Oliver Tigges
(oliver.tigges@innoq.com)
ist Principal Consultant bei der innoQ Deutschland GmbH. Er befasst sich seit zehn Jahren mit der Architektur und Realisierung von Webanwendungen und verteilten Unternehmensanwendungen. Sein besonderes Interesse gilt momentan den Themen DevOps, Continuous Delivery und Graphendatenbanken.

Skalierbare Softwaresysteme – Vom Entwickler-Notebook bis zur Serverfarm

Skalierbarkeit ist eines der klassischen Qualitätsmerkmale von IT-Systemen. Wenn wir diesen Begriff hören, denken wir üblicherweise sofort an die Skalierung des Systems „nach oben". Es scheint also meistens darum zu gehen, inwieweit unser System durch das Hinzufügen von mehr RAM, CPU oder zusätzlichen Maschinen auch entsprechend mehr Durchsatz erreicht oder mehr Last vertragen kann. Oft ist aber auch die Skalierbarkeit „nach unten" interessant, also die Lauffähigkeit des Systems bei sehr wenig verfügbaren Ressourcen. Eine solche Flexibilität ist zum Beispiel sehr hilfreich, wenn ein komplexes System automatisiert durch viele unterschiedlich dimensionierte Entwicklungs-, Test- und Abnahmeumgebungen wandern muss, bevor es in der Produktivumgebung ankommt.

Beim Entwurf der Systemarchitektur haben wir in der Regel das Produktionssystem im Auge. Das ist letztendlich die einzige Umgebung, die zählt. Für diese Umgebung legen wir fest, wie viele physikalische und virtuelle Maschinen wir aufsetzen, wie die Software darauf verteilt wird und wie die Netzwerkinfrastruktur mit Load Balancern und Firewalls aussieht (vgl. Abbildung 1).

Bevor der Rechenzentrumsbetrieb die Software aber in der Produktionsumgebung installieren kann, muss sie sich noch in verschiedenen Tests auf unterschiedlichen Umgebungen bewähren. Üblicherweise existieren hierzu mehrere Testumgebungen, auf denen unterschiedliche Aspekte überprüft werden: generelle Start- und Lauffähigkeit, fachliche Funktionalität, Integration in die Systemlandschaft, Schnittstellen, Performance und Verhalten unter Last.

Erfahrungsgemäß nimmt die Anzahl der Testumgebungen mit der Lebenszeit eines Systems zu.

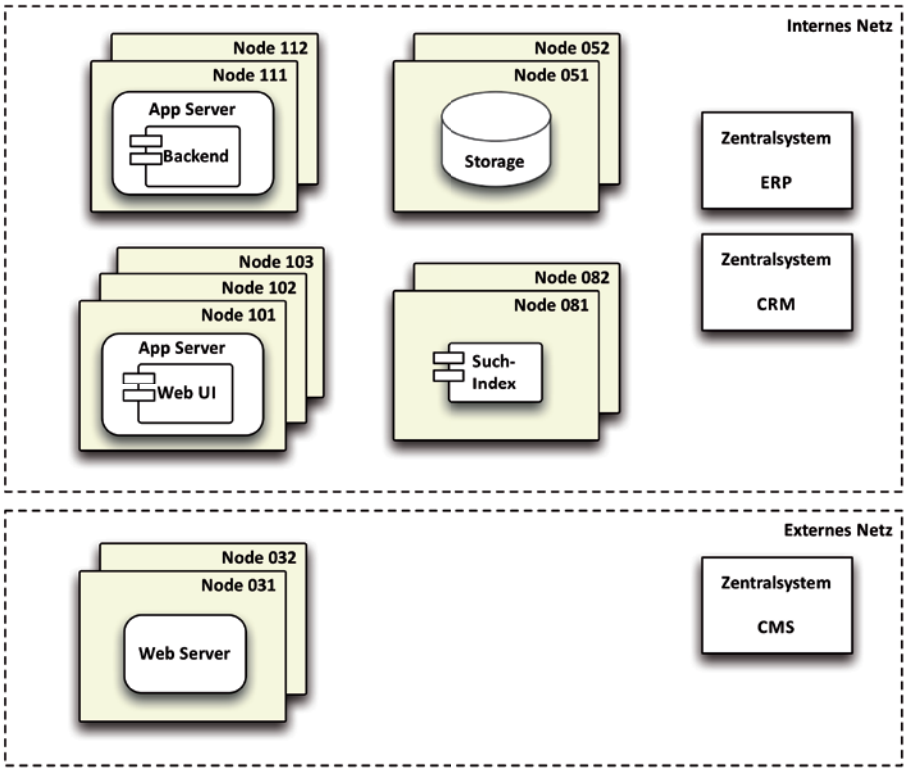

Abb. 1: Beispielhafte Systemarchitektur in der Produktionsumgebung

An dieser Stelle tauchen oft mehrere Herausforderungen und Fragen auf: Wie kann man effektiv und kostenschonend neue Umgebungen aufsetzen und die bestehenden pflegen und betreiben? Und wie stellen wir sicher, dass sich das Verhalten unserer Software auf einer Testumgebung auch auf die Produktion übertragen lässt?

Offensichtlich ist es aufwendig und teuer, wenn jede Maschine und jeder Application-Server manuell installiert und konfiguriert werden muss und für jede Umgebung individuelle Routing- und Firewall-Regeln definiert werden müssen.

Eine oft praktizierte, aber nicht zielführende Lösung ist es, in den Testumgebungen auf viele Details zu verzichten, die das Leben schwer machen: Es gibt dann für die Tests keine Firewall, keinen vorgelagerten Web-Server, keine Zertifikate und keine Verschlüsselung. Oder von jedem Application-Server wird nur eine einzelne Instanz eingesetzt, um den Aufwand für Clustering und Replikation zu sparen.

In dieser Konstellation kann man sich leider nie ganz sicher sein, dass ein erfolgreicher Systemtest in der Testumgebung auch ein erfolgreiches Rollout in die Produktion garantiert. Schon eine minimale Konfigurationsabweichung im Application-Server oder eine einzige vergessene Firewall-Regel kann das Rollout scheitern lassen.

Automatisierung des Rollouts

Eine tatsächliche Lösung für diese Problemstellung verspricht hingegen die Automatisierung der Serverkonfiguration und der Softwareinstallation, was auch als „Infrastructure as Code“ bezeichnet wird.

Diese Automatisierung des Rollouts könnte grundsätzlich durch einen Satz von Skripten auf Betriebssystemebene erfolgen. Aus guten Gründen etabliert hat sich aber der Einsatz spezieller Werkzeuge für Konfigurationsmanagement und Server-Automatisierung (vgl. [Puppet], [Chef]). Solche Werkzeuge erlauben, in einer auf den spezifischen Zweck ausgerichteten Syntax die „Infrastruktur zu programmieren“ (siehe auch Box „Puppet, Chef und Vagrant“).

Die Quelltexte für Server- und Rollout-Automatisierung – ob es nun Batch-Skripte, Chef-Rezepte oder Puppet-Module sind – sollten genau wie die Quelltexte der Software in einer Versionsverwaltung gepflegt werden. Sie sind ein ebenso elementarer Bestandteil eines jeden Releases wie die Software-Artefakte der Anwendung selbst.

Bei einem Blick in die Versionsverwaltung sollten wir neben den Anwendungsquelltexten auch Code für diese Infrastrukturelemente finden:

- Systembenutzer- und gruppen
- Systemdienste (z. B. cron, syslog, sendmail)
- Netzwerk und Load Balancer
- Lokale Firewall (iptables), Zertifikate und Keystores
- Logging und Monitoring
- Laufzeitumgebungen (Java JRE/JDK)
- Webserver (Apache, Nginx) und Web Caches (Squid)
- Application-Server
- Datenbankkonfigurationen und Migrationswerkzeuge

Ein bewährter Ansatz mit Puppet und Chef ist die Zusammenfassung dieser einzelnen Konfigurationselemente zu größeren Einheiten, die mit einem eindeutigen Namen identifiziert werden können. Beispielsweise können wir ein Modul „App-Server“ aus der Anlage eines Betriebssystem-Users, einigen Installationspaketen und einem Satz von Konfigurationsdateien zusammenstellen. Das gesamte Modul kann dann als Einheit einem Zielknoten – einer virtuellen oder physikalischen Maschine – zugewiesen werden.

Sobald uns jetzt die Kollegen aus dem Test-Rechenzentrum einen frisch aufgesetzten Server zur Verfügung stellen, können wir dort innerhalb von Minuten unser System bzw. einen bestimmten Teil unseres verteilten Systems installieren lassen. Sofern dafür ein Job im zentralen Build-Server existiert, läuft das Ganze sogar sprichwörtlich „auf Knopfdruck“.

An dieser Stelle möchte ich nicht die notwendigen Investitionen für eine weitgehende IT-Automatisierung unterschlagen. Programme zu schreiben (und vor allem auch zu testen), die einen Service zuverlässig installieren und konfigurieren, ist in der Regel nicht trivial und um ein Vielfaches aufwendiger, als den Service manuell einzurichten. Neue Technologien und Methoden müssen erschlossen werden und auch organisatorisch müssen die Verantwortlichkeiten und Schnittstellen zwischen Entwicklung und Betrieb angepasst werden.

Wann sich die Investitionen auszahlen, müssen also alle Unternehmen individuell für sich selber einschätzen. Es ist nicht garantiert, dass dies schon im ersten Projekt der Fall sein wird.

Too big to scale

Um ein Softwaresystem flexibel auf unterschiedliche Hardwareumgebungen verteilen zu können, muss es zunächst einmal überhaupt etwas zu verteilen geben. Die Verantwortlichen für den Entwurf der System- und Softwarearchitektur sollten deshalb vermeiden, riesige, monolithische Blöcke zu schaffen, die nur als Einheit auf einem Server installiert und betrieben werden können.

Diese können nur als Gesamtheit skaliert werden, auch wenn vielleicht nur ein einziges Modul in der Gesamtanwendung unter hoher Last steht. Wenn das System in mehrere einzeln installierbare Einheiten aufgeteilt wird, kann für jede Einheit individuell festgelegt werden, wie viele Instanzen laufen sollen und welche Systemressourcen ihnen zugeteilt werden. Natürlich wird die Systemarchitektur durch viele kleine Anwendungen komplexer, aber gerade durch die Automatisierung von Konfiguration und Rollout bleibt diese Komplexität beherrschbar.

Ein weiterer Einflussfaktor auf die Flexibilität des Systems und damit die Skalierbarkeit ist die gewählte Verteilungsarchitektur. Ein klassisches Vorgehen im Bereich der Java Application-Server ist der Betrieb eines Clusters, in dem Komponenten verteilt und Daten und Zustandsinformationen repliziert werden. Durch die Replikationsanforderung ist aber in der Regel einerseits die Anzahl der Knoten im Cluster begrenzt und es ist andererseits nicht möglich, die Server an beliebig weit auseinander liegenden Orten zu betreiben.

Wenn wir stattdessen eine *Shared Nothing Architektur* [Sto86] entwerfen, unsere Application-Server also keinen Zustand teilen und gänzlich unabhängig voneinander laufen, können wir bei Bedarf beliebig viele weitere Instanzen hochfahren. Das einzige, was die Betreiber des Rechenzentrums tun müssen: sie dem Load Balancer und den Firewalls bekannt machen.

Gerade in der Java-Welt machen es einem die meist zustandsbehafteten Web-Framewoks hier schwer. Zustandslose Web-Frameworks, wie z. B. Rails oder Django, erleichtern es hingegen, viele Instanzen einer Web-Anwendung parallel aufzusetzen.

Abweichungen minimieren

Durch die Versionierung des Infrastruktur-Codes und den Verzicht auf jegliche manuellen Eingriffe in die Systemkonfiguration kann ein zentraler Unsicherheits-

faktor aus dem Rechenzentrum verbannt werden: Unbewusste Abweichungen zwischen den Testumgebungen und der Produktion. Durch die Automatisierung wird sichergestellt, dass in allen Umgebungen klar definierte Versionen von Betriebssystem, Systemdiensten, Java-Laufzeitumgebung sowie Web- und Application-Servern installiert sind.

Natürlich muss es gewisse Abweichungen zwischen den einzelnen Umgebungen geben, allein wenn man an IP-Adressen, Host-Namen oder SSL-Zertifikate denkt. Und Unterschiede bei der Anzahl und Dimensionierung der Maschinen sollen durch die Automatisierung der Server-Konfiguration doch gerade unterstützt werden.

Wichtig ist hierbei, dass alle Abweichungen bewusst und aktiv verwaltet werden.

Für jede Umgebung muss es einen Satz an Parametern für den Infrastruktur-Code geben. In einer Java-Umgebung würden dadurch unter anderem Optionen für die Application-Server definiert, um beispielsweise folgende Eigenschaften festzulegen:

- Speicherverwaltung
- Garbage Collection
- Thread- und Connection-Pools
- Einstellungen zu Logging und Monitoring
- Ausnahmebehandlung und Timeouts

Idealerweise kann über diese Parametrisierung für jede Systemkomponente pro Umgebung individuell definiert werden, wie viel Speicher und wie viele Prozesse oder Threads genutzt werden dürfen. Die Herausforderung liegt darin, die entsprechenden Stellschrauben zu identifizieren und sinnvolle Default-Werte zu definieren.

Flexibilität durch Konfigurierbarkeit

Neben den oben genannten direkten Vorteilen der Serverautomatisierung, die während der Entwicklung und beim Betrieb viel manuelle Arbeit, Fehlersuche und Sorgen vor dem Rollout nehmen, bringt sie auch noch etwas anderes: Flexibilität in der Gestaltung unserer Umgebungen.

Wir können durch die Programmierung der Infrastruktur sehr schnell die Zuordnung von Komponenten und Diensten zu Zielknoten ändern. Auf der Produktionsumgebung gibt es vermutlich genügend Maschinen, um das System aus Gründen der Lastverteilung und Ausfallsicherheit so zu verteilen, dass von jeder Komponente mindestens zwei Instanzen auf verschiedenen physikalischen Maschinen laufen.

Sofern in einer Fachtestumgebung hingegen nur wenige virtuelle Server zur Verfügung stehen, müssen mehrere Komponenten auf einer Maschine gebündelt werden (vgl. Abbildung 2).

Wenn dem Rechenzentrumsbetrieb auffällt, dass eine einzelne Komponente unter ganz besonders hoher Last steht, kann er für genau diese Komponente einen zusätzlichen Server bereitstellen. Mit wenigen Zeilen Code in der Konfiguration ordnet er die Komponente dann diesem Server zu. Gerade wenn Rechenzentren immer weiter auf Virtualisierung setzen und sehr schnell und flexibel Server zur Verfügung stellen können, zahlen sich die Investitionen in die Automatisierung von Serverkonfiguration und Rollout schnell aus.

Konfigurierbarkeit verbessern

Bei der Automatisierung von Servern und Softwareverteilung können wir einen Effekt beobachten, den auch viele Anwender von Test-Driven-Development (TDD) erlebt haben: Ganz unabhängig vom Aufwand-Nutzen-Verhältnis der Tests selbst führt TDD in der Regel zu einer besseren Strukturierung der Software. Um jede Klasse isoliert von ihrem Kontext testen zu können, müssen die Entwickler schließlich viel Wert auf eine saubere Trennung der Zuständigkeiten legen.

Bei der Einführung von Infrastruktur als Code können wir ähnliches feststellen: Um die Komponenten eines Systems auf verschieden dimensionierte Zielmaschinen zu verteilen, müssen wir uns zwangsläufig Gedanken über eine flexible Parametrisierung der Komponenten machen. Und wenn wir dann erst einmal für jedes Zielsystem über einen Satz von Parametern definieren können, wie viele Instanzen einer Komponente mit welcher Laufzeitkonfiguration auf welche Zielknoten installiert werden, dann haben wir auch direkt die Werkzeuge an der Hand, um unser Produktivsystem bei Bedarf sehr flexibel horizontal oder vertikal zu skalieren – d. h. entweder durch Hinzufügen von RAM und CPUs oder durch das Beistellen von mehr Maschinen.

Die IT-Automatisierung funktioniert am besten mit einer skalierbaren Architektur. Im Umkehrschluss treibt die Automatisierung dazu an, das System skalierbarer zu gestalten.

Lokale Entwicklungsumgebungen virtualisieren

Jetzt haben wir uns viele Gedanken über Automatisierung und effektivere Nutzung der Entwicklungs- und Testumgebungen gemacht. Aber wo bleibt eigentlich das im Titel erwähnte Notebook?

Die Integration der Systemkomponenten sollte nicht erst der Build-Server in einer zentralen Umgebung verifizieren. Viel

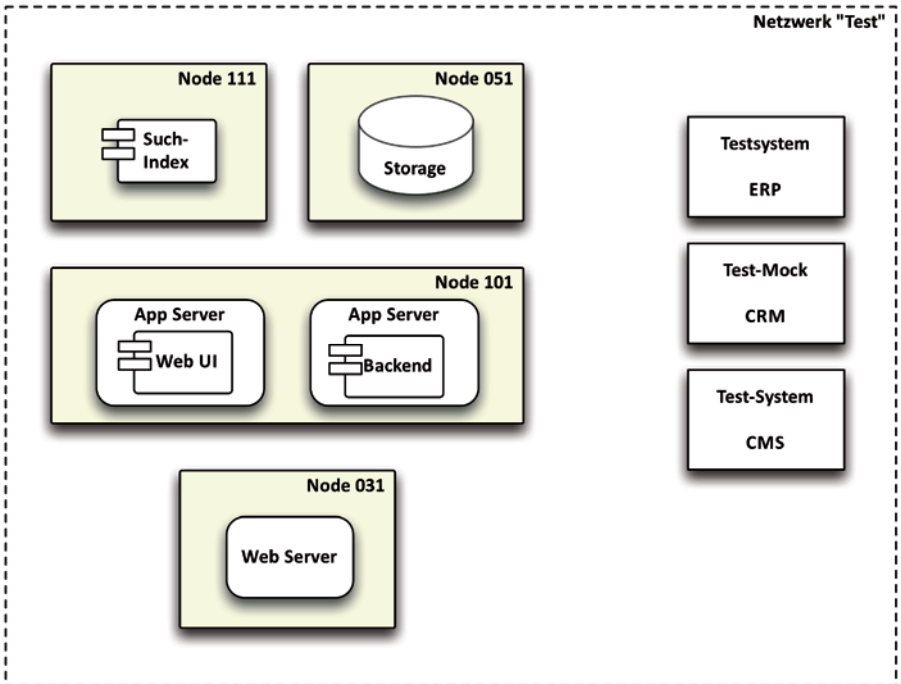

Abb. 2: Angepasste Verteilungsarchitektur für eine Testumgebung

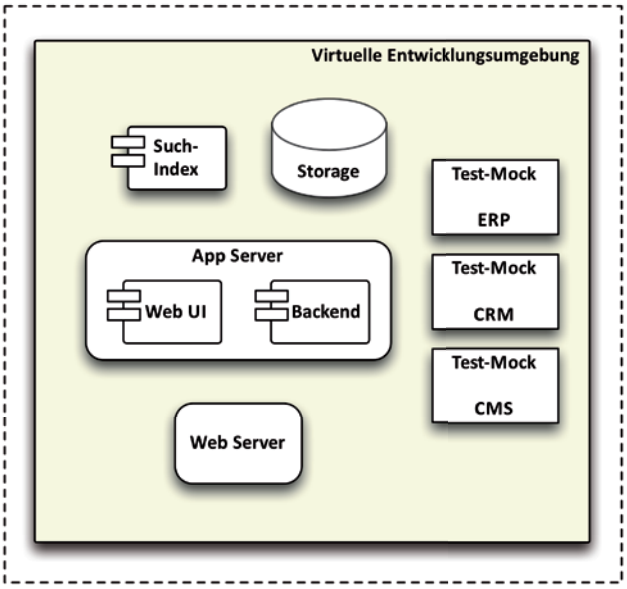

Abb. 3: Verteilungsarchitektur für das Notebook

besser wäre es, wenn die Entwickler schon direkt auf ihrem Notebook (oder auch Desktop) ausprobieren könnten, ob eine geänderte Systemkomponente weiter mit den anderen kommunizieren kann.

Grundsätzlich könnten die Entwickler an dieser Stelle die eingecheckten Automatisierungsskripte abrufen und lokal ausführen, um die Dienste und Infrastrukturkomponenten in der richtigen Version zu installieren und passend zu konfigurieren.

Für einfache Szenarien ist das auch ein plausibler Weg. Aber wenn ein Entwickler an mehreren Projekten beteiligt ist, will er nicht seine lokale Umgebung mit eventuell konkurrierenden Systemkonfigurationen zerstören. Richtig umständlich würde es zudem, wenn das Zielsystem ein UNIX oder Linux ist, der Entwicklungsrechner aber unter Windows läuft und dort jetzt *cron*, *iptables* und *syslog* eingesetzt und konfiguriert werden sollen.

Hier bietet es sich stattdessen an, für jedes Projekt eine virtuelle Maschine mit geeignetem Betriebssystem aufzusetzen und diese Umgebung dann automatisch zu konfigurieren.

Exakt die gleichen Systemkomponenten und die Netzwerkkonfiguration, die mit unseren Konfigurationswerkzeugen auf den zentralen Systemumgebungen ausgerollt werden, können dies also auch auf den Entwicklerrechnern. Komplexe Umsysteme, wie ERP und CRM, werden entweder auf einer zentralen Entwicklungsumgebung betrieben und über das Netzwerk aufgerufen oder lokal durch minimale Mock-Anwendungen simuliert.

Wenn für die Bereitstellung des Systems (*Provisioning*) Chef oder Puppet eingesetzt werden, drängt sich zum Einsatz auf den Entwicklerrechner ein Werkzeug wie Vagrant [Vag] geradezu auf. Vagrant erlaubt es, über die Kommandozeile virtuelle Maschinen (VirtualBox oder VMware) für die Entwicklung zu erstellen und mit Chef oder Puppet zu konfigurieren.

Alle Entwickler eines Projekts können durch die automatische Konfiguration

Box „Puppet, Chef und Vagrant“

„Eine Marionette, ein Koch und ein Landstreicher machen einen Ausflug ins Rechenzentrum...“ Die drei könnten auch die Protagonisten in einem Kneipenwitz sein, aber Puppet, Chef und Vagrant sind drei Werkzeuge, mit denen Infrastrukturen „programmiert“ und getestet werden können. Sie helfen dabei, verteilte Systeme auf unterschiedlich dimensionierte Umgebungen zu installieren, u. a. auch auf Entwicklerrechner.

Puppet und **Chef** sind Ruby-basierte Open Source-Werkzeuge für das Konfigurationsmanagement, die entweder in einem Client-Server-Modell betrieben oder direkt auf eine einzelne Maschine angewandt werden. Vom Funktionsumfang sind sie sehr ähnlich, unterscheiden sich aber bei der Herangehensweise, ihren Konzepten und Begriffen sowie der Syntax.

Mit beiden Werkzeugen wird die Zielkonfiguration für ein System deklarativ beschrieben. Es wird also nicht definiert, wie etwas getan wird, sondern was erwartet wird, z. B. dass ein User existiert, ein Paket installiert ist oder eine Konfigurationsdatei an einem bestimmten Ort liegt. Zur Strukturierung und Zusammenstellung von einzelnen Konfigurationselementen gibt es in Puppet *Klassen* und *Module*. Bei Chef werden die einzelnen Ressourcen in *Rezepten* zusammengefasst und diese wiederum in *Kochbüchern* gebündelt.

Der zunächst auffälligste Unterschied zwischen den beiden Werkzeugen ist die Sprache. Puppet nutzt eine ganz eigene Syntax, um die Konfiguration zu beschreiben, während Chef-Programme in einer internen Ruby DSL (*Domain Specific Language*, eine Programmiersprache für eine sehr spezielle Anwendungsdomäne) geschrieben werden und somit grundsätzlich alle Sprachmittel von Ruby zur Verfügung stehen. Wahrscheinlich ist das auch der Grund, warum Entwickler meistens Chef bevorzugen, während Systemadministratoren oftmals eher zu Puppet tendieren.

Mit **Vagrant** können virtuelle Entwicklungsumgebungen erstellt und konfiguriert werden. Es werden die Virtualisierungslösungen VirtualBox und VMware unterstützt, weitere sind geplant, z. B. KVM, stecken aber leider noch in den Kinderschuhen.

Zur Konfiguration der Umgebungen können u. a. Puppet oder Chef eingesetzt werden. Durch den Einsatz von Vagrant können projektspezifische Entwicklungsumgebungen schnell und einfach auf Entwicklungsrechnern aufgesetzt werden, unabhängig vom Wirts-Betriebssystem und sehr nah an der Produktivumgebung. Anstatt also z. B. lokal auf einem Windows-PC einen Apache Web-Server und einen JBoss Application-Server selber zu installieren und zu konfigurieren, wird eine Vagrant-Box mit einem passenden Linux erstellt und mit dem aktuellen Stand der Konfiguration aus der Versionsverwaltung provisioniert.

von virtuellen Maschinen also schnell und unkompliziert eine lokale Entwicklungs- und Testumgebung aufsetzen. Noch wichtiger ist aber, dass sie die Software lokal mit exakt derselben Infrastrukturkonfiguration testen, die später auch auf den zentralen Umgebungen verwendet wird (vgl. Abbildung 3).

Jetzt endlich kann die Integration wirklich *kontinuierlich* stattfinden. D. h., Entwickler können testen, ob das System trotz ihrer Anpassung noch funktioniert, bevor sie die Änderung einchecken. Die Feedback-Schleife wird noch einmal viel kürzer, als wenn die Entwickler erst eine halbe Stunde später eine E-Mail vom Build-Server erhalten.

Fazit

Zum Schluss noch einmal eine Zusammenfassung der wichtigsten Aspekte, um ein verteiltes System in Produktion und Testumgebungen flexibel skalieren zu können – wenn möglich bis runter zum Notebook:

1. Statt weniger großer Einheiten sollte es viele unabhängig voneinander verteilbare Komponenten geben.
2. Serverkonfiguration und Installation werden programmiert statt administriert.
3. Der Infrastrukturcode liegt in der Versionsverwaltung und ist Bestandteil eines jeden Releases.
4. Die Zuordnung von Systemkomponenten zu Zielknoten und Application-Servern muss pro Umgebung flexibel konfigurierbar sein.
5. Die einzelnen Komponenten müssen pro Umgebung parametrisiert werden können, bei Java EE-Systemen beispielsweise Speicherverwaltung und Threadpool.
6. Durch Einsatz von Virtualisierung und IT-Automatisierung werden schnelle Anpassungen an Servern und Verteilung der Software ermöglicht.
7. Die gesteigerte Komplexität der Verteilungsarchitektur muss durch eine reproduzierbare und testbare Automatisierung des Rollouts beherrschbar bleiben.

Natürlich bleibt es eine Herausforderung, ein komplexes, verteiltes Anwendungssystem auf einem Notebook zu betreiben. Aber mit einer geschickten Kombination aus Virtualisierung, Konfigurationsmanagement sowie Server- und Rollout-Automatisierung ist es möglich, die Systemkomponenten und ihre Kommunikation sehr realitätsnah lokal zu betreiben und zu testen. ■

Links und Ressourcen

[Puppet] http://puppetlabs.com/puppet/puppet-open-source
[Chef] http://www.opscode.com/chef/
[Vag] http://www.vagrantup.com/
[Sto86] Michael Stonebraker, „The Case for Shared Nothing Architecture“, 1986
[Fow] http://www.martinfowler.com/articles/continuousIntegration.html
[HaF10] Jez Humble, David Farley: „Continuous Delivery“, Addison-Wesley, 2010