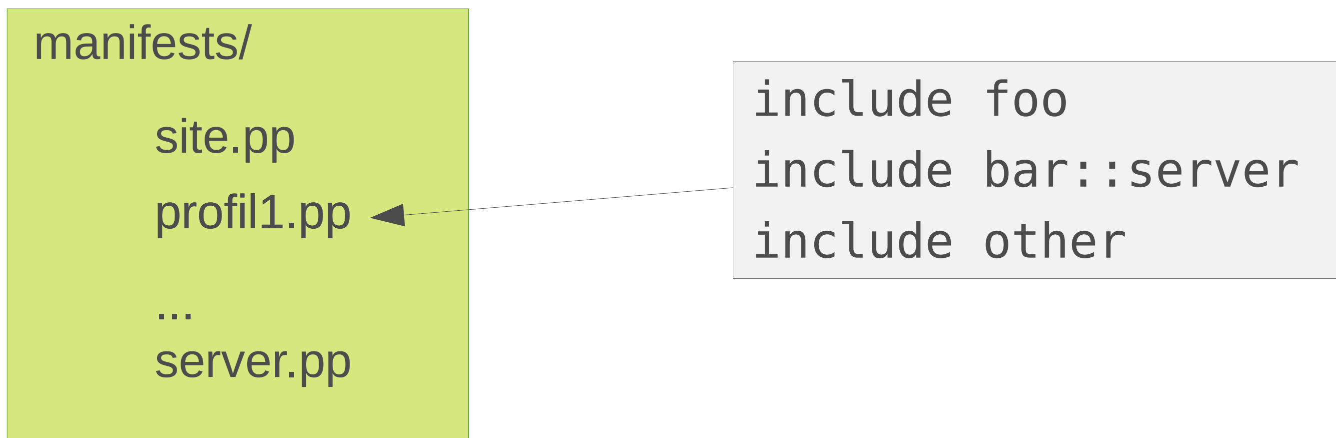
  - Valeur par défaut dans la classe

    ```
    class foo($param = 1) { … }
    ```

  - Sinon, c'est une erreur (valeur non définie)

Hiera

- L'objectif de Hiera est de simplifier les manifests
  - Ils ne contiennent plus que les classes à utiliser
- Les paramètres de ces classes sont gérés par Hiera qui est plus souple



External Node Classifier

- La configuration à appliquer sur un nœud dépend de :
  - son environnement
  - son profil
- Le profil est basé sur une ou plusieurs classes avec éventuellement des variables
- En plus des manifests et de Hiera, Puppet offre un mécanisme supplémentaire pour cette configuration :
  - External Node Classifier – ENC

External Node Classifier

- L'ENC est une commande externe qui, connaissant le nom du nœud, renvoie à Puppet :
  - L'environnement
  - Une liste de classes
  - Une liste de variables (et même plus…)
- Cela permet de stocker de façon externe cette correspondance, avec sa propre logique
- Permet de coupler Puppet à un autre outil

External Node Classifier

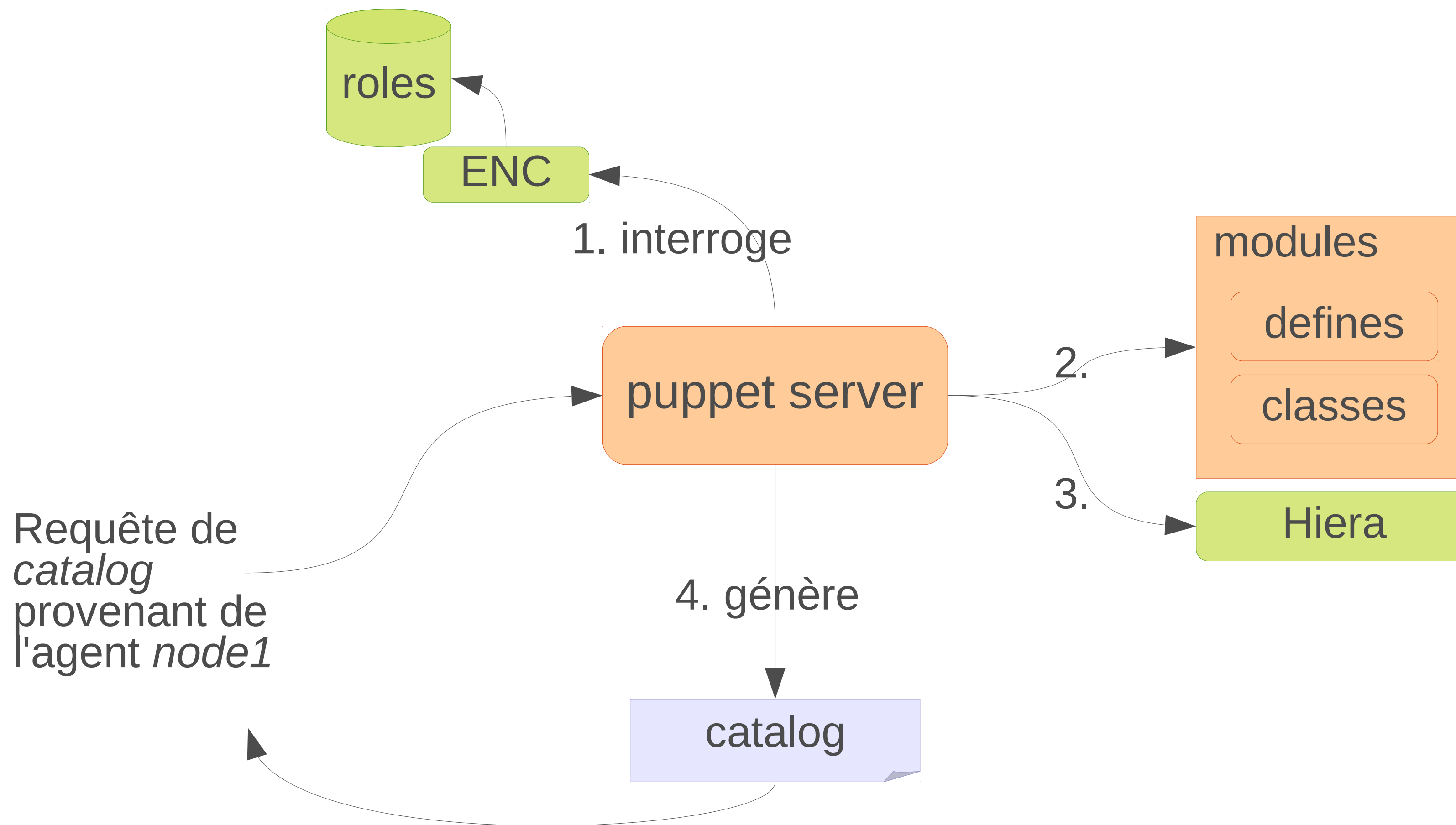
- Les clusters utilisent des noms à base de range
- Un ENC adapté pourrait déclarer simplement le *mapping* entre un nœud et son environnement
  - `production: compute[0-100]`
  - `testing:    compute[101-110]`
- Ou alors des rôles :
  - `admin:     cluster[1-5]`
  - `login:     cluster[10-12]`
  - `compute:   cluster[50-200]`

External Node Classifier

- L'ENC est une commande externe qui pour un nœud donnée peut renvoyer plusieurs informations, comme :
  - les classes à inclure
  - l'environnement
  - des variables

```yaml
---
classes:
    - common
    - puppet
    - ntp
    - yumconfig
parameters:
    ntp_servers:
        - 0.pool.ntp.org
        - ntp.example.com
    mail_server: mail.example.com
environment: production
```

Organisation des configurations



Résumé

- Puppet permet de gérer l'état des ressources du système
- Mode client-serveur
  - La configuration est centralisée sur un serveur
  - Elle est appliqué par un agent
- L'organisation des configurations est modulaire

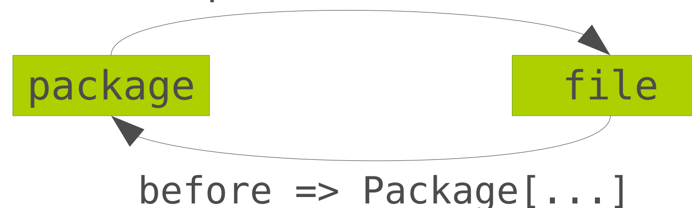
Dépendances entre ressources

- Par défaut, il n'y aucune notion d'ordre entre les ressources
- Si, par exemple, l'on souhaite modifier un fichier de configuration installé par un package, il faut être sur que le package est bien installé auparavant.
- Pour cela, toutes les ressources peuvent utiliser des propriétés déclarant une dépendance :
  - Nécessite que le package soit déjà installé

```
require => Package["foo"]
```

  - Ou son pendant. La ressource doit être traitée avant le fichier foo

```
before => File["/etc/foo"]
```



Facter

- Analyse le système et génère une liste de variables le décrivant
- Ces variables peuvent être utilisée dans les *manifests* ou les configurations Hiera

```
if $facts['os']['family'] == 'RedHat' {
    ...
}
```

```
$ facter
...
os => {
  architecture => "x86_64",
  distro => {
    id => "CentOS",
    release => {
      full => "7.4.1708",
      major => "7",
      minor => "4"
    },
  },
  family => "RedHat",
  hardware => "x86_64",
  name => "CentOS",
  release => {
    full => "7.4.1708",
    major => "7",
    minor => "4"
  },
}
...
```

Conditions

- Le code puppet peut s'adapter en fonction de conditions : if, else, case, etc...

```
if $facts['is_virtual'] {
  # Our NTP module is not supported on virtual machines:
  warning('Tried to include class ntp on virtual machine')
}
elsif $facts['os']['name'] == 'Darwin' {
  warning('This NTP module does not yet work on our Mac laptops.')
}
else {
  # Normal node, include the class.
  include ntp
}
```

Backup

- File properties, file content, templates