KAPITEL 1

Was ist Infrastructure as Code?

Arbeiten Sie in einem Team, das IT-Infrastruktur erstellt und betreibt, sollten Ihnen Automatisierungstechniken für Cloud und Infrastruktur dabei helfen, in weniger Zeit zuverlässiger mehr Wert bieten zu können. Aber in der Praxis geht es vor allem darum, die stetig zunehmende Größe, Komplexität und Diversität der verschiedenen Elemente im Griff zu behalten.

Diese Technologien sind besonders relevant, wenn Organisationen digital werden. Wenn Business-Menschen »Digital« sagen, meinen sie damit, dass Softwaresysteme für das, was die Organisation tut, von zentraler Bedeutung sind.[1] Der Wechsel ins Digitale verstärkt den Druck auf Sie, mehr Aufgaben schneller zu erledigen. Sie müssen zusätzliche Services hinzufügen und betreuen. Mehr Business-Aktivitäten. Mehr Mitarbeiterinnen und Mitarbeiter. Mehr Kundinnen und Kunden, Lieferpartner und andere Stakeholder.

Cloud- und Automatisierungs-Tools helfen dabei, es viel leichter zu machen, Infrastruktur hinzuzufügen und anzupassen. Aber viele Teams haben Probleme damit, ausreichend Zeit dafür zu finden, mit der schon vorhandenen Infrastruktur Schritt zu halten. Da ist es nicht sehr hilfreich, das Erstellen von noch mehr Infrastruktur einfacher zu machen. Einer meiner Kunden sagte mir: »Durch Cloud wurden die Barrieren eingerissen, die unseren Reifenbrand unter Kontrolle behielten.«[2]

Viele haben auf die Drohung eines ausufernden Chaos durch ein Anziehen ihres Änderungsmanagement-Prozesses reagiert. Sie haben die Hoffnung, dass sich das Chaos durch Begrenzen und Kontrollieren der Änderungen vermeiden lässt. Also legen sie die Cloud in Ketten.

1 Das steht im Gegensatz zu dem, was die gleichen Personen ein paar Jahre zuvor gesagt haben, nämlich dass Software »nicht Teil unseres Kerngeschäfts ist«. Nachdem sie diesem Ratschlag gefolgt sind und ihre IT outgesourced haben, stellten die Organisationen fest, dass diejenigen an ihnen vorbeizogen, die bessere Software als Wettbewerbsvorteil und nicht als Kostenersparnis ansahen.

2 Laut (englischsprachiger) Wikipedia (*https://oreil.ly/IkDu_*) kann es einen *Reifenbrand* in zwei Ausprägungen geben: »Schnell brennende Ereignisse, die fast sofort dazu führen, dass das Feuer außer Kontrolle gerät, und langsam brennende Pyrolyse, die sich mehr als ein Jahrzehnt hinziehen kann.«

Damit gibt es zwei Probleme. Das eine ist, dass so die Vorteile der Cloud-Technologien verloren gehen, das andere, dass die Benutzerinnen und Benutzer die Vorteile der Cloud-Technologie auch haben *wollen*. So ist man bemüht, die Personen zu umgehen, die versuchen, das Chaos im Griff zu behalten. Im schlimmsten Fall wird das Risikomanagement vollständig ignoriert, und man entscheidet für sich, dass das in der schönen neuen Cloud-Welt nicht notwendig sei. Sie setzen auf Cowboy-IT, was zu verschiedenen Problemen führt.[1]

Die Prämisse dieses Buchs ist, dass Sie Cloud- und Automatisierungs-Technologie nutzen können, um Änderungen einfach, sicher, schnell und verantwortungsbewusst durchführen zu können. Diese Vorteile entstehen nicht automatisch durch den Einsatz von Automatisierungs-Werkzeugen oder Cloud-Plattformen. Sie hängen davon ab, wie Sie diese Technologie verwenden.

DevOps und Infrastructure as Code

DevOps ist eine Bewegung, die Hindernisse abbaut und Reibungen zwischen der Silo-Entwicklung, Operations und anderen Stakeholdern reduziert, die am Planen, Bauen und Ausführen von Software beteiligt sind. Auch wenn der Technologie-Anteil der sichtbarste und in mancherlei Hinsicht auch einfachste Aspekt von DevOps ist, haben Kultur, Personen und Prozesse den größten Einfluss auf den Fluss und die Effektivität. Technologie und Entwicklungspraktiken wie Infrastructure as Code sollten zum Einsatz kommen, um die Aktivitäten zu unterstützen, die Gräben überbrücken und die Zusammenarbeit verbessern.

In diesem Kapitel erkläre ich, dass moderne, dynamische Infrastruktur eine Mentalität aus dem »Cloud-Zeitalter« erfordert. Die Mentalität unterscheidet sich grundlegend vom klassischen »Eisenzeit«-Ansatz, den wir bei statischen Prä-Cloud-Systemen genutzt haben. Ich definiere drei zentrale Praktiken für das Implementieren von Infrastructure as Code: Definiere alles als Code, teste und liefere alles fortlaufend während des Entwickelns aus und baue das System aus kleinen, lose gekoppelten Elementen auf.

Ich beschreibe zudem in diesem Kapitel die Gründe für den »Cloud-Zeitalter«-Ansatz zur Infrastruktur. Dieser löst sich von der fälschlich angenommenen Gegensätzlichkeit von Geschwindigkeit und Qualität, die man gegeneinander abwägen müsse. Stattdessen nutzen wir die Geschwindigkeit als eine Möglichkeit, die Qualität zu verbessern, und die Qualität, um mit hoher Geschwindigkeit auszuliefern.

1 Mit »Cowboy-IT« meine ich Leute, die IT-Systeme ohne eine Vorgehensweise oder Überlegung hinsichtlich der Konsequenzen in der Zukunft bauen. Meist wählen Personen, die noch nie Produktivsysteme betreut haben, den einfachsten Weg, um etwas zum Laufen zu bekommen, ohne auf Sicherheit, Wartbarkeit, Performance und andere Operability-Aspekte zu achten.

Aus der Eisenzeit in das Cloud-Zeitalter

Technologie des Cloud-Zeitalters ermöglicht ein schnelleres Provisionieren und Ändern von Infrastruktur, als dies mit den klassischen Technologien aus der Eisenzeit möglich wäre (Tabelle 1-1).

Tabelle 1-1: Technologische Änderungen im Cloud-Zeitalter

Eisenzeit	Cloud-Zeitalter
Physische Hardware	Virtualisierte Ressourcen
Provisionieren dauert Wochen	Provisionieren dauert Minuten
Manuelle Prozesse	Automatisierte Prozesse

Aber diese Technologien machen es nicht notwendigerweise einfacher, Ihre Systeme zu managen und wachsen zu lassen. Überführen Sie ein System mit technischen Schulden (*https://oreil.ly/3AqHB*) in eine schrankenlose Cloud-Infrastruktur, beschleunigen Sie nur das Chaos.

Vielleicht konnten Sie auf bewährte, klassische Governance-Modelle zurückgreifen, um die Geschwindigkeit und das Chaos zu kontrollieren, das neuere Technologien mit sich bringen. Ein umfassendes, vorher ausgearbeitetes Design, rigorose Change Reviews und strikt getrennte Zuständigkeiten werden schon für Ordnung sorgen!

Aber leider sind diese Modelle für die Eisenzeit optimiert, in der Änderungen langsam und teuer sind. Sie sorgen für zusätzlichen Aufwand im Vorhinein mit der Hoffnung, später den Zeitaufwand für die Änderung zu verringern. Das ist durchaus sinnvoll, wenn es später teuer und langsam ist, Änderungen vorzunehmen. Aber durch die Cloud werden Änderungen schnell und günstig. Sie sollten diese Geschwindigkeit zu Ihrem Vorteil nutzen, um kontinuierlich zu lernen und Ihr System zu verbessern. Arbeiten Sie weiter wie in der Eisenzeit, sind Ihr Lernen und Ihre Verbesserungen massiv eingeschränkt.

Statt also langsame Prozesse aus der Eisenzeit auf schnellere Technologie des Cloud-Zeitalters anzuwenden, sollten Sie eine neue Mentalität übernehmen. Nutzen Sie eine schnellere Technologie, um Risiken zu verringern und die Qualität zu verbessern. Das erfordert einen grundlegend anderen Ansatz und neue Wege, über Änderungen und Risiken nachzudenken (Tabelle 1-2).

Tabelle 1-2: Arbeitsweisen im Cloud-Zeitalter

Eisenzeit	Cloud-Zeitalter
Änderungskosten sind hoch	Änderungskosten sind niedrig
Änderungen stehen für Fehler (Änderungen müssen »gemanagt« oder »kontrolliert« werden)	Änderungen stehen für Lernen und Verbesserungen

Tabelle 1-2: Arbeitsweisen im Cloud-Zeitalter (Fortsetzung)

Eisenzeit	Cloud-Zeitalter
Gelegenheit zu Fehlern verringern	Geschwindigkeit von Verbesserungen maximieren
In großen Batches ausliefern, am Ende testen	Kleine Änderungen ausliefern, fortlaufend testen
Lange Release-Zyklen	Kurze Release-Zyklen
Monolithische Architektur (wenige und größere Komponenten)	Microservices-Architektur (viele und kleinere Komponenten)
Konfiguration über GUI oder direkt an der Hardware	Konfiguration als Code

Infrastructure as Code

Infrastructure as Code ist ein Ansatz für die Infrastruktur-Automatisierung, der auf Praktiken aus der Softwareentwicklung basiert. Er baut auf konsistenten, wiederholbaren Routinen zum Provisionieren und zur Änderung von Systemen und deren Konfiguration auf. Sie nehmen Änderungen am Code vor und nutzen dann Automation, um diese Änderungen zu testen und auf Ihre Systeme anzuwenden.

In diesem Buch erläutere ich, wie Sie agile Entwicklungspraktiken wie Test-driven Development (TDD), Continuous Integration (CI) und Continuous Delivery (CD) nutzen können, um das Ändern von Infrastruktur schnell und sicher zu gestalten. Ich beschreibe zudem, wie ein modernes Softwaredesign für resiliente, gut gewartete Infrastruktur sorgen kann. Diese Praktiken und Design-Ansätze unterstützen sich gegenseitig. Gut designte Infrastruktur lässt sich leichter testen und bereitstellen. Automatisiertes Testen und Bereitstellen führt zu einfacherem und sauberem Design.

Vorteile von Infrastructure as Code

Zusammengefasst kann man sagen, dass Organisationen, die Infrastructure as Code nutzen, um dynamische Infrastruktur zu verwalten, unter anderem auf Folgendes hoffen:

- Mit der IT-Infrastruktur ein schnelles Bereitstellen von Werten ermöglichen.
- Aufwand und Risiken von Änderungen an der Infrastruktur reduzieren.
- Anwenderinnen und Anwender der Infrastruktur dazu befähigen, die notwendigen Ressourcen dann zu bekommen, wenn sie sie brauchen.
- Gemeinsame Werkzeuge für Entwicklung, Operations und andere Stakeholder bereitstellen.
- Systeme aufbauen, die zuverlässig, sicher und kostengünstig sind.
- Steuerelemente für Governance, Sicherheit und Compliance sichtbar machen.
- Die Geschwindigkeit verbessern, mit der Fehler gefunden und gelöst werden.

Infrastructure as Code nutzen, um für Änderungen zu optimieren

Angesichts dessen, dass Änderungen das größte Risiko für ein Produktivsystem sind, sich kontinuierliche Änderungen nicht vermeiden lassen und ein System nur durch Änderungen verbesserbar ist, ist es sinnvoll, Ihre Fähigkeit zu optimieren, Änderungen sowohl schnell *als auch* zuverlässig zu machen.[1] Die Forschungsergebnisse im *Accelerated State of DevOps Report* unterstützen diese Ansicht. Häufige und zuverlässige Änderungen korrelieren mit dem Erfolg von Organisationen.[2]

Es gibt eine Reihe von Einwänden, die ich zu hören bekomme, wenn ich einem Team empfehle, zum Optimieren von Änderungen Automation einzuführen. Ich glaube, diese entstehen aus Missverständnissen, wie Sie Automation verwenden können und sollten.

Einwand »Wir haben gar nicht so häufig Änderungen, sodass sich Automation nicht lohnt«

Wir möchten gerne glauben, dass wir ein System bauen, und dann ist es »fertig«. Bei einer solchen Sichtweise nehmen wir nicht viele Änderungen vor, daher ist das Automatisieren von Änderungen nur Zeitverschwendung.

In der Realität gibt es nur an sehr wenigen Systemen keine Veränderungen mehr – zumindest nicht, bevor sie ausgemustert werden. Manche gehen davon aus, dass die aktuelle Menge an Änderungen nur vorübergehend ist. Andere schaffen aufwendige Prozesse zum Änderungsmanagement, um Personen davon abzuhalten, nach Änderungen zu fragen. Aber diejenigen reden es sich schön. Die meisten Teams, die aktiv genutzte Systeme betreuen, haben es mit einem kontinuierlichen Strom von Änderungen zu tun.

Denken Sie nur an die folgenden Beispiele für Änderungen an der Infrastruktur:

- Ein wichtiges neues Anwendungsfeature erfordert das Hinzufügen einer neuen Datenbank.
- Für ein neues Anwendungsfeature muss der Anwendungsserver aktualisiert werden.
- Die App wird häufiger genutzt als erwartet. Sie brauchen mehr Server, neue Cluster und zusätzliche Networking- und Storage-Kapazitäten.

1 Gene Kim, George Spafford und Kevin Behr schreiben in *The Visible Ops Handbook* (IT Process Institute), dass Änderungen für 80 Prozent der ungeplanten Ausfälle verantwortlich sind.

2 Die Berichte aus der *Accelerate*-Forschung finden sich im jährlich erscheinenden *State of DevOps Report* (*https://oreil.ly/0Q3FE*) und im Buch *Accelerate* von Dr. Nicole Forsgren, Jez Humble und Gene Kim (IT Revolution Press).

- Beim Performance-Profiling zeigt sich, dass die aktuelle Deployment-Architektur für die Anwendung die Performance einschränkt. Sie müssen die Anwendungen auf mehrere andere Anwendungsserver neu deployen. Dazu sind Änderungen am Clustering und an der Netzwerk-Architektur erforderlich.
- Es gibt eine neu bekannt gewordene Sicherheitslücke in Systempaketen für Ihr Betriebssystem. Sie müssen dutzende Produktivserver patchen.
- Sie müssen Server aktualisieren, auf denen veraltete Versionen des Betriebssystems und zentraler Pakete laufen.
- Auf Ihren Webservern gibt es immer wieder Aussetzer. Sie müssen eine Reihe von Konfigurationsänderungen vornehmen, um das Problem untersuchen zu können. Dann müssen Sie ein Modul aktualisieren, um das Problem zu lösen.
- Sie finden eine Konfigurationsänderung, die die Performance Ihrer Datenbank verbessert.

Eine zentrale Wahrheit des Cloud-Zeitalters ist: *Stabilität entsteht durch Änderungen.*

Ungepatchte Systeme sind nicht stabil – sie sind angreifbar. Können Sie Probleme nicht beheben, sobald Sie sie finden, ist Ihr System nicht stabil. Können Sie nach einem Ausfall nicht schnell wieder auf die Füße kommen, ist Ihr System nicht stabil. Wenn für Ihre Änderungen eine deutliche Downtime erforderlich ist, ist Ihr System nicht stabil. Wenn Änderungen immer wieder fehlschlagen, ist Ihr System nicht stabil.

Einwand »Wir sollten erst bauen und danach automatisieren«

Wenn Sie mit Infrastructure as Code beginnen, haben Sie eine steile Lernkurve vor sich. Das Aufsetzen der Tools, Services und Arbeitsweisen zum Automatisieren der Infrastruktur-Bereitstellung erfordert viel Arbeit, insbesondere, wenn Sie sich gleichzeitig auch noch in eine neue Infrastruktur-Plattform einarbeiten. Der Wert dieser Arbeit wird erst dann sichtbar, wenn Sie beginnen, Services damit zu bauen und zu deployen. Und auch dann wird er eventuell denen nicht deutlich werden, die nicht direkt mit der Infrastruktur arbeiten.

Stakeholder drängen Infrastruktur-Teams oft dazu, schnell und mal eben manuell neue in der Cloud gehostete Systeme zu bauen und sich erst später um das Automatisieren zu kümmern.

Es gibt drei Gründe dafür, warum das eine schlechte Idee ist:

- Durch die Automation sollte das Bereitstellen schneller gehen, auch für neue Elemente. Implementieren Sie die Automation erst, nachdem ein Großteil der Arbeit erledigt ist, opfern Sie viele der Vorteile.
- Automation erleichtert das Schreiben automatisierter Tests für das, was Sie bauen. Und sie macht es einfacher, etwas schnell zu korrigieren und neu zu bauen, wenn Sie Probleme finden. Wenn Sie dies als Teil des Build-Prozesses tun, hilft es Ihnen dabei, eine bessere Infrastruktur zu bauen.

- Das Automatisieren eines bestehenden Systems ist sehr schwer. Automation ist Teil des Designs und der Implementierung eines Systems. Um ein System, das ohne Automation aufgebaut wurde, um diese zu ergänzen, müssen Sie das Design und die Implementierung des Systems deutlich anpassen. Das gilt auch für das automatisierte Testen und Deployen.

Cloud-Infrastruktur, die ohne Automation aufgebaut wurde, müssen Sie schneller abschreiben, als Sie denken. Die Kosten für das manuelle Warten und Beheben von Fehlern im System können schnell deutlich wachsen. Ist der Service, der darauf läuft, erfolgreich, werden Sie die Stakeholder dazu drängen, ihn zu erweitern und neue Features hinzuzufügen, statt innezuhalten, um ihn neu zu bauen.

Das Gleiche gilt, wenn Sie ein System als Experiment bauen. Haben Sie einen Proof of Concept zum Laufen gebracht, wird es den Druck geben, sich mit dem nächsten Thema zu beschäftigen, statt zurückzukehren und das System richtig neu zu bauen. Und eigentlich sollte Automation auch Teil des Experiments sein. Wollen Sie Automation zum Managen Ihrer Infrastruktur einsetzen, müssen Sie verstehen, wie das funktionieren wird, daher sollte es auch Teil Ihres Proof of Concept sein.

Die Lösung besteht darin, Ihr System inkrementell aufzubauen und die Automation direkt mit zu berücksichtigen. Stellen Sie sicher, dass Sie Werte bieten, während Sie gleichzeitig für die Möglichkeit sorgen, das kontinuierlich zu tun.

Einwand »Wir müssen zwischen Geschwindigkeit und Qualität entscheiden«

Es ist ganz natürlich, davon auszugehen, dass Sie nur dann schnell vorankommen können, wenn Sie Abstriche bei der Qualität machen, und dass Sie nur dann Qualität erhalten, wenn Sie sich langsam bewegen. Sie sehen das vielleicht als Kontinuum (siehe Abbildung 1-1).

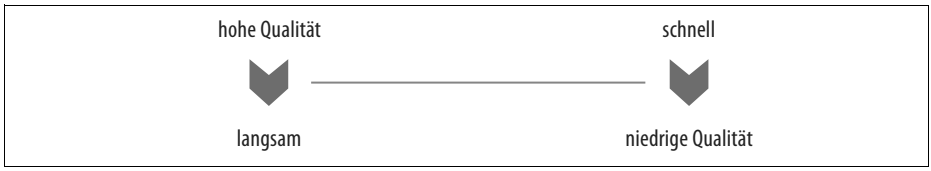

Abbildung 1-1: Die Idee, dass sich Geschwindigkeit und Qualität an entgegengesetzten Enden eines Spektrums befinden, ist eine fälschlich angenommene Gegensätzlichkeit.

Die Accelerate-Forschungsdaten, die ich weiter oben erwähnt habe (siehe »Infrastructure as Code nutzen, um für Änderungen zu optimieren« auf Seite 35), zeigen aber etwas anderes:

> Diese Ergebnisse zeigen, dass es keinen Kompromiss zwischen dem Verbessern der Performance und dem Erreichen einer besseren Stabilität und Qualität gibt. Stattdessen sind High-Performer bei all diesen Messwerten besser. Das ist genau das, was die Agile- und Lean-Bewegungen predigen, aber ein Glaubenssatz in unserer Branche basiert immer noch auf der falschen Annahme, dass man ein schnelles Vorankommen gegen andere Performance-Ziele abwägen muss, statt alles zu unterstützen und sich gegenseitig verstärken zu lassen.
>
> *– Nicole Forsgren, PhD, Accelerate*

Kurz gesagt können Organisationen nicht wählen, ob sie entweder Veränderungen oder Stabilität gut hinbekommen. Sie tendieren dazu, entweder in beidem gut oder in beidem schlecht zu sein.

Ich ziehe es vor, Qualität und Geschwindigkeit als Quadrant statt als Kontinuum zu betrachten,[1] wie in Abbildung 1-2 zu sehen ist.

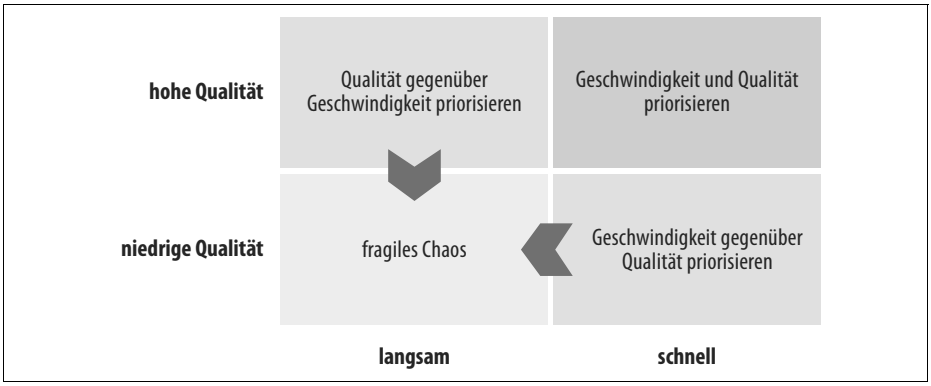

Abbildung 1-2: Geschwindigkeit und Qualität auf Quadranten abgebildet

Dieses Quadrantenmodell zeigt, warum es auf jeden Fall zu Mittelmäßigkeit führen wird, wenn man versucht, zwischen Geschwindigkeit und Qualität zu wählen:

Unterer rechter Quadrant: Geschwindigkeit gegenüber Qualität priorisieren
: Das ist die Philosophie »move fast and break things«. Teams, die auf Geschwindigkeit optimieren und dafür die Qualität opfern, schaffen chaotische, fragile Systeme. Sie rutschen in den linken unteren Quadranten, weil sie von ihren schlampigen Systemen ausgebremst werden. Viele Start-ups, die eine Zeitlang so gearbeitet haben, beschweren sich darüber, ihr »Mojo« zu verlieren. Einfache Änderungen, die sie in der guten alten Zeit mal eben rausgehauen hätten, brauchen jetzt Tage oder Wochen, weil das System ein verworrenes Chaos ist.

Oberer linker Quadrant: Qualität gegenüber Geschwindigkeit priorisieren
: Auch bekannt als »wir haben es hier mit ernsthaften und wichtigen Dingen zu tun, daher müssen wir es *richtig* machen«. Der Druck von Deadlines führt

1 Ja, ich arbeite als Berater, warum fragen Sie?

dann zu »Workarounds«. Aufwendige Prozesse schaffen Hürden für Verbesserungen, daher wachsen die technischen Schulden zusammen mit der Liste der »bekannten Probleme«. Diese Teams rutschen in den linken unteren Quadranten. Sie landen bei Systemen schlechter Qualität, *weil* es zu schwer ist, sie zu verbessern. Als Reaktion auf Fehler führen sie noch mehr Prozesse ein. Diese Prozesse machen es noch schwerer, Verbesserungen umzusetzen, und sorgen so für zusätzliche Fragilität und Risiken. Das führt zu mehr Fehlern und mehr Prozessen. Viele Personen, die in so agierenden Organisationen tätig sind, gehen davon aus, dass das normal ist[1] – insbesondere, wenn sie in risikosensiblen Branchen unterwegs sind.[2]

Der obere rechte Quadrant ist das Ziel moderner Ansätze wie Lean, Agile und DevOps. Sich schnell bewegen und ein hohes Qualitätsniveau beibehalten zu können, scheint wie ein Traum zu wirken. Aber die *Accelerate*-Forschung zeigt, dass viele Teams das erreichen. Daher finden Sie in diesem Quadranten »High Performer«.

Die Four Key Metrics

DORAs *Accelerate*-Forschungsteam hat vier zentrale Metriken für die Performance der Softwareauslieferung und von Operations identifiziert.[3] Dafür wurden diverse Messwerte begutachtet, und es wurde herausgefunden, dass diese vier die stärkste Korrelation dazu haben, wie gut eine Organisation ihre Ziele erreicht:

Auslieferungsdurchlaufzeit
: Die Zeit, die erforderlich ist, um Änderungen am Produktivsystem zu implementieren, zu testen und auszuliefern.

Deployment-Häufigkeit
: Wie oft Sie Änderungen an Produktivsystemen deployen.

Anteil an Änderungsfehlschlägen
: Welcher Prozentsatz an Änderungen entweder einen Service beeinträchtigt hat oder eine sofortige Korrektur erforderte, wie zum Beispiel ein Rollback oder ein Notfall-Fix.

Mean Time to Recovery (MTTR)
: Wie lange es dauert, einen Service wiederherzustellen, wenn es einen ungeplanten Ausfall oder eine Beeinträchtigung gibt.

---

1 Das ist ein Beispiel für die »Normalisierung der Abweichung« – die Menschen gewöhnen sich an eine Arbeitsweise, die das Risiko ansteigen lässt. Diane Vaughan hat diesen Begriff in *The Challenger Launch Decision* (University of Chicago Press) definiert.

2 Es ist ironisch (und beängstigend), dass so viele Mitarbeiterinnen und Mitarbeiter in Branchen wie dem Finanzwesen, in Regierungen und in der Gesundheitsversorgung fragile IT-Systeme – und Prozesse, die deren Verbesserung behindern – als normal, ja sogar als wünschenswert ansehen.

3 DORA, jetzt Teil von Google, ist das Team hinter dem *Accelerate State of DevOps Report* (*https://oreil.ly/ysk9n*).

Organisationen, die ihre Ziele gut erreichen – seien es der Umsatz, der Aktienpreis oder andere Kriterien – funktionieren auch gut in Bezug auf diese vier Metriken und umgekehrt. Die Ideen in diesem Buch sollen Ihrem Team und Ihrer Organisation dabei helfen, diese Metriken gut zu erfüllen. Drei zentrale Praktiken für Infrastructure as Code können Sie dabei unterstützen.

Drei zentrale Praktiken für Infrastructure as Code

Das Konzept des Cloud-Zeitalters nutzt die dynamische Natur moderner Infrastruktur- und Anwendungsplattformen aus, um häufig und zuverlässig Änderungen durchführen zu können. Infrastructure as Code ist ein Ansatz, mit dem Sie Infrastruktur bauen, die kontinuierliche Änderungen unterstützt, um eine hohe Zuverlässigkeit und Qualität zu erreichen. Wie kann Ihr Team das schaffen?

Es gibt drei zentrale Praktiken beim Implementieren von Infrastructure as Code:

- Definieren Sie alles als Code.
- Testen Sie all Ihre aktuelle Arbeit kontinuierlich und liefern Sie sie aus.
- Bauen Sie kleine, einfache Einheiten, die Sie unabhängig voneinander austauschen können.

Ich werde jede dieser Thesen nun vorstellen, um den Rahmen für weitere Diskussionen vorzugeben. Im weiteren Verlauf des Buchs werde ich dann jeweils in einem eigenen Kapitel die Prinzipien für das Implementieren dieser drei Praktiken ausführlich erläutern.

Zentrale Praktik: Definieren Sie alles als Code

Es ist eine zentrale Praktik für zügige und zuverlässige Änderungen, alles »als Code« zu definieren. Es gibt dafür ein paar Gründe:

Wiederverwendbarkeit
: Definieren Sie etwas als Code, können Sie viele Instanzen davon erzeugen. Sie können Ihre Elemente reparieren und schnell neu bauen, und andere Personen können identische Instanzen des Elements erzeugen.

Konsistenz
: Elemente, die aus Code gebaut werden, werden jedes Mal gleich gebaut. Dadurch wird das Verhalten von Systemen vorhersagbar, das Testen zuverlässiger und es werden kontinuierliches Testen und Ausliefern möglich.

Transparenz
: Alle können sehen, wie ein Element gebaut wird, indem sie sich den Code anschauen. Die Leute können den Code begutachten und Verbesserungen vorschlagen. Sie können daraus lernen, um Elemente im eigenen Code einzusetzen, Einblicke für das Troubleshooting liefern und zu Compliance-Zwecken Reviews und Audits durchführen.

Ich werde in Kapitel 4 genauer auf die Konzepte und Implementierungs-Prinzipien zum Definieren von Elementen als Code eingehen.

Zentrale Praktik: Kontinuierliches Testen und die gesamte aktuelle Arbeit ausliefern

Effektive Infrastruktur-Teams sind beim Testen rigoros. Sie nutzen Automation, um jede Komponente ihres Systems zu deployen und zu testen, und sie integrieren die gesamte aktuelle Arbeit. Sie testen schon während der Arbeit, statt zu warten, bis sie fertig sind.

Die Idee ist, *Qualität einzubauen*, statt zu versuchen, die *Qualität zu testen*.

Dabei wird gerne übersehen, dass dazu das Integrieren und Testen *der gesamten aktuellen Arbeit* gehört. Bei vielen Teams arbeiten die Entwicklerinnen und Entwickler am Code in eigenen Branches und integrieren erst, wenn sie fertig sind. Laut den Forschungsergebnissen von *Accelerate* liefern Teams allerdings bessere Ergebnisse, wenn sie ihre Arbeit mindestens täglich integrieren. Zum CI gehört das Mergen und Testen des Codes aller Beteiligten während des Entwickelns. CD geht noch einen Schritt weiter und sorgt dafür, dass der gemergte Code immer bereit für die Produktivumgebung ist.

Details zum kontinuierlichen Testen und Ausliefern von Infrastruktur-Code finden Sie in Kapitel 8.

Zentrale Praktik: Kleine einfache Elemente bauen, die Sie unabhängig voneinander ändern können

Teams bekommen Probleme, wenn ihre Systeme groß und eng gekoppelt sind. Je größer ein System ist, desto schwieriger wird es, es zu ändern, und umso einfacher kann man es zerstören.

Schauen Sie sich die Codebasis eines leistungsfähigen Teams an, sehen Sie den Unterschied. Das System ist aus kleinen, einfachen Elementen aufgebaut. Jedes Element lässt sich leicht verstehen, und es besitzt sauber definierte Schnittstellen. Das Team kann jede Komponente für sich einfach ändern und jede Komponente isoliert deployen und testen.

Genauer gehe ich auf die Implementierungsprinzipien dieser zentralen Praktik in Kapitel 15 ein.

Zusammenfassung

Um etwas aus der Cloud- und Infrastruktur-Automation herauszuholen, benötigen Sie eine Mentalität, die zum Cloud-Zeitalter passt. Sie müssen dafür die Geschwindigkeit ausnutzen, um die Qualität zu verbessern, und Qualität einbauen, um Geschwindigkeit aufzunehmen. Es ist Arbeit, Ihre Infrastruktur zu automatisieren, insbesondere wenn Sie erst lernen, wie Sie das tun können. Aber es hilft Ihnen dabei, Änderungen vorzunehmen – und vor allem, das System überhaupt zu bauen.

Ich habe die Elemente eines typischen Infrastruktur-Systems beschrieben, da diese die Grundlage für die Kapitel legen, in denen erklärt wird, wie Sie Infrastructure as Code implementieren.

Schließlich habe ich drei zentrale Praktiken für Infrastructure as Code beschrieben: Definieren Sie alles als Code, testen und liefern Sie kontinuierlich und bauen Sie kleine Elemente.

KAPITEL 11

Server als Code bauen

Infrastructure as Code entwickelte sich zunächst als Möglichkeit zum Konfigurieren von Servern. Systemadministratoren haben Shell-, Batch- und Perl-Skripte geschrieben. CFEngine unterstützte als Erste deklarative, idempotente DSLs zum Installieren von Paketen und Managen von Konfigurationsdateien auf Servern, Puppet und Chef folgten. Diese Tools gingen davon aus, dass Sie mit einem bestehenden Server beginnen – oft mit einem realen Server in einem Rack, manchmal mit einer virtuellen Maschine per VMware und später als Cloud-Instanz.

Jetzt konzentrieren wir uns entweder auf Infrastruktur-Stacks, in denen Server nur ein Element sind, oder wir arbeiten mit Container-Clustern, in denen es sich bei den Servern um ein zugrunde liegendes Detail handelt. Und selbst Teams, die Cluster betreiben, müssen normalerweise Server für die Host-Knoten bauen und laufen lassen.

Server sind komplexer als andere Infrastruktur-Typen (wie Networking oder Storage). Es gibt mehr Komponenten und Variationen, daher verbringen die meisten Systemteams immer noch ziemlich viel Zeit mit Betriebssystemen, Paketen und Konfigurationsdateien.

Dieses Kapitel stellt Ansätze zum Bauen und Managen von Serverkonfigurationen als Code vor. Es beginnt mit dem Inhalt von Servern (was muss konfiguriert werden) und dem Lebenszyklus von Servern (wann geschehen Konfigurationsaktivitäten). Dann geht es weiter zu Code und Tools zur Serverkonfiguration. Der zentrale Inhalt dieses Kapitels dreht sich um verschiedene Möglichkeiten, Server-Instanzen zu erstellen, Server so weit vorzubereiten, dass sich mehrere konsistente Instanzen erstellen lassen, und Ansätze zum Anwenden der Serverkonfiguration über den Lebenszyklus des Servers hinweg.

Es kann nützlich sein, den Lebenszyklus eines Servers als mehrere Übergangsphasen zu betrachten, wie dies in Abbildung 11-1 dargestellt ist.

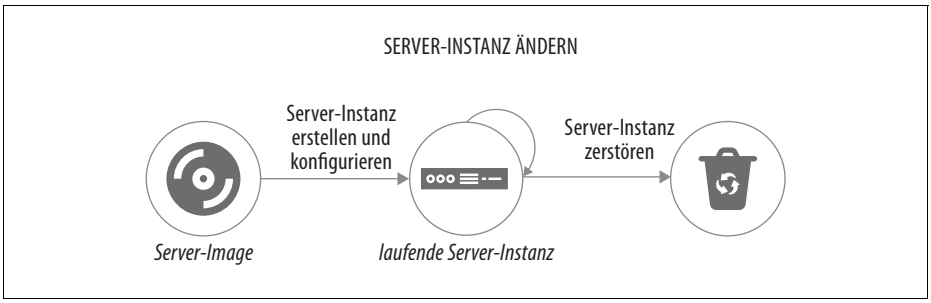

Abbildung 11-1: Der grundlegende Lebenszyklus eines Servers

Der hier gezeigte grundlegende Lebenszyklus besteht aus drei Übergangsphasen:

1. Eine Server-Instanz erstellen und konfigurieren
2. Eine bestehende Server-Instanz verändern
3. Eine Server-Instanz zerstören

Dieses Kapitel dreht sich um das Erstellen und Konfigurieren von Servern. In Kapitel 12 geht es um das Vornehmen von Änderungen an Servern, und Kapitel 13 beschreibt das Erstellen und Aktualisieren von Server-Images, die Sie zum Erstellen von Server-Instanzen einsetzen können.

Was gibt es auf einem Server

Es ist sinnvoll, über die verschiedenen Elemente auf einem Server und ihre Herkunft nachzudenken.

Ein Weg ist dabei, die Elemente in Software, Konfiguration und Daten zu unterteilen. Diese Kategorien (beschrieben in Tabelle 11-1) helfen dabei, zu verstehen, wie Tools zum Konfigurations-Management eine bestimmte Datei oder einen Satz an Dateien behandeln sollten.

Der Unterschied zwischen Konfiguration und Daten liegt darin, ob Automations-Tools automatisch den Inhalt der Datei managen. So gehört beispielsweise ein Systemlog zwar zur Infrastruktur, aber ein Tool zur automatisierten Konfiguration behandelt es als Daten. Und wenn eine Anwendung beispielsweise ihre Accounts und Voreinstellungen für die Benutzer in Dateien auf dem Server speichert, behandelt das Serverkonfigurationstool diese Datei ebenfalls als Daten.

Tabelle 11-1: Kategorien von Elementen auf einem Server

Element-Typ	Beschreibung	Behandlung durch das Konfigurations-Management
Software	Anwendungen, Bibliotheken und anderer Code. Das müssen keine ausführbaren Dateien sein – es kann sich um so gut wie jede Datei handeln, die statisch ist und sich nicht von einem System zum anderen unterscheidet. Ein Beispiel dafür sind Datendateien mit Zeitzonen auf einem Linux-System.	Stellt sicher, dass sie auf jedem relevanten Server gleich ist; kümmert sich nicht darum, was sich darin befindet.
Konfiguration	Dateien, die steuern, wie das System und/oder die Anwendung arbeitet. Die Inhalte können sich von Server zu Server unterscheiden – abhängig von den Rollen, Umgebungen, Instanzen und so weiter. Diese Dateien werden als Teil der Infrastruktur gemanagt – im Gegensatz zu Konfigurationsdateien für Anwendungen. Besitzt eine Anwendung beispielsweise ein UI zum Verwalten der Benutzerprofile, würden die Datendateien, die diese Profile speichern, aus Infrastruktur-Sicht nicht als Konfiguration betrachtet werden, sondern als Daten. Aber eine Konfigurationsdatei für eine Anwendung, die auf dem Dateisystem abgelegt und durch die Infrastruktur verwaltet wird, würde als Konfiguration angesehen werden.	Baut die Dateiinhalte auf dem Server; stellt sicher, dass sie konsistent sind.
Daten	Dateien, die durch das System und die Anwendungen erzeugt und aktualisiert werden. Die Infrastruktur kann für diese Daten zum Teil verantwortlich sein, wie zum Beispiel zum Verteilen, Sichern oder Replizieren. Aber sie behandelt den Inhalt der Dateien als Blackbox und kümmert sich nicht darum. Datenbank-Datendateien und Logdateien sind Beispiele für solche Daten.	Normales Auftreten und Ändern; muss sie eventuell sichern, kümmert sich aber nicht um die Inhalte.

Woher Dinge kommen

Software, Konfiguration und Daten, aus denen sich eine Server-Instanz zusammensetzt, können hinzugefügt werden, wenn der Server erstellt und konfiguriert wird oder wenn er sich ändert. Es gibt eine Reihe möglicher Quellen für diese Elemente (siehe Abbildung 11-2):

Basis-Betriebssystem
: Das Installations-Image für das Betriebssystem kann eine physische Festplatte, eine ISO-Datei oder ein Stock-Server-Image sein. Der Prozess der OS-Installation wählt eventuell optionale Komponenten aus.

OS-Paket-Repositories
: Die OS-Installation oder ein Post-Installations-Konfigurations-Schritt kann ein Tool starten, das Pakete aus einem oder mehreren Repositories herunterlädt und installiert (in Tabelle 10-1 finden Sie mehr zu OS-Paketformaten). OS-Anbieter stellen normalerweise ein Repository mit unterstützten Paketen

bereit. Sie können zudem Fremd-Repositories für Open-Source- oder kommerzielle Pakete einsetzen. Außerdem können Sie ein internes Repository für Pakete betreiben, die in der Firma entwickelt oder kuratiert wurden.

Sprach-, Framework- und andere Plattform-Repositories
: Neben OS-spezifischen Paketen installieren Sie eventuell Pakete für Sprachen oder Frameworks, wie Java-Bibliotheken oder Ruby-Gems. Wie bei OS-Paketen holen Sie vielleicht Pakete aus Repositories, die von Sprachanbietern, Fremdherstellern oder internen Gruppen betreut werden.

Nicht-Standard-Pakete
: Manche Anbieter und interne Gruppen bieten Software mit ihren eigenen Installern an oder solche, die neben dem Ausführen eines Standard-Paket-Management-Tools noch mehrere Schritte erfordern.

Separates Material
: Eventuell fügen Sie noch Dateien außerhalb einer Anwendung oder Komponente hinzu oder verändern welche – zum Beispiel ergänzen Sie Benutzer-Accounts oder setzen lokale Firewall-Regeln.

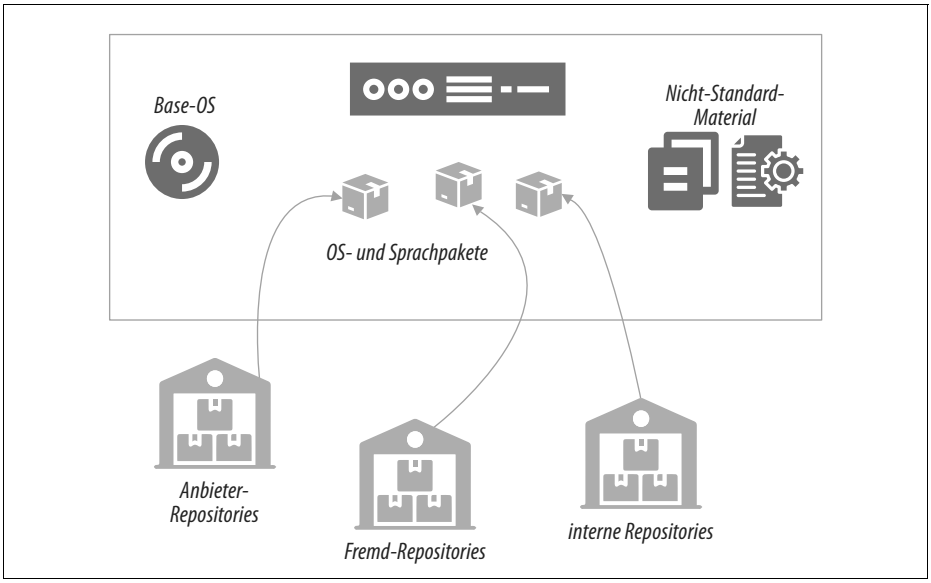

Abbildung 11-2: Pakete und andere Elemente einer Server-Instanz

Die nächste Frage dreht sich darum, wie Sie Server-Instanzen beim Erstellen oder Ändern um Elemente erweitern.

Server-Konfigurationscode

Die erste Generation von Tools für Infrastructure as Code hat sich auf die automatisierte Serverkonfiguration konzentriert. Einige der Tools in diesem Bereich sind:

- Ansible (*https://www.ansible.com*)
- CFEngine (*https://cfengine.com*)
- Chef (*https://www.chef.io*)
- Puppet (*https://puppet.com*)
- Saltstack (*https://www.saltstack.com*)

Viele dieser Tools nutzen einen Agenten, der auf jedem Server installiert ist und dem Configuration-Pattern folgt (siehe »Pattern: Pull Server Configuration« auf Seite 238). Sie stellen einen Agenten bereit, den Sie entweder als Service installieren oder periodisch aus einem Cron-Job ausführen können. Andere Tools sind so entworfen, dass sie auf einem zentralen Server laufen und sich mit jedem gemanagten Server verbinden (orientiert am Push-Pattern, siehe »Pattern: Push Server Configuration« auf Seite 236).

Sie können sowohl das Push- wie auch das Pull-Pattern mit den meisten dieser Tools implementieren. Verwenden Sie ein Tool wie Ansible, das für das Push-Modell designt ist, können Sie es auf Servern vorinstallieren und per Cron ausführen. Nutzen Sie andererseits ein Tool wie Chef oder Puppet, die beide einen Agenten zum Ausführen mit dem Pull-Modell bereitstellen, können Sie stattdessen einen Befehl auf einem zentralen Server ausführen, der sich auf jeder Maschine anmeldet und den Client startet. Daher sind Sie durch das Tool nicht auf ein Pattern eingeschränkt.

Viele Hersteller von Serverkonfigurationstools bieten Repository-Server an, auf denen man Konfigurationscode hosten kann. Beispiele dafür sind Ansible Tower, Chef Server und Puppetmaster. Diese Tools haben eventuell noch zusätzliche Funktionalität, wie zum Beispiel eine Konfigurations-Registry (siehe »Konfigurations-Registry« auf Seite 129), Dashboards, Logging oder eine zentralisierte Ausführung.

Wählt man einen Anbieter, der ein Ökosystem von Tools bereitstellt, das alles abdeckt, vereinfacht das natürlich die Arbeit des Infrastruktur-Teams. Aber es ist hilfreich, wenn Elemente des Ökosystems durch andere Tools ausgetauscht werden können, sodass das Team die Elemente auswählen kann, die ihre Bedürfnisse am besten. Nutzen Sie beispielsweise mehrere Tools, die mit einer Konfigurations-Registry integrieren, wählen Sie vielleicht eher eine eigenständige, allgemein einsetzbare Konfigurations-Registry statt einer, die mit Ihrem Serverkonfigurationstool verknüpft ist.

Die Codebasis ist ein signifikanter Teil im Leben eines modernen Infrastruktur-Teams. Die Prinzipien und Richtlinien aus Kapitel 4 gelten für Server-Code, Stack-Code und so weiter.

Code-Module für die Serverkonfiguration

Server-Konfigurationscode scheint es wie jedem Code zu gefallen, mit der Zeit zu einem unübersichtlichen Chaos anzuwachsen. Es sind Disziplin und gute Gewohnheiten erforderlich, um das zu vermeiden. Zum Glück erhalten Sie mit diesen Tools Möglichkeiten, den Code in unterschiedlichen Einheiten zu organisieren und zu strukturieren.

Viele Serverkonfigurationstools lassen Sie Code in Modulen organisieren. So besitzt Ansible beispielsweise Playbooks, Chef gruppiert Rezepte in Cookbooks und Puppet nutzt Module mit Manifesten. Sie können diese Module individuell organisieren, versionieren und bereitstellen.

Rollen sind ein Weg, eine Gruppe von Modulen für das Anwenden auf einen Server zu kennzeichnen und deren Zweck zu definieren. Anders als bei anderen Konzepten scheinen sich die meisten Tool-Anbieter dafür auf den Begriff *Rolle* geeinigt zu haben, statt sich selbst einen auszudenken.[1]

Viele dieser Tools unterstützen zudem Erweiterungen für den Kern ihres Ressourcenmodells. Eine Sprache für das Werkzeug stellt ein Modell von Server-Ressourcen-Entitäten bereit – zum Beispiel Anwender, Pakete und Dateien. So können Sie beispielsweise einen Chef Lightweight Resource Provider (LWRP) schreiben, um eine Java-Anwendungsressource zu ergänzen, die Anwendungen installiert und konfiguriert, die von Ihren Entwicklungsteams geschrieben wurden. Diese Erweiterungen ähneln Stack-Modulen (siehe Kapitel 16).

In Kapitel 15 finden Sie eine allgemeinere Diskussion über das Modularisieren von Infrastruktur mit Hinweisen zu gutem Design.

Code-Module für die Serverkonfiguration designen

Sie sollten jedes Code-Modul zur Serverkonfiguration (zum Beispiel jedes Cookbook oder Playbook) entlang eines einzelnen, abgeschlossenen Aspekts designen und schreiben. Dieser Rat orientiert sich am Software-Designprinzip der *Separation of Concerns* (*https://oreil.ly/CPXDu*). Ein üblicher Ansatz ist, ein eigenes Modul für jede Anwendung zu haben.

Ein Beispiel: Sie erstellen ein Modul zum Managen eines Anwendungsservers wie Tomcat. Der Code im Modul würde die Tomcat-Software installieren, Benutzer-Accounts und -Gruppen anlegen, unter denen sie laufen soll, und Ordner mit den

1 Es hätte wirklich zu den Leuten bei Chef gepasst, wenn sie eine Rolle dort als *Hat* bezeichnet hätten, sodass sie über Server reden könnten, die einen *Chef Hat* tragen. Zum Glück haben sie das nicht getan.

korrekten Berechtigungen erstellen, in denen Logs und andere Dateien abgelegt werden. Er würde die Konfigurationsdateien für den Tomcat bauen und darin unter anderem die Ports und Einstellungen konfigurieren. Der Modul-Code würde den Tomcat zudem in eine Reihe von Services integrieren, wie zum Beispiel Log-Aggregation, Monitoring oder Process Management.

Viele Teams finden es nützlich, Serverkonfigurationsmodule abhängig von ihrem Einsatzzweck zu entwerfen. So können Sie Ihre Module beispielsweise in Bibliotheksmodule und Anwendungsmodule unterteilen (bei Chef wären das Bibliotheks-Cookbooks und Anwendungs-Cookbooks).

Ein Bibliotheksmodul verwaltet ein Kernkonzept, das von Anwendungsmodulen wiederverwendet werden kann. Nehmen wir das Beispiel mit dem Tomcat-Modul: Sie könnten das Modul so entwerfen, dass es Parameter übernimmt und damit genutzt werden kann, um eine Instanz von Tomcat zu konfigurieren, die sich für unterschiedliche Zwecke optimieren lässt.

Ein Anwendungsmodul – auch als Wrapper-Modul bezeichnet – importiert ein oder mehrere Bibliotheksmodule und setzt ihre Parameter für einen spezifischeren Zweck. Das ShopSpinner-Team könnte ein Servermaker-Modul bauen, das Tomcat zum Ausführen ihre Produkt-Katalog-Anwendung installiert, und ein zweites, das Tomcat für die Kundenmanagement-Anwendung erstellt.[1] Beide Module nutzen eine gemeinsame Bibliothek, die Tomcat installiert.

Server-Code versionieren und weitergeben

Wie bei jedem Code sollten Sie Code zur Serverkonfiguration testen und zwischen den Umgebungen weiterreichen können, bevor Sie ihn auf Ihre Produktivsysteme anwenden. Manche Teams versionieren all ihre Server-Code-Module gemeinsam und reichen sie auch zusammen als eine Einheit weiter, indem sie sie entweder gemeinsam bauen (siehe »Pattern: Build-Time Project Integration« auf Seite 369) oder das Bauen und Testen getrennt vornehmen, bevor sie sie für den Einsatz in Produktivumgebungen zusammenstellen (siehe »Pattern: Delivery-Time Project Integration« auf Seite 373). Andere versionieren jedes Modul getrennt und reichen es auch alleine weiter (siehe »Pattern: Apply-Time Project Integration« auf Seite 375).

Behandeln Sie jedes Modul als eigenständige Einheit, müssen Sie sich um alle Abhängigkeiten zwischen diesen kümmern. Viele Tools zur Serverkonfiguration ermöglichen es Ihnen, die Abhängigkeiten für ein Modul in einem Deskriptor zu beschreiben. In einem Chef-Cookbook führen Sie beispielsweise Abhängigkeiten im Feld `depends` der Metadatendatei für das Cookbook auf (*https://oreil.ly/_4eEP*). Das Tool zur Serverkonfiguration nutzt diese Spezifikation, um abhängige Module

1 Vielleicht erinnern Sie sich an Kapitel 4 und daran, dass es sich bei Servermaker um ein fiktives Tool zur Serverkonfiguration handelt, das Ansible, Chef und Puppet ähnelt.

herunterzuladen und auf eine Server-Instanz anzuwenden. Das Abhängigkeitsmanagement für Server-Konfigurationsmodule funktioniert genauso wie die Systeme zum Software-Paketmanagement – zum Beispiel für RPMs oder Python-PIP-Pakete.

Sie müssen auch verschiedene Versionen Ihres Serverkonfigurationscodes speichern und verwalten. Oft haben Sie eine Version des Codes aktuell auf die Produktivumgebungen angewandt, während andere Versionen entwickelt und getestet werden.

In den Kapiteln 18 und 19 gehe ich genauer auf das Organisieren, Verpacken und Bereitstellen Ihres Infrastruktur-Codes ein.

Serverrollen

Wie schon erwähnt ist eine Serverrolle eine Möglichkeit, eine Gruppe von Serverkonfigurationsmodulen zu definieren, die auf einen Server angewendet werden. Eine Rolle kann auch ein paar Standardparameter setzen. So könnten Sie beispielsweise eine Rolle `application-server` erstellen:

```
role: application-server
  server_modules:
    - tomcat
    - monitoring_agent
    - logging_agent
    - network_hardening
  parameters:
    - inbound_port: 8443
```

Weisen Sie diese Rolle einem Server zu, wenden Sie die Konfiguration für Tomcat, Monitoring- und Logging-Agenten und das Netzwerk-Hardening an. Zudem wird ein Parameter für einen eingehenden Port gesetzt – vermutlich einer, den das Modul `network_hardening` verwendet, um diesen Port in der lokalen Firewall zu öffnen, während es alles andere dicht macht.

Rollen können unübersichtlich werden, daher sollten Sie sie bedacht und konsistent einsetzen. Sie können feingranulare Rollen nutzen, die Sie kombinieren, um spezifische Server zu erstellen. Sie weisen einem gegebenen Server mehrere Rollen zu – wie zum Beispiel `ApplicationServer`, `MonitoredServer` und `PublicFacingServer`. Jede dieser Rollen enthält eine kleine Zahl von Servermodulen für einen eng begrenzten Zweck.

Alternativ können Sie High-Level-Rollen einsetzen, die mehr Module enthalten. Dann hat jeder Server vermutlich nur eine Rolle, die recht spezifisch sein kann – zum Beispiel `ShoppingServiceServer` oder `JiraServer`.

Oft nutzt man Rollenvererbung. Sie definieren eine Basisrolle, die Software und Konfiguration enthält, die für alle Server zum Einsatz kommen. Diese Rolle kann beispielsweise Netzwerk-Hardening, administrative Benutzer-Accounts und Agenten für das Monitoring und Logging enthalten. Spezifischere Rollen, wie zum Bei-

spiel für Anwendungsserver oder Container-Hosts, übernehmen die Basisrolle und ergänzen dann ein paar zusätzliche Module und Parameter zur Serverkonfiguration:

```
role: base-role
  server_modules:
    - monitoring_agent
    - logging_agent
    - network_hardening

role: application-server
  include_roles:
    - base_role
  server_modules:
    - tomcat
  parameters:
    - inbound_port: 8443

role: shopping-service-server
  include_roles:
    - application-server
  server_modules:
    - shopping-service-application
```

Dieses Codebeispiel definiert drei Rollen in einer Hierarchie. Die Rolle `shopping-service-server` erbt alles von der Rolle `application-server` und fügt ein Modul zum Installieren der spezifischen Anwendung hinzu. Die Rolle `application-server` erbt von der Rolle `base-role` und ergänzt den Tomcat-Server und die Konfiguration für die Netzwerk-Ports. Die Rolle `base-role` definiert einen gemeinsamen Satz allgemein nutzbarer Konfigurationsmodule.

Server-Code testen

In Kapitel 8 wurde erklärt, wie automatisiertes Testen und CD in der Theorie auf Infrastruktur angewendet werden, während Sie in Kapitel 9 Ansätze kennengelernt haben, wie Sie dies mit Infrastruktur-Stacks implementieren. Viele der Konzepte in diesen beiden Kapiteln gelten auch für das Testen von Server-Code.

Server-Code progressiv testen

Es gibt eine sich selbst verstärkende Dynamik zwischen dem Design und dem Testen von Code. Es ist einfacher, Tests für eine sauber strukturierte Codebasis zu schreiben und zu betreuen. Und durch das Schreiben von Tests (und Ihren Einsatz für ihr erfolgreiches Bestehen) sind Sie gezwungen, diese klare Struktur beizubehalten. Indem Sie jede Änderung in eine Pipeline stecken, die Ihre Tests ausführt, helfen Sie Ihrem Team dabei, die Disziplin zum kontinuierlichen Refaktorieren für ein Minimieren der technischen und Design-Schulden aufrechtzuerhalten.

Die weiter oben beschriebene Struktur von Serverrollen, die aus Modulen zur Serverkonfiguration zusammengesetzt werden, die wiederum in Bibliotheksmodulen und Anwendungsmodulen organisiert sein können, passt gut zu einer progressiven Teststrategie (siehe »Progressives Testen« auf Seite 148). Eine Folge von Pipeline-Stages kann jedes dieser Module mit zunehmender Integration testen, wie in Abbildung 11-3 zu sehen ist.

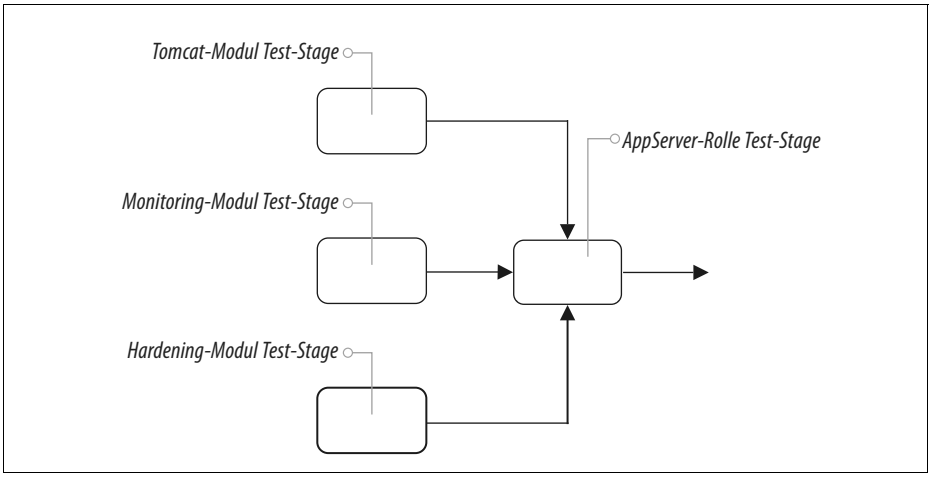

Abbildung 11-3: Server-Code-Module progressiv testen

Eine eigene Stage testet jedes Code-Modul, wann immer eine Änderung für dieses Modul eingecheckt wird. Die Serverrolle besitzt ebenfalls eine Test-Stage. Diese Stage führt die Tests immer dann aus, wenn sich eines der Module ändert, das von dieser Rolle genutzt wird und wenn dessen Test erfolgreich war. Die Stage zum Testen der Rolle läuft außerdem, wenn eine Änderung am Code der Rolle vorgenommen wird – zum Beispiel beim Hinzufügen oder Entfernen eines Moduls oder bei Änderungen an den Parametern.

Was Sie bei Server-Code testen

Es ist knifflig, zu entscheiden, was Sie bei Server-Code testen sollen. Sie landen dann wieder bei der Frage, ob es einen Wert hat, deklarativen Code zu testen (siehe »Herausforderung: Tests für deklarativen Code haben häufig nur einen geringen Wert« auf Seite 143). Module für Server-Code – insbesondere in einer gut designten Codebasis – tendieren dazu, klein, einfach und fokussiert zu sein. Ein Modul, das eine Java JVM installiert, kann beispielsweise aus einer einzelnen Anweisung mit ein paar Parametern bestehen:

```
package:
  name: java-${JAVA_DISTRIBUTION}
  version: ${JAVA_VERSION}
```

In der Praxis besitzen wahrscheinlich selbst einfach erscheinende Module zum Installieren von Paketen mehr Code, es werden Dateipfade und Konfigurationsdateien angepasst, und vielleicht kommt auch noch ein Benutzer-Account dazu. Aber oft gibt es nicht viel zu testen, das nicht einfach den Code selbst neu formuliert.

Tests sollten sich auf häufige Probleme, variable Ergebnisse und die Kombination von Code konzentrieren.

Denken Sie an häufige Probleme, Dinge, die in der Praxis gerne schiefgehen, und wie Sie eine erfolgreiche Umsetzung prüfen können. Bei einer einfachen Paketinstallation wollen Sie vielleicht prüfen, dass der Befehl im Standardpfad verfügbar ist. Ein Test würde also den Befehl aufrufen und sicherstellen, dass er ausgeführt wurde:

```
given command 'java -version' {
  its(exit_status) { should_be 0 }
}
```

Liefert das Paket abhängig von den übergebenen Parametern sehr unterschiedliche Ergebnisse, sollten Sie Tests schreiben, um sicherzustellen, dass es sich in den unterschiedlichen Situationen korrekt verhält. Beim JVM-Installationspaket könnten Sie den Test mit unterschiedlichen Werten für die Java-Distribution schreiben und beispielsweise prüfen, dass der Befehl `java` unabhängig von der gewählten Distribution verfügbar ist.

In der Praxis steigt der Wert des Testens mit zunehmender Integration von mehr Elementen. Daher schreiben Sie vielleicht mehr Tests für die Anwendungsserver-Rolle und führen sie aus, als für die Module, die die Rolle enthält.

Das gilt insbesondere, wenn diese Module untereinander integriert sind und miteinander interagieren. Ein Modul, das Tomcat installiert, wird vermutlich nicht mit einem kollidieren, das einen Monitoring-Client installiert, aber ein Modul zum Absichern des Servers könnte zu Problemen führen. In diesem Fall werden Sie vielleicht Tests haben wollen, die auf der Anwendungsserver-Rolle laufen, um sicherzustellen, dass der Anwendungsserver auch dann läuft und Requests annimmt, wenn Ports versperrt wurden.

Wie Sie Server-Code testen

Die meisten automatisierten Tests für Server-Code führen Befehle auf einem Server oder in einer Container-Instanz aus und überprüfen die Ergebnisse. Diese Tests können die Existenz und den Status von Ressourcen kontrollieren, wie zum Beispiel Pakete, Dateien, Benutzer-Accounts und laufende Prozesse. Tests können sich auch Ergebnisse anschauen – sich beispielsweise mit einem Netzwerk-Port verbinden, um zu prüfen, ob ein Service das erwartete Ergebnis zurückliefert.

Zu den bekanntesten Tools für das Testen von Bedingungen auf einem Server gehören unter anderem Inspec (*https://www.inspec.io*), Serverspec (*https://serverspec.org*) und Terratest (*https://oreil.ly/W6H5Q*).

Die Strategien zum Testen vollständiger Infrastruktur-Stacks beinhalten das Ausführen von Offline- und Onlinetests (siehe »Offline-Test-Stages für Stacks« auf Seite 165 und »Online-Test-Stages für Stacks« auf Seite 168). Bei Onlinetests werden Elemente auf der Infrastruktur-Plattform erzeugt. Normalerweise können Sie Server-Code offline mit Containern oder lokalen virtuellen Maschinen testen.

Infrastruktur-Entwicklerinnen und -Entwickler, die lokal arbeiten, können eine Container-Instanz oder eine lokale VM mit einer minimalen Betriebssystem-Installation erzeugen, den Server-Konfigurationscode anwenden und dann die Tests laufen lassen.

Pipeline-Stages, die Server-Code testen, können das Gleiche tun und die Container-Instanz oder VM auf dem Agenten laufen lassen (zum Beispiel dem Jenkins-Knoten, der den Job ausführt). Alternativ könnte die Stage eine eigenständige Container-Instanz in einem Container-Cluster starten. Das funktioniert gut, wenn die Pipeline-Orchestrierung selbst containerisiert ist.

Sie können den Ratschlägen für Stacks zum Orchestrieren von Server-Code-Tests folgen (siehe »Test-Orchestrierung« auf Seite 183). Dazu gehört das Schreiben von Skripten, die die Vorbereitungen für das Testen treffen – zum Beispiel Container-Instanzen hochfahren –, bevor die Tests ausgeführt und die Ergebnisse zusammengefasst werden. Sie sollten die gleichen Skripte zum lokalen Testen laufen lassen, die Sie auch zum Testen in Ihrem Pipeline-Service nutzen, damit die Ergebnisse konsistent sind.

Eine neue Server-Instanz erstellen

Der weiter oben beschriebene grundlegende Server-Lebenszyklus beginnt mit dem Erstellen einer neuen Server-Instanz. Ein umfassenderer Lebenszyklus enthält Schritte für das Erstellen und Aktualisieren von Server-Images, aus denen Sie Server-Instanzen erstellen können. Aber um dieses Thema kümmern wir uns in einem anderen Kapitel (Kapitel 13). Hier soll unser Lebenszyklus mit dem Erstellen einer Instanz beginnen, wie in Abbildung 11-4 zu sehen ist.

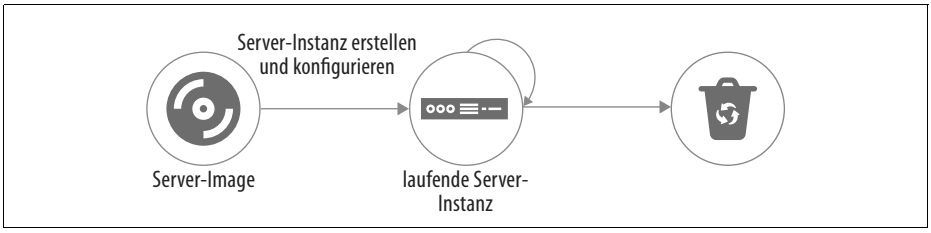

Abbildung 11-4: Eine Server-Instanz erstellen

Zum vollständigen Prozess des Erstellens eines Servers gehört das Provisionieren der Server-Instanz, sodass sie bereit für den Einsatz ist. Zu den Aktivitäten beim Erstellen und Provisionieren einer Server-Instanz gehören unter anderem:

- Infrastruktur-Ressourcen zum Instanziieren der Server-Instanz allokieren. Dazu wird ein realer Server aus einem Pool gewählt, oder ein Host-Server wird genutzt, auf dem der Server als virtuelle Maschine läuft. Für eine virtuelle Maschine allokiert die Hypervisor-Software auf dem Host Speicher und andere Ressourcen. Der Prozess kann auch Storage für Disk Volumes für die Server-Instanz allokieren.
- Das Betriebssystem und die initiale Software installieren. Das Betriebssystem kann vielleicht auf ein Disk Volume kopiert werden, wenn von einem Server-Image wie einem AMI gebootet wird. Oder die neue Instanz führt einen Installationsprozess aus, der Dateien für die neue Instanz auswählt und dorthin kopiert – möglicherweise mit einem skriptbaren Installer. Beispiele für geskriptete OS-Installer sind Red Hat Kickstart (*https://oreil.ly/k6d53*), Solaris JumpStart (*https://oreil.ly/flDLL*), Debian Preseed (*https://oreil.ly/tXdcu*) und das Windows Installation Answer File (*https://oreil.ly/oOq0C*).
- Es wird zusätzliche Konfiguration auf die Server-Instanz angewendet. In diesem Schritt führt der Prozess das Serverkonfigurationstools aus, wobei er sich an einem in »Wie Sie Serverkonfigurationscode anwenden« auf Seite 236 beschriebenen Pattern orientiert.
- Networking-Ressourcen und Policies werden konfiguriert und angebunden. Dazu können das Zuweisen des Servers zu einem Netzwerk-Adressblock, das Erstellen von Routen und das Hinzufügen von Firewall-Regeln gehören.
- Die Server-Instanz wird bei Services registriert. So kann der neue Server beispielsweise zu einem Monitoring-System hinzugefügt werden.

Diese Punkte schließen sich nicht gegenseitig aus – Ihr Prozess zum Erstellen eines Servers kann eine oder mehrere Methoden für das Ausführen dieser unterschiedlichen Aktivitäten nutzen.

Es gibt eine Reihe von Mechanismen, die Sie verwenden können, um das Erstellen einer Server-Instanz anzustoßen. Schauen wir uns jeden einzelnen an.

Eine neue Server-Instanz per Hand erstellen

Infrastruktur-Plattformen bieten Tools zum Erstellen neuer Server an, meist über ein Web-UI, als Befehlszeilentool und manchmal auch als GUI-Anwendung. Immer dann, wenn Sie einen neuen Server erstellen, wählen Sie die gewünschten Optionen aus – unter anderem das Ausgangs-Image, die zu allokierenden Ressourcen und die Networking-Details:

```
$mycloud server new \
    --source-image=stock-linux-1.23 \
    --memory=2GB \
    --vnet=appservers
```

Es ist zwar nützlich, mit UIs und Befehlszeilen-Tools herumzuspielen, um mit einer Plattform zu experimentieren, aber Server, die von anderen gebraucht werden, sollten man auf diesem Weg nicht erstellen. Die Prinzipien, die weiter oben für das Erstellen und Konfigurieren von Stacks besprochen wurden (siehe »Patterns zum Konfigurieren von Stacks« auf Seite 110) gelten auch für Server. Setzen Sie Optionen manuell (wie in »Antipattern: Manual Stack Parameters« auf Seite 110 beschrieben), passieren schneller Fehler und die Wahrscheinlichkeit für inkonsistent konfigurierte Server, nicht planbare Systeme und zu viel Wartungsarbeiten steigt.

Einen Server mit einem Skript erstellen

Sie können Skripte schreiben, die Server konsistent erzeugen. Das Skript verpackt ein Befehlszeilen-Tool oder nutzt die API der Infrastruktur-Plattform, um eine Server-Instanz zu erstellen und die Konfigurationsoptionen im Skript-Code oder über Konfigurationsdateien zu setzen. Ein Skript, das Server erstellt, kann wiederverwendet werden, ist konsistent und transparent. Dies ist das gleiche Konzept wie das Scripted Parameters Pattern für Stacks (siehe »Pattern: Scripted Parameters« auf Seite 114):

```
mycloud server new \
    --source-image=stock-linux-1.23 \
    --memory=2GB \
    --vnet=appservers
```

Dieses Skript ist identisch mit dem vorigen Befehlszeilenbeispiel. Aber weil sich der Befehl in einem Skript befindet, können andere Personen in meinem Team sehen, wie ich den Server erstellt habe. Sie können mehr Server erstellen und darauf vertrauen, dass sich diese genauso verhalten wie der von mir erstellte.

Vor Terraform und anderen Tools zum Stack-Management schrieben die meisten Teams, mit denen ich zusammengearbeitet habe, Skripte zum Erstellen von Servern. Wir haben diese Skripte im Allgemeinen durch Konfigurationsdateien anpassbar gemacht – wie beim Stack-Configuration-Pattern (siehe »Pattern: Stack Configuration Files« auf Seite 117). Aber wir haben zu viel Zeit mit dem Verbessern und Korrigieren dieser Skripte verbracht.

Einen Server mit einem Stack-Management-Tool erstellen

Mit einem Stack-Management-Tool (siehe Kapitel 5) können Sie einen Server im Kontext von anderen Infrastruktur-Ressourcen definieren (siehe Listing 11-1). Das Tool nutzt die Plattform-API, um die Server-Instanz zu erstellen oder zu aktualisieren.

Listing 11-1: Stack-Code, der einen Server definiert

```
server:
  source_image: stock-linux-1.23
  memory: 2GB
  vnet: ${APPSERVER_VNET}
```

Es gibt eine Reihe von Gründen, warum der Einsatz eines Stack-Tools zum Erstellen von Servern so praktisch ist. Einer ist, dass sich das Tool um Logik kümmert, die Sie sonst in Ihrem eigenen Skript implementieren müssten, wie zum Beispiel die Behandlung von Fehlern. Ein weiterer Vorteil eines Stacks ist, dass dieser die Integration mit anderen Infrastruktur-Elementen umsetzt – zum Beispiel das Verbinden des Servers mit Netzwerkstrukturen und Storage, der im Stack definiert ist. Der Code in Listing 11-1 setzt den Parameter `vnet` mit einer Variablen `${APPSERVER_VNET}`, die sich vermutlich auf eine Netzwerkstruktur bezieht, die in einem anderen Teil des Stack-Codes definiert ist.

Die Plattform für das automatische Erstellen von Servern konfigurieren

Die meisten Infrastruktur-Plattformen können automatisch aufgrund bestimmter Umstände neue Server-Instanzen erstellen. Zwei häufige Fälle sind *Autoscaling*, das Hinzufügen von Servern, um eine wachsende Last verarbeiten zu können, und *Auto-Recovery*, das Ersetzen einer Server-Instanz, wenn diese ausfällt. Sie können das normalerweise durch Stack-Code definieren (siehe Listing 11-2).

Listing 11-2: Stack-Code zum automatischen Skalieren

```
server_cluster:
  server_instance:
    source_image: stock-linux-1.23
    memory: 2GB
    vnet: ${APPSERVER_VNET}
  scaling_rules:
    min_instances: 2
    max_instances: 5
    scaling_metric: cpu_utilization
    scaling_value_target: 40%
  health_check:
    type: http
    port: 8443
    path: /health
    expected_code: 200
    wait: 90s
```

Dieses Beispiel weist die Plattform an, mindestens zwei und höchstens fünf Instanzen des Servers laufen zu lassen. Die Plattform ergänzt oder entfernt Instanzen nach Bedarf, um die CPU-Auslastung nahe bei 40 Prozent zu halten.

Die Definition enthält zudem einen Health Check. Dieser Check führt einen HTTP-Request an `/health` auf Port 8443 aus und geht davon aus, dass der Server in Ordnung ist, wenn der Request einen HTTP-Response-Code `200` zurückgibt. Liefert der Server den erwarteten Code nicht innerhalb von 90 Sekunden, nimmt die Plattform an, dass der Server ausgefallen ist, zerstört ihn und erzeugt eine neue Instanz.

Einen Server mit einem Network-Provisioning-Tool erstellen

In Kapitel 3 habe ich Bare-Metal Clouds erwähnt, die Hardware-Server dynamisch provisionieren. Der entsprechende Prozess enthält meist die folgenden Schritte:

1. Wähle einen ungenutzten realen Server und sorge dafür, dass er in einem »Network Install«-Modus bootet, der von der Server-Firmware unterstützt wird.[1]
2. Der Netzwerk-Installer bootet den Server aus einem einfachen Bootstrap-OS-Image, um die OS-Installation zu starten.
3. Lade ein OS-Installations-Image herunter und kopiere es auf die primäre Festplatte.
4. Starte den Server neu, um das OS im Setup-Modus zu booten und einen unbeaufsichtigten, geskripteten OS-Installer auszuführen.

Es gibt eine Reihe von Tools, um diesen Prozess zu managen, unter anderem:

- Crowbar (*http://opencrowbar.github.io*)
- Cobbler (*http://cobbler.github.io*)
- FAI oder Fully Automatic Installation (*http://fai-project.org*)
- Foreman (*http://theforeman.org*)
- MAAS (*https://maas.io*)
- Rebar (*https://rebar.digital*)
- Tinkerbell (*https://tinkerbell.org*)

Statt ein OS-Installations-Image zu booten, könnten Sie auch ein vorbereitetes Server-Image booten. Damit schaffen Sie die Gelegenheit, ein paar der anderen Methoden zum Vorbereiten von Servern zu implementieren, die im nächsten Abschnitt beschrieben werden.

FaaS-Events können dabei helfen, einen Server zu provisionieren

FaaS-Serverless-Code kann beim Provisionieren eines neuen Servers helfen. Ihre Plattform kann Code an unterschiedlichen Stellen im Prozess ausführen – vor, während und nach dem Erstellen einer neuen Instanz. Beispiele sind das Zuweisen von Sicherheitsrichtlinien zu einem neuen Server, das Registrieren bei einem Monitoring-Service oder das Ausführen eines Serverkonfigurationstools.

1 Oft muss man eine Funktionstaste beim Starten des Servers drücken, damit er PXE zum Booten eines Netzwerk-Image verwendet. Das kann unbeaufsichtigt zu einem Problem werden. Aber viele Hardware-Anbieter bieten eine »Lights-Out Management«-(LOM-)Funktionalität (*https://oreil.ly/N-85U*), die diesen Schritt aus der Ferne ermöglicht.

Server vorbereiten

Wie schon erwähnt (siehe »Woher Dinge kommen« auf Seite 207) gibt es eine Reihe von Quellen für die Inhalte auf einem neuen Server, unter anderem die Installation des Betriebssystems, Paket-Downloads aus Repositories und eigene Konfigurationsdateien und Anwendungsdateien, die auf den Server kopiert werden.

Sie können das zwar alles zusammensetzen, wenn Sie den Server erstellen, es gibt aber auch eine Reihe von Möglichkeiten, Serverinhalte im Voraus vorzubereiten. Diese Ansätze können den Prozess des Bauens eines Servers optimieren, ihn beschleunigen und vereinfachen und es leichter machen, mehrere konsistente Server zu erstellen. Im Folgenden stelle ich ein paar der Ansätze vor. Danach werden Optionen für das Anwenden der Konfiguration vor, während und nach dem Provisionierungsprozess beschrieben.

Hot-Cloning eines Servers

Viele Infrastruktur-Plattformen machen es leicht, einen laufenden Server zu duplizieren. Das Hot-Cloning eines Servers auf diese Art und Weise geht schnell und einfach, und Sie erhalten so Server, die zum Zeitpunkt des Klonens konsistent sind.

Das Klonen eines laufenden Servers kann nützlich sein, wenn Sie einen Server untersuchen oder dort Fehler beheben wollen, ohne ihn in seiner eigentlichen Arbeit zu unterbrechen. Aber es birgt auch Risiken. Eines ist, dass eine Kopie eines Produktivservers, die Sie zum Experimentieren erstellen, die Produktivumgebung beeinflussen kann. Ist beispielsweise ein geklonter Anwendungsserver so konfiguriert, dass er die Produktivdatenbank nutzt, verändern oder beschädigen Sie eventuell unabsichtlich Produktivdaten.

Geklonte Server, die in der Produktivumgebung laufen, besitzen im Allgemeinen historische Daten des Originalservers, wie zum Beispiel Logdateien. Diese Datenverschmutzung kann die Fehlerbehebung verkomplizieren. Ich habe schon gesehen, wie Leute viel Zeit mit Fehlermeldungen verbracht haben, die letztendlich gar nicht von dem Server kam, auf dem sie debuggt haben.

Und geklonte Server sind nicht wirklich reproduzierbar. Zwei Server, die vom gleichen Ursprungsserver, aber zu unterschiedlichen Zeiten geklont wurden, werden sich unterscheiden, und Sie können nicht einfach zurückkehren, einen dritten Server bauen und genau wissen, ob und wie er sich von den anderen unterscheidet. Geklonte Server sind eine Ursache für Konfigurationsdrift (siehe »Konfigurationsdrift« auf Seite 48).

Einen Server-Snapshot verwenden

Statt neue Server zu bauen, indem ein laufender Server direkt geklont wird, können Sie einen Snapshot eines laufenden Servers erstellen und Ihre Server daraus erstellen. Auch dies wird von den meisten Infrastruktur-Plattformen durch Befehle und API-Aufrufe erleichtert, mit denen Sie ein statisches Image erstellen. So können Sie so viele Server erzeugen, wie Sie möchten, und dabei sicher sein, dass jeder einzelne identisch zum Ausgangsserver ist.

Erstellen Sie einen Server aus einem Snapshot, der von einem laufenden Server gemacht wurde, erhalten Sie allerdings auch viele der Nachteile des Hot-Clonings. Der Snapshot kann durch Logs, Konfiguration und andere Daten des Originalservers verschmutzt sein. Es ist effektiver, von Grund auf ein sauberes Server-Image zu erzeugen.

Ein sauberes Server-Image erstellen

Ein Server-Image ist ein Snapshot, den Sie aus sauberen, bekannten Quellen erstellen, sodass Sie mehrere, konsistente Server-Instanzen erzeugen können. Das tun Sie eventuell mit den gleichen Features der Infrastruktur-Plattform, mit denen Sie auch einen Snapshot eines Live-Servers erstellen, aber die Original-Server-Instanz wird nie als Teil Ihres Gesamtsystems verwendet. So stellen Sie sicher, dass jeder neue Server sauber ist.

Ein typischer Prozess für das Erstellen eines Server-Image sieht so aus:

1. Erstellen Sie eine neue Server-Instanz aus einer bekannten Quelle, zum Beispiel aus dem Installations-Image eines Betriebssystem-Anbieters oder aus einem Image, das von Ihrer Infrastruktur-Plattform für diesen Zweck bereitgestellt wird.
2. Wenden Sie den Code zur Serverkonfiguration an oder führen Sie andere Skripte auf dem Server aus. Damit werden vielleicht Standard-Pakete und Agenten installiert, die Sie auf all Ihren Servern haben wollen, die Konfiguration wird abgesichert, und die neuesten Patches werden angewendet.
3. Erstellen Sie einen Snapshot des Servers, erzeugen Sie ein Image, das Sie als Quelle für das Anlegen mehrerer Server-Instanzen nutzen können. Dazu können Schritte gehören, mit denen Sie den Snapshot als Image zum Erstellen neuer Server auszeichnen. Auch Tags und Versionsnummer können dabei helfen, eine Bibliothek mit vielen Server-Images im Griff zu behalten.

Manchmal werden Server-Images als *Golden Images* bezeichnet. Auch wenn manche Teams diese Images von Hand bauen – und dabei vielleicht eine Checkliste mit Schritten abarbeiten –, sehen Sie beim Lesen dieses Buchs vermutlich sofort die Vorteile des Automatisierens dieses Prozesses, wenn Server-Images als Code gemanagt werden. Das Bauen und Bereitstellen von Server-Images ist Thema von Kapitel 13.

Eine neue Server-Instanz konfigurieren

Dieses Kapitel hat beschrieben, was die Elemente eines Servers sind, woher sie kommen, wie man eine neue Server-Instanz erzeugt und worin der Wert des Vorbereitens von Server-Images liegt. Das letzte Puzzleteil beim Erstellen von Servern und beim Provisionieren ist das automatisierte Anwenden des Server-Konfigurationscodes auf neue Server. Es gibt in diesem Prozess eine Reihe von Stellen, an denen Sie das umsetzen können, wie in Abbildung 11-5 zu sehen ist.

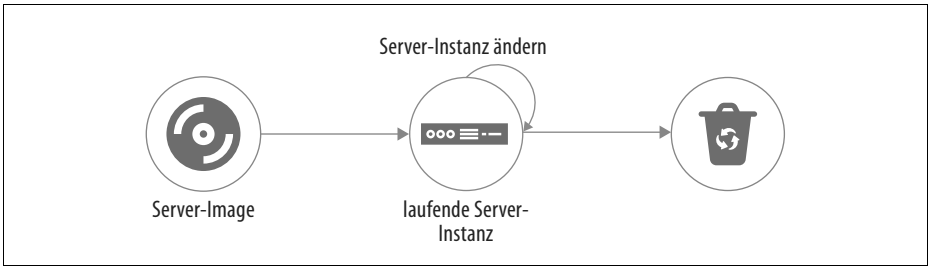

Abbildung 11-5: Wo die Konfiguration im Server-Lebenszyklus angewendet werden kann

Die Konfiguration kann angewendet werden, wenn das Server-Image erstellt wird, wenn die Server-Instanz aus dem Image erzeugt wird oder wenn der Server läuft:

Ein Server-Image konfigurieren
: Sie wenden die Konfiguration an, wenn Sie ein Server-Image bauen, das genutzt wird, um mehrere Server-Instanzen zu erstellen. Betrachten Sie es als einmaliges Konfigurieren, das dann mehrfach zum Einsatz kommt. Das wird oft als *Backen (Baking)* eines Server-Image bezeichnet.

Eine neue Server-Instanz konfigurieren
: Sie wenden die Konfiguration an, wenn Sie eine neue Server-Instanz erstellen. So konfigurieren Sie viele Male. Das wird manchmal als *Ausbacken (Frying)* einer Server-Instanz bezeichnet.

Eine laufende Server-Instanz konfigurieren
: Sie wenden die Konfiguration auf einen Server an, der schon im Einsatz ist. Ein häufiger Grund dafür ist das Umsetzen einer Änderung, wie zum Beispiel das Anwenden von Sicherheits-Patches. Ein weiterer Grund ist das Zurücknehmen von Änderungen, die außerhalb der automatisierten Konfiguration vorgenommen wurden, um die Konsistenz sicherzustellen. Manche Teams wenden auch die Konfiguration an, um einen bestehenden Server umzuwandeln – zum Beispiel einen Webserver in einen Anwendungsserver umzubauen.

Die letzte dieser drei Optionen – das Konfigurieren einer laufenden Server-Instanz – geschieht meist, um einen Server zu ändern, war Thema von Kapitel 12 ist. Die ersten beiden sind Optionen für das Anwenden einer Konfiguration beim Erstellen eines neuen Servers, daher passen sie besser in den Rahmen dieses Kapitels. Die

entscheidende Frage ist, den richtigen Zeitpunkt zum Anwenden der Konfiguration auf einen neuen Server zu bestimmen – jede neue Server-Instanz ausbacken oder in das Server-Image backen?

Eine Server-Instanz ausbacken

Wie beschrieben gehört zum Ausbacken eines Servers das Anwenden der Konfiguration, wenn Sie die neue Server-Instanz erstellen. Sie können das ins Extreme treiben, indem Sie jedes Server-Image auf ein absolutes Minimum reduzieren und alles für einen gegebenen Server Spezifische in das Live-Image einbauen. So haben neue Server immer die neuesten Änderungen, unter anderem System-Patches, die neuesten Versionen der Softwarepakete und aktuelle Konfigurationsoptionen. Das Ausbacken eines Servers ist ein Beispiel für eine Delivery-Time Integration (siehe »Pattern: Delivery-Time Project Integration« auf Seite 373).

Dieser Ansatz vereinfacht das Image-Management. Es gibt nicht viele Images – vermutlich nur eines für jede in der Infrastruktur genutzte Kombination aus Hardware und OS-Version. So kann es beispielsweise ein Image für 64-Bit-Windows 2019 und jeweils eines für 32-Bit- und 64-Bit-Ubuntu 18.x geben. Die Images müssen nicht sehr häufig aktualisiert werden, da sich an ihnen nicht sehr viel ändert. Und Sie können immer auch noch die neuesten Patches anwenden, wenn Sie einen Server provisionieren.

Manche Teams, mit denen ich gearbeitet habe, haben sich bei der Umsetzung der Infrastruktur-Automation frühzeitig darauf festgelegt, Server auszubacken. Das Aufsetzen des Toolings und der Prozesse für das Verwalten der Server-Images erfordert viel Aufwand. Wir haben diese Arbeit in unser Backlog übernommen, um sie nach dem Bauen unseres zentralen Infrastruktur-Managements zu erledigen.

In anderen Fällen ist das Ausbacken sinnvoll, weil Server hochverfügbar sind. Eine Hosting-Firma, die ich kenne, lässt Kundinnen und Kunden aus einer großen Zahl von Anpassungen für die Server wählen. Die Firma managt nur sehr wenige Basis-Server-Images und steckt mehr Aufwand in das Automatisieren, dass jeden neuen Server konfiguriert.

Es gibt eine Reihe potenzieller Probleme beim Installieren und Konfigurieren von Elementen eines Servers, wenn eine Instanz erzeugt wird. Dazu gehören:

Geschwindigkeit
: Aktivitäten, die beim Bauen eines Servers notwendig sind, kosten Zeit. Diese zusätzliche Zeit kann insbesondere dann ein Problem sein, wenn es darum geht, Server hochzufahren, um Lastspitzen abzufangen oder um nach Ausfällen wieder Services anzubieten.

Effizienz
: Zum Konfigurieren eines Servers gehört oft das Herunterladen von Paketen aus Repositories über das Netzwerk. Das kann unnötig und langsam sein.

Müssen Sie beispielsweise in kurzer Zeit 20 Server hochfahren, ist es unwirtschaftlich, jeden von ihnen die gleichen Patches und Anwendungsinstaller herunterladen zu lassen.

Abhängigkeiten
: Das Konfigurieren eines Servers hängt meist von Artefakt-Repositories und anderen Systemen ab. Ist eine dieser Quellen offline oder nicht erreichbar, können Sie keinen neuen Server erstellen. Das kann insbesondere in einem Notfall schmerzhaft sein, wenn Sie schnell viele Server neu bauen müssen. In solchen Situationen sind eventuell auch Netzwerk-Devices oder Repositories nicht verfügbar, was zu einem komplexen geordneten Graphen der Systeme führt, um herauszufinden, welche davon zuerst neu gestartet oder neu gebaut werden müssen, um die nächsten Systeme starten zu können.

Server-Images backen

Am anderen Ende des Spektrums liegt der Ansatz, beim Erstellen von Servern fast alles in das Server-Image hinein zu konfigurieren. Das Bauen neuer Server geht dann sehr schnell und einfach vonstatten, weil Sie nur eine instanzspezifische Konfiguration vornehmen müssen. Das Backen eines Server-Image ist ein Beispiel für die Build-Time Integration (siehe »Pattern: Build-Time Project Integration« auf Seite 369).

Die Vorteile des Backens ergeben sich aus den Nachteilen des Ausbackens. Es ist besonders sinnvoll, die Konfiguration in Server-Images zu backen, die viele gleiche Server-Instanzen nutzen, und wenn Sie dazu in der Lage sein müssen, Server häufig und schnell erzeugen zu müssen.

Eine Herausforderung beim Backen von Server-Images ist, dass Sie das Tooling und die Automations-Prozesse aufsetzen müssen, mit denen es leicht wird, neue Versionen zu aktualisieren und auszurollen. Wird beispielsweise ein wichtiger Sicherheits-Patch für Ihr Betriebssystem oder für zentrale Pakete, die in Ihre Server-Images gebacken sind, veröffentlicht, wollen Sie schnell neue Images bauen und sie ohne große Unterbrechung auf Ihre Server ausrollen können. Das ist Thema von Kapitel 13.

Backen und Ausbacken kombinieren

In der Praxis nutzen die meisten Teams eine Kombination aus Backen und Ausbacken, um neue Server zu konfigurieren. Sie können einen Mittelweg dafür finden, welche Konfiguration Sie im Server-Image vornehmen und welche beim Erstellen einer Server-Instanz. Manche Konfigurationselemente lassen sich auch in beiden Teilen des Prozesses anwenden.

Die wichtigsten Überlegungen bei der Frage, wann die richtige Zeit zum Anwenden einer bestimmten Konfiguration ist, sind der Zeitaufwand für die Änderung und deren Häufigkeit. Dinge, deren Anwenden länger braucht und die sich selte-

ner ändern, sind klare Kandidaten für das Backen in das Server-Image. So können Sie beispielsweise Anwendungsserver-Software auf ein Server-Image installieren und damit das Hochfahren mehrerer Server-Instanzen deutlich beschleunigen und in diesem Prozess Netzwerk-Bandbreite einsparen.

Am anderen Ende dieser Skala liegen Dinge, die sich schneller installieren lassen oder häufiger ändern – diese werden besser ausgebacken. Ein Beispiel dafür sind Anwendungen, die in der Firma entwickelt wurden. Einer der häufigsten Anwendungsfälle für das Hochfahren neuer Server bei Bedarf ist das Testen als Teil eines Software-Release-Prozesses.

Hochproduktive Entwicklungsteams sorgen vielleicht für ein Dutzend oder mehr neue Builds ihrer Anwendung pro Tag, wobei sie auf einen CI-Prozess setzen, der jeden Build automatisch deployt und testet. Das Backen eines neuen Server-Image für jeden neuen Anwendungs-Build ist für diese Arbeitsgeschwindigkeit zu langsam, daher ist es effizienter, die Anwendung beim Erstellen des Testservers zu deployen.

Eine andere Möglichkeit, wie Teams Backen und Ausbacken kombinieren, ist, so viel wie möglich in Server-Images zu backen, aktualisierte Versionen aber auszubacken. Ein Team backt vielleicht Server-Images mit niedrigerer Geschwindigkeit – wöchentlich oder monatlich. Müssen sie etwas aktualisieren, das sie normalerweise in das Image backen würden, wie zum Beispiel einen Sicherheits-Patch oder Verbesserungen an der Konfiguration, können sie dies in den Prozess zum Erstellen der Server stecken, um es zusammen mit dem gebackenen Image auch noch auszubacken.

Ist der Zeitpunkt gekommen, ein aktualisiertes Image zu backen, stecken sie die Updates mit hinein und entfernen sie gleichzeitig aus dem Erstellungsprozess. So kann das Team schnell neue Änderungen übernehmen, ohne sich viel Overhead aufzuhalsen. Teams nutzen das oft in Kombination mit dem kontinuierlichen Anwenden von Code (siehe »Pattern: Continuous Configuration Synchronization« auf Seite 232), um bestehende Server zu aktualisieren, ohne sie neu bauen zu müssen.

Serverkonfiguration beim Erstellen eines Servers anwenden

Die meisten Tools, die zum Erstellen von Servern auf den oben besprochenen Wegen zum Einsatz kommen (siehe »Eine neue Server-Instanz erstellen« auf Seite 216) – seien es Befehlszeilentools, Plattform-API-Aufrufe oder ein Stack-Management-Tool, bieten eine Möglichkeit an, Code zur Serverkonfiguration anzuwenden. So sollte beispielsweise ein Stack-Management-Tool eine Syntax zum Unterstützen verbreiteter Tools oder zum Ausführen eines beliebigen Befehls auf einer neuen Server-Instanz besitzen, wie in »Stack-Code führt mein fiktives Server-Konfigurationstool aus.« auf Seite 227 gezeigt wird.

Listing 11-3: Stack-Code führt mein fiktives Server-Konfigurationstool aus.

```
server:
  source_image: stock-linux-1.23
  memory: 2GB
  vnet: ${APPSERVER_VNET}
  configure:
    tool: servermaker
    code_repo: servermaker.shopspinner.xyz
    server_role: appserver
    parameters:
      app_name: catalog_service
      app_version: 1.2.3
```

Dieser Code führt das Servermaker-Tool aus, übergibt ihm den Hostnamen des Servers, der den Code für die Serverkonfiguration hostet, die Rolle, die auf den Server angewendet werden soll (`appserver`), und ein paar Parameter für den Serverkonfigurationscode (`app_name` und `app_version`).

Manche Tools erlauben es Ihnen auch, Serverkonfigurationscode oder auszuführende Shell-Befehle direkt in den Code für den Stack einzubetten. Es kann verlockend sein, darüber die Logik für die Serverkonfiguration zu implementieren, und für einfache Situationen kann das auch in Ordnung sein. Aber in den meisten Fällen wächst dieser Code bezüglich Größe und Komplexität an. Daher ist es besser, den Code zu extrahieren, um Ihre Codebasis sauber und wartbar zu halten.

Zusammenfassung

Dieses Kapitel hat eine Reihe von Aspekten beim Erstellen und Provisionieren neuer Server angesprochen. Zu den Elementen auf einem Server gehören Software, Konfiguration und Daten. Diese werden meist aus der Installation des Basis-Betriebssystems und Paketen diverser Repositories geholt, unter anderem aus denen, die vom Betriebssystem und von Sprachanbietern, Fremdfirmen und internen Teams verwaltet werden. Typischerweise bauen Sie einen Server, indem Sie Inhalte aus einem Server-Image kombinieren und ein Server-Konfigurationstool nutzen, um zusätzliche Pakete und Konfiguration anzuwenden.

Um einen Server zu erstellen, können Sie ein Befehlszeilentool ausführen oder eine Benutzeroberfläche nutzen, aber es ist besser, einen Code-gesteuerten Prozess zu verwenden. Heutzutage werden Sie eher kein eigenes Skript dafür schreiben, sondern Ihr Stack-Management-Tool verwenden. Auch wenn ich ein paar unterschiedliche Ansätze zum Bauen von Servern beschreibe, empfehle ich im Allgemeinen, Server-Images zu bauen.