Container à la K

Container-Orchestrator Kubernetes: Einstieg für Docker-Kenner

Betreiber von großen Container-Umgebungen stoßen irgendwann an die Grenzen dessen, was mit Docker möglich ist. Der Container-Verwalter Kubernetes verspricht Abhilfe, der Einstieg ist aber auch für erfahrene Docker-Nutzer nicht ganz leicht. Mit einem Beispiel-Setup aus Raspis kann man erste Erfahrungen sammeln.

Von Jan Mahn und Merlin Schumacher

Docker hat die Technik der Linux-Container handlich und damit populär gemacht und ist die erste Wahl für kleine und mittelgroße Umgebungen. Wird die Infrastruktur irgendwann größer, steigen viele auf Docker Swarm um und betreiben mehrere Docker-Hosts im Verbund.

Bei wirklich großen Umgebungen – etwa einer Webseite, die weltweit erreichbar sein soll, hohe Lastspitzen abfangen muss und auf vielen Servern in mehreren Clouds und auch im eigenen Rechenzentrum zugleich laufen soll, braucht es einen ausgewachsenen Container-Orchestrator wie Kubernetes. Die Software ist schnell zum Industriestandard für die Container-Verwaltung geworden und die Nachfrage nach Kubernetes-Experten in Unternehmen ist groß. Wer aus Neugier oder beruflichem Interesse den Schritt von Docker zu Kubernetes gehen will, steht vor der Herausforderung, nicht nur neue Begriffe lernen zu müssen, sondern nebenbei eine realistische Testumgebung, am besten aus mehreren Maschinen, betreiben zu müssen – oder Geld bei Cloud-Anbietern auszugeben. Mit ein paar Raspis kann man ein ausfallsicheres Kubernetes-Cluster zum Ausprobieren schnell selbst einrichten und in Kubernetes einsteigen.

Kubernetes ist Open-Source-Software, die 2015 von Google an die Cloud Native Computing Foundation übergeben wurde. Das Projekt entstand aus den Erfahrungen, die Googles Entwickler mit ihrem eigenen Container-Orchestrator Borg gemacht haben. Borg ist noch heute als Grundlage für praktisch alle von Google betriebenen Dienste im Einsatz. Der etwas sperrige Name Kubernetes kommt aus dem Griechischen und bedeutet Steuermann – um Platz zu sparen, wird er gern abgekürzt zu K8S (ein K, acht Zeichen und ein S). Kubernetes gehört mit mehr als 81000 Commits von 2200 Entwicklern und zu den größten Open-Source-Projekten.

Hardcore

Einige der Funktionen, die Kubernetes mitbringt, hat Docker inzwischen auch gelernt. Dennoch ist Kubernetes in diesem Bereich der Cloud-Dienste der Industriestandard. Ohne sich um das zugrundeliegende Betriebssystem Gedanken machen zu müssen, kann man bei Google, Amazon, Microsoft & Co. in den Clouds fertige Kubernetes-Nodes starten und ganz nebenbei der Kreditkarte beim Schwitzen zusehen.

Um die Beispiele in diesem Artikel nachzuvollziehen, brauchen Sie nicht so tief in die Tasche zu greifen. Für den Betrieb eines ersten Kubernetes-Clusters reichen schon zwei Raspis. Version 2 sollte es mindestens sein. Die Anleitung funktioniert aber auch mit beliebigen x86-PCs oder virtuellen Maschinen. Solide Docker-Kenntnisse sind für das Verständnis erforderlich.

Begriffsklärung

Die Architektur und die Begriffswelt von Kubernetes ist auf den ersten Blick kaum zu durchschauen, was vor allem daran liegt, dass Dinge aus der realen Welt (Computer, Festplatten, Prozessoren) über mehrere Ebenen bis zur Unkenntlichkeit wegabstrahiert werden. Es gibt also einige Vokabeln zu lernen: Die gesamte Einheit von Diensten, Containern und Servern wird als **Cluster** bezeichnet. Die Struktur eines Clusters sehen Sie in der Infografik auf dieser Seite.

Die kleinste Einheit ist der **Pod.** Er fasst mehrere **Container** und zugehörige Ressourcen (zum Beispiel persistenten Speicher) zusammen und stellt deren Dienste über eine gemeinsame IP-Adresse bereit. Ein Pod kann beliebig oft kopiert und verschoben, allerdings nie aufgeteilt werden.

Wie die Pods ausgestaltet sind, also welche Container darin laufen und wie diese konfiguriert sind, definiert ein **Deployment.** Das Deployment beschreibt auch, wie viele Replicas, also Kopien, eines Pods existieren sollen. Kubernetes verfolgt permanent das Ziel, diese Zahl zu erreichen: Fällt ein Pod aus, erzeugt Kubernetes automatisch einen neuen, um die Anforderungen des Deployments zu erfüllen. Wer sich bei der Arbeitsweise von Deployments an Docker Swarm erinnert fühlt, liegt richtig.

Die verschiedenen Kubernetes-Instanzen, auf denen Pods laufen, werden als **Nodes** bezeichnet. Wo diese Nodes liegen, ist unerheblich. So kann ein Node ein physischer Server (etwa ein Raspi), eine virtuelle Maschine oder sogar selbst ein Container sein, in dem wiederum Kubernetes läuft.

Pods sind flüchtig, das heißt, dass sie jederzeit gelöscht und wieder neu erzeugt werden können. Auf welchem Node ein Pod läuft oder welche IP-Adresse er bekommt, weiß man nicht und braucht man auch nicht zu wissen – man sollte auf keinen Fall mit festen IP-Adressen hantieren.

Eine Ansammlung von identischen Pods wird zu einem **Service** zusammengefasst. Angesprochen wird also nie ein Pod direkt, sondern immer der Service. Kommt eine Anfrage von außerhalb beim Service an, leitet er diese an einen der Pods weiter. An welchen Pod Kubernetes die Anfrage leitet, weiß man nicht – das ist aber auch unerheblich, denn jeder Pod arbeitet gleich. Es gibt unterschiedliche Typen von Services, die entweder ausgeklügelte Load-Balancing-Funktionen bereitstellen oder nur der Reihe nach die Pods durchwechseln.

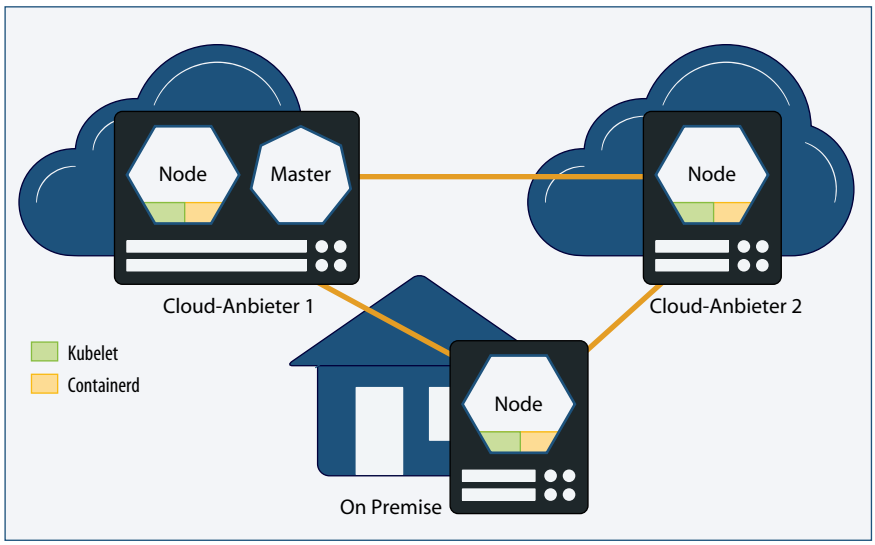
Die Steuerung der Nodes, Verteilung der Pods und die Verwaltung des Clusters übernimmt der **Master.** Der Master nimmt Anweisungen über ein API entgegen und wendet diese an. Wenn man zum Beispiel die Menge der Kopien eines Pods erhöht, erzeugt der Master neue, indem er den Nodes mitteilt, wie viele Pods welchen Typs sie erzeugen müssen. Der Master kontrolliert auch, ob die Anwendungen in den Containern wie geplant ihrer Arbeit nachgehen und stellt sicher, dass die Anforderungen der Deployments eingehalten werden.

Eine Besonderheit von Kubernetes ist der **Ingress-Router** für eingehenden Verkehr. Er kann HTTP(S)-Anfragen von Diensten an Services extrem flexibel verteilen. Der Administrator kann Regeln aufstellen, die dafür sorgen, dass Anfra-

Aufbau eines Kubernetes-Clusters

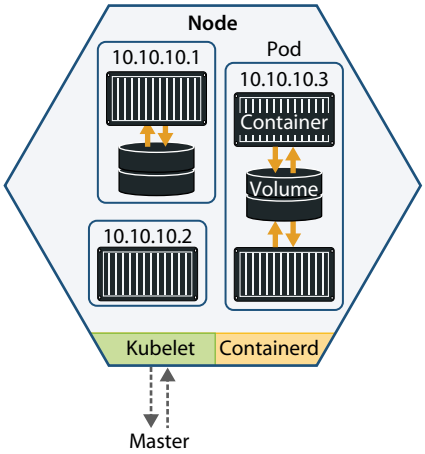
Ein Kubernetes-Cluster umfasst alle Dienste und Ressourcen, die für die Bereitstellung einer oder mehrerer Anwendungen nötig sind. Innerhalb des Clusters gibt es verschiedene Nodes. Das können Server, VMs oder auch Container sein, in denen eine Container-Runtime wie Containerd und ein Verwaltungsdienst namens Kubelet läuft, über den der Node verwaltet wird. Die Verwaltung übernimmt der Master.

Kubernetes-Cluster



Aufbau eines Kubernetes-Nodes

Innerhalb eines Kubernetes-Nodes laufen immer zwei Dinge: Der Verwaltungsdienst Kubelet, der ein API bereitstellt, über das der Master Dienste und Ressourcen verwalten kann. Dazu kommt eine Container-Runtime. Bei Kubernetes ist das üblicherweise Containerd. Innerhalb der Nodes laufen sogenannte Pods. Diese können verschiedene Ressourcen beinhalten, wie etwa Container oder Volumes.



gen auf www.beispiel.de von einem anderen Service beantwortet werden, als solche an www.example.com. Auch TLS-Terminierung gehört zu den Aufgaben des Ingress-Routers. Der Webserver im Service muss sich also nicht mit Zertifikaten belasten und kann hinter dem Ingress HTTP ausliefern. Die für das Routing eingesetzte Software wird als Ingress-Controller bezeichnet. Es gibt einige Programme, die als Ingress-Controller für Kubernetes arbeiten können. In diesem Artikel kommt die leichtgewichtige Kubernetes-Distribution K3S zum Einsatz, die den Router Traefik 1.7 nutzt. Traefik ist im Container-Umfeld aufgrund seiner Flexibilität sehr populär.

Auch wenn Kubernetes nach all den Begriffen komplex und unbezwingbar wirkt, muss man nicht viel dafür tun, Nodes, Services und Pods zu starten – wie auch bei Docker-Compose passiert das meiste in schön strukturierten YAML-Dateien.

K3S einrichten

Der Kubernetes-Download ist mit über 400 MByte ein ziemlicher Brocken. Entpackt werden daraus etwa 1,6 GByte. Da man auf dem Raspi mit dem Arbeitsspeicher meist haushalten muss, sind Binaries in dieser Größe zu viel.

Das K3S-Paket ist entpackt nur etwa 80 MByte groß und läuft somit auch auf dem Raspi oder kleinen Cloudservern mit wenig Arbeitsspeicher. Das Schöne an echter Hardware, also einer Handvoll Raspis ist aber, dass man wirklich mal testen kann, was mit einem Cluster passiert, wenn ein kompletter Node ausfällt, weil jemand gerade den Stecker gezogen hat.

Die Installation von K3S auf dem Raspi ist schnell erledigt. Schreiben Sie ein aktuelles Raspbian-Buster-Image (zu finden über ct.de/ya59) auf eine SD-Karte und bringen Sie es auf den neusten Stand. Falls Sie bereits Raspis am Laufen haben, können Sie auch die bestehenden Raspbian-Installationen weiternutzen. Ob Buster oder Stretch ist unerheblich.

Laden Sie jetzt das Installations-Skript herunter und führen Sie es im selben Schritt aus:

```
curl -sfL https://get.k3s.io | sh -
```

Wenn Sie auf Nummer sicher gehen wollen, laden Sie die Datei erst herunter und kontrollieren Sie, ob in der Datei k3s.sh das korrekte Shell-Skript und kein Müll gelandet ist.

Zentrales Werkzeug bei der Arbeit mit Kubernetes ist das Programm Kubectl. Es spricht über ein API mit dem Master. Nach der Installation können Sie mit Kubectl testen, ob Kubernetes auf Ihrer Maschine läuft:

```
sudo kubectl get nodes
```

Eventuell dauert es einen Moment, bis der Master antwortet. Anschließend sollten Sie eine Ausgabe bekommen, die der folgenden ähnelt:

```
NAME      STATUS ROLES  AGE VERSION
k3smaster Ready  master 47h v1.14.4
```

Der Name k3smaster ist der Hostname des Systems. Haben Sie den Ihres Raspi nicht geändert, dürfte dort raspberrypi stehen.

Öffnen Sie nun die Datei `/var/lib/rancher/k3s/server/node-token`. Sie enthält den Node-Token. Er wird benötigt, um weitere Nodes beim Master anzumelden.

Anschließend müssen Sie weitere Nodes – in diesem Fall Raspberry Pis – in das Cluster aufnehmen. Installieren Sie Raspbian und verbinden Sie sich per SSH. Zwei Umgebungsvariablen müssen gesetzt sein, bevor das Installationsskript startet: Die IP-Adresse des Masters und das Token:

```
export K3S_URL=https://
  ↳[MEINSERVER]:6443
export K3S_TOKEN=[MEINTOKEN]
curl -sfL https://get.k3s.io | sh -
```

Diesen Prozess wiederholen Sie nun für alle weiteren Nodes. Wenn Sie nun `sudo kubectl get node` auf dem Master aufrufen, sollten Sie alle hinzugefügten Nodes sehen können:

```
NAME      STATUS ROLES  AGE VERSION
k3smaster Ready  master 5m  v1.14.4
k3snode1  Ready  worker 5m  v1.14.4
```

Das Kubernetes-Cluster ist nun bereit für einen Test. Nach oben gibt es für die An-

```
01  ---
02  apiVersion: apps/v1
03  kind: Deployment
04  metadata:
05    name: whoami-deployment
06  spec:
07    selector:
08      matchLabels:
09        app: whoami
10    replicas: 6
11    template:
12      metadata:
13        labels:
14          app: whoami
15      spec:
16        containers:
17        - image:
ctmagazin/whoami
18          name: whoami-container
19          ports:
20          - containerPort: 80
21  ---
22  apiVersion: v1
23  kind: Service
24  metadata:
25    name: whoami-port
26  spec:
27    selector:
28      app: whoami
29    type: NodePort
30    ports:
31    - nodePort: 30123
32      port: 80
33      targetPort: 80
```

whoami.yml: Eine YAML-Datei reicht, um einen einfachen, aber redundanten Dienst über Kubernetes bereitzustellen.

zahl der Nodes keine Grenze – zumindest keine, die Sie mit Raspis erreichen können.

Ein Beispiel bitte

Kubernetes wird wie Docker-Compose auch über YAML-Dateien gesteuert. Die Syntax ist aber wesentlich komplexer. Die von der YAML-Datei beschriebenen Dienste werden als **Objekte** bezeichnet. Im Listingkasten auf Seite 158 sehen Sie ein Beispiel für einen grundlegenden HTTP-Dienst mit allen dafür nötigen Objekten. Basis ist ein Container, der per HTTP Informationen über sich selbst ausgibt. Darunter auch seinen Hostnamen. Darüber kann man erkennen, in welchem Pod ein Aufruf gelandet ist.

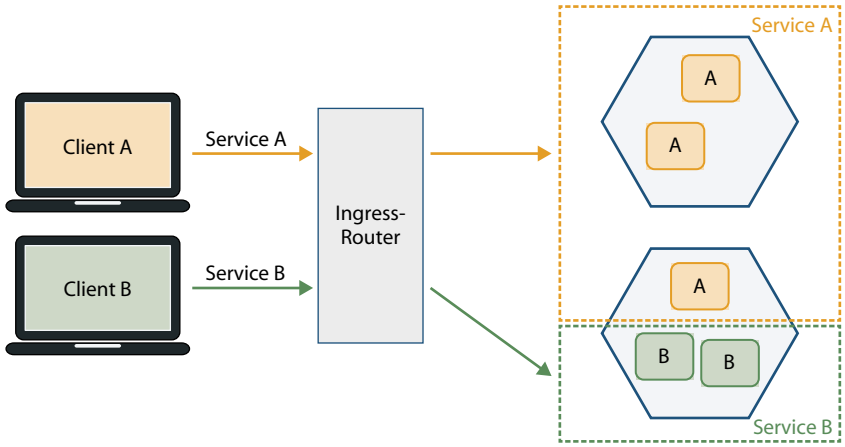
Alle Beispieldateien, die in diesem Artikel auftauchen, finden Sie über ct.de/ya59. Das spart bei den arg länglichen YAML-Blöcken viel Tipparbeit.

Die YAML-Datei gliedert sich in zwei Definitionen, die durch drei Minuszeichen getrennt sind. Der obere Block legt die

Routing und Services

Damit Clients immer den gewünschten Dienst innerhalb eines Kubernetes-Clusters erreichen, müssen deren Anfragen geroutet werden. Dafür ist der Ingress-Router zuständig.

Pods sind bei Kubernetes flüchtig, das heißt, sie können jederzeit rückstandslos zerstört werden. Pods werden in Services zusammengefasst. Kubernetes weiß, welche Pods den gewünschten Dienst bereitstellen und kann externe Anfragen durchleiten. Der Ingress-Router greift wiederum auf die Services zurück und leitet die externen Anfragen weiter.



Listingkasten (yaml)

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: whoami-service
spec:
  ports:
  - targetPort: 80
    port: 80
  selector:
    app: whoami
---
apiVersion:
networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: whoami-ingress
spec:
  rules:
  - http:
      paths:
        - path: /
          backend:
            serviceName:
whoami-service
            servicePort: 80
```

whoami-ingress.yaml: Die Konfiguration eines Ingress-Routers macht kaum Aufwand, bietet aber sehr viel Flexibilität und automatisches Load-Balancing. Den Block für das Deployment können Sie aus dem ersten Listing übernehmen.

Eigenschaften des Deployments fest. Der untere definiert den Service. Die erste Zeile beider Blöcke legt jeweils die API-Version fest, die Kubectl nutzen soll, um die Anweisungen umzusetzen.

Darauf folgt die Art (`kind`) von Element, die dieser Block definiert. Je nach Art unterscheidet sich teilweise auch die API-Version. Eine Übersicht der aktuellen API-Versionen finden Sie über ct.de/ya59. Im anschließenden Block `metadata` legt man den Namen des bereitgestellten Dienstes fest. Das hilft bei der eindeutigen Identifikation. Mit dem `spec`-Abschnitt beginnen die ersten großen Unterschiede, denn dieser beschreibt die Eigenschaften des Objekts. Die möglichen Werte und Strukturen unterscheiden sich aber je nach Objekt. Eine vollständige Dokumentation aller möglichen `spec`-Definitionen wäre zu ausführlich für einen Artikel, daher beschreiben wir nur die Grundlagen – mehr ist für die ersten Gehversuche aber auch nicht nötig.

Beim Deployment wird zunächst der `selector` definiert. Seine Aufgabe ist es festzulegen, welche Pods (neu erzeugt oder bereits bestehend) von diesem Deployment und den damit einhergehenden Änderungen betroffen sind. Das Attribut `matchLabels` legt fest, welches Label die Pods tragen müssen, damit das Deployment sie mit einbezieht. Die Labels können auch dazu dienen, Objekte aufzufinden und sind in der Form „Schlüssel=Wert“ definiert. So kann man zum Beispiel mit folgendem Befehl alle Pods anzeigen lassen, die das Label `app=whoami` tragen.

```
kubectl get pods -l app=whoami
```

Mit `replicas`, gefolgt von einer Zahl definieren Sie, wie viele Pods erzeugt werden sollen. Kubernetes verteilt diese dann gleichmäßig auf die vorhandenen Nodes. Wenn Sie also wie im Beispiel sechs Pods und zwei Nodes haben, landen jeweils drei Pods auf den Nodes. Die Zahl lässt sich theoretisch beliebig skalieren – in der Praxis ist Schluss, wenn die Ressourcen der Nodes erschöpft sind.

Die nun folgende `template`-Definition legt die Ausgestaltung der Pods fest. Mit `metadata.label` bekommen die Pods ein oder mehrere Label zugewiesen. Das sind die Label, nach denen in `matchLabels` gesucht wird. Stimmen diese nicht überein, wird auch das spätere Deployment fehlschlagen, denn aus Sicht von Kubernetes gibt es keine Pods, die von diesem Deployment betroffen wären. Damit wäre das Deployment nichtig.

`spec` legt die innerhalb des Pods benötigten Ressourcen fest. Das ist in diesem Beispiel ein einzelner Container. Es könnten aber auch mehrere Container und auch Volumes enthalten sein. Der Container wird aus einem `image` erzeugt und bekommt einen Namen (`name`) zugewiesen. Der Block `ports` legt fest, welche Ports des Containers erreichbar sind.

Anschließend kommt die Service-Definition. Der hier erzeugte Service ist vom Typ `NodePort`. Das bedeutet, dass der Dienst auf jedem Node über einen festgelegten Port von außen erreichbar ist. Jeder Node stellt aber nur Ausgaben der auf ihm laufenden Pods bereit. Es findet also kein Routing oder Load-Balancing statt.

Ein NodePort-Service muss nur wissen, für welche Pods er zuständig ist. Dafür kommt der `selector` zum Einsatz. `type` bestimmt den Typ des Services. Die `ports` gliedern sich in drei Typen: `nodePort` ist der Port, über den der Service extern, also auf der IP-Adresse des Nodes erreichbar ist. `port` bestimmt, über welchen Port der Service innerhalb des Clusters erreichbar ist. Auf welchen Port die jeweiligen Pods lauschen, legt `targetPort` fest.

Um das Beispiel zu starten, geben Sie folgenden Befehl auf dem Master ein:

```
kubectl apply -f whoami.yaml
```

Kubectl schickt die Anweisungen der YAML-Datei per API-Aufruf an den Kubernetes Master, der dann die Anforderungen des Deployments umsetzt. Er weist die Nodes an, die Pods und Services zu erzeugen. Um zu sehen, ob alles umgesetzt ist, können Sie kubectl get services und kubectl get deployments aufrufen. Laufen Deployment und Service, starten Sie Ihren Browser und öffnen Sie die URL http://<RASPI-IP>:30123. Daraufhin sollten Sie nun Informationen über den Container bekommen. Wenn Sie die Seite mit abgeschaltetem Cache (etwa über Strg+F5) neu laden, werden Sie sehen, dass der Hostname sich ändert. Der Service wechselt die antwortenden Pods einfach durch.

Neben `NodePort` gibt es noch weitere Typen von Services: Der Standard heißt `ClusterIP` und bedeutet lediglich, dass der Service nur innerhalb des Clusters erreichbar ist. Mit `LoadBalancer` erzeugt man eine Load-Balancer-Instanz, die versucht, die Last auf alle Nodes gleichmäßig zu verteilen. Das Load-Balancing kann man sowohl lokal als auch über einen externen Cloud-Provider bereitstellen. Auch hier

```
Hostname: whoami-deployment-7f444c5bc-fh7kk
IP: 127.0.0.1
IP: ::1
IP: 10.42.0.9
IP: fe80::9842:59ff:fe26:21d4
GET / HTTP/1.1
Host: 192.168.73.210
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: de-DE,de;q=0.8,en-US;q=0.5,en;q=0.3
Cache-Control: no-cache
```

Dank des whoami-Containers sieht man, welcher Pod antwortet.

richtet Kubernetes die nötigen Routen und notwendige Portfreigaben automatisch ein. Der letzte Typ heißt `ExternalName` und stellt den Service unter einem festlegten DNS-Namen bereit.

Aktuell antworten immer nur die Pods auf dem Node, den man per Browser angesprochen hat. Eleganter ist eine Lösung, die Load-Balancing kann – das ist das Problem, das Kubernetes lösen soll. Die eleganteste Lösung dafür ist der Ingress-Router. Er ist weit mächtiger als der Service-Typ `LoadBalancer`. Für den Einsatz des Ingress-Routers muss die Konfiguration des Services nur minimal angepasst werden. Hinzu kommt noch die Ingress-Konfiguration. Am Deployment ändert sich nichts. Eine Beispielkonfiguration für Ingress-Router und Service sehen Sie im Listingkasten auf Seite 160. Beim Service verschwindet der `type`, sodass er im ClusterIP-Modus läuft und damit nur innerhalb des Clusters erreichbar ist. Das reicht für den Ingress-Router. Die Option `nodePort` fehlt hier ebenfalls.

Die Konfiguration des Ingress-Routers ist genauso aufgebaut wie die der anderen Objekte. Lediglich die API-Version und `spec` unterscheiden sich. Bei der `spec` legt `rules` Regeln fest, nach denen der Ingress-Router entscheidet, welche Anfragen wo landen sollen. In diesem Beispiel soll der Router lediglich `http`-Anfragen für den Root-Pfad `/` beantworten. Für die Antwort ist dann der Service zuständig.

Wenden Sie die geänderte Konfiguration mit dem folgenden Befehl an:

```
kubectl apply -f whoami-ingress.yaml
```

Kubectl wird Ihnen mitteilen, dass es das Deployment nicht geändert hat, aber den Service aktualisiert und einen Ingress-Router erzeugt hat.

Wenn Sie nun die IP-Adresse des Masters im Browser aufrufen – diesmal auf dem normalen HTTP-Port 80, sehen Sie wieder die Ausgabe eines Pod. Wenn Sie die Seite neu laden, antwortet jedes Mal ein anderer Pod von einem beliebigen Node. Dadurch wird die Last auf alle Pods auf allen Nodes gleichmäßig verteilt und alle Pods sind über eine IP-Adresse erreichbar.

Weiter gedacht

Damit haben Sie einen ersten einfachen Dienst auf Basis von Kubernetes erzeugt – und damit etwa 1 Prozent des Funktionsumfangs von Kubernetes ausprobiert. Der Dienst kann zum Beispiel noch keine Daten speichern, weil noch keine Volumes enthalten sind. Auch das automatische Skalieren von Diensten oder Multi-Cloud-Umgebungen und zahlreiche andere Kubernetes-Spezialitäten sind nicht zur Sprache gekommen. Alle Funktionen zu beschreiben, würde schätzungsweise mehrere c't-Ausgaben füllen. Für den Einstieg und das weitere Erforschen von Kubernetes haben Sie jetzt aber das Rüstzeug. Weiterführende Dokumentationen und Beispiele haben wir für Sie beim folgenden Link hinterlegt. *(mls@ct.de)* c't

Links und Beispiele: **ct.de/ya59**