Puppen schnitzen

Praktische Erfahrungen aus einem Puppet-Migrationsprojekt

Axel Bock

Automatisierungswerkzeuge für größere Serverumgebungen gewinnen an Beliebtheit und Bedeutung. Für Puppet-verwaltete Umgebungen ist ein Rollen- und Profil-basierter Ansatz recht populär. In diesem Artikel sollen einige praktische Lösungen und Herangehensweisen dargelegt werden, die der Autor bei der Umsetzung dieses Ansatzes in einem Projekt als Antwort auf auftauchende Probleme entwickelt hat.

Worum geht es

In meinem letzten Projekt musste ich eine bestehende, sehr organisch gewachsene Puppet-Codebasis auf das Rollen-Konzept migrieren. Dieser Artikel verarbeitet hauptsächlich die Richtlinien, die sich bei der Umsetzung für uns als nützlich erwiesen haben, und einige der gewonnenen Erfahrungen.

Beim Einsatz des Konfigurationsmanagement-Tools Puppet [Puppet] beschreibt man den gewünschten Zielzustand eines Servers deklarativ mit Hilfe einer domänenspezifischen Sprache. Der zentrale Bestandteil sind Puppet-Manifest-Dateien, in denen Ressourcen beschrieben werden, die von Puppet konfiguriert werden. Ich setze im Artikel voraus, dass der Leser Puppet kennt und zumindest schon in Ansätzen damit gearbeitet hat.

Es wäre weiterhin schön, wenn das Rollen- und Profilkonzept als Begriff inhaltlich zumindest grob bekannt ist (andernfalls empfehle ich, den sehr lesenswerten Original-Blog-Eintrag von Craig Dunn [Dunn12] vorher durchzulesen).

Dieser Artikel zusammen mit den referenzierten Informationen sollte ein guter Startpunkt sein, um selbst ein solches Projekt anzugehen, und eventuell einige Überlegungen vorwegnehmen, die sicherlich auftauchen werden.

Eine kleine Rekapitulation

Eine *Rolle* ist die funktionale Sicht (oder auch „Business-Sicht“) eines Servers ohne spezifischen technologischen Bezug, wie „Mailserver“, „DNS-Server“ usw. Das wesentliche Kriterium ist: Nicht-Technikern kann die Rolle ein Begriff sein. („Wir brauchen einen neuen DNS-Server [=*Rolle!*] für eine neue interne Domain.“ „Wir brauchen wegen erhöhter Last jetzt mehr Applikationsserver [=*Rolle!*] im Load Balancing.“)

Ein *Profil* ist eine Klasse, die eine einzelne Funktion für die einbindende Rolle „verwendungsfertig“ konfiguriert. „webserver“, „apache“, „dnsmasq_server“ oder „pam_login_configuration“ könnten Beispiele für Profil-Namen sein. Das für uns wesentliche Kriterium war hierbei „verwendungsfertig“.

Ein *Modul* ist eine Sammlung von Klassen und Ressourcen, die sich unter einem singulären Oberbegriff („apache“, „postgresql“) der portierbaren Einrichtung beziehungsweise Verwaltung einer einzelnen technologischen Komponente widmen, zum Beispiel „apache“. Das wesentliche Kriterium hierbei war für uns: Portierbarkeit.

Rein prinzipiell geht es bei dem Konzept darum, die Verwaltung einer Serverlandschaft von einem eher naiven Ansatz (s. Listing 1) in einen abstrahierten Ansatz zu überführen (s. Listing 2). Sämtliche Beispiele gehen übrigens von einem Linux/Unix-System aus.

```
# file: site.pp
node 'webserver.mydomain.com' {
  include ssh
  ssh_authorized_key { 'admin_logins':
    ensure => 'present',
    user => 'root',
    key => 'a-long-string-...',
  }
  include ::apache
  apache::vhost { 'www.booyah.de':
    document_root => '/srv/www/booyah',
    default_ssl_vhost => false,
  }
  file { 'booyah_index':
    path => '/srv/www/booyah/index.html',
    ensure => 'present',
    content => 'puppet:///modules/apachestuff/booyah/index.html',
  }
  mount { '/srv/www/booyah':
    ensure => 'present',
    atboot => true,
    device => '/dev/mapper/www-booyah',
    fstype => 'ext4',
    before => File['booyah_index'],
  }
  include ::logrotate
  # ... und z.B. noch ein paar logrotate-Regeln für apache,
  # hier nicht näher definiert.
}
```

Listing 1: Hier wollen wir weg von ...

```
# file: site.pp
node 'webserver.mydomain.com' {
  include roles::infrastructure::dns_server
}
# file: modules/roles/manifests/infrastructure/dns_server.pp
class roles::infrastructure::dns_server
inherits roles::base::standard_server {
  include ::profiles::ssh
  include ::profiles::ssh::admin_logins
  include ::profiles::webserver::apache

  # das geht auch anders/besser, dient hier nur ein Beispiel
  include ::profiles::apache::hosts::booyah_de
}
```

Listing 2: ... da wollen wir hin

Ros, Pros und Mods – die Eigenschaften

Rein technisch gesehen sind Rollen, Profile und Module ganz einfach Puppet-Klassen, die allerdings durch ihre Klassifizierung eine semantische Bedeutung bekommen und gewissen künstlichen Restriktionen unterliegen. Wir haben für uns folgende Richtlinien aufgestellt, deren Einhaltung für alle Beteiligten zwingend war.

Die Eigenschaften einer Rolle sind:

- Rollen sind nicht kombinierbar.
- Rollen können von einer anderen Rollen-Klasse erben.
- Rollen bestehen ausschließlich aus „include profiles::...“-Anweisungen (oder notfalls „class{ 'profiles::...': }“-Anweisungen).
- Das Zuweisen einer Rolle zu einem Server stellt diesen vollständig fertig, und nach einem Puppet-Lauf sind keine manuellen Nacharbeiten mehr notwendig.

Die Eigenschaften eines Profils sind:

- Profile müssen kombinierbar sein (bis auf offensichtliche Ausnahmen).
- Profile dürfen nicht von anderen Profile-Klassen erben.
- Profile sind funktional klar von jedem anderen Profil abgegrenzt, ohne Überschneidungen.
- Profile kennen nur Module, und gegebenenfalls andere Profile (beispielsweise darf das „tomcat"-Profil das „java"-Profil einbinden).
- Profile enthalten unternehmensspezifische Konstrukte, sind daher nicht portierbar auf andere Umgebungen, und außerdem der Ort für „krumme Geschäfte" – potenziell unsaubere oder fehleranfällige Konstrukte.
- Profile konfigurieren sich selbst vorzugsweise aus Hiera [Hiera].
- Profile befassen sich mit *ausschließlich einer* Technologie beziehungsweise einem „Objekt" („java" verwaltet natürlich Java, und „tomcat" verwaltet tomcat, bindet aber „java" für die dafür notwendige Installation des JDK ein).

Die Eigenschaften eines Moduls sind:

- Am besten holt man sich ein bestehendes Modul von der Puppet Forge und nutzt dieses (oder holt es sich, erweitert es und stellt die Änderungen zur Verfügung).
- Nur wenn das überhaupt nicht geht, sollte man ein eigenes entwickeln.
- Dieses muss dann ohne Änderungen der Allgemeinheit zur Verfügung gestellt werden können (in Form einer Veröffentlichung auf der Puppet Forge). Ob man das letztlich tut, ist egal, aber es muss *möglich* sein.
- Daher sollte ein Modul auch keine obskuren Abhängigkeiten haben, außer zu „allgemein genutzten" Modulen mit Hilfsfunktionen (wie puppetlabs-concat, ein wirklich sehr hilfreiches Modul).

Im Fokus: Rollen

Rollen sind intern eine Sammlung von include-Anweisungen auf Profil-Klassen. Das bedeutet auch: Sämtliche nicht-funktionalen Anforderungen an Rollen müssen aus Profilen kommen. Da solche Einstellungen typischerweise auf *jedem* Host notwendig sind, haben wir zwei Basis-Rollen geschaffen, die (über Vererbung) die Standard-Konfiguration weitergeben (s. Abb. 1).

Der „empty_server" ist ein leerer Server, ohne jede Konfiguration, allerdings in unserem Fall mit unseren Repositories konfiguriert.

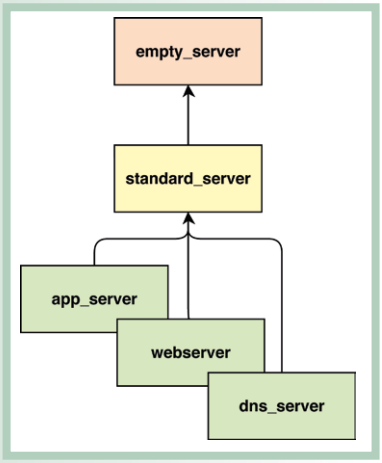

Abb. 1: Rollen-Vererbungshierarchie

„standard_server" enthält dann alle Konfigurationen, die „ein Server halt nun mal braucht" – PAM-Login-, Mail-, DNS-, Scheduler-Einstellungen usw.). Listing 3 zeigt, wie wir diese Funktionen in die Rolle eingebracht haben.

Beachtenswert hier ist vielleicht die eine Abweichung vom „Rolle enthält Profile"-Dogma in Form von „company::prepare". Hier hätte man natürlich auch „profiles::preparation::root_user_password" nutzen können, da die Aktionen der verschiedenen Module allerdings eher trivial sind, haben wir zur Verdeutlichung „company::prepare::..." gewählt.

Man beachte, dass wir nicht eine einzige „Riesen-Klasse" mit allen notwendigen Anpassungen an die Unternehmensanforderungen geschrieben haben, sondern viele kleine. Hier gilt wieder: Eine Funktion je Profil, auch wenn es etwas mehr Aufwand bedeutet. Hieraus haben wir im Übrigen eine Art „Richtlinie" abgeleitet: Das Design ist dann gut, wenn Änderungen an den Anforderungen die minimal möglichen Diff-Zeilen an bestehenden Dateien verursachen. Oder anders: Fällt das Setzen des Kernel-Schedulers weg, dann wird nur *eine* Zeile entfernt.

```
# file: modules/roles/manifests/base/standard_server.pp
class roles::base::standard_server
inherits roles::base::empty_server {
  include company::prepare::pam_ldap_config
  include company::prepare::kernel_scheduler
  include company::prepare::root_user_password
  include profiles::managed::files
  include profiles::managed::users
}
```

Listing 3: Auszüge der „standard_server"-Rolle

Im Fokus: Profile

Kommen wir zum meiner Ansicht nach wichtigsten Teil des Engineerings – den Profilen. Unser Profil-Design-Ansatz wird am ehesten an einem Beispiel deutlich. In Listing 4 habe ich exemplarisch ein Beispiel entworfen, wie ich die Apache-Profile, die ich in Listing 2 eingeführt habe, möglicherweise umgesetzt hätte: Ein „webserver::apache"-Profil bindet apache ein und hält offen, ob man später zum Beispiel als Alternative noch „webserver::nginx" parallel erstellen möchte (daher nicht einfach „profiles::webserver"). Für die apache-spezifische Webseite existiert ein Unter-Profil im „profiles::apache"-Zweig.

Man kann außerdem schon erahnen, dass der Aufwand für die Schaffung einer sauberen Profil-Struktur nicht unerheblich ist. Unsere Erfahrung bestätigt dies, und noch etwas: Dieser Aufwand zahlt sich doppelt und dreifach aus, wenn unvorhergesehene Änderungen auftauchen.

Noch einmal zurück zu Listing 4, mit einem Augenmerk auf „profiles::logrotate::apache". Nehmen wir an, dieses Profil definiere ausschließlich eine logrotate-Regel für /var/httpd. Natürlich könnte man diese Regel auch einfach direkt ins „webserver::apache"-Profil packen. Doch angenommen, wir wollten statt logrotate etwas anderes verwenden? Dann würde unsere Minimale-Diff-Zeilen-Regel von oben klar ein „include ..."-Statement favorisieren.

```
# file: modules/profiles/manifests/webserver/apache.pp
class profiles::webserver::apache {
  include ::apache
  include ::profiles::logrotate

  # hier ginge natürlich auch profiles::apache::logrotate,
  # das ist Geschmackssache, aber Hauptsache überall konsistent!
  include ::profiles::logrotate::apache
}

# file: modules/profiles/manifests/apache/hosts/booyah_de
class profiles::apache::hosts::booyah_de {
  include ::profiles::webserver::apache
  apache::vhost { 'www.booyah.de':
    document_root => '/srv/www/booyah',
    default_ssl_vhost => false,
  }
```

```
  file { 'booyah_index':
    path => '/srv/www/booyah/index.html',
    ensure => 'present',
    content =>
      'puppet:///modules/profiles/apache/hosts/booyah_de/index.html',
  }
  mount { '/srv/www/booyah':
    ensure => 'present',
    atboot => true,
    device => '/dev/mapper/www-booyah',
    fstype => 'ext4',
    before => File['booyah_index'],
  }
}
```

Listing 4: Die „apache“-Profile

Interessant war für mich die Erfahrung, dass ich in Profilen sogar relativ unsauber arbeiten kann – solange ich eine saubere funktionale Struktur einhalte (worauf ich beim Thema „Profile“ klar den Schwerpunkt legen möchte), geht das ziemlich gut, und, falls man sich mal bemüßigt fühlt aufzuräumen, kann man sehr zügig vorankommen. Das Schaffen dieser Struktur ist jedoch mehr Arbeit, als man annehmen möchte …

Schwerpunkt Module

Nach Rollen und Profilen jetzt die Module. Generell möchte ich hier feststellen, dass sich das „install/config/service“-Pattern (vgl. [Piena12]) für mich bewährt hat, und auch zur Konsistenz beiträgt. Man findet sich einfach schnell zurecht. Darüber hinaus möchte ich auf einige Dinge eingehen, die mir bei der Entwicklung aufgefallen sind.

Wie sollten Konfig-Dateien aussehen?

Module verwalten Software, und damit auch generell die Konfigurationsdateien dieser Software. Das kann „so original wie möglich“ oder „minimalistisch“ erfolgen, was ich gerne anhand der Software „postfix“ erläutern würde.

Die Begriffe muss man erklären. Die Standard-Konfigurationsdatei [Apple] von postfix, „main.cf“, ist normalerweise ein Traum – für Neulinge. *Jede* Einstellung ist mit mindestens drei Zeilen Kommentar einzeln aufgeführt. Der Nachteil: Eine etwa 25 KB große Datei – auch ohne getroffene Einstellungen.

„So original wie möglich“ bedeutet, dass das Puppet-Modul versucht, die Original-Datei so gut wie möglich beizubehalten und wirklich nur exakt die Zeilen zu ändern, die geändert werden müssen. Hintergrund: Man sieht dann in dem von Puppet erzeugten Diff exakt, welche Einstellungen angepasst wurden.

Ich persönlich mag präzise Konfigurationsdateien. Ich möchte „cat main.cf“ machen und auf einen Blick sehen, was los ist, und nicht aus Hunderten Kommentarzeilen die relevanten Einstellungen heraussuchen müssen. Daher bevorzuge ich, wenn Puppet-Module minimale Konfigurationsdateien erstellen. (Die dürfen allerdings durchaus lesbar formatiert sein!)

```
# ...
$config_file = '/etc/postfix/main.cf'
exec { 'save_config_file':
  command => "cp '${config_file}' '${config_file}.orig'",
  creates => "${config_file}.orig",
  requires => Package['postfix_package'],
  before => File[$config_file],
}
```

Listing 5: Sichern von Konfigurationsdateien vor dem Überschreiben

Ich persönlich habe mir allerdings angewöhnt, die Original-Datei aufzuheben. Meine Puppet-Module, die eine Konfigurationsdatei anlegen oder überschreiben, sichern das Original mit der Endung „.orig“. Listing 5 liefert ein Beispiel dafür, wie das gehen kann.

Wohin mit den Parametern?

Es gibt für die main.cf-Datei über 40 unterschiedliche Parameter. Nur, wie sollte das API hierfür aussehen? Der direkte Ansatz wäre, so vorzugehen wie in Listing 6 und diese alle der Klasse als Parameter zu übergeben.

```
# file: modules/postfix/init.pp
class postfix (
  $alias_database = undef,
  $alias_maps = undef,
  $command_directory = undef,
  $daemon_directory = undef,
  # und so weiter ...
) {
  # do-the-magic-stuff
}
```

Listing 6: Postfix-Modul mit Klassenparametern

Hier bin ich persönlich zwiegespalten. Einerseits sind die Parameter dann offen sichtbar, und es ist klar, welche Datentypen sie erwarten. Andererseits kann dies zu ziemlich viel Code führen, wenn man die Parameter anschließend validieren möchte (was man tun sollte – viel Arbeit anfänglich, aber es vereinfacht die Fehlersuche später manchmal deutlich). Da die Puppet-DSL leider nicht zum Programmieren geeignet ist, kann das sehr unübersichtlich werden und von der eigentlichen Funktion der Klasse (oder Ressource) ablenken. Neugierige können sich unter [PuppetLogRotate] mal anschauen, welche Formen das annehmen kann, wenn man es gründlich machen möchte.

Ich lagere daher ganz gerne die gesamte Validierung in das Template der Konfig-Datei aus. Diese bekommt anschließend einfach einen Hash mit den gesetzten Konfigurationsdaten (vgl. Listing 7).

```
# file: modules/postfix/init.pp
class postfix (
  $configuration_data = {},
  $save_config_file = true,
) {
  file { '/etc/postfix/main.cf':
    ensure => 'present',
    content => template('postfix/main.cf.erb'),
  }
}

#file: modules/postfix/templates/main.cf.erb
<%
  # hier validieren. das geht NICHT einfach so im Modul ...
  cd = @configuration_data
  cd.keys.each { |key|
    raise "NO UPPERCASE PLEASE. found in: #{key}" if key.index(/[A-Z]/)
  }
%>
# und so weiter
```

Listing 7: Postfix-Modul mit der Intelligenz im Template

Zwar kann man die verfügbaren Parameter nicht mehr auf Anhieb erkennen (aber dafür gibt es RDoc), doch die Interpretation und Validierung der Parameterwerte ist in einem Template schlichtweg einfacher, und die Menge Boilerplate-Code reduziert sich bisweilen ungemein. Das gilt insbesondere dann, wenn sich Parameter gegenseitig beeinflussen sollen, um einen zusätzlichen Komfort-Nutzen einzubringen, der über die „ein-

fache“ Weiterleitung der Parameterwerte in die Konfig-Dateien hinausgeht.

Die Grenze ist jedoch sehr fließend und vom persönlichen Geschmack abhängig. Für Inspirationen kann man zum Beispiel die gelungene „vhost“-Ressource des Puppetlabs-apache-Moduls [vhost] betrachten.

Auch Module erben

Kurz – Klassen in Modulen sollten von Parameterklassen erben dürfen, und müssen das auch, wenn sie Standard-Parameter gesetzt haben. Warum ist in Listing 8 verdeutlicht.

```
# file: modules/i_inherit/manifests/init.pp
class i_inherit (
  $package_name = $::i_inherit::params::package_name,
) inherits ::i_inherit::params {
  package { $package_name: ensure => 'present' }
}

# file: modules/i_dont/manifests/init.pp
class i_dont (
  # das wird nicht funktionieren, $package_name ist <undef>!
  $package_name = $::i_dont::params::package_name,
) {
  include ::i_dont::params
  package { $package_name: ensure => 'present' }
}
```

Listing 8: Erben von Parameterklassen

Der Wert von $my_parameter ist **<undef>** in der i_dont-Klasse, während in der i_inherit-Klasse der Wert dem in der Params-Klasse gesetzten Wert entspricht. Bis mindestens Puppet in Version 3.7.3 ist das Verhalten so; wer mir nicht glaubt, findet unter [PuppetTest] einen Test.

Parameterklassen an sich finde ich sehr hilfreich, da diese zum Beispiel distributionsspezifische Werte herausfinden und übergeben können. Dieser funktional irrelevante Code „verschmutzt“ dann nicht die eigentliche Klasse.

Schwerpunkt: Alles

Konsistenz

Es wurden viele Dinge angesprochen, die man mal so oder so machen kann: Profil-Strukturen, Modul-Parameter, sicherlich kann man auch die eine oder andere Richtlinie überdenken, und so weiter. Für uns war sehr wichtig, dass wir uns beim Treffen von Design-Entscheidungen immer an bestehenden Formaten orientiert haben. So war gewährleistet, dass jeder sich in den Strukturen des Anderen auf Anhieb zurechtfand. Das betraf Namensgebung genauso wie Klassenparameter, Modulaufbau oder Informationsfluss. In meinen Augen ist dies einer der wichtigsten Botschaften dieses Artikels.

Neuaufbau von Maschinen

Ändert sich ein Host eher grundlegend (z. B. durch einen Wechsel der verwendeten Profile), sollte das Credo sein: Neuaufbau. Bei einer entsprechend ausgestatteten Infrastruktur (wir nutzten hierfür „The Foreman“ und hauptsächlich VMs) dauert das keine 10 Minuten. Positiver Nebeneffekt: Wenn man alles danach ausrichtet, dass alle Maschinen jederzeit neu aufgebaut werden können und direkt danach laufen, dann wird man sehr entspannt, wenn mal eine ausfällt.

Continuous Integration

Wir haben mit unseren Puppet-Rollen ein Continuous Integration-Verfahren entwickelt, das jede Nacht alle Rollen einmal aufgebaut hat. So hatten wir zumindest die tägliche Kontrolle, dass die Checkins nicht den Aufbau einer (oder vieler) Rollen kaputt gemacht haben. Das hat sich als sehr hilfreich erwiesen.

Dokumentation

Eine Klasse, die nicht dokumentiert ist, sollte nicht existieren. Wir hatten oft das Problem, dass uns bei vielen Dingen nicht mehr bekannt war, warum sie genau so gelöst wurden und nicht anders. Und dann vereinfacht man, weil es sich anbietet, und dann klappt es manchmal, und manchmal nicht – und das kostet Zeit und Nerven. Also gewöhnt euch Dokumentation an, für alles, auch den kleinsten Test. Zwei Sätze (ja, zwei!) reichen.

Letzte Worte

Wer früher noch mit ein paar Skripten und „vim a.conf b.conf“ einen Server installiert hat, der sieht sich heute einem Werkzeug gegenüber, das eine eigene Infrastruktur braucht, eigene Dokumentation, und für das man eigenen Code schreibt.

Denn Puppet-Code *ist* Code, und sollte auch so betrachtet werden. Der Abschnitt über „Continuous Integration“ war schon ein Hinweis. Für Puppet-Manifeste sollten die gleichen Regeln und Anforderungen gelten wie für Software. (Insbesondere, was Unit-Tests betrifft, die ich jetzt gar nicht besprochen habe, die man aber *unbedingt* verwenden sollte! Mehr gibt es dazu allerdings auch nicht zu sagen)

Jeder Systemadministrator, der Puppet-Klassen, -Module, -Funktionen oder -Facts schreibt, ist also ein Entwickler und API-Designer.

Links

[Apple] main.cf.default, `http://bit.ly/1zj9TJD`
[Dunn12] C. Dunns Blog, Designing Puppet – Roles and Profiles, main.cf.default, `http://www.craigdunn.org/2012/05/239/`
[Hiera] `https://docs.puppetlabs.com/hiera/1/complete_example.html`
[Piena12] R. I. Pienaar, Simple Puppet Module Structure Redux, 2012, `http://www.devco.net/archives/2012/12/13/simple-puppet-module-structure-redux.php`
[Puppet] `http://www.puppetlabs.com`
[PuppetWikipedia] `http://de.wikipedia.org/wiki/Puppet_(Software)`
[PuppetLogRotate] `https://github.com/rodjek/puppet-logrotate/blob/master/manifests/rule.pp` oder `http://bit.ly/1zjchQm`
[PuppetTest] `https://github.com/flypenguin/puppetplayground` oder `http://bit.ly/1wSyohm`
[vhost] apache::vhost von Puppetlabs, `https://github.com/puppetlabs/puppetlabs-apache/blob/master/manifests/vhost.pp` oder `http://bit.ly/1u1zPYo`

Axel Bock ist freiberuflicher Entwickler und Systemadministrator mit den Schwerpunkten Java, Python und Puppet sowie einer ausgeprägten Abneigung gegen sich wiederholende Tätigkeiten. Privat liest er gerne und mag Kampfsport.

E-Mail: mr.axel.bock@gmail.com