

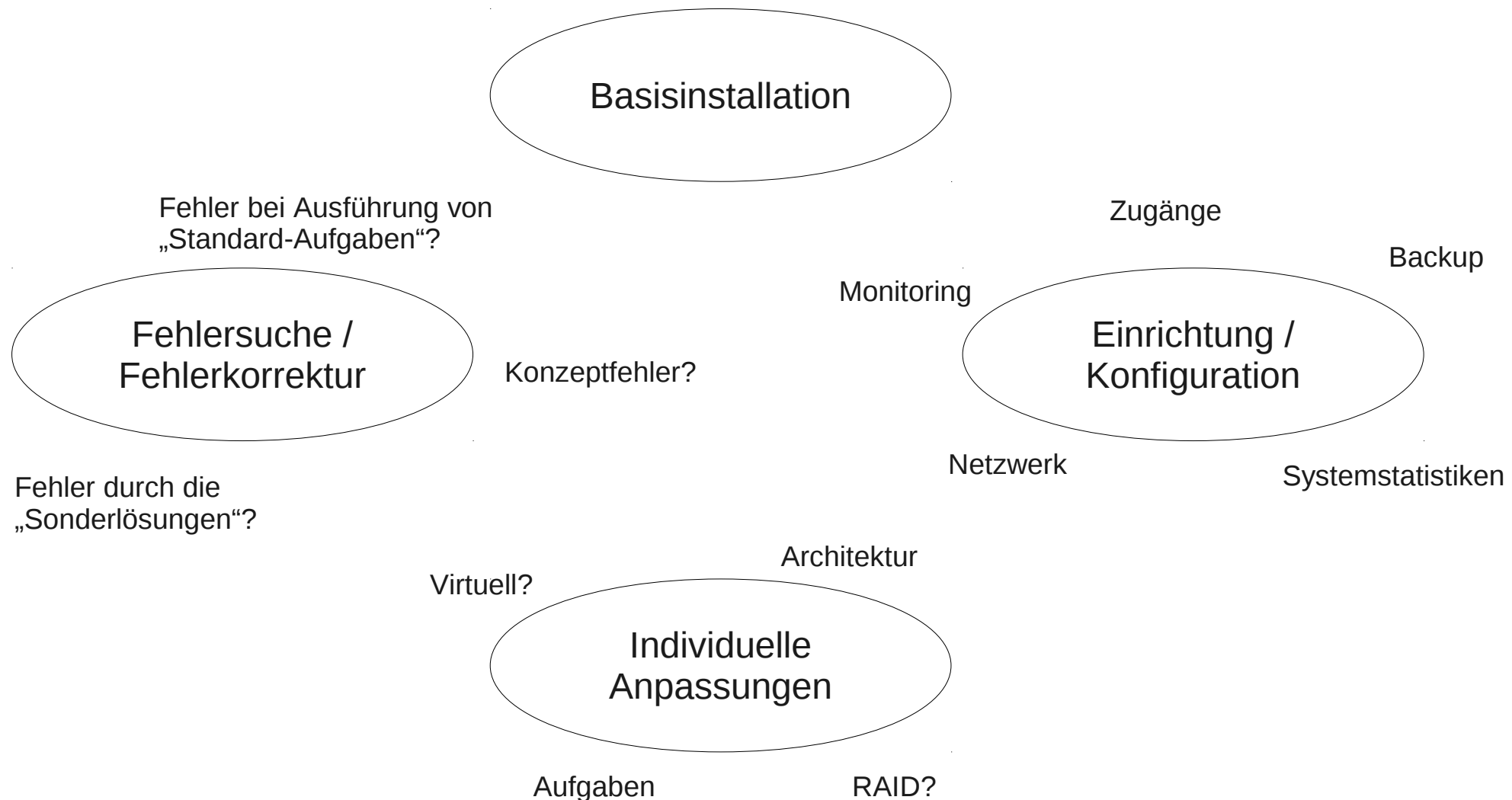
System-Management-Trio

Zentrale Verwaltung mit factor, puppet und augeas

Aufbau / Ziele:

- Einführung / Überblick
- Begriffsklärung
- Aufbau / Arbeitsweise von Manifests
- Templates, Klassen, Typen, Module
- Facter: Umgebungsparameter ermitteln
- Augeas: Konfigurationen bearbeiten
- Client-Server-Betrieb
- Arbeit mit mehreren Umgebungen
- Links / Hilfen

- Stefan Neufeind
- Mit-Geschäftsführer der SpeedPartner GmbH aus Neuss ein Internet-Service-Provider (ISP)
 - Individuelle TYPO3-Entwicklungen
 - Hosting, Housing, Managed Services
 - Domains / Domain-Services
 - IPv6, DNSSEC, ...
- Aktive Mitarbeit im Community-Umfeld (PHP/PEAR, TYPO3, Linux)
- Freier Autor für z.B. t3n, iX, Internet World, ...

Alltägliche Administrationsaufgaben für ein (Server-)System

Herausforderungen bei Administration mehrerer Systeme

- Einsparungen (Zeit und Kosten)
- Minimierung von Fehlern
- Einheitliche Konfiguration
- Einfache Anpassbarkeit
 - Zentral?
 - Automatisiert?

Umsetzung erfordert:

- Klare Regeln / Entscheidungsgrundlagen
- Abgleich Vorgaben und Realität
- Strukturierte Herangehensweise
- Unterstützung durch geeignete Hilfsmittel

Das „System-Management-Trio“ bestehend aus:

- Puppet
 - Zentrale Komponente
 - Definiert Regeln, Abhängigkeiten, Aktionen, ...
 - Vorgaben basierend auf Fakten
 - Abgleich Planung / Vorgaben und Realität
- Facter
 - Liefert „Fakten“ für Puppet
 - Ermittelt Umgebungsparameter
 - Bereitstellung in einheitlicher Form
- Augeas
 - Anpassung von Konfigurationen
 - Abstraktion für Zugriff auf Konfigurationseinstellungen

Zentrale Begriffe im puppet-Umfeld:

- Manifests:
 - „Verzeichnis“ von Objekten, Eigenschaften und Beziehungen
 - Dateierdung: .pp
- Ressourcen:
 - Definition über den „Resource abstraction layer“ (RAL)
 - Beziehen sich auf Entsprechungen im System
 - Unabhängig z.B. vom verwendeten Paketmanager
 - Zentrale Ressourcentypen: Dateien, Pakete, Dienste, Benutzer/Gruppen
 - Bestehend aus einem „Titel“ und „Attributen“
 - Jede Ressource eindeutig
 - Modellierung von Abhängigkeiten, Reihenfolge der Definition beliebig
 - Puppet auto-generiert sinnvolle Reihenfolgen
(z.B. erst Verzeichnisse anlegen, dann darin enthaltene Dateien)
- Knoten („nodes“):
 - Sammlung von Ressourcen

Zentrale Begriffe im puppet-Umfeld:

- Templates:
 - Vorlagen für Ressourcentyp Datei („file“)
 - Ausgabe auf Basis von Variablen (Fakten), Kontrollstrukturen und umgebener Notation
 - Notation mit Hilfe von ERB (eine Implementierung von „embedded ruby“, auch eRuby genannt)

- Module:
 - Sammlung von Code und Daten
 - Einheitliche Verzeichnis-/Dateistruktur
 - Wiederverwendbar
 - Fertige Module für verschiedene Einsatzzwecke verfügbar (siehe z.B. auf „Puppet Forge“)

Notation in „Manifests“:

- Puppet-eigene Syntax

Ressourcen-Typ gefolgt von
Block in geschweifte Klammern

Block mit eindeutigem Titel
(je Ressourcen-Typ)
gefolgt von Doppelpunkt

```
file { '/etc/my.cnf':  
  ensure => file,  
  mode   => 600,  
  source => '/home/demo/my.cnf',  
}
```

Definierte Werte /
Zahlen ohne
Klammerung

Attribute mit
Komma getrennt

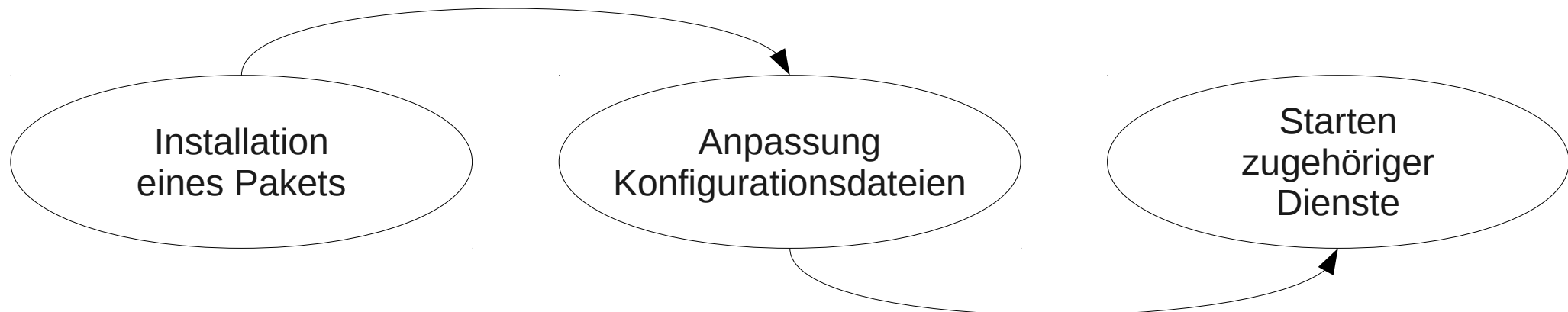
Notation mit key => value

- Zeichenketten in einfachen Hochkommata ('...') oder Anführungszeichen ("...")
 - Hochkommata: keine weitere Verarbeitung der Inhalte
 - Anführungszeichen: Ersetzung von z.B. Zeilenumbrüche ("\n"), Variablen ("\${fileame}") oder Platzhaltern (wie z.B. \$0, \$1, ... bei der Verwendung von regulären Ausdrücken)
- Listen von Werten (Arrays): in eckigen Klammern, Werte mittels Kommas getrennt

Arbeitsweise von „Manifests“:

- Definitionen von Eigenschaften, keine sequentielle „Abarbeitung“ / Reihenfolge
- Keine Löschung / Überschreibung von einmal definierten Ressourcen oder zugewiesene Variablen
- Compiler löst vor Anwendung der Definitionen etwaige Bedingungen auf
- Notwendige Reihenfolge (Abhängigkeiten) von Ressourcen bei Definition von Ressourcen angeben

Beispiel:



Abhängigkeiten zwischen Ressourcen:

- Definitionen über Meta-Parameter innerhalb Ressourcen-Definitionen
 - 'before': diese Ressource vor der angegebenen berücksichtigen
 - 'require': diese Ressource nach der angegebenen berücksichtigen
 - 'notify': ein/mehrere andere Ressourcen benachrichtigen (sofern Änderungen durchgeführt wurden)
 - 'subscribe': auf Änderungen einer anderen Ressource reagieren
- Explizite Definition (separate Zeile in Manifest)
 - Angabe einer Reihenfolge (Pfeil mit Bindestrich)
 - Angabe einer Benachrichtigung (Pfeil mit Tilde)

```
package { 'mysql-server': } -> file { '/etc/my.cnf': }
```

Das erste Manifest:

- Notation in einer beliebigen Datei (hier: demo1.pp)

```
package { 'mysql-server':
  ensure => present,
}
file { '/etc/my.cnf':
  ensure  => file,
  mode    => 600,
  source  => '/home/demo/my.cnf',
}
service { 'mysqld':
  ensure      => running,
  enable      => true,
  hasrestart  => true,
  hasstatus   => true,
  subscribe   => File['/etc/my.cnf'],
  require     => Package['mysql-server'],
}
```

Dienst soll (aktuell) gestartet sein
Dienst soll automatisch starten
Startskripte unterstützen „restart“ / „status“
Bei Änderungen an Datei restart
Paket muss installiert sein

- Anwenden des Manifest auf das System

```
puppet apply demo1.pp
```

Fakten und Entscheidungen:

- Testweise händischer Aufruf von „Factor“ für Übersicht Eigenschaften (Auszug)

```
# factor
architecture => x86_64
augeasversion => 0.9.0
domain => example.com
facterversion => 1.6.5
fqdn => demohost.example.com
hostname => demohost
is_virtual => true
virtual => kvm
kernel => Linux
kernelmajversion => 2.6
kernelrelease => 2.6.32-220.4.2.el6.x86_64
kernelversion => 2.6.32
manufacturer => Red Hat
operatingsystem => CentOS
operatingsystemrelease => 6.2
osfamily => RedHat
physicalprocessorcount => 4
processorcount => 4
uptime_hours => 136
uptime_seconds => 490098
```

Fakten und Entscheidungen:

- Factor ermittelt Fakten über das System
- Factor-Module stellen Daten in Variablen zur Verfügung
- Fakten können für Entscheidungen innerhalb des Manifest genutzt werden
 - Standardoperationen (==, !=, <, >, <=, >=)
 - Reguläre Ausdrücke (=~ und !~)
 - „in“-Operator (Prüfung gegen eine Liste von Werten)
- Mehrfach-Entscheidung per „case“
- Selektoren (ähnlich ternären Operatoren aus anderen Sprachen)

```
case $operatingsystem {
  centos, redhat, fedora:
    { $nameserver = "named" }
  debian:   { $nameserver = "bind9" }
  default:  { fail("Unknown OS") }
}
```

```
$nameserver = $operatingsystem ? {
  centos    => 'named',
  redhat    => 'named',
  fedora    => 'named',
  debian    => 'bind9',
  default   => undef,
}
```

Pfade:

- Bis hier nur Manifest-Dateien im aktuellen Pfad bzw. mit absoluter Adressierung verwendet
- Empfehlung: Angaben relativ zum puppet Modul-Pfad
 - System-unabhängig
 - Transparent für spätere Verwendung von puppet im Client-/Server-Modus

- Ermittlung Pfad für Manifests:

```
# puppet apply --configprint manifestdir
```

Alternativ: z.B. gesamte Konfiguration über Schlüsselwort „all“ ermitteln

- Umstellung Adressierung auf relative Angaben, z.B. für Template-Dateien

```
source => 'puppet:///files/etc/my.cnf'
```

Klassen:

- Zusammenfassung wiederverwendbarer Definitionen
- Definierte Klassen dann noch als Ressourcen einbinden
 - Einbindung per „class“ oder „include“
 - Übergabe von Parametern bei Verwendung von „class“-Notation möglich

```
# Klasse definieren
class db {
    # Definitionen innerhalb Klasse wie gewohnt
}
# Klasse einbinden
class { 'db': }
# Alternativ per include
include db
```

- Autoloader: Klassen (und Module) werden automatisch geladen
 - Konvention: Verzeichnis „classes“, Dateiname „klassenname.pp“

Klassen und Variablen:

- Variablen innerhalb Klasse bilden separaten Geltungsbereich („Scope“)
- Falls Variable nicht im aktuellen „Scope“ auffindbar Rückgriff auf übergeordnete Ebene
- Klassenvariablen von außerhalb per voll qualifiziertem Namen erreichbar, Notation mit zwei Doppelpunkten, z.B. `$db::variable`
- Aus Klasse expliziter Zugriff auf globale Variablen möglich, die z.B. per `facter` bereitgestellt werden: `$::operatingsystem`

Parametrisierte Klassen:

- Erlaubt Übergabe von Werten

```
# Definition
class db ($dbname, $servername = 'localhost') {
    ...
}

# Einbindung als Ressource
class {'db':
    $dbname => 'demodb',
}
```

Definition von Typen:

- Problem: Je Ressource nur eine eindeutige Definition möglich.
Wie einfache „Mehrfach-Nutzung“?
- Lösung: Eindeutige Definitionen verwenden.
Arbeitserleichterung durch Verwendung von Typen.
- Ansatz: Ressourcen benötigen pro Ressourcen-Typ eindeutigen Titel

```
define sshkeys ($user = $title, $content) {  
    file {"${user}/keys":  
        path      => "/home/${user}/.ssh/authorized_keys",  
        ensure    => file,  
        content   => $content,  
        mode      => 0644,  
        owner     => $user,  
        require   => User[$user],  
    }  
}  
  
# Verwendung  
sshkeys { 'joeuser': content => '...' }
```

← beliebiger, eindeutiger Titel

Module:

- Fassen Code und Daten zusammen
- Leichte Wiederverwendbarkeit
- Modul-Sammlung z.B. bei „Puppet Forge“ (siehe Hersteller-Website) verfügbar
- Automatische Initialisierung innerhalb Moduls durch init-Manifest
- Einheitliche Struktur:

```
{module}/  
  files/  
  lib/  
  manifests/  
    init.pp ← automatisch geladen,  
               ermöglicht weitere Initialisierungen  
    {class}.pp  
    {namespace}/  
      {class}.pp  
  templates/  
  tests/
```

Templates:

- Ermöglicht Dateiinhalte mit (teilweise) dynamischen Inhalten
- Verwendung von ERB-Notation (Implementierung von „embedded ruby“, auch eRuby genannt)
 - Code innerhalb spitzer Klammern mit Prozentzeichen

```
<% ...ruby-Code... %>
```

- Kurzschreibweise zur Ausgabe von Variablen / Ausdrücken: Gleichheitszeichen am Anfang

```
<%= hostname %>
```

- Schleife zur Ausgabe über ein Array
 - Definition Werte in Manifest:

```
$servernamen = ['server1', 'server2', 'server3', ]
```

- Ausgabe im Template:

```
<% servernamen.each do |srv| -%>  
<%= srv %>  
<% end -%>
```

Konfigurationsdateien bearbeiten:

- Einheitliche, hierarchische Sicht auf verschiedene Konfigurationsformate
- Unterstützung von Dateien/Formaten über passende „Linsen“ („lenses“)
- Zugriff / Änderung von relevanter Teile einfach, Rest bleibt unberührt

```
# augtool
augtool> print /files/etc/php.ini
[...]
/files/etc/php.ini/PHP/expose_php = "On"
/files/etc/php.ini/PHP/max_execution_time = "30"
/files/etc/php.ini/PHP/max_input_time = "60"
/files/etc/php.ini/PHP/max_input_vars = "1000"
/files/etc/php.ini/PHP/memory_limit = "128M"
/files/etc/php.ini/PHP/error_reporting = "E_ALL & ~E_DEPRECATED"
/files/etc/php.ini/PHP/display_errors = "Off"
/files/etc/php.ini/PHP/display_startup_errors = "Off"
/files/etc/php.ini/PHP/log_errors = "On"
/files/etc/php.ini/PHP/log_errors_max_len = "1024"
/files/etc/php.ini/mail function
/files/etc/php.ini/mail function/SMTP = "localhost"
/files/etc/php.ini/mail function/smtp_port = "25"
/files/etc/php.ini/mail function/sendmail_path = "/usr/sbin/sendmail -t -i"
augtool> set /files/etc/php.ini/PHP/expose_php Off
```

Konfigurationsdateien bearbeiten:

- Integration von Augeas in puppet verfügbar

```
class php {
  package { ['php','php-mysql','php-gd','php-pdo'] :
    ensure => installed,
  }
  Augeas { "/etc/php.ini":
    context => "/files/etc/php.ini",
    changes => [
      "set PHP/expose_php off",
      "set mail\ function/mail.add_x_header off",
      "set Date/date.timezone Europe/Berlin",
      "set PHP/display_errors on",
      "set PHP/error_reporting E_ALL\ &\ ~E_NOTICE",
      "set PHP/memory_limit 128M",
      "set PHP/post_max_size 64M",
      "set PHP/upload_max_filesize 64M",
      "set PHP/max_execution_time 180",
      "set Session/session.save_path /tmp",
    ],
    notify => [ Service['httpd'] ],
  }
}
```

Fakten schaffen:

- Erweiterung der factor-Funktionalität über eigene Module möglich
- Werte stehen in puppet in Variablen zur Verfügung
- Beispielskript zur (simplen) Ermittlung aller Laufwerke unterhalb eigenen Modulpfads ablegen, etwa `modules/mymodules/lib/factor/harddrives.rb`

```
Factor.add(:harddrives) do
  setcode do
    Factor::Util::Resolution.exec("ls /dev/sd? /dev/hd? 2>/dev/null | tr -s '\n' ','")
  end
end
```

Fakten schaffen:

- Beispielskript zur Erkennung Hard-/Software-RAID

```
Factor.add("raidtype") do
  confine :kernel => :linux
  ENV["PATH"]="/bin:/sbin:/usr/bin:/usr/sbin"
  setcode do
    raidtype = []
    if FileTest.exists?("/proc/mdstat")
      txt = File.read("/proc/mdstat")
      raidtype.push("software ") if txt =~ /^md/i
    end

    lspciexists = system "/bin/bash -c 'which lspci >&/dev/null'"
    if $? .exitstatus == 0
      output = %x{lspci}
      output.each { |s|
        raidtype.push("hardware ") if s =~ /RAID/i
      }
    end
  end
end
end
```


Vorteile:

- Zentrale Verwaltung von Definitionen und Regeln
- Zentrale Ablage der „Fakten“ für alle Systeme
- Bildung von Gruppen möglich
(alle Webserver, alle Server mit bestimmtem Hostnamen-Schema, ...)

Konfiguration Server:

- Installation Server-Paket „puppet“, inkl. facter und (optional) Augeas
- CentOS, RedHat, Fedora: Paket „puppet-server“, bei Debian „puppetmaster“
- Laden aller Definitionen für Nodes und Klassen,
z.B. über Einträge in /etc/puppet/manifests/site.pp vornehmen

```
import "nodes/*.pp"  
import "classes/*.pp"
```

- Je Node eine Datei im nodes-Verzeichnis anlegen
für Definition von Ressourcen, Verwendung von Klassen, ...

```
node "testclient.example.com" {  
    include basenode  
}
```

Konfiguration Server:

- Klassen können dann beispielsweise weitere Klassen includen oder Ressourcen definieren

```
class basenode {
  if ($mta == "")
  {
    $mta = 'postfix'
  }

  include basepackages, crond, puppet, sshd, logrotate, logwatch, collectd, mailalias, nrpe

  if ($is_virtual == "false")
  {
    include ntp
    include lm_sensors
    include dns_resolver
  }
}
```

- Ablegen aller weiteren benötigten Dateien auf dem Server (files, templates, ...)
- Verwendung von puppet:// URLs für Zugriff auf Dateien vom Server
- puppetmaster-Daemon starten :-)

Client-Server-Verbindung:

- Puppet auf Client installieren, ggf. Server-Einstellung anpassen (/etc/puppet/puppet.conf)
 - Standard-Servername ist „puppetmaster“

```
[agent]
server = puppetsrv.example.com
```

- Betrieb Client als Daemon möglich (Standardmäßig Ausführung alle 30min.)
oder händischer Start (Parameter --test beim Aufruf)
- Erster Verbindungsaufbau erzeugt einen Zertifikatsrequest
 - Praxistipp: Client kann auf Akzeptieren des Zertifikat warten

```
# puppetd --waitforcert 60 --test
```

- Zertifikatsanfragen am Server prüfen und signieren

```
# puppetca --list
# puppetca --sign <Clientname> oder
# puppetca --sign --all Aufpassen was signiert wird!
```

- Prüfen welche Änderungen auf dem Client durchgeführt würden (Trockenübung)

```
# puppetd --test ---noop
```

Arbeit mit mehreren Umgebungen:

- Trennung in z.B. „dev“ und „production“
- Angabe des Environment für einen Node auf Server oder Client
 - Client-seitige Wahl des Environment empfohlen
- Standard-Environment: „production“

- Konfiguration des Environment auf dem Server (puppet.conf):
 - Standard-Einstellungen aus [master] für unbekanntes Environment

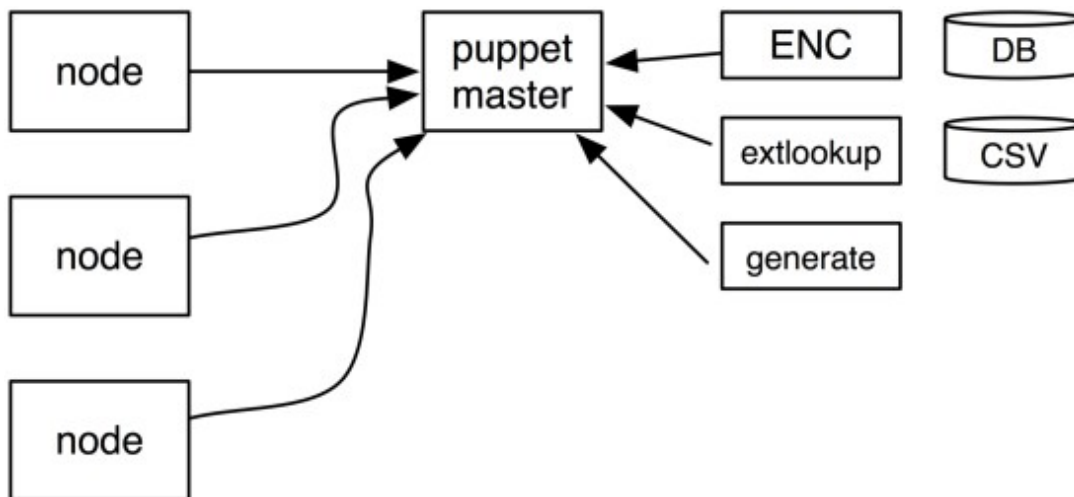
```
[master]
manifest = $confdir/manifests/site.pp
[production]
manifest = $confdir/env/prod/manifests/site.pp
[dev]
manifest = $confdir/env/$environment/manifests/site.pp
```

- Konfiguration des Environment auf dem Client (puppet.conf):

```
[agent]
environment = dev
```

Vielfältige Möglichkeiten zur Integration:

- Generierung Node-Konfiguration über „external node classifier“ (ENC)
 - Externes Programm erhält Node-Namen, liefert YAML-Dokument zurück
 - z.B. Abfrage gegen beliebige Datenbanken



Grafik / Beispiel von Jan-Piet Mens, jpmens.net

```
---
classes:
  dyn:
    ports:
      - 25
      - 587
    smtpmx: gw.mens.de
    users:
      alex:
        home: /nfs/alex
        uid: 502
      jane:
        home: /home/jane
        uid: 201
    ssh: "
parameters:
  location: Chicago
  manager: Jane Doe
  my_memory: 216
  nodename: cp09.mens.de
```

Vielfältige Möglichkeiten zur Integration:

- Von facter gesammelte „Fakten“ für Generierung von Konfigurationen nutzen
 - Pro Node ermittelt
 - Daten zentral auf dem puppet-master verfügbar (/var/lib/puppet/yaml/facts)
- Monitoring-Konfiguration
 - Welche Hosts
 - Welche Dienste / Ports
- Backup-Konfiguration
 - Automatische Einrichtung auf Backuptarget
 - Backup-Rollover, ...
- SSHFP-Records generieren / publizieren
 - „ssh“-Fact sammelt bereits RSA/DSA-Keys
 - Hieraus SSHFP-Records generieren und ins DNS bringen

- Definition und Umsetzung klarer Regeln
- Leichte Administration einer größeren Anzahl Maschinen / Umgebungen
- Flexible Konfigurationsmöglichkeiten

- Nutzung lokal oder im Client-Server-Betrieb
- Zentrale Sammlung aller „Fakten“ auf Server
 - Ermöglicht weitere Verarbeitung für z.B. Monitoring-Konfiguration

- Leichter Einstieg für einzelne Teilaspekte einer Konfiguration möglich
- Ausbaufähig zu umfangreichen Regelwerken

- Aufwand für Definition aller Abhängigkeiten und Sonderfälle nicht unterschätzen!

- Cheatsheet für alle Kern-Ressourcetypen und ihre Attribute:
http://docs.puppetlabs.com/puppet_core_types_cheatsheet.pdf
- „Puppet Forge“ (Sammlung fertiger Module):
<http://forge.puppetlabs.com/>
- External Node Classifiers (ENC):
<http://jpmens.net/2011/07/25/external-node-classification-and-data-in-puppet/>
- Generieren von Konfigurationen aus „facts“, z.B. SSHFP-Record-Einträge
<https://github.com/jpmens/facts2sshfp>
<http://jpmens.net/2012/02/01/on-collecting-ssh-host-keys-for-sshfp-dns-records/>

Danke fürs Zuhören
sowie
viel Erfolg und Spaß
mit puppet & Co.!

Link zu den Slides: <http://talks.speedpartner.de/>

Bei Fragen stehen wir selbstverständlich gerne zur Verfügung:

Stefan Neufeind, neufeind@speedpartner.de
SpeedPartner GmbH, <http://www.speedpartner.de/>