

Continuous Delivery with Puppet

Anders Malmborg und Michael Haslgrübler

14. Oktober 2012

1 Einleitung

1.1 Ausgangssituation

Iteratives Vorgehen bei der Entwicklung von Applikationen ist mittlerweile sehr verbreitet. Jede Iteration erweitert meist die Applikation mit neuer Funktionalität. Jede Implementierung und Konfiguration einer neuen Funktionalität sollte außerdem in der Iterationsphase qualitätsgesichert werden. Zusätzlich zu UnitTests, welche einzelne Teile prüfen, sind Integrationstests mit einer laufenden Applikation zwingend erforderlich - sei es automatisch oder manuell.

Jede Applikation beinhaltet außerdem Konfigurationen, im banalsten Fall sind das Einstellungen für den Zugriff auf eine Datenbank oder Schalter welche Teilfunktionalitäten aktivieren. Je nach den Konfigurationsmöglichkeiten können mehrere Installationen notwendig sein um die korrekte Funktionsweise einer Applikation, in all ihren Variationen, sicherzustellen.

Für die Source-Code-Übersetzung, Testausführung und Paketierung der Applikation helfen Continuous Integration Systeme wie [Jenkins]. Aber wie automatisiert man den nächste Schritt - die automatische Installation und Konfiguration von mehreren Servern?

Für diese Problemstellung haben wir uns nach einer Lösung umgesehen und mit Puppet einen gangbaren Weg gefunden.

[Puppet] oder auch [Chef] sind so genannte Configuration Management(CM) Lösungen. Zusammengefasst bietet ein CM-Lösung die Möglichkeit den erwarteten Zustand eines Systems (hier Server) zu beschreiben und bei Abweichungen Mechanismen um das System in dem erwarteten Zustand zu versetzt.

Folgende Probleme wollten wir adressieren und dafür Lösungen finden:

- Automatische Installation und Konfiguration nach erfolgreichen Build im Jenkins.
 - trotz Paketierung in Web Archivs (WARs) waren einige Konfigurationen, wie Datenbank-Parametern, manuell einzutragen.
- Zentrale Definition, welche Applikationen mit welcher Konfiguration wo, auf welchen Servern, zu laufen haben.
- Neue virtuelle Servers sollten einfach von einem Basis-Image aufgesetzt werden und mit Puppet fertig konfiguriert werden, inklusive beispielsweise Apache und Tomcat.
- Gleiche Mechanismen und selbes Vorgehen bei der Test-/QA- und Produktionsumgebung.

In der Entwicklung heißt das in erster Linie Softwarecode der committed wird, der gebaut werden kann und die automatisierten Tests besteht soll auch installiert werden. Damit können wir gewährleisten bzw. überprüfen das die Software zu jedem Zeitpunkt einsatzbereit ist und nicht nur auf einem Entwicklungs-PC funktioniert.

1.2 Puppet Einführung

Puppet benutzt eine Domain-Specific-Language(DSL) um den Zustand eines System zu beschreiben. Der Code wird strukturiert in Manifeste und Module.

Ein Manifest ist ein Puppet "Program". Module sind für die Puppet-EntwicklerIn ähnlich wie Libraries für Programmierer. Ein Großteil der Manifeste und Module machen die Definition von

Ressourcen aus. Eine Ressource ist ein atomarer Typ eines Systems, es entspricht einer physischen Identität eines Computersystems. Ein Beispiel für eine solche Ressource, wäre ein Benutzer oder eine Datei. In Listing 1 wird die Ressource `user puppetdemo` und dazugehörige Ressource `file /home/puppetdemo` definiert. Der Aufruf `sudo puppet apply manifest/user.pp` führt es aus und muss mit 'sudo' aufgerufen werden damit Verzeichnis und Benutzer angelegt werden können, da dies nur der Administrator darf. Mit `puppet resource user puppetdemo` wird die Informationen zum neu angelegten Benutzer ausgegeben, siehe Listing 2.

Für die Entwicklung von Puppet Module und Manifeste bietet sich [Geppeto] an. Geppeto bringt Code Completion und Syntax Highlighting mit und kommt als Standaloneapplikation oder als Plugin für eine bestehende Eclipse-Installation.

```

1 $ cat manifest/user.pp
2 node default {
3   user {
4     'puppetdemo' :
5       ensure => present,
6       home  => '/home/puppetdemo',
7       shell => '/bin/bash',
8   }
9   file {
10    '/home/puppetdemo' :
11      ensure => 'directory',
12      owner  => 'puppetdemo',
13      group  => 'puppetdemo',
14    }
15 }
16 $ sudo puppet apply manifest/user.pp
17 notice: /Stage[main]/Node[default]/User[puppetdemo]/ensure: created
18 notice: /Stage[main]/Node[default]/File[/home/puppetdemo]/ensure: created
19 notice: Finished catalog run in 0.36 seconds

```

Listing 1: User mit Puppet anlegen

```

1 user { 'puppetdemo':
2   ensure => present,
3   gid    => '1004',
4   home   => '/home/puppetdemo',
5   shell  => '/bin/bash',
6   uid    => '1002',
7 }

```

Listing 2: Anzeige der Benutzerinformation in Puppet

Eine Aufzählung der vom Puppet unterstützten Ressourcen wird mit `puppet resource --types` ausgegeben. Die Ressourcen sind im Core Types Cheat Sheet http://docs.puppetlabs.com/puppet_core_types_cheatsheet.pdf auch gut beschrieben.

1.3 Testbox mit Vagrant

Für die Entwicklung und Tests von Puppet Modulen und Manifeste empfiehlt sich eine virtuelle Maschine. Puppet Labs stellt solche für VMware und VirtualBox zur Verfügung unter <http://info.puppetlabs.com/download-learning-puppet-VM.html>.

Eine andere Möglichkeit ist [Vagrant], welche wir auch verwenden. Vagrant ist eine Konfigurationstool für die Verwaltung von virtuellen Maschinen mit VirtualBox. Es kann in weiterer Folge auch Puppet, Chef oder Shell Scripts benutzen kann um die virtuelle Maschine zu konfigurieren. Analog zu Virtualbox kann Vagrant via Paketmanager für Linux installiert werden oder von den entsprechenden Downloadseiten heruntergeladen werden.

Eine Liste mit vorgefertigten Vagrant Boxen gibt es übrigens auf <http://www.vagrantbox.es/>

Nachdem Vagrant und VirtualBox installiert worden sind, können wir eine Box zum Testen aufsetzen. In unserem Fall verwenden wir eine 64 Bit Version von Debian Squeeze. Diese beinhaltet eine Minimalinstallation mit den für Vagrant üblichen Vorbereitungen: SSH Key Setup, VirtualBox Guest Additions, Puppet und Ruby, siehe auch http://vagrantup.com/v1/docs/base_boxes.html.

```
1 vagrant box add debian_squeeze_64 http://dl.dropbox.com/u/937870/VMs/squeeze64.box
```

Listing 3: Download der Vagrant Box

Nachdem dem Download steht uns jetzt die Vagrant Box *debian_squeeze_64* zur Verfügung. Nun können wir in ein beliebiges Verzeichnis wechseln und eine initiale Konfiguration basierend auf der Box anlegen, siehe Listing 4.

```
1 vagrant init debian_squeeze_64
```

Listing 4: Vagrant initialisieren

Diese initiale Konfiguration beinhaltet alles was Vagrant zum Konfigurieren und Starten der Maschine braucht, es sind keine weiteren Einstellungen mehr nötig und wir können diese starten, siehe Listing 5.

```
1 vagrant up
```

Listing 5: Starten der Vagrant Maschine

Nachdem die virtuelle Maschine gestartet worden ist, können wir mit ssh einsteigen, siehe Listing 6. Unter Windows ist dieser Befehl derzeit nicht verfügbar und man muss deshalb mit Tools wie [Putty] darauf zugreifen.

```
1 vagrant ssh
```

Listing 6: Mit ssh in der Vagrant Maschine einsteigen

Zum Aktivieren von Puppet müssen wir unseren *Vagrantfile* bearbeiten. Dieser liegt im Verzeichnis wo wir den Befehl *vagrant init* ausgeführt haben. Diese Datei beinhaltet bereits Einträge für Puppet, welche jedoch auskommentiert sind. Nach dem entfernen der Kommentarzeichen für die Sektion *config.vm.provision :puppet*, tragen wir unter *puppet.manifests_path* und *puppet.module_path* ein wo unsere Manifeste und Module liegen. Als *puppet.manifest_file* tragen wir *user.pp* ein. Zum Testen von einer späteren Apache-Installation wird Port 6400 im Host auf Port 80 im Guest weitergeleitet: *config.vm.forward_port 80, 6400*. Das ganze sollte dann in etwa wie in Listing 7 aussehen.

```
1 config.vm.forward_port 80, 6400
2 config.vm.provision :puppet do |puppet|
3   puppet.manifests_path = "~/git/puppet-demo/puppet/manifests"
4   puppet.module_path = "~/git/puppet-demo/puppet/modules"
5   puppet.manifest_file = "user.pp"
6 end
```

Listing 7: Puppet Provisioning in Vagrantfile konfigurieren

Bei dieser Änderung müssen wir die Vagrantbox neu laden, da geteilte Verzeichnisse nur beim Starten des Hosts erkannt und automatisch gemounted werden. Zusätzlich dazu wird beim Starten das Provisioning, durch Puppet mit den Anweisungen aus *user.pp* durchgeführt. Mit *puppet ssh* steigen wir und verifizieren dass der User 'puppetdemo' mit Home-Verzeichnis /home/puppetdemo vorhanden ist, siehe Listing 8.

Die Vagrant Boxen werden unter Unix in *\$HOME/.vagrant.d/boxes* installiert. Falls man dies ändern will kann man die Umgebungsvariable *VAGRANT_HOME* setzen:

```
export VAGRANT_HOME=
$HOME/vagrant_home
```

```

1 vagrant reload
2 notice: /Stage[main]//Node[default]/User[puppetdemo]/ensure: created
3 notice: /Stage[main]//Node[default]/File[/home/puppetdemo]/ensure: created
4 notice: Finished catalog run in 0.35 seconds
5
6 $ vagrant ssh
7 Welcome to your Vagrant-built virtual machine.
8
9 $ sudo su - puppetdemo
10 puppetdemo@precise32:~$ pwd
11 /home/puppetdemo

```

Listing 8: Vagrant Box neu laden

Mit *vagrant provisioning* kann das Puppet Manifest erneut ausgeführt werden. Ohne Änderungen im Manifest oder in der virtuellen Maschine passiert nichts. Würde der Benutzer *puppetdemo* zum Beispiel entfernt, wird er wieder beim nächsten Provision-Vorgang wieder angelegt. Will man das Puppet Manifest in der virtuellen Maschine manuell ausführen, ist der *puppet.manifests.path* als */tmp/vagrant-puppet/manifest* gemounted, siehe Listing 9.

```

1 $ sudo puppet apply /tmp/vagrant-puppet/manifests/user.pp
2 No LSB modules are available.
3 notice: Finished catalog run in 0.04 seconds
4 $ sudo deluser puppetdemo
5 Removing user 'puppetdemo' ...
6 Warning: group 'puppetdemo' has no more members.
7 Done.
8 $ sudo puppet apply /tmp/vagrant-puppet/manifests/user.pp
9 No LSB modules are available.
10 notice: /Stage[main]//Node[default]/User[puppetdemo]/ensure: created
11 notice: Finished catalog run in 0.39 seconds

```

Listing 9: Puppet apply im Box

Vagrant kann wie VirtualBox die Maschine stilllegen mit *vagrant suspend* bzw mit *vagrant resume* wieder fortsetzen. Zum Starten und Stoppen kann man *vagrant up* bzw mit *vagrant halt* verwenden. Sollte man die Box nicht mehr benötigen kann Sie mit *vagrant destroy* unwiderruflich löschen. Das Vagrantfile bleibt jedoch erhalten, somit kann man wieder bei Null anfangen und mit *vagrant up* die Box inklusive Provisioning mit Puppet wieder aufsetzen.

2 Apache Webserver und eine Applikation mit HTML und JavaScript mit Puppet installieren

Nachdem wir die Grundlagen von Puppet und Vagrant jetzt kurz kennengelernt haben, installieren wir einen Apache Webserver. Da es wahrscheinlich ist, dass Apache für andere Applikationen auch genutzt wird, wird der Puppet Code dafür in einem Module abgelegt. Module sind wiederverwendbare Einheiten vom Code und Daten. Die Struktur eines Modules ist auf <http://docs.puppetlabs.com/learning/modules1.html> beschrieben, wie auszugsweise in Abbildung 10 dargestellt.

```

1 my_module
2 |-- files
3 |   '-- my_file.txt
4 |-- manifests
5 |   '-- init.pp
6 '-- templates
7   '-- my_template.erb

```

Listing 10: Puppet Module Struktur



Abbildung 1: Apache aufrufen

Auf die gleiche Ebene wie das manifests-Verzeichnis legen wir das Verzeichnis *modules* an, darunter die Datei *apache/manifests/init.pp* und entsprechenden Verzeichnisse, siehe Listing 11.

```

1 puppet
2 |-- manifests
3 |   |-- user.pp
4 |-- modules
5 |   |-- apache
6 |       |-- manifests
7 |           |-- init.pp
    
```

Listing 11: Verzeichnisstruktur für den apache-Module

Die Datei *modules/apache/manifests/init.pp* ist recht kompakt und zeigt auf einen Blick einen der großen Vorteile von Puppet. Die Ressourcen *package* und *service* verstecken die Komplexität und die plattformspezifische Vorgänge für Installation von Paketen und das Starten der Dienste, siehe Listing 12.

```

1 class apache {
2     package {
3         'apache2' :
4             ensure => present,
5     }
6     service {
7         'apache2' :
8             ensure => running,
9             require => Package["apache2"]
10    }
11 }
    
```

Listing 12: Inhalt von modules/apache/manifests/init.pp

Um ein Modul in einem Manifest verwenden zu können, reicht es lediglich *include apache*, siehe Listing 13 zu schreiben.

```

1 include apache
    
```

Listing 13: Inhalt von manifests/setupapache.pp

In Vagrantfile ändern wir jetzt die Zeile *puppet.manifest_file = "user.pp"* auf *puppet.manifest_file = "setupapache.pp"* und geben auf der Kommandozeile *vagrant provision* ein, siehe Listing 14.

```

1 $ vagrant provision
2 [default] Running provisioner: Vagrant::Provisioners::Puppet...
3 [default] Running Puppet with /tmp/vagrant-puppet/manifests/setupapache.pp...
4 notice: /Stage[main]/Apache/Package[apache2]/ensure: ensure changed 'purged' to 'present'
5 notice: Finished catalog run in 97.25 seconds
    
```

Listing 14: vagrant provisioning für Apache

Wenn wir auf unseren Browser die lokale Adresse über, den weitergeleiteten, Port 6400 aufrufen, können wir feststellen dass der Apache Webserver läuft, siehe Abbildung 1.

Als nächstes lassen wir Puppet die Dateien für die Applikation in das Apache Wurzel Verzeichnis (*/var/www/*) kopieren. Folgender Eintrag in Vagrantfile stellt das Host-Verzeichnis *~/git/puppet-demo/puppet-demoapp/src/main/webapp* als Guest-Verzeichnis */demoapp* zur Verfügung, siehe Listing 15. Das Guest-Verzeichnis dient als Quelle für die Kopie.

```
1 config.vm.share_folder "demoapp", "/home/vagrant/demoapp", "~/git/puppet-demo/puppet -
demoapp/src/main/webapp"
```

Listing 15: Shared folders in Vagrantfile konfigurieren

Mit dem Manifest *manifests/demoapp.pp* für die Installation der Applikation, siehe Listing 16, definieren wir das Puppet die Dateien vom zuvor spezifizierten Verzeichnis */demoapp* nach */var/www/* kopieren sollen, das Ganze ist im Module *demoapp* gekapselt, siehe Abbildung 17.

```
1 include apache
2 include demoapp
```

Listing 16: Puppet Manifest für die Applikation

```
1 class demoapp($demoappname='demoapp') {
2   file {
3     "/var/www/$demoappname" :
4     ensure => directory,
5     source => '/home/vagrant/demoapp',
6     require => Package['apache2'],
7     recurse => true,
8   }
9 }
```

Listing 17: Puppet Module demoapp

Durch die Änderung der geteilten Verzeichnisse im Vagrantfile ist es notwendig die Vagrant Box neu zu starten um die Änderung aktiv werden zu lassen: *vagrant reload*, siehe Listing 18.

```
1 $ vagrant reload
2 [default] Attempting graceful shutdown of VM...
3 [default] Clearing any previously set forwarded ports...
4 [default] Forwarding ports...
5 [default] -- 22 => 2222 (adapter 1)
6 [default] -- 80 => 6400 (adapter 1)
7 [default] Creating shared folders metadata...
8 [default] Clearing any previously set network interfaces...
9 [default] Booting VM...
10 [default] Waiting for VM to boot. This can take a few minutes.
11 [default] VM booted and ready for use!
12 [default] Mounting shared folders...
13 [default] -- demoapp: /demoapp
14 [default] -- v-root: /vagrant
15 [default] -- v-pp-m0: /tmp/vagrant-puppet/modules-0
16 [default] -- manifests: /tmp/vagrant-puppet/manifests
17 [default] Running provisioner: Vagrant::Provisioners::Puppet...
18 [default] Running Puppet with /tmp/vagrant-puppet/manifests/demoapp.pp...
19 notice: /Stage[main]//File[/var/www/demoapp]/ensure: created
20 ...
21 notice: /File[/var/www/demoapp/index.html]/ensure: defined content as '{md5}90
a8d419b9c7b43b09ba73abebaf8f4c'
22 ...
23 notice: Finished catalog run in 1.34 seconds
```

Listing 18: Puppet reload mit Provisioning der Applikation

Nach Abschluss des Neustarts erfolgt auch automatisch wieder der Provisioning Vorgang durch Puppet und wir können unsere neue Webapplikation über den Browser aufrufen, siehe Abbildung 2.



Abbildung 2: Die Applikation im Browser

Eine Änderung in der Datei *index.html* wird bei Puppet im Provisioning-Vorgang (*vagrant provision*) erkannt und Puppet aktualisiert die Datei auch in */var/www/*, siehe Listing 19.

```

1 $ vagrant provision
2 [default] Running provisioner: Vagrant::Provisioners::Puppet...
3 [default] Running Puppet with /tmp/vagrant-puppet/manifests/demoapp.pp...
4
5 notice: /File[/var/www/demoapp/index.html]/content: content changed '{md5}90
   a8d419b9c7b43b09ba73abebaf8f4c' to '{md5}0a4ee5bb63c3e5c29cc54cf36a4be23c'
6
7 notice: Finished catalog run in 0.71 seconds

```

Listing 19: Puppet Provisioning nach Änderung von index.html

3 Puppet am Server

Nachdem dem erfolgreichen erstellen erster Manifeste, geht es nun darum wie man die Manifeste auf einem Server verlagert und eine ganze Serverfarm damit betreibt. Grundsätzlich unterscheiden wir bei Puppet zwischen zwei Typen Puppet Master und Puppet Agent. Der Puppet Agent ist auf einem x-beliebigen Server installiert und kontaktiert eine zentrale Einheit, den Puppet Master, um von ihm gesteuert zu werden.

3.1 Puppet Server-Agent-Workflow

Der Puppet Server-Agent-Workflow funktioniert grundsätzlich nach dem Pull Prinzip, d.h. der Puppet Agent fragt aktiv beim Puppet Master nach was zu tun ist, er fordert einen *Catalog* an in dem er dem Master seinem Namen und seine Informationen über sich selbst, sogenannte *facts* mitteilt. Ein *fact* wäre zum Beispiel das Betriebssystem des Agent. Der Puppet Master identifiziert und sucht nach Arbeitsanweisungen für den Agent. Der Master kompiliert aus allen anwendbaren Manifesten einen *Catalog* und sendet ihn zurück an den Agent. Dieser wendet den *Catalog* an d.h. es wird versucht den durch den *Catalog* definierten Zustand herzustellen. Das Herstellen dieses Zustandes wird protokolliert und dann an den Puppet Master gesendet. Dieser Report kann zu einem späteren Zeitpunkt ausgewertet werden.

3.2 Arbeitsanweisungen für den Agent suchen

Wenn der Puppet Master vom Puppet Agent kontaktiert wird, wird er mithilfe seines Hostnamen identifiziert und der Puppet Master liest das Manifest */etc/puppet/manifests/site.pp* ein und sucht nach einer passenden *node* Definition. Wenn der Master nun vom *server1.example.org* kontaktiert wird wird dieser die Ausgabe *hello server1!* erzeugen, analog für *server2*. Statt alle Server einzeln zu definieren kann man auch das Schlüsselwort

default verwenden damit werden die beinhaltende Anweisungen auf allen Servern die den Master kontaktieren ausgeführt werden.

```

1 node server1.example.org {
2     notice("hello server1!")
3 }
4
5 node server2.example.org {
6     include apache
7     notice("hello server2!")
8 }
9
10 node default {
11     notice("hello server")
12 }

```

Listing 20: Node Definitionen in `/etc/puppet/manifests/site.pp`

3.3 Alternative: External Node Classifier

```

1 [master]
2     node_terminus = exec
3     external_nodes = /usr/local/bin/my_node_classifier

```

Listing 21: External Node Classifier Konfiguration des Puppet Master

Alternativ zu obigen deklarativen Ansatz kann Puppet zusätzlich auch einen External Node Classifier verwenden. Ein External Node Classifier ist ein Script dem als Argument der Puppet Agent Name übergeben wird und der eine *YAML* Datei ausgibt. Die *YAML* Datei enthält eine Liste von Klassen und Parametern welche dem Knoten zugewiesen werden.

Im vorigen Beispiel hat die Knotendeklaration, siehe Listing 20 für den server2 das *apache* Modul enthalten, analog dazu müsste der ENC das *YAML* Dokument, siehe Listing 22, ohne die letzte Zeile zurückliefern.

Interessant wird das Ganze erst wenn der ENC mit einem Konfigurationswerkzeug gekoppelt wird, in dem dynamisch Konfigurationen erstellt werden können. Ein solches Werkzeug wäre das [Puppet Dashboard]. Nachteil hierbei ist es, das parametrisierte Klassen derzeit nicht verwaltet werden können. In unserem obigen Beispiel könnten wir die Demoapplikation unter einer anderen URL auch verfügbar machen in dem wir den Klassenparameter *demoappname* überschreiben, siehe Listing 22.

```

1 name: server2.example.org
2
3 classes:
4     apache:
5     demoapp:
6         demoappname: mydemoapp

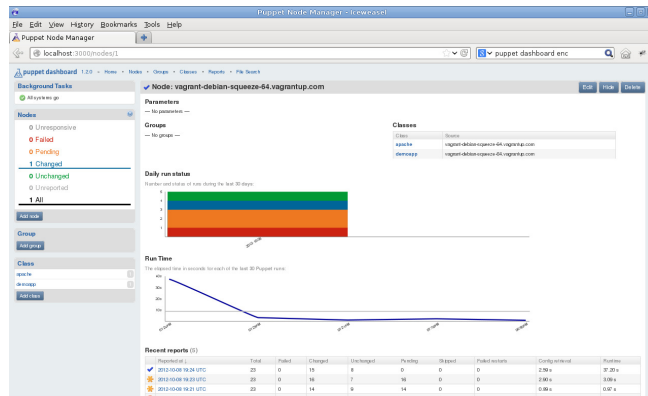
```

Listing 22: ENC's *YAML* mit Klassenparameter

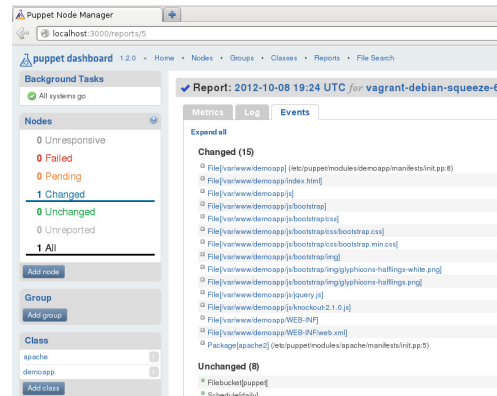
3.3.1 Dashboard

Das Puppet Dashboard kann nicht nur als ENC verwendet werden sondern ist primär eigentlich ein Reporting Tool. In Abbildung 3(a) sieht man eine Übersicht über den aktuellen Knoten. Im oberen Teil sieht man welche Gruppen, Klassen und Parameter

dem Knoten zugeordnet sind. In unserem Fall sind das die Klassen *apache* und *demoapp*. Die Klassen müssen im Modulpfad (*/etc/puppet/modules/*) des Puppet Master liegen. Desweiteren sieht man eine Übersicht über die Änderungen am Knoten, als sich der Puppet Agent mit dem Master verbunden hat und das Provisioning durchgeführt wurde. Der Provisioning-Vorgang wird im Dashboard, siehe Abbildung 3(b) und in der Ausgabe des Puppet Agent festgehalten, siehe Abbildung 23.



(a) Übersicht eines Knoten



(b) Änderungen im Dashboard

Abbildung 3: Puppet Dashboard

```

1 $ puppet agent --test --server vagrant-debian-squeeze-64.vagrantup.com
2 info: Caching catalog for vagrant-debian-squeeze-64.vagrantup.com
3 info: Applying configuration version '1349723897'
4 notice: /Stage[main]/Apache/Package[apache2]/ensure: ensure changed 'purged' to 'present'
5 notice: /Stage[main]/Demoapp/File[/var/www/demoapp]/ensure: created
6 ...
7 notice: /File[/var/www/demoapp/js/knockout-2.1.0.js]/ensure: defined content as '{md5
      }235475c7c3dc43c7cb7f6125be536c32'
8 notice: Finished catalog run in 35.07 seconds

```

Listing 23: Puppet Agent Run durchführen

4 Resumé

Wir wollten für uns mit Puppet und Vagrant Lösungen für folgende Probleme finden:

- Automatische Installation und Konfiguration nach erfolgreichen Build im Jenkins
- Zentrale Definition, welche Applikationen mit welcher Konfiguration wo, auf welchen Servern, zu laufen haben.
- Neue virtuelle Server sollten einfach von einem Basis-Image aufgesetzt werden und mit Puppet fertig konfiguriert werden, inklusive beispielsweise Apache und Tomcat.
- Gleiche Mechanismen und Vorgehen bei der Test-/QA- und Produktionsumgebung.

Beim Jenkins Build wird nicht nur die Software kompiliert sondern es erfolgt auch eine Paketierung. Die Konfiguration kann aber nicht Teil der Paketierung sein sondern muss zu einem späteren Zeitpunkt bestimmt werden. Weiters muss definiert werden, wenn möglich zentral, wo diese Applikationen mit welcher Konfiguration zu laufen haben. Puppet in Verknüpfung mit einem ENC bietet uns genau diese Funktionalität. Wir definieren im ENC, welcher Server mit welcher Software von Puppet ausgestattet werden soll und

über Parameter der dazugehörigen Puppet Module definieren wir, wie diese Software konfiguriert wird.

Mithilfe von Puppet können wir auf Basis eines Standard-Image, wie in obigen bei Vagrant eingesetzt, einen Server fertig konfigurieren und auf unsere Bedürfnisse anpassen und das mit dem selben Mechanismus – wiederholbar – in jeder Umgebung, sei es QA oder Produktion.

Autoren

Anders Malmborg



hat jahrezehntelange Erfahrung in der Applikations- und Produktentwicklung im C++ und JavaEE Umfeld und arbeitet als IT Freelancer im automotive Bereich.

Michael Haslgrübler



hat mehrjährige Erfahrung in JavaEE Entwicklungsumfeld in der Automotive und Immobilienbranche. Er administriert seit Jahren einen Linux-Root-Server für diverse Kunden.

Literatur

[Chef] Chef. Chef is a systems integration framework, built to bring the benefits of configuration management to your entire infrastructure.

[Geppeto] Geppeto. Eclipse plugin for developing puppet modules and manifests.

[Jenkins] Jenkins. An extendable open source continuous integration server.

[Puppet] Puppet. Puppet is it automation software that helps system administrators manage infrastructure throughout its lifecycle, from provisioning and configuration to patch management and compliance.

[Puppet Dashboard] Puppet Dashboard. Web interface und reporting tool für puppet.

[Putty] Putty. Ein ssh client für windows.

[Vagrant] Vagrant. Virtualized development for the masses.