www.javaspektrum.de

Ausgabe 6/2020
D € 9,90 · A € 10,90 · CHF 18,20

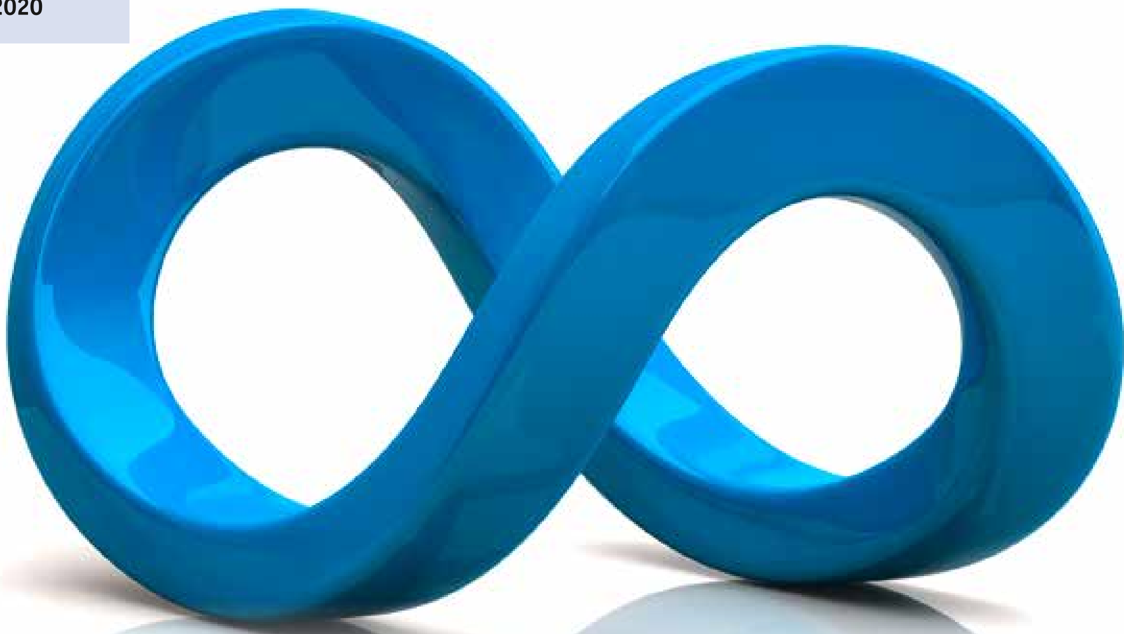
06
4 194156 109901

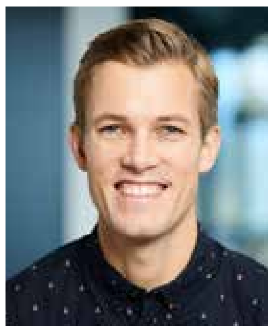
JavaSPEKTRUM

Magazin für professionelle Entwicklung und digitale Transformation

DevOps – Patterns und Praktiken

Sonderdruck aus JavaSPEKTRUM 6/2020





Wie DevOps mit Kubernetes Qualitätssicherungen ermöglicht

Automatische Verteilung v... Microservices – ein P...

Interview

André Christ, Gründer von LeanIX über die zentrale Rolle der Architektur und den W... zur produktorienti...

Fach...

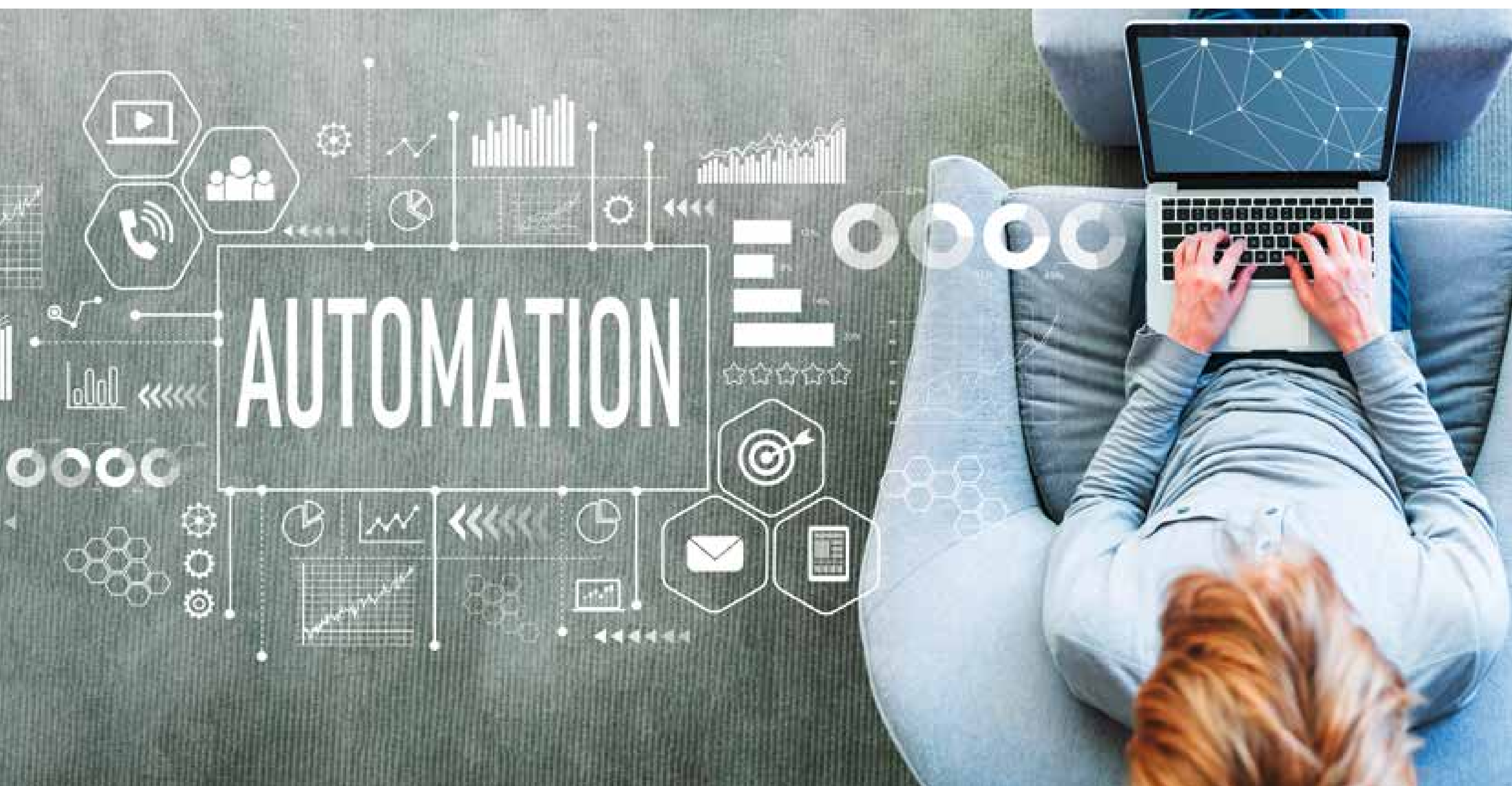
...k für Data Science ...on und JavaScript ...für die Digitalisierung ...hilfe der richtigen Architektur

Sonderdruck für CONSIST Business Information Technology

DevOps – Patterns und Praktiken

G30325

Sonder... ...sting Magazin „Lean Quality Management"



(Copyright/Quelle: ©Tierney - stock.adobe.com)

Ganz automatisch

DevOps in der Praxis – Microservices verteilen mit Terraform

Matthias Hänsgen

Die Verteilung von Microservices ist eine komplexe Herausforderung, die nur vollautomatisiert effizient erfolgen kann. Manuelle Arbeitsschritte innerhalb dieses Prozesses sind langsam und fehleranfällig und stellen somit ein grundlegendes Problem dar. Eine Lösung ist der Einsatz von Infrastructure-as-Code (IaC)-Werkzeugen zur Automatisierung dieses Prozesses. Der Artikel stellt eine Best Practice zur Verteilung von Microservices mit Fokus auf Spring Boot und Terraform vor.

Auch in der Welt der monolithischen Softwarearchitekturen stellt sich der Softwareverteilungsprozess als ein komplexer Vorgang dar. In der Zeit vor DevOps war dieser Prozess durch die folgenden Randbedingungen geprägt:

- Trennung der Verantwortlichkeiten zwischen Entwicklung (Dev) und Betrieb (Ops), was in der Realität zu entsprechenden Reibungsverlusten führte,
- langwierige Prozesse zur Beschaffung, Installation und dem Management von Hardware,
- viele manuelle Prozess-Schritte, die langsam, fehleranfällig und oftmals nicht sicher reproduzierbar waren,
- langer Zeitraum bis zur Produktivsetzungen einer neuen Softwareversion (Time-to-Market),
- nur wenige Releases pro Jahr mit einem hohen Fehlerrisiko.

Die Herausforderungen bei der Softwareverteilung von Microservices

In einer Microservices-Architektur potenziert sich dieses Problem, da nicht ein Service (Monolith) mit allen seinen beteiligten Komponenten (z. B. Datenbank) betrieben werden muss, sondern in Services. In der Praxis verschärft sich dieses Problem noch, da nicht nur eine Instanz der Applikation pro Release ausgerollt werden muss, sondern viele (Testumgebung Entwickler, Testumgebung Fachbereich, Produktivumgebung, ggf. mit Blue-Green-Deployment usw.). Hieraus ergibt sich für den Produktivbetrieb von Microservices-Applikationen die zwingende Notwendigkeit einer automatisierten Softwareverteilung. Basis hierfür sind die Konzepte und Werkzeuge aus der DevOps-Bewegung.

IaC: der Schlüssel zur Automatisierung

Die Kernidee hinter Infrastructure as Code (IaC) ist, dass sich jede Infrastrukturkomponente durch Software erzeugen, konfigurieren und zerstören lässt. Dieser Ansatz eröffnet völlig neue Möglichkeiten für die Softwareverteilung und den Betrieb von Applikationen, da sich diese Prozesse nun vollständig durch Software automatisieren lassen. Grundlage hierfür sind IaC-Werkzeuge, die die folgenden Aufgaben unterstützen:

Matthias Hänsgen arbeitet als Systemarchitekt für die Consist Software Solutions GmbH. Seit über 25 Jahren entwickelt er geschäftskritische Anwendungen auf Basis von objektorientierten Technologien. Sein aktueller Tätigkeitsschwerpunkt liegt in der Entwicklung von Microservices-Anwendungen.
E-Mail: haensgen@consist.de

- Konfigurationsmanagement,
- Server Templating,
- (Anwendungs-)Orchestrierung,
- Provisionierung von Ressourcen.

Konfigurationsmanagement-Werkzeuge dienen zur Installation und zum Management von Software auf bestehenden Servern. Sie stellen sicher, dass auf einem Server die richtige Software in der richtigen Version und in der richtigen Konfiguration vorhanden ist.

Server-Templating-Werkzeuge dienen zur Erstellung eines Server- oder Container-Images mit einem spezifischen Softwareinstallationsstand. Beide Werkzeugtypen dienen prinzipiell demselben Zweck, einen Server/Container in einem spezifischen Softwarestand bereitzustellen. Sie unterscheiden sich aber in der Arbeitsweise und dem Zustand der erzeugten Infrastruktur.

Ein Orchestrierungswerkzeug organisiert und koordiniert den Betrieb der verschiedenen Einzel-Services/-Dienste, aus denen eine gesamte Applikation besteht. Hierzu gehören Aufgaben wie das Ausrollen von Softwareversionen, Lastverteilung, Skalierung, Überwachung und Neustart von Services.

Die Erzeugung von Infrastrukturkomponenten wie Servern, Netzwerken, Datenbanken, Caches usw. erfolgt mittels spezieller Provisionierungswerkzeuge. Diese Werkzeug-Palette bildet die Grundlage für die Automatisierung der Softwareverteilung. Für die zielgerichtete Auswahl eines (oder mehrerer) IaC-Werkzeuge ist eine Betrachtung der wesentlichen Differenzierungsmerkmale von grundlegender Bedeutung (s. Abb. 1).

IaC-Werkzeuge – Was sind die prinzipiellen Unterschiede?

Aus dem Blickwinkel der automatisierten Softwareverteilung sind die folgenden Kriterien von maßgeblicher Relevanz für die Bewertung von IaC-Werkzeugen:

- Welche Funktionalitäten zur Provisionierung und Konfiguration bietet ein Werkzeug?
- Ist die erzeugte Infrastruktur veränderbar oder nicht?

Konfigurationsmanagement	Server Templating
• Ansible • Chef • Puppet • SaltStack	• Docker • Packer • Vagrant

(Anwendungs)-Orchestrierung	Provisionierung
• Amazon Elastic Container Service (Amazon ECS) • Docker Swarm • Kubernetes • Marathon / Mesos	• CloudFormation • OpenStack Heat • Terraform

Abb. 1: Exemplarische Übersicht IaC-Werkzeuge

- Auf welchem Programmier-Paradigma basiert das Werkzeug?
- Erfordert das Werkzeug die Installation zusätzlicher Software für den Betrieb?
- Wie ausgereift und verbreitet ist das Werkzeug?

Die Grenze zwischen Konfigurationsmanagement- und Provisionierungswerkzeugen ist fließend. Jeder der beiden Typen bietet auch Funktionalitäten des jeweils anderen Typs an. Basis für eine Automatisierung der Softwareverteilung ist ein Provisionierungswerkzeug, das in der Lage ist, für die Ziel-Plattform (z. B. AWS Cloud) alle benötigten Infrastrukturkomponenten bereitzustellen und zu verwalten. Wenn dieses Werkzeug darüber hinaus grundlegende Konfigurationsmanagement-Funktionalitäten bietet, kann auf ein zusätzliches Konfigurationsmanagement-Werkzeug verzichtet werden. Hierdurch wird in der Praxis der Softwareverteilungsprozess vereinfacht.

Wie bereits beschrieben, dienen sowohl Konfigurationsmanagement-Werkzeuge als auch Server-Templating-Werkzeuge der Bereitstellung von Servern/Container mit einem definierten Softwarestand. Sie unterscheiden sich aber in der Arbeitsweise, da die über Konfigurationsmanagement-Werkzeuge erstellten Komponenten veränderbar sind, die über Server Templating erzeugten nicht (immutable). Der Ansatz auf Basis von „Immutable Infrastructure" sorgt in der Praxis für Klarheit, da immer ein definierter, reproduzierbarer Stand in Betrieb ist. Unbeabsichtigte, manchmal subtile Konfigurationsänderungen mit Seiteneffekten an einem Produktivsystem werden hierdurch vermieden.

IaC-Werkzeuge nutzen zur Automatisierung von Funktionalitäten eine Programmiersprache. Hierbei kommen die folgenden Programmier-Paradigmen zum Einsatz:

- Prozedurale Programmierung
- Deklarative Programmierung

Die prozedurale Programmierung definiert die (Arbeits-)Schritte, wie ein bestimmtes Ergebnis erreicht wird (z. B.: Erzeuge einen Server vom Typ X). Bei der deklarativen Programmierung steht die Beschreibung des Ergebnisses/Problems im Vordergrund (z. B.: Es sollen immer 5 Server-Instanzen vom Typ X vorhanden sein). Da Infrastrukturkomponenten in großem Maße zustandsbehaftet sind, ist die prozedurale Programmierung für diesen Anwendungsfall deutlich komplexer und schlechter zu modularisieren als eine deklarative Lösung. Deklarative Programmierung basiert auf einer umfassenden Abstraktion des zugrunde liegenden Problems, was eine längere Einarbeitung erfordert als der Einstieg in eine prozedurale Programmiersprache. Nachdem die Einstiegshürde für eine deklarative Programmiersprache erklommen wurde, führt dies in der Praxis aber zu einfacherem und wartbarerem Code.

Einige Werkzeuge erfordern die Installation von zusätzlicher Software unter anderem auf den administrierten Servern. Hierbei kann es sich um einen zusätzlichen Master-Server handeln, der Informationen zentral bereitstellt und verwaltet, oder auch um Agenten, die auf den jeweiligen Servern installiert werden müssen. Ein solcher Ansatz bedingt immer zusätzliche Wartungsaufwände für den Betrieb dieser Komponenten und erzeugt darüber hinaus ein zusätzliches, potenzielles Sicherheitsrisiko durch eine vergrößerte Angriffsfläche.

Neben den rein technischen Aspekten eines Werkzeugs sind daneben auch die Ausgereiftheit und dessen Zukunftsperspektiven von Bedeutung. Indikatoren hierfür sind die Verbreitung des Produkts, die Größe und Agilität der Community und wie lange ein Produkt bereits im praktischen Einsatz ist.

Die Eignung und Bewertung eines spezifischen Werkzeugs ist in hohem Maße von den Charakteristika des zu automatisierenden Szenarios abhängig. Grundlegende Faktoren sind hierbei die Ziel-

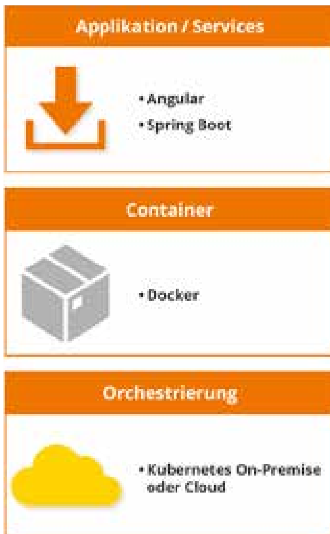


Abb. 2: Exemplarische Microservices-Architektur

plattform, auf der eine Applikation betrieben werden soll, und deren Architektur.

Die Basis - Entscheidung für eine Microservices-Architektur

Die Wahl einer konkreten Architektur ist immer eine sehr individuelle Entscheidung eines Unternehmens. Sie wird sowohl geprägt durch deren Historie (z. B. Programmiersprachen-Know-how) als auch durch aktuelle Rahmenbedingungen (was fordert der Markt, Plattformen bei Kunden usw.). In dem Anwendungsfall stellten sich die Rahmenbedingungen wie folgt dar:

- langjähriges Know-how in Projekten, basierend auf dem Spring Framework,
- die Kunden nutzen sowohl monolithische als auch Microservices-basierte Lösungen in Abhängigkeit von der Projektgröße,
- fundiertes Know-how in der Entwicklung von User Interfaces auf Basis von Angular,
- die Applikationen werden derzeit sowohl On-Premise als auch in der Cloud betrieben.

Basierend auf diesen Prämissen, wurde eine (Microservices-)Architektur und Zielplattform (s. Abb. 2) mit den folgenden Eckpunkten definiert:

- Nutzung von Spring Boot sowohl zur Entwicklung monolithischer Applikationen (kleine Anwendungen) als auch von Microservices-Applikationen (große Anwendungen),
- Angular als User Interface Framework,
- Docker als eine einheitliche (Container-basierte) Laufzeitumgebung für alle Zielplattformen (On-Premise, Cloud),
- Kundenspezifische Docker-basierte Orchestrierungsplattform.

Der Einsatz von Spring Boot bietet den Vorteil, für die unterschiedlichen Applikationsarchitekturen ein einheitliches Framework einzusetzen. Cloud-Spezifika können mittels Spring Cloud abgebildet werden.

Der Einsatz von Docker ist ein leichtgewichtigerer Ansatz als die Verwendung von virtuellen Maschinen und reduziert den Overhead in den Entwicklungszyklen durch kürzere Erzeugungszeiten für die Docker Images und durch kürzere Startzeiten. Darüber hinaus bietet dieser Ansatz den Vorteil, dass die Betriebssystemspezifika abstrahiert werden und die erzeugte Laufzeitumgebung immutable ist (in Analogie zu Server Templating).

Die Orchestrierungsplattform einer Applikation hat in der Praxis die größten Auswirkungen auf deren Betrieb. Eine umfassende Plattform bietet umfangreiche Funktionalitäten wie automatisches Deployment ohne Unterbrechung, automatisierte Skalierung, Überwachung und gegebenenfalls Neustarten von defekten Komponenten. Diese Funktionalitäten sind nur aufwendig selbst zu realisieren, sodass der Einsatz einer Orchestrierungsplattform für eine Microservices-Architektur ein Muss ist. Für Kubernetes spricht die Verfügbarkeit auf vielen Cloud-Plattformen als SaaS und die Möglichkeit, alternativ für eine On-Premise Lösung selbst ein Kubernetes Cluster zu betreiben. Ein Manko hierbei ist die Komplexität der Installation und des Betriebs eines solchen Clusters On-Premise.

Im nächsten DevOps-Schritt muss nun für diese konkrete Anwendungsarchitektur und Laufzeitumgebung im Rahmen der CI/CD-Aktivitäten eine automatisierte Softwareverteilung konzipiert werden.

Die passende Plattform für die automatisierte Softwareverteilung

Die Auswahl einer spezifischen Plattform mit ihren Werkzeugen ist in hohem Maße von den individuellen Prioritäten eines Unternehmens/Teams abhängig. In diesem Anwendungsfall waren die wichtigsten Einflussfaktoren die folgenden:

- Die Infrastruktur soll aus Robustheitsgründen „immutable“ erzeugt werden.
- Einfachheit der Lösung – möglichst wenige Standard-Werkzeuge mit einem geringen Anteil an Individuallösungen und Anpassungen.
- Ein einheitliches Verfahren und Werkzeug-Set für alle Ziel-Plattformen (On-Premise, verschiedene Cloud-Anbieter).
- Automatisierung des CI-/CD-Prozesses auf Basis des Open-Source-Automation-Servers Jenkins.

Da durch die Architekturentscheidung bereits der Fokus auf dem Einsatz von Server Templating gerichtet war, hat das Konfigurationsmanagement bei der Entscheidung für eine Werkzeug-Lösung nur eine untergeordnete Rolle gespielt. Kern der Automatisierung ist ein leistungsfähiges Provisionierungswerkzeug. Die Entscheidung für Terraform basierte hierbei auf den folgenden Gründen:

- Nutzung einer deklarativen Programmiersprache,
- Erzeugungsmöglichkeiten für alle benötigten Infrastrukturkomponenten (z. B. Datenbanken, Cloud-Ressourcen usw.),
- Unterstützung verschiedener Cloud-Anbieter,

On-Premise	Cloud
Server-Templating-Applikation	
• Docker	• Docker
Orchestrierung	
• Kubernetes • Server Templating - Packer	• Kubernetes SaaS - Amazon EKS - Azure Kubernetes Service - Google Kubernetes Engine
Provisionierung und Konfiguration	
• Terraform - Netzwerk - Datenbanken - Benutzer - Zugriffsrechte etc.	• Terraform - Netzwerk - Datenbanken - Benutzer - Zugriffsrechte etc.

Abb. 3: Exemplarische Plattform für die automatisierte Softwareverteilung

Aufbau Terraform Modul

- Jedes Verzeichnis in einem Terraform-Projekt ist ein Modul
- Jedes Modul hat
  - Eingangsparameter (variable),
  - Ausgangsparameter (output) und
  - Ressourcendefinitionen (resource).

Modul-Definition in ./templates/ecs-service/main.tf

```
// define input parameter
variable "service_name" {
  description = "the name of the service"
}
resource " aws_ecs_service" "ecs_service_template" {
  name = var.service_name
// uses variable  ...  // more parameters must be defined  ...
}
// define output parameter
output "service_arn" {
// ressources can be referenced by its name
 value       = aws_ecs-service.ecs_service_template.service_name
 description = "the name of the service"
}
```

Modul-Verwendung in ./services/service1/ecs-service.tf

```
// create ecs-service
module “service1" {
  source         = "templates/ecs-service"
  service_name   = "myService1"
}
```

Abb. 4: Aufbau Terraform-Modul

- befriedigende Konfigurationsmanagement-Funktionalitäten.

Als unterstützendes Server-Templating-Werkzeug zur Bereitstellung von Kubernetes On-Premise wurde aufgrund der Eignung für den Produktivbetrieb und der Erzeugung von Immutable-Infrastrukturen Packer ausgewählt (s. Abb. 3).

Best Practice – Terraform in der Praxis

Die Praxis hat gezeigt, dass für einen effektiven Terraform-Einsatz die Beachtung der folgenden Punkte von grundlegender Bedeutung ist.

Terraform State File und Remote Backends

Terraform speichert den Zustand seiner Ausführung und damit auch den Zustand der provisionierten und konfigurierten Infrastruktur in einem State File. Die Verfügbarkeit und Integrität dieser Datei ist von zentraler Bedeutung für Terraform. Für Produktivumgebungen oder Umgebungen, auf die mehrere Personen administrativen Zugriff haben, muss diese Datei an einer zentralen, sicheren Stelle gespeichert werden. Das bedeutet, dass der State File geschützt werden müssen vor versehentlichen Änderungen, gleichzeitigem Zugriff mehrerer Benutzer und unbefugter Einsicht in die im Klartext enthaltenen Passwörter. Hierfür bietet Terraform vorgefertigte Remote Backends, die die sichere Speicherung in verschiedenen Umgebungen ermöglichen (z. B. AWS S3 Bucket).

Abbildung von Stages in Terraform

In einem nächsten Schritt muss konzeptionell das Problem der Handhabung der State Files für die verschiedenen Stages einer Applikation gelöst werden. Ein Ansatzpunkt hierfür ist die Nutzung von Terraform-Workspaces zur Abbildung der verschiedenen Stages. Der Nachteil hierbei ist, dass alle Workspace-Varianten/Stages als verschiedene State Files innerhalb eines zentralen Remote Backends gespeichert werden – das heißt, es gibt zum Beispiel keine physische Trennung zwischen Test- und Produktions-Stage. Eine Abbildung von Stages für den Produktivbetrieb mittels Workspaces ohne zusätzliche Maßnahmen ist nicht empfehlenswert, da keine echte Isolation der einzelnen Stages gegeben ist. Hierdurch kann bereits durch eine kleine Unachtsamkeit ein Ausfall des gesamten Produktivsystems erfolgen.

Die einfachste Lösung für dieses Problem ist die Abbildung und Isolation der einzelnen Stages über eine Terraform-Directory-Struktur mit je einem Subdirectory pro Stage. Jedes Stage-Subdirectory enthält den Terraform-Sourcecode für die jeweilige Stage und es wird bei der Terraform-Ausführung jeweils ein separater Remote State File erzeugt, der auch über eine eigene Autorisierung verfügen kann (z. B. Trennung Zugang Test-System/Produktiv-System). Ohne weitere Maßnahmen würde dieser Ansatz jedoch zu einer erheblichen Code-Duplizierung zwischen den einzelnen Stages führen. Um dies zu verhindern, bietet Terraform den Modul-Mechanismus an.

Terraform-Module

Terraform-Modules (s. Abb. 4) unterstützen die Möglichkeit, Funktionsbibliotheken mit wiederverwendbaren Bausteinen aufzubauen. Im konkreten Microservices-Anwendungsfall wurden zur Strukturierung des Terraform-Codes die folgenden Modul-Typen verwendet:

- Resource-Definition-Modul,
- Infrastructure-Layer-Modul,
- Microservices-Modul.

Resource-Definition-Module dienen der zentralen Definition von Ressourcen, die Microservices übergreifend genutzt werden. Im konkreten Fall, der in einer Amazon-AWS-Umgebung betrieben wird, fallen hierunter zum Beispiel die Definitionen von zentralen Message-Queues für den Datenaustausch zwischen Microservices.

Die Infrastructure-Layer-Module dienen dazu, die verschiedenen Schichten (Netzwerk, Security, Datenbank, eigentliche Applikation) einer Applikation abzubilden. Da die Erzeugung von

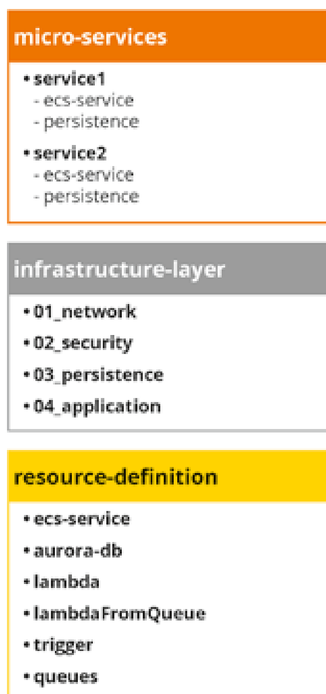


Abb. 5: Terraform-Module - Beispielstruktur

Infrastrukturkomponenten und Datenbanken in der Regel zeitaufwendig ist, bietet es sich unter anderem zur Beschleunigung von Entwicklertests an, auch die Terraform-Module zu „schichten". Hierdurch ist es zum Beispiel beim Testen möglich, nur die Applikationsschicht zu zerstören und neu zu deployen – ohne die unterliegende Infrastruktur zeitaufwendig neu zu erzeugen (s. Abb. 5).

Die Layer unter `infrastructure-layer` müssen der Reihenfolge nach aufsteigend ausgeführt werden, um eine Anwendungsinstanz aufzubauen, und in der umgekehrten Reihenfolge ausgeführt werden, um die Instanz wieder abzubauen. Dazu werden die Terraform-Befehle `apply` und `destroy` in jedem Layer-Verzeichnis/Modul ausgeführt.

Die Microservices-Module beinhalten alle Terraform-Definitionen für die konkreten Microservices-Komponenten, die nicht über die Infrastructure-Layer- und Resource-Definition-Module bereitgestellt werden (jeder Microservice ist hierbei ein eigenes Modul). Zur Minimierung der Cloud-Kosten im Rahmen von Entwicklertests wurden die Layer `03_persistence` und `04_application` separiert, sodass die Applikation gestoppt/zerstört werden kann, ohne die Daten zu verlieren.

Die Versorgung der einzelnen Terraform-Module mit Ausgabeparametern aus anderen Modulen in einer Remote-Backend-Umgebung wurde durch die Verwendung von Terraform-Remote-State-Data-Sources gelöst.

Terraform – Sourcecode-Management und Versionierung

Der Terraform-Sourcecode muss wie jeder andere produktive Sourcecode vollständig und versioniert in einem Sourcecode-Management-System verwaltet werden (z. B. Git). Darüber hinaus bietet Terraform die Möglichkeit, seine eigenen Module als Git-Module zu verwalten. So können Terraform-Module direkt über entsprechende Git-Referenzen miteinander verbunden werden. Im konkreten Fall wurden von einem Kernteam komplexe Infrastruktur-Basis-Funktionalitäten als zentrale Terraform Module entwickelt und in einem Git-Repository zur Verfügung gestellt. Diese Module wurden dann von den Entwickler-Teams zur Erstellung der Microservices-spezifischen Terraform-Module genutzt.

Terraform – Was ist sonst noch zu beachten?

Die sichere, verschlüsselte Speicherung von Autorisierungsinformationen (z. B. Datenbank-Benutzer und dessen Passwort) im Rahmen einer automatisierten Softwareverteilung ist ein prinzipielles Problem. Terraform unterstützt dies durch die Anbindung verschiedener, sicherer Speichermedien (z. B. AWS Secrets Manager). Ein weiterer Punkt, der für Produktivumgebungen berücksichtigt werden muss, ist, dass persistente Informationen wie Datenbankinhalte, Logdateien usw. beim Zerstören der Umgebung erhalten bleiben müssen.

Für die Entwicklung und das Testen von Terraform ist die Erkennung von Abweichungen an einer deployten Stage in Bezug zur definierten Konfiguration (Drift) nützlich. Terraform unterstützt die Erkennung durch den Befehl `terraform refresh`.

Abschließend sei noch erwähnt, dass eine produktionsreife, automatisierte Softwareverteilung generell ein (zeit-)aufwendiges Unterfangen ist. Erste Erfolge, indem man „schnell" eine kleine Applikation ausgerollt hat, täuschen oft. Um die Softwareverteilung einer durchschnittlichen Applikation in der Praxis vollständig und stabil zu automatisieren, vergehen in der Regel mehrere Monate.

Fazit und Ausblick

Der vorliegende Artikel beschreibt anhand eines Praxisbeispiels auf Basis von Spring Boot, wie eine Microservices-Applikation automatisiert verteilt und betrieben werden kann. Schlüssel hierfür ist der Einsatz von IaC-Werkzeugen für die Erzeugung, Konfiguration und das Zerstören „beliebiger" Infrastrukturkomponenten mittels Software.

Kernidee der Lösung ist der Einsatz von Server Templating zur Erzeugung von Immutable Infrastructure. Hierdurch kann auf ein zusätzliches, dediziertes Konfigurations-Management-Werkzeug verzichtet werden, was die Gesamtlösung technisch vereinfacht. Die Provisionierung und Konfiguration der Applikations-Ressourcen erfolgt mittels Terraform. Ausschlaggebend für die Auswahl ist das Terraform-Provider-Konzept, das transparent den Betrieb der Applikation auf unterschiedlichen (Cloud-)Zielplattformen unterstützt.

Die Grundlage der Best Practice beim Einsatz von Terraform ist die Verwendung einer Directory-Struktur, die die einzelnen Stages einer Applikation isoliert, um somit Seiteneffekte im Betrieb zu verhindern. Darüber hinaus ist die Nutzung des Terraform-Modul-Konzepts zur Erzeugung von wartbarem Code und zur Vermeidung von Code-Duplizierung von fundamentaler Bedeutung. Im Praxisbeispiel wurden Module zur Definition von wiederverwendbaren Ressourcen, von Infrastruktur-Komponenten in Form von Layern und zur Beschreibung vollständiger Microservices verwendet.

Durch die DevOps-Bewegung und den Einsatz von IaC hat sich in den letzten Jahren bereits eine dramatische Verbesserung bezüglich Geschwindigkeit, Häufigkeit und Qualität der Softwareverteilung ergeben. Da die Entwicklung in diesem Bereich rasant verläuft, ist sicherlich auch in den nächsten Jahren mit spannenden Neuerungen zu rechnen.

Literatur und Links

[Bri19] Y. Brikman, Terraform: Up & Running, 2nd Edition, O'Reilly Media, Inc., 2019
[Kim16] G. Kim, J. Humble, P. Debois, J. Willis, The DevOps Handbook, IT Revolution Press, 2016
[Tur17] J. Turnbull, The Packer Book, Turnbull Press, 2017