Architekturdokumentation mit jQAssistant S. 56
Spring-Boot-Anwendungen mit CRaSH S. 90
Flux-Architektur mit JavaFX S. 76

JAVA Mag

Java™ magazin

Java | Architektur | Software-Innovation

ICH HABE FERTIG!

Das Leben nach dem Release

www.javamagazin.de

Ausgabe 1.2017

Deutschland € 9,80
Österreich € 10,80
Schweiz sFr 19,50
Luxemburg € 11,15

4 194586 709801 01

Stabile Produktion und Betrieb von Software in der Industrie

ICH HABE FERTIG? NICHT GANZ!

In Zeiten immer kleiner werdender und schneller evolutionierender Softwaresysteme rückt ein bislang eher stiefmütterlich betrachteter Arbeitsteil eines Entwicklungsteams mehr und mehr in den Fokus: der stabile Softwarebetrieb.

von Joachim Arrasz und Sascha Rüssel

Keine Software ist einfach fertig. Mit dem ersten Release, der Version 1.0, geht die eigentliche Arbeit erst los. Nun wird sich zeigen, ob das Entwicklungsteam auch den nachhaltigen Betrieb der Lösung im Auge hatte. Um Struktur und Priorität in das Thema zu bekommen, haben wir es in drei einzelne Artikel aufgespalten, die sich auf je eine Anforderung fokussieren (**Abb. 1**).

Los geht es mit der Herausforderung, die Weichen für die weitere Entwicklung in einem Projekt zu stellen, obwohl es aktiv in Produktion ist. Man muss Anpassungen an vorhandenen und neuen Features sowie Bugfixes planen. Diese Änderungen sollten die Stabilität der Anwendung weiter verbessern – oder zumindest erhalten. Um das zu erreichen, stellen wir zuerst einmal jene Weichen, mit denen aktuelle Versionen der Entwicklung schneller an den Kunden gebracht werden können. Das erreichen wir über Automatisierung möglichst vieler betriebsrelevanter Themen. Wir bekommen deutlich früher Feedback zu den Änderungen, und der Kunde kann so früher von den Änderungen profitieren. Darüber hinaus schaffen wir es über die Automatisierung, den Menschen als Fehlerfaktor herauszunehmen. Somit entwickeln wir einen stabilen Pfad, um unsere Entwicklungsleistung auch zu installieren. Grundvoraussetzung für einen solchen Pfad ist eine vereinheitlichte Konfiguration.

Es gibt verschiedene Werkzeuge, die uns dabei unterstützen, den Infrastrukturcode vom Businesscode zu trennen und die Konfiguration für verschiedene Umgebungen, Tests und Produktion, aber auch diverse Plattformen herzustellen. Diese Gedanken sollte man sich bereits zu Beginn einer Entwicklung machen.

Ein wichtiger Faktor ist die Laufzeit des Projekts. Lohnt es sich überhaupt, den Aufwand für eine solche Lösung zu betreiben? Ist die geplante Laufzeit weniger als zwei Jahre, kann man erfahrungsgemäß davon ausgehen, dass es real circa drei bis vier Jahre werden, in Ausnahmefällen sicherlich noch mehr. Nun stellt sich die Frage, ob der gewählte Technikstack in dieser Zeit mehr als ein Major-Version-Update erhalten wird oder nicht. Erfahrungsgemäß wird das beispielsweise in der Java-EE-Welt eher nicht der Fall sein, in anderen Umgebungen ist das unter Umständen eher zu erwarten. Bei Laufzeiten darüber hinaus, wie man sie im Besonderen in der Industrie und der Logistik vorfindet, ist es aus unserer Sicht unerlässlich. Um Investitionen, die man in die Entwicklung einer Lösung gesteckt hat, vor dem Verfall zu schützen, sollte man sich zuvor Gedanken machen, ob das überhaupt relevant und nötig ist. Um unsere These zu untermauern, wollen wir euch auf eine kurze Reise durch ein exemplarisches Projekt einladen.

Initiierungsphase: Los gehts!

Eine mittelständische Firma entscheidet sich, ihre Prozesse in Zukunft digital zu unterstützen. Excel alleine reicht aber nicht mehr aus. Die Firma beginnt nach und nach, alle ihre Prozesse aufzuzeichnen und zu dokumentieren. Hierbei entstehen die ersten Ideen, wie man die Prozesse weiter verbessern kann. Parallel werden die

Abb. 1: Phasen eines Softwareprojekts

bisher erkannten Anforderungen in Software umgesetzt und implementiert. Man kann die reine Entwicklungszeit schlecht mit einer realen Zeit untermauern, da sich diese natürlich stark durch Einflüsse wie den Betrieb oder das Management der Firma verändert. Man kann es aber durchaus prozentual quantifizieren. Unserer Erfahrung nach dauert diese Phase 10 bis 15 Prozent der Gesamtbetriebszeit einer Softwarelandschaft.

Stabilisierungsphase

Die Firma hat ihr entwickeltes Softwaresystem in Produktion. Die ersten Tage und Wochen verbringt man damit, Logfiles, die Last des Systems, das Speicherverhalten und andere Metriken einzupendeln. Wir haben dafür die Eckpunkte, Werte aus der betriebsvorbereitenden Phase, als Anhaltspunkte. Die Erfahrung sagt aber, dass unser System vom Endanwender in ande-

Was bisher geschah

In einer vorangegangenen Artikelserie im Java Magazin [1] haben wir gezeigt, wie man grundlegende Aspekte der Softwareentwicklung so gestaltet, dass die Software sauber und nachhaltig zu betreiben ist. Dabei haben wir aufgezeigt, wie wichtig unter anderem das so unbeliebte Logging ist. Konkret gingen wir auf Perfomance und Failover bzw. Loadbalancing-Problemstellungen ein und wie man diese Risiken minimieren kann. Wegzaubern geht leider immer noch nicht. Zu guter Letzt haben wir gezeigt, wie wir aus den statistischen Informationen Trends ableiten können, die uns letztlich helfen, eine stabile, produktive Umgebung bereitzustellen.

ren Zyklen und auch anderer Intensität genutzt wird. Daher wird es nötig, das echte Endnutzerverhalten in die Eckpunkte für die Metriken einfließen zu lassen. Darüber hinaus werden in dieser Phase meist viele kleinere Bugfixes und kosmetische Eingriffe in das System vorgenommen, die man natürlich auch vom Speichermanagement und der Last im Griff haben will, die das System erzeugt.

Betriebsphase: hallo Supportteam

Die Entwicklungsmannschaft zieht sich vom Projekt zurück, bekommt entweder neue Aufgaben und Systeme zur Entwicklung oder muss sich um weitere neue Features kümmern. Allmählich gerät das konkrete Wissen über die Entwicklung des Systems in Vergessenheit. Zu diesem Zeitpunkt muss der Betrieb an die Supportmannschaft übergehen. Das Wissen, das noch in den Köpfen der Entwickler steckt, muss nun auch dem Supportteam übermittelt werden. Das Vorgehen bei Problemen im nun nicht mehr ganz so neuen System verändert sich dahingehend, dass ein Supportteam die Pflege und die Hilfe bei Problemen im System übernimmt. Es entscheidet, wann es softwaretechnische Änderungen am System gibt. Nun wird sich herausstellen, wie gut sich die Metriken, Logfiles und auch Dokumentationen eignen, die während der ersten zwei Phasen entstanden sind, um schnell und effizient Änderungen oder Erweiterungen vornehmen zu können.

Bei der schnellen und effizienten Umsetzung der anfallenden Änderungen hilft Automatisierung ungemein. Darum wollen wir nun näher darauf eingehen und beleuchten, welche Rolle das Konfigurationsmanagement spielt und welche Tools eine automatisierte Infrastruktur unterstützen können.

Um den Verwaltungs- und Kommunikationsoverhead zwischen Entwicklern und Betreibern zu minimieren, lassen sich sowohl die Infrastruktur als auch die Konfiguration der Applikation selbst durch die gleiche Software verwalten, ein Konfigurationsmanagementsystem. Es gibt diverse Konfigurationsmanagementsysteme und Infrastrukturautomatisierungsansätze, einhergehend mit Tools, die beides unterstützen. Zu den ältesten und bekanntesten gehören Chef und Puppet, aber auch modernere wie Ansible wollen hier genannt werden.

Ziele von Automatisierung und Konfigurationsmanagement

- Erwartungen an die Infrastruktur manifestieren
- Nachvollziehbarkeit von Konfigurationsänderungen gewährleisten
- Testbarkeit der Infrastruktur erhöhen
- Manuelle Interaktion minimieren
- Schnelle und sichere Herstellung eines definierten Zustands ermöglichen

Die Vorteile von Puppet

Puppet [2] verfolgt einen deklarativen Ansatz. Man kümmert sich in Manifesten per einfacher DSL um die Beschreibung des gewünschten Zustands eines Systems. Man beschreibt geforderte Ressourcen, wie Dateien, Pakete, Benutzer und Services, deren gewünschter Zustand und Abhängigkeiten untereinander. Die Abhängigkeiten dieser Ressourcen bestimmen die Ausführungsreihenfolge, was zu Beginn durchaus etwas gewöhnungsbedürftig sein kann. Zusammen mit Facts, also Eigenschaften eines Computersystems, wie eingesetzte Hardware, Betriebssystem und dessen Spezifika, die vor jedem Lauf ermittelt werden, werden Manifeste serverseitig zu einem Katalog kompiliert. Dieser Katalog dient dem Puppet-Agent, der lokal auf dem zu provisionierenden System läuft, und dessen Providern – Hilfsprogramme auf dem Clientsystem – als Instruktion, um die eigentliche Arbeit zu tun.

Diese Herangehensweise bringt einen gewichtigen Vorteil mit sich. Wenn Ressourcen doppelt definiert sind, im schlimmsten Fall mit widersprüchlichen Parametern, fällt das bereits bei der Kompilierung auf dem Server auf. Es findet keine Änderung mit ungewissem Ausgang auf dem zu provisionierenden System statt. Die nächste Abstraktion von Manifesten sind Klassen, die mit den gängigen Klassen wenig gemein haben. So können sie sich beispielsweise der Vererbung bedienen und sind einmalig pro Katalog. Es kann mitunter schwierig werden, die Abhängigkeiten und Reihenfolge der Ressourcen über verschiedene Klassen hinweg zu gestalten. Da hilft oft nur ein Blick in den Abhängigkeitsgraphen. Indem man weiter zu Modulen abstrahiert und mit Designpatterns wie dem Rollen-und-Profilpattern arbeitet, versucht man dem entgegenzuwirken. So versieht man einfache Klassen mit zusätzlicher Semantik und erhebt sie zu Rollen, Profilen und logischen Ressourcen, deren Benennung an dieser Stelle leider nicht ganz eindeutig ist. Eine Rolle enthält dann mehrere Profile und diese wiederum die eigentliche Implementierung in wiederverwendbaren logischen Ressourcen.

Rezepte und Cookbooks in Chef

Ähnlich wie Puppet arbeitet Chef ebenfalls deklarativ, jedoch mit unterschiedlichen Konzepten. Was Puppet als Klassen abbildet, wird von Chef als Rezept betitelt (recipe). Gesammelt werden sie in Cookbooks. Im Gegensatz zu Puppets DSL besteht die von Chef aus absolut validem Ruby-Code. Das bringt zwar deutlich mehr Flexibilität, aber auch Komplexität mit sich. Außerdem lässt das vermuten, dass die Hauptzielgruppe von Chef eher die geneigten Entwickler als die klassischen Systemadministratoren sind. Aber hier verwischen sich die Grenzen unter anderem durch solche Automatisierungs- und Konfigurationsmanagementwerkzeuge logischerweise zunehmend. Chef basiert ebenfalls auf Ressourcen und Providern, die die nötigen Schritte abbilden, um eine Ressource in einen gewünschten Zustand zu bringen. Chef kennt aber keine Ausführungsreihenfolge per

Abhängigkeitsdefinition, sondern führt die Rezepte in der Reihenfolge aus, in der sie im Cookbook stehen. So bietet es wiederum mehr Spielraum, um definierte Zustände innerhalb eines Laufs herzustellen, wie einen Container zu stoppen, eine Aktion durchzuführen und anschließend wieder zu starten. Es werden ebenso die diversen Attribute eines Systems berücksichtigt – Entsprechung zu Puppet-Facts –, die durch Ohai ermittelt werden. Durch den eher entwicklungsgetriebenen Ansatz und die etwas spätere Erscheinung sind einige der Konzepte, die bei Puppet erst durch Konventionen realisiert werden, fest in der unterhaltsamen und dennoch präzisen Terminologie von Chef verankert.

Chef verfügt zudem über ein sehr mächtiges Tool namens Knife. Es dient der Interaktion zwischen der Workstation und dem Chef-Server. Knife verfügt über Mechanismen, um Cookbooks von der Workstation auf den Server hochzuladen und zu inspizieren. Darüber hinaus kann man damit Nodes und Environments, Parameter und Rezepte anlegen, verwalten, suchen und ansehen. Auch den Chef-Client auf einem noch unberührten System über Knife zu installieren, ist mit Knife Bootstrap ein Kinderspiel. Ein solch einheitliches und umfangreiches Tooling scheint für Puppet in weiter Ferne.

Ansible: der Youngster

Ansible, der Youngster in dieser Runde, unterscheidet sich deutlich von Chef und Puppet. Das betrifft vor allem die Art der Ausführung. Außerdem arbeitet hier Python unter der Haube. Es gibt keinen Agenten auf den Zielsystemen. Ansible nutzt die bewährte Kommunikation über SSH. Dieses Set-up ist auch am ehesten skalierbar, denn es gibt schlicht keine zu skalierenden Komponenten. Von Haus aus ist auch kein zentraler Server vorgesehen. Diese Funktionalität kann man durch den kostenpflichtigen Ansible Tower erwerben, der das Problem löst, dass der Entwickler und seine Workstation zum Zeitpunkt der Ausführung Zugriff und Verbindung auf alle zu provisionierenden Systeme haben muss. So genannte Playbooks werden in YAML verfasst und sehen auf den ersten Blick aus, als würde auch Ansible deklarativ arbeiten. Dieser Eindruck täuscht, denn in Wirklichkeit funktioniert es imperativ. Man verfasst Tasks, die wie bei Chef in der Reihenfolge ausgeführt werden, wie sie im Playbook stehen. Dabei muss Idempotenz geschaffen werden, wie bei anderen Automatisierungstools. Das heißt, auch bei mehrmaliger Ausführung eines Tasks oder Playbooks muss immer noch dasselbe Ergebnis entstehen. Dabei wird man stellenweise nicht so gut unterstützt wie bei Puppet, da dort die Ressourcen selbst für Idempotenz sorgen. Eine Ausnahme bildet die generische Exec-Ressource, die eine Bedingung erfordert. Die Ausführung eines Playbooks ist nicht an einen Node gekoppelt, sondern kann über mehrere Systeme hinweg auch komplexere Orchestrierungsaufgaben erledigen. Man ist auch nicht unbedingt an das vorgegebene Push-Prinzip gebunden. So kann man die Architektur mit Ansible Pull quasi umdrehen. Das bedeutet aber einen wesentlich höheren Verwaltungsaufwand, da die Playbooks und Tasks auf die Maschinen gelangen müssen, auf denen sie benötigt werden. Außerdem muss die regelmäßige Ausführung gewährleistet werden, z. B. durch Cron. Das gilt im Übrigen auch für den masterlosen Betrieb von Puppet oder Chef. Bei Ansible können ebenfalls Parameter von einem generischen Playbook getrennt gehalten werden. Ein Defizit dabei ist allerdings die manchmal undurchsichtige Zuweisung und Überschreibung von Parametern, die aus allen möglichen Dateien kommen oder überschrieben werden können. Dass man Variablen explizit in einem Playbook zu Anfang lädt und möglichst nicht oder nur einmal überschreibt, hat man per Konvention adressiert.

> In der Betriebsphase gerät allmählich das konkrete Wissen über die Entwicklung des Systems in Vergessenheit.

Im Gegenzug bietet Puppet mit Hiera ein einfaches und mächtiges System, um eine Hierarchie von Parametern nach eigenem Gusto herzustellen. Chef verfügt auch über ein ausgeklügeltes, aber etwas undurchsichtigeres Design, um Parameter per Priorität zu überschreiben. Dabei gibt es auch eine Gewichtung der Stellen, an denen man das Überschreiben veranlasst. So oder so muss man sich klar werden, welche Parameter man justieren möchte und was auf die Werte der Parameter Einfluss hat, wie die logische Umgebung (z. B. Test oder Produktion), die physikalische Umgebung oder gar fachliche Mandantenanforderungen.

Bei allen Unterschieden gibt es auch viele Gemeinsamkeiten der Tools. So ist das Templating bei allen dreien vorhanden und bietet kaum funktionale Unterschiede. Man achtet bei allen auf Wiederverwendbarkeit, auch durch Dritte. Jede Menge Puppet-Module, Ansible-Rollen sowie Chef Cookbooks sind bereits vorgefertigt und jeweils in der Puppet Forge, dem Chef Supermarket oder der Ansible Galaxy zu finden. Wobei natürlich nicht immer alle oder spezielle Bedürfnisse befriedigt werden. In diesem Fall sollte man, bevor man etwas Eigenes entwickelt, in Erwägung ziehen, bereits Vorhandenes zu erweitern, den Bedürfnissen anzupassen und fairerweise und im Sinne der DevOps-Bewegung in die Codebasis zurückfließen zu lassen. Welches der Konfigurationsmanagementsysteme man letztendlich benutzen möchte, hängt natürlich davon ab, ob es bereits Erfahrungen mit dem ein oder anderen Tool im Team gibt, oder wie sehr man die Konzepte eines Tools als sinnig unterschreiben kann. Dabei sind auch Mischformen denkbar, wie mit Puppet den Zustand der Systeme zu gestalten und für

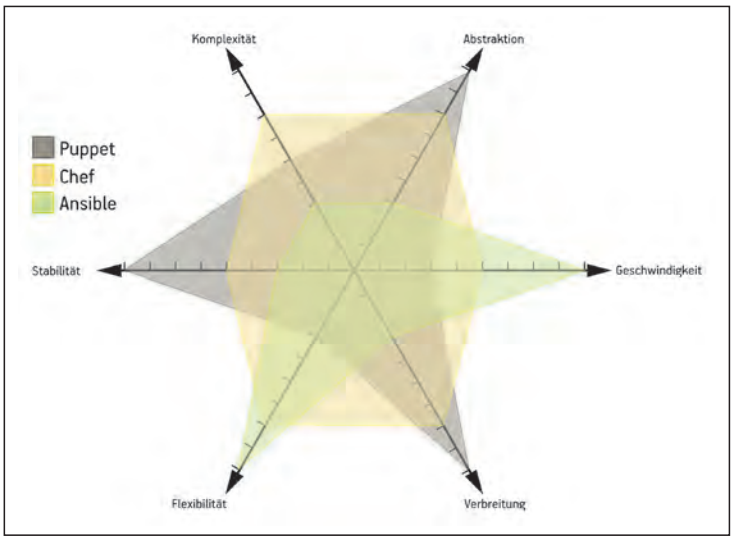

Abb. 2: Die Vorteile der Konfigurationsmanagementsysteme Puppet, Chef und Ansible

komplexere Orchestrierungs- und Deployment-Aufgaben zusätzlich auf Ansible zurückzugreifen. In jedem Fall wird ein solches System hilfreich dabei sein, den Betrieb zu stabilisieren und die übergreifende Zusammenarbeit zwischen Betreibern und Entwicklern zu fördern (**Abb. 2**).

Auch die Infrastruktur ändert sich

Das gemeinsame Gestalten der durch Code abstrahierten Infrastruktur führt dazu, dass der geneigte Entwickler anerkennt, dass sich ein Projekt eben nicht nur softwaretechnisch, sondern auch infrastrukturell oder im dazugehörigen Datenbankschema weiterentwickeln kann.

Während der ersten Wochen des Betriebs einer Lösung ist es sehr wahrscheinlich, dass die Änderungsrate innerhalb der Infrastruktur, die dann zumindest teilweise automatisiert ist, und ihrer Beschreibungen höher als die innerhalb des Sourcecodes der Lösung ist. Dadurch kann das Betriebsteam auch agil auf Änderungen reagieren, da der Austausch oder die Anpassung einer Infrastruktur nicht mehr mit Ausfällen oder Vergleichbarem einher kommt. Das trifft natürlich nicht nur für Infrastrukturbeschreibungen zu, sondern auch auf Werkzeuge, die andere Dinge auf dieselbe oder ähnliche Art automatisieren, beispielsweise Flyway [3] oder Liquibase [4] im Datenbankumfeld. Wobei man sich aus Entwicklersicht natürlich fragen muss, ob nicht auch die Datenbank zur Infrastruktur gehört. Natürlich ist es sinnvoll, den Stand einer Datenbankmodellierung zum Stand der Software auch zu versionieren und zu verwalten. Zu guter Letzt ist eine solche Lösung, wenn sie denn wirklich funktioniert und alle Systeme (Dev, Test, Stage, Produktion) auf diese Art aufgesetzt wurden, ein sehr effizienter Start für neue Mitarbeiter. Und wenn man sich vorstellt, dass man drei Jahre später an einem Projekt weiterarbeiten darf, aber keine Ahnung mehr hat, wie die Produktion eigentlich aussieht, sind die Beschreibungen dann nicht nur sehr effizient nutzbar, um eine solche Umgebung wieder zu restaurieren oder neu zu erzeugen, sie helfen auch als Dokumentation.

Infrastrukturdateien ablegen

1. Sie werden innerhalb des Softwarequellcodes mitverwaltet
2. Sie werden in einem dedizierten VCS bzw. Repository verwaltet

Beide Vorgehensweisen haben Vor- und Nachteile. Infrastruktur- und Softwarequellcode in einem gemeinsamen Repository zu verwalten, führt zu einer sehr engen Zusammenarbeit von Entwicklern und Betreibern. Es macht es aber auch sehr komplex, den für das Konfigurationsmanagement benötigten Code einzusammeln und zu integrieren. Auch muss man bei Konfigurationsänderungen ein neues Softwarerelease erstellen. Ein extra Repository für Infrastruktur hingegen stellt eine Art Barriere dar. Man sollte dann darauf achten, dass die Mitarbeit nicht durch fehlende Rechte oder mangelnde Sichtbarkeit behindert wird.

Fazit

Ein ordentlich adressiertes Konfigurationsmanagement gehört unbedingt in das Set-up einer langlaufenden Lösung und deren Betrieb. Ab wann sich der Aufwand für ein Konfigurationsmanagement lohnt, muss jedes Team für sich selbst entscheiden. Wir haben versucht, einen Lösungs- oder besser ausgedrückt einen Annäherungsansatz aufzuzeigen, damit man zu Beginn nicht komplett mit leeren Händen dasteht. Im kommenden Teil der Serie werden wir das Konfigurationsmanagement noch weiter strapazieren und mit einer weiteren Aufgabe belasten: vorbereitende Dinge für die Themen Management und Monitoring. In diesem Sinne ...

Joachim Arrasz ist als Software- und Systemarchitekt in Karlsruhe bei der synyx GmbH & Co. KG als Leiter der CodeClinic tätig. Darüber hinaus twittert und bloggt er gerne.

@arrasz **http://blog.synyx.de/**

Sascha Rüssel ist als Systemadministrator bei der synyx GmbH & Co. KG in Karlsruhe tätig, regelmäßig auf Konferenzen wie der FrOSCon, FOSDEM und OSDC sowie auf Twitter anzutreffen.

@da_zivi

Links & Literatur

[1] Arrasz, Joachim; Rüssel, Sascha: „Production-ready“, in: Java Magazin 11.2014–1.2015

[2] Puppet: https://puppet.com

[3] Flyway: https://flywaydb.org

[4] Liquibase: http://www.liquibase.org