SERVERLESS DEPLOYMENTS MIT JENKINS X, TEIL 1

Mit ohne Server

Build und Verteilung von Softwareprojekten mittels Serverless Computing.

Der Name Serverless ist völlig irreführend, da er den Eindruck erweckt, dass kein Server beteiligt ist. Aber selbstverständlich sind bei diesem Konzept Server beteiligt. Nur erlaubt Serverless Computing den Nutzern, deren Existenz zu ignorieren. Gerade beim Bau von Anwendungen und der Verteilung kann diese Technologie helfen.

Um aber die neue Spielart zu verstehen, muss man sich zunächst mit den Hürden auseinandersetzen, die man normalerweise beim traditionellen Deployment von Anwendungen überspringen muss.

Früher lief das Deployment von Anwendungen meist direkt auf Servern ab. Man musste die Größe (Speicher und CPU) der Nodes bestimmen, auf denen die Anwendung laufen sollte. Auch mussten die Server zunächst eingerichtet und dann instand gehalten werden.

Mit dem Einzug von Cloud Computing verbesserte sich die Situation. Die Aufgaben waren die gleichen, doch die Schnittstellen und Services der Cloud-Anbieter haben viele Prozesse vereinfacht und beschleunigt.

Die Möglichkeiten stiegen damit ins Unendliche – zumindest gefühlt: Einfach einen Befehl ausführen, und wenige Minuten später sind die Server als virtuelle Maschinen (VMs) erstellt. Konzepte wie die der Unveränderlichkeit wurden ebenfalls Mainstream. Eine höhere Verlässlichkeit, eine deutlich geringere Durchlaufzeit sowie viel mehr Elastizität waren die Vorteile.

Dennoch blieben damit einige Fragen unbeantwortet: Sollten Server weiterlaufen, selbst wenn keiner auf die Anwendungen zugreift? Falls nein, wie stellt man sicher, dass die Anwendungen dennoch leicht verfügbar sind, wenn man sie wieder braucht?

Wer ist für die Wartung der Server verantwortlich? Das eigene Infrastruktur-Team oder der Cloud-Anbieter? Oder lässt sich ein System erstellen, das diese Aufgaben ohne menschliche Intervention erledigt?

Die Lage änderte sich mit dem Aufkommen von Containern und Schedulern. Nach einigen Jahren der Unsicherheit, die dadurch entstanden war, dass es zu viele Optionen gab, sta-

bilisierte sich die Situation rund um Kubernetes [1], das zum De-facto-Standard wurde.

Parallel zur zunehmenden Beliebtheit von Containern und Schedulern begannen sich Lösungen für Serverless-Computing-Konzepte zu entwickeln. Diese Lösungen standen in keinem Zusammenhang zueinander – genauer gesagt, nicht in den ersten Jahren. Kubernetes bot die Möglichkeit, sowohl Microservices als auch herkömmlichere Arten von Anwendungen auszuführen. Serverless hingegen konzentrierte sich darauf, Funktionen ablaufen zu lassen, die oft nur aus wenigen Zeilen Code bestanden.

Nicht für alle Szenarien gut

Die großen Cloud-Anbieter (AWS, Microsoft Azure und Google) haben alle Lösungen für Serverless Computing im Angebot. Entwickler könnten sich also auf das Schreiben von Funktionen mit ein paar zusätzlichen Codezeilen konzentrieren, die speziell für den Anbieter von Serverless Computing entwickelt wurden. Alles andere, was zum Ausführen und Skalieren dieser Funktionen benötigt wird, wird transparent.

Aber nicht alles funktioniert in der serverlosen Welt einwandfrei. Die Anzahl der Anwendungsfälle, die durch Schreiben von Funktionen – im Gegensatz zu Anwendungen – realisiert werden können, ist begrenzt. Selbst wenn wir genügend Anwendungsfälle haben, um den Aufwand für Serverless Computing zu rechtfertigen, lauert ein noch bedeutenderes Problem gleich um die Ecke.

Die Wahrscheinlichkeit einer Abhängigkeit von einem Anbieter (Vendor Lock-in) ist groß, da es noch keine anerkannten Industriestandards gibt. Egal ob AWS Lambda, Azure Functions oder Google Cloud Functions: Der Code wird nicht von einem Anbieter auf einen anderen übertragbar sein.

Das bedeutet nicht, dass es keine serverlosen Frameworks gibt, die nicht an einen bestimmten Cloud-Provider gebunden sind. Es gibt sie, aber man müsste sie selbst pflegen, sei es On-Prem oder innerhalb von Clustern in einer Public Cloud. Damit entfällt einer der wesentlichen Vorteile von Serverless-Konzepten. Aber genau an dieser Stelle setzt Kubernetes an.

Verteil mich!

Nimmt man an, dass der Großteil der Unternehmen mindestens eine, wenn nicht alle Anwendungen mit Kubernetes laufen lässt und Kubernetes sich damit zur Standardschnittstelle entwickelt, dann besitzt früher oder später auch jedes Unternehmen einen Kubernetes-Cluster.

Sie alle werden viel Zeit mit der Wartung verbringen, und jeder wird über ein gewisses Maß an Fachwissen darüber verfügen, wie es funktioniert.

Wenn diese Annahme zutrifft, liegt es nahe, dass Kubernetes die beste Wahl einer Plattform wäre, um auch Serverless-Anwendungen auszuführen. Dadurch wird die Bindung an den Anbieter umgangen, da Kubernetes eben (fast) überall laufen kann.

Kubernetes-basiertes Serverless Computing würde noch eine ganze Reihe weiterer Vorteile bieten. Es wäre beispielsweise möglich, Anwendungen in jeder beliebigen Sprache zu schreiben. Entwickler müssen dann nicht die Sprache nutzen, die von den Function-as-a-Service-Lösungen der Cloud-Anbieter unterstützt werden. Auch wäre dann die Begrenzung auf Funktionen allein ausgehebelt.

Ein Microservice oder gar ein Monolith könnte als serverlose Anwendung laufen. Dazu braucht es nur eine Lösung, die dies ermöglicht. Schließlich verwenden proprietäre Cloud-spezifische Serverless-Lösungen auch Container (aller Art), und der gängige Mechanismus für den Einsatz von Containern ist Kubernetes. Entsprechend wächst die Anzahl von Kubernetes-Plattformen für Serverless. Es ist gut möglich, dass Knative [2] der De-facto-Standard dafür werden wird, wie man serverlose Loads mit Kubernetes bereitstellt.

Knative ist ein Open-Source-Projekt, das Komponenten für die Erstellung und Ausführung von serverlosen Anwendungen auf Kubernetes bereitstellt. Es dient zum Skalieren bis auf null, zum Autoskalieren, für In-Cluster-Builds und als Eventing-Framework für Anwendungen auf Kubernetes. Besonders interessant an Knative ist die Fähigkeit, Anwendungen in Serverless Deployments zu konvertieren. So spart man nicht nur Ressourcen (Speicher und CPU) im Ruhezustand der Anwendungen, sie können auch bei steigendem Datenverkehr schnell skaliert werden.

Diese Anwendungen eignen sich für Serverless

Ursprünglich waren nur Funktionen für Serverless vorgesehen. Dafür wären nur Codeteile notwendig, die einem einzigen Zweck dienen und aus wenigen Zeilen Code bestehen. Ein klassisches Beispiel für eine serverlose Anwendung wäre eine Bildverarbeitungsfunktion, die auf eine einzelne Anforderung reagiert und für einen begrenzten Zeitraum ausgeführt werden kann. Einschränkungen wie die Größe der Anwendungen (Funktionen) und deren maximale Laufzeit werden durch die Implementierung von Serverless Computing von Seiten der Cloud-Anbieter auferlegt.

Doch mit Kubernetes als Plattform für den Betrieb serverbasierter Implementierungen sind diese Einschränkungen unter Umständen nicht mehr relevant. Grundsätzlich kann jede Anwendung, die in ein Container-Image eingebettet werden kann, als Serverless Deployment in Kubernetes laufen. Das bedeutet jedoch nicht, dass jeder Container infrage kommt. Je kleiner die Anwendung, oder genauer gesagt, je schneller ihre Boot-up-Zeit ist, desto besser ist sie für Serverless Deployments geeignet.

Allerdings ist die Realität nicht ganz so einfach. Denn nur weil eine Anwendung in der Theorie nicht für Serverless geeignet ist, heißt das nicht, dass sie dafür vollkommen unbrauchbar ist. Wie viele andere serverlose Frameworks auch erlaubt Knative es, Konfigurationen zu verfeinern. Beispielsweise ist es möglich, mit Knative festzulegen, dass immer mindestens ein Replikat einer Anwendung vorliegen muss. Dies würde das Problem eines langsamen Boot-Ups beiseiteräumen und gleichzeitig einige der Vorteile von Serverless Deployments beibehalten.

Die Größe und die Boot-up-Zeit sind allerdings nicht die einzigen Kriterien, nach denen sich beurteilen lässt, ob eine Anwendung Serverless sein soll oder nicht. Auch der Traf- ▶

fic sollte berücksichtigt werden. Wenn die Anwendung beispielsweise konstant viele Anfragen erhält, muss sie möglicherweise nie auf null Replikate reduziert werden. Ebenso ist die Anwendung möglicherweise nicht so konzipiert, dass jede Anfrage von einem anderen Replikat bearbeitet wird.

Schließlich können die meisten Anwendungen eine Vielzahl von Anfragen mit einem einzigen Replikat bearbeiten. In solchen Fällen ist Serverless Computing, das von Cloud-Anbietern implementiert wird und auf Function as a Service basiert, möglicherweise nicht die richtige Wahl.

Aber wie bereits erwähnt gibt es auch andere serverlose Plattformen, und die auf Kubernetes basierenden Plattformen folgen nicht diesen Regeln. Da jeder Container als Serverless ausgeführt werden kann, kann auch jede Art von Anwendung als solche bereitgestellt werden. Das bedeutet, dass ein einziges Replikat so viele Anfragen verarbeiten kann, wie es das Design erlaubt. Außerdem können Knative und andere Plattformen so konfiguriert werden, dass sie eine Mindestanzahl von Replikaten besitzen, sodass sie sich auch für Anwendungen mit einem konstanten Datentraffic gut eignen.

Quelle: https://jenkins.io

Jenkins X zu Diensten

Zusammenfassend lässt sich sagen: Kann eine Anwendung in einem Container ausgeführt werden, so ist sie auch für Serverless Deployment geeignet. Man sollte sich jedoch stets bewusst sein, dass kleinere Anwendungen mit schnelleren Boot-up-Zeiten besser geeignet sind. Was aber bei der Frage nach Serverless oder Nicht-Serverless unbedingt beachtet werden soll, ist der Zustand der Anwendung – oder genauer gesagt der fehlende Zustand. Ist eine Anwendung zustandslos (stateless), könnte sie der richtige Kandidat für Serverless Computing sein.

Doch auch wenn sie nicht der richtige Kandidat ist, bedeutet das noch nicht, dass man keinen Vorteil aus Frameworks wie Knative ziehen kann. Dies ist möglich, da die Frage des Deployments in verschiedenen Umgebungen noch offen ist.

Grundsätzlich gibt es permanente und temporäre Umgebungen. Beispiele für Erstere sind Staging und Produktion. Soll die Anwendung in der Produktion keinesfalls Serverless werden, so wird es im Staging nicht anders aussehen. Sonst würde sie sich in den verschiedenen Phasen unterschiedlich verhalten. Dann könnte man keine zuverlässigen Tests durchführen, die vom Verhalten im Staging auf das in der Produktion schließen lassen.

Wenn also eine Anwendung in der Produktion nicht Serverless sein sollte, gilt dies meist auch für andere permanente Umgebungen. Für temporäre Umgebungen hingegen gilt diese Regel nicht.

Betrachtet man beispielsweise eine Umgebung, in der eine Anwendung als Ergebnis auf einen Pull-Request bereitgestellt wird, so handelt es sich um eine temporäre Umgebung. Sie würde in dem Moment entfernt werden, in dem der Pull-Request geschlossen wird. Die Zeitspanne dieser Umgebung ist daher relativ kurz – von wenigen Minuten bis ein paar Wochen ist alles möglich. Die Wahrscheinlichkeit, dass die Anwendung in diesem temporären Umfeld einen geringen Daten-Traffic aufweist, ist hoch.

In der Regel führen Entwickler eine Reihe von automatisierten Tests durch, wenn der Pull-Request gestellt oder geändert wird. Dies würde sicherlich einen Spitzenwert im Traffic verursachen.

Danach wäre der Traffic jedoch deutlich geringer und meist sogar nicht vorhanden. Voraussichtlich würde man die Anwendung öffnen, einen Blick darauf werfen und manuelle Tests durchführen, um dann darauf zu warten, dass der Pull-Request genehmigt wird, oder – falls nötig – zusätzliche Änderungen eingepflegt werden. In all dieser Zeit würde das eigentliche Deployment nicht genutzt werden.

Wäre es jedoch ein traditionelles Deployment, würde es ohne erkennbaren Grund dennoch Ressourcen verbrauchen. Dies würde gegen temporäre Umgebungen und die damit verbundenen hohen Kosten sprechen.

Da Implementierungen, die auf Pull-Requests basieren, nicht für die endgültige Validierung vor dem Deployment in die Produktion verwendet werden (dafür sind permanente Umgebungen vorgesehen), ist es nicht erforderlich, dass sie mit der Produktion identisch sind.

Andererseits sind die Anwendungen in solchen Pull-Request-Umgebungen meist ungenutzt. Diese Fakten deuten darauf hin, dass temporäre (oft Pull-Request-basierte) Umgebungen ein idealer Kandidat für Serverless Deployments sind, unabhängig vom Deployment-Typ in permanenten Umgebungen (zum Beispiel Staging und Produktion).

Jenkins X und Serverless

Mit dem traditionellen Jenkins gibt es einige Probleme: Jenkins (ohne X) skaliert nicht, ist nicht fehlertolerant, hat einen hohen Ressourcenverbrauch, ist langsam, ist nicht API-gesteuert und so weiter. Kurzum: Jenkins ist zwar nicht veraltet, doch die angesprochenen Punkte hatten bei Entstehung von Jenkins noch nicht die gleiche Relevanz wie heute. Der Schritt von Jenkins zu Jenkins X war daher ein großer.

Ursprünglich enthielt Jenkins X eine abgespeckte Version von Jenkins, aber seit dem Release 2 ist keine einzige Zeile des traditionellen Jenkins mehr in Jenkins X übrig. Jetzt ist Jenkins X dank Tekton [3] und einer Vielzahl von neu entwi-

ckeltem Code vollständig Serverless – abgesehen von einem sehr dünnen Layer, der meist als API-Gateway fungiert.

So kommt Jenkins X dem Bedarf an einer modernen Kubernetes-basierten Lösung nach. Ist kein Build-Vorgang aktiv, läuft bei Jenkins X nichts, und für jede Auslastung wird entsprechend skaliert. Dies ist eines der besten Beispiele für Serverless Computing.

Abläufe für Continuous Integration und Continuous Delivery sind per se temporär. Bei einer Änderung an einem Git-Repository wird der Cluster benachrichtigt, der eine Reihe von Prozessen in Gang setzt. Jede Git-Webhook-Anfrage führt zum Start einer Pipeline, die einen neuen Release erstellt, ihn validiert und bereitstellt und, sobald diese Prozesse abgeschlossen sind, aus dem System verschwindet. Ohne Aktivitäten in der Pipeline, und davon sind beliebig viele parallel möglich, wird kein Befehl ausgeführt. Diese Prozesse sind elastisch und ressourcenschonend und die Schwerstarbeit übernimmt Tekton.

Tools für Continuous Integration and Continuous Delivery zählen zu den besten Use Cases für das Konzept von Serverless Computing.

Knative, Tekton und wie es zu Jenkins X passt

Tekton ist ein Spin-off-Projekt von Knative und ein wesentlicher Bestandteil der Lösung. Es ist dafür verantwortlich, bei Bedarf Pipeline-Prozesse (eine spezielle Art von Pods) zu erstellen und diese nach Beendigung zu löschen. Dank Tekton ist der Ressourcenverbrauch von Serverless Jenkins X im Ruhezustand sehr klein. Ebenso kann die Lösung durch Tekton auf fast jede erforderliche Größe skaliert werden. Tekton ist dabei nur für spezielle Prozesse ausgelegt, meist solche, die mit Continuous-Integration- und Continuous-Delivery-Pipelines verbunden sind.

Es ist jedoch nicht für langfristig laufende Anwendungen geeignet, die für die Bearbeitung von Requests konzipiert sind. Doch auch wenn Tekton es nicht erlaubt, Anwendungen ohne Server zu betreiben, so ist es aufgrund seiner Herkunft dennoch relevant.

Tekton wurde als Knative-Spin-off entwickelt, um bessere Funktionen für Continuous Integration und Continuous Delivery zu bieten. Oder genauer: Tekton wurde aus einer Knative-Build-Komponente geboren, die heute nicht mehr zeitgemäß ist. Aber Knative ist nach wie vor der beste Weg, um serverlose Anwendungen in Kubernetes auszuführen.

Jenkins X bietet die Möglichkeit, auszuwählen, ob ein Quickstart erstellt oder ein bestehendes Projekt importiert werden soll, das als Serverless-Anwendung mit Knative bereitgestellt wird. Im letzteren Fall wird Knative die Erstellung der benötigten Definition (YAML-Datei) übernehmen.

Jenkins X ist also ein gutes Beispiel sowohl für eine Reihe von serverlosen Anwendungen, aus denen sich die Lösung zusammensetzt, als auch für ein Tool, das es uns ermöglicht, unsere bestehenden Anwendungen in serverlose Implementierungen zu konvertieren. Jenkins X muss dies lediglich als konkreten Wunsch kommuniziert bekommen, und wir müssen die richtigen Definitionen für die Anwendungen erstellen und sie durch ihre Lebenszyklen bewegen.

Kubernetes-Cluster mit Jenkins X

Das folgende Beispiel zeigt, wie man Serverless-Anwendungen erstellt. Zunächst erstellen Sie über *gke-jx.sh* einen Kubernetes-Cluster mithilfe des installierten Jenkins X [4].

Wir könnten jetzt die Knative-Dokumentation aufrufen, den Anweisungen zur Installation und Konfiguration folgen und anschließend Jenkins X neu konfigurieren. Aber wir werden nichts davon tun, denn Jenkins X bietet bereits eine Methode zur Installation und Integration von Knative. Genauer gesagt erlaubt uns Jenkins X die Installation des Gloo-Add-ons [5], das wiederum Knative installiert.

Gloo ist sozusagen eine Einlasskontrolle für Kubernetes und ein API-Gateway. Der Hauptgrund für die Verwendung in unserem Kontext liegt in der Möglichkeit, Anfragen an von Knative verwaltete und autoskalierte Anwendungen weiterzuleiten. Die Alternative zu Gloo wäre Istio [6], das zwar sehr beliebt, aber leider auch äußerst komplex ist. Installieren Sie Gloo über

```
jx create addon gloo
```

Anhand der Ausgabe können wir sehen, dass der Prozess überprüft hat, ob *glooctl* installiert ist, und wenn nicht, hat er es für uns eingerichtet.

Das Befehlszeilen-Tool bietet einige Funktionen an, aber die einzige, die (vorerst) wichtig ist, ist, dass es Knative installiert. Darüber hinaus hat der Prozess Gloo und Knative in unserem Cluster installiert und er hat unsere Umgebung so konfiguriert, dass sie Knative als Standardbereitstellungsart verwendet. Das bedeutet, dass von nun an jede neue Anwendung, die wir durch einen Schnellstart oder durch den Import eines bestehenden Projekts hinzufügen, als Knative bereitgestellt wird, sofern wir nichts anderes angeben. Der standardmäßige Bereitstellungsmechanismus kann jederzeit geändert werden, aber dazu später mehr. Lassen Sie uns vorerst durch die Erkundung der Namespaces einen Blick auf das werfen, was wir erreicht haben:

```
kubectl get namespaces
```

Die folgenden Outputs stammen aus Serverless Jenkins X in GKE. Wenn Sie eine andere Kombination verwenden, kann es zu Unterschieden im Output kommen.

```
NAME             STATUS AGE
cd               Active 66m
cd-production    Active 59m
cd-staging       Active 59m
default          Active 67m
gloo-system      Active 2m1s
knative-serving  Active 117s
kube-public      Active 67m
kube-system      Active 67m
```

Wie Sie sehen, gibt es zwei neue Namespaces. *gloo-system* enthält Gloo-Komponenten, während Knative in *knative-serving* läuft. Bitte beachten Sie, dass nicht alle Knative-Kom- ▶

ponenten aufgeführt sind, sondern nur diejenigen, die dafür verantwortlich sind, Pods als Serverless Loads auszuführen.

Von nun an werden alle unsere neuen Projekte als Serverless Deployments ausgeführt. Wenn wir wollen, dass neue Projekte standardmäßig nicht Serverless sind, können wir das korrigieren, indem wir die Deployment-Einstellungen für das gesamte Team (eine Jenkins-X-Installation) bearbeiten:

```
jx edit deploy \
    --team \
    --kind default \
    --batch-mode
```

Die Ausgabe sollte bestätigen, dass die Art des Team-Deploys auf Standard gesetzt wurde.

Um diese Einstellung wieder rückgängig zu machen, also festzulegen, dass alle neuen Projekte standardmäßig Serverless sein sollen, muss der Befehl wie folgt lauten:

```
jx edit deploy \
    --team \
    --kind knative \
    --batch-mode
```

Von diesem Moment an werden alle neuen Projekte serverlos sein, sofern wir nichts anderes festlegen.

Neues Projekt für Serverless-Anwendungen

Jenkins X versucht, die Abläufe so einfach wie möglich zu halten. Getreu diesem Ziel gibt es nichts Besonderes, was ein Nutzer tun muss, um ein neues Projekt für Serverless Deployments zu erstellen.

Es gibt keinen zusätzlichen Befehl und auch keine zusätzlichen Argumente. Der Befehl *jx edit deploy* gibt Jenkins X bereits vor, dass tatsächlich alle neuen Projekte standardmäßig serverlos sind, sodass wir nur einen neuen Quickstart erstellen müssen.

```
jx create quickstart \
    --language go \
    --project-name jx-knative \
    --batch-mode
```

Wie Sie sehen, war dieser Befehl nicht anders als die anderen, den wir zuvor erstellt haben.

Das Projekt benötigt einen einzigartigen Namen. Der wird nun über den Parameter *--project-name jx-knative* als *jx-knative* festgelegt.

Wenn man sich den Output ansieht, gibt es dort auch nichts Neues. Wenn jemand anderes die Art des Deployments ändern würde, wüssten Sie nicht einmal, dass ein Quickstart damit enden wird, dass der erste Release in serverloser Manier in der Staging-Umgebung laufen wird.

Listing 1: Die Datei ksvc.yaml

```
{{- if .Values.knativeDeploy }}
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
{{- if .Values.service.name }}
  name: {{ .Values.service.name }}
{{- else }}
  name: {{ template "fullname" . }}
{{- end }}
  labels:
    chart: "{{ .Chart.Name }}-{{ .Chart.Version |
      replace "+" "_" }}"
spec:
  runLatest:
    configuration:
      revisionTemplate:
        spec:
          container:
            image: "{{ .Values.image.repository }}
                   :{{ .Values.image.tag }}"
            imagePullPolicy:
                   {{ .Values.image.pullPolicy }}
            env:
{{- range $pkey, $pval := .Values.env }}
            - name: {{ $pkey }}
              value: {{ quote $pval }}
{{- end }}
            livenessProbe:
              httpGet:
                path: {{ .Values.probePath }}
              initialDelaySeconds: {{ .Values
                .livenessProbe.initialDelaySeconds }}
              periodSeconds: {{ .Values
                .livenessProbe.periodSeconds }}
              successThreshold: {{ .Values
                .livenessProbe.successThreshold }}
              timeoutSeconds: {{ .Values
                .livenessProbe.timeoutSeconds }}
            readinessProbe:
              httpGet:
                path: {{ .Values.probePath }}
              periodSeconds: {{ .Values
                .readinessProbe.periodSeconds }}
              successThreshold: {{ .Values
                .readinessProbe.successThreshold }}
              timeoutSeconds: {{ .Values
                .readinessProbe.timeoutSeconds }}
            resources:
{{ toYaml .Values.resources | indent 14 }}
{{- end }}
```

Es gibt jedoch einen Unterschied, und wir müssen das Projektverzeichnis aufrufen, um ihn zu finden:

```
cd jx-knative
```

Der entscheidende Wert befindet sich in der Datei mit Namen *values.yaml*.

```
cat charts/jx-knative/values.yaml
```

Die relevanten Teile der Ausgabe lauten:

```
...
# enable this flag to use knative serve to
# deploy the app
knativeDeploy: true
...
```

Wie Sie sehen können, wird die Variable *knativeDeploy* auf *true* gesetzt. Früher war dieser Wert auf *false* gesetzt, einfach weil das Gloo-Add-on nicht installiert war.

Aber nachdem wir das geändert haben, wird *knativeDeploy* für alle neuen Projekte auf *true* gesetzt, es sei denn, wir ändern die Deployment-Einstellung erneut.

Möglicherweise denken Sie, dass eine Helm-Variable an sich nicht viel bedeutet, es sei denn, sie wird verwendet. Das stimmt. Es ist nur eine Variable, und wir müssen noch den Grund für ihre Existenz nachvollziehen.

Werfen wir mit dem folgenden Befehl einen Blick auf das Template Directory des Charts:

```
ls -l charts/jx-knative/templates
```

Die Ausgabe ist wie folgt:

```
NOTES.txt
_helpers.tpl
deployment.yaml
ksvc.yaml
service.yaml
```

Wir sind bereits mit *deployment.yaml*- und *service.yaml*-Dateien vertraut, aber wir haben vielleicht ein wichtiges Detail übersehen. Also, schauen wir uns an, was sich dahinter verbirgt.

```
cat charts/jx-knative/templates/deployment.yaml
```

Der Output, begrenzt auf den oberen und unteren Teil, ist:

```
{{- if .Values.knativeDeploy }}
{{- else }}
...
{{- end }}
```

Die Definition des Deployments liegt zwischen *{{- else }}* und *{{- end }}*. Das mag auf den ersten Blick ungewöhnlich erscheinen, bedeutet aber, dass die Deployment-Ressource nur erstellt werden sollte, wenn *knativeDeploy* auf *false* gesetzt ist. Wenn Sie einen Blick in die Datei *service.yaml* werfen, werden Sie das gleiche Muster feststellen.

In beiden Fällen werden die Ressourcen nur dann erstellt, wenn wir nicht die Option Knative-Deployment ausgewählt haben. Das bringt uns zur Datei *ksvc.yaml*.

```
cat charts/jx-knative/templates/ksvc.yaml
```

Der Output ist in Listing 1 zu sehen. Zunächst erkennen Sie dort, dass die bedingte Logik vertauscht ist. Die in dieser Datei definierte Ressource wird nur erstellt, wenn die Variable *knativeDeploy* auf *true* gesetzt ist.

Wir werden nicht auf die Details der Spezifikation eingehen. Es ähnelt dem, was wir als Pod-Spezifikation definieren würden. Blättern Sie selbst einmal durch die Knative-Serving-API-Spezifikation [7].

Wo sich die Knative-Definition deutlich von dem unterscheidet, was wir gewohnt sind, wenn wir beispielsweise mit Deployments und StatefulSets arbeiten, ist, dass wir vieles nicht spezifizieren müssen.

Es ist nicht erforderlich, ein Deployment zu erstellen, das ein ReplicaSet definiert, welches wiederum Pod-Templates definiert. Es gibt keine Definition eines Service, der den Pods zugeordnet ist.

Knative wird alle Objekte erstellen, die erforderlich sind, um unsere Pods in eine skalierbare Lösung umzuwandeln, die für unsere Nutzer zugänglich ist.

Fazit

Damit sind wir am Ende des ersten Teils angelangt. In ihm haben Sie erfahren, warum Serverless Deployment mit Jenkins X von Vorteil ist. Im zweiten Teil geht es dann darum, die Lösung zu konfigurieren, um sie an die unterschiedlichen Gegebenheiten anzupassen. ■

[1] Kubernetes, https://kubernetes.io/de
[2] Knative, www.dotnetpro.de/SL2002Serverless1
[3] Tekton, www.dotnetpro.de/SL2002Serverless2
[4] Skript zur Cluster-Erzeugung, www.dotnetpro.de/SL2002Serverless3
[5] Gloo, https://gloo.solo.io
[6] Istio, https://istio.io
[7] API-Spezifikation von Knative-Serving, www.dotnetpro.de/SL2002Serverless4

Viktor Farcic
ist Senior Consultant bei CloudBees, ein Mitglied der Docker Captains Group und Buchautor. Er programmierte schon mit C, Python, Visual Basic und C#. Seine Leidenschaft gilt Microservices, CI und Test Driven Development.

dnpCode A2002Serverless