c't Teil 2: Dickschiffkapitän

Der Weg zum Kubernetes-Kenner muss nicht steinig sein. Kleine Server und solides Docker-Vorwissen reichen für den Einstieg. Im zweiten Teil der Reihe füllen Sie Ihren Kubernetes-Cluster mit Containern, spielen Updates ohne Downtime aus und lernen verschiedene Administrationsstile kennen.

Von Jan Mahn

c't 23/2022, Seite 158

kompakt

- In Kubernetes stecken Container in sogenannten Pods. Kubernetes kann sehr gut überwachen, wann ein solcher bereit ist.
- Damit Netzwerkverkehr in einem Pod ankommt, braucht man zusätzlich einen Service – mit eingebautem Load Balancer.
- Anders als in Docker können Sie mit etwas Konfigurationsarbeit sicherstellen, dass Updates ohne Auswirkungen auf den Betrieb stattfinden.

Container-Umgebungen funktionieren im Prinzip alle gleich: Im Hintergrund läuft eine Container-Runtime, die Container auf Basis eines Abbildes (Container-Image) startet, den Prozess im Container vom Rest des Betriebssystems trennt und mit Ressourcen wie Netzwerk und Speicherplatz versorgt. Das ist auf einem einzelnen Raspberry Pi mit Docker nicht anders als in einem gigantischen Kubernetes-Cluster, der die Dienste eines Weltkonzerns bereitstellt. Und auch wenn das Prinzip dasselbe ist, gibt es beim Umstieg von Docker auf Kubernetes eine Menge Tücken und Irrwege. Die will diese mehrteilige Reihe vermeiden und Docker-Kennern einen praktikablen Lernpfad aufzeigen. Im ersten Teil haben Sie erfahren, wie aus drei Linux-Maschinen ein Kubernetes-Cluster wird [1]. Einen solchen brauchen Sie für diesen zweiten Teil, ob nun selbstgebaut, im Rechenzentrum Ihres Unternehmens oder bei einem Provider fix und fertig gemietet. Auf Ihrer lokalen Maschine sollte das Kommandozeilenwerkzeug Kubectl installiert und mit den Zugangsdaten des Clusters versorgt sein – all diese Schritte sind in Teil 1 beschrieben.

Der erste Teil endete mit einem kleinen Erfolgserlebnis: Ihr erster Kubernetes-Container lief und zeigte eine Beispielseite auf Port 30000. Die Definition haben Sie aus unserem Beispiel heruntergeladen und in Ihren Cluster gebracht. In diesem zweiten Teil erfahren Sie, wie die YAML-Definitionen funktionieren. Damit Sie wieder mit einem leeren Cluster starten, müssen Sie zunächst aufräumen, wenn Sie das Beispiel bereits im Cluster laufen haben:

```
kubectl delete pod my-first-nginx
kubectl delete service my-service
```

Damit Sie nichts abtippen müssen (was bei YAML immer fehlerträchtig ist), finden Sie alle YAML-Schnipsel der Anleitung zum Download über ct.de/ynfw.

Hülsenfrüchte

In der Docker-Welt sind Container die kleinste Einheit, die man starten, stoppen und entfernen kann. Jeder Container entsteht aus einem Abbild, wird vom Rest des Betriebssystems gekapselt und mit einer virtuellen Netzwerkkarte für Kontakte mit der Außenwelt versehen. Kubernetes erweitert dieses Konzept und steckt Container immer in eine Hülle, die Pod genannt wird. In so einem Pod dürfen ein oder mehrere Container laufen, sie teilen sich eine gemeinsame Netzwerkverbindung und können sich auf Wunsch auch Ordner eines Dateisystems teilen. Nach außen erscheinen sie als eine Einheit und können immer nur zusammen verschoben und geklont werden.

Wie immer bei zusätzlichen Abstraktionsebenen, die Kubernetes im Vergleich zu Docker einzieht, gilt auch hier: Man muss dieses Angebot nicht nutzen und im Alltag enthalten viele Pods (wenn nicht gar die meisten) nur einen Container. Auf keinen Fall sollte man seine gesamte Anwendung (zum Beispiel Frontend, Backend und Datenbank) in einen Pod stopfen, dann könnte man es mit der Containerisierung auch ganz lassen, weil man ein unflexibles Monstrum geschaffen hätte.

Auf keinen Fall sollte man seine gesamte Anwendung in einen einzigen Pod stopfen.

Die Grundregel: Jede Komponente, die einzeln aktualisiert und skaliert werden soll, bekommt ihren eigenen Pod. Mehrere Container in einem Pod kann man zum Beispiel dann einsetzen, wenn ein Container ein kleines Helferlein in Form eines anderen Containers braucht, der einmalig oder regelmäßig eine Konfiguration einliest und in ein Pod-internes Dateisystem legt. So ein Hilfscontainer heißt in der Kubernetes-Welt Sidecar, früher oder später werden Ihnen solche Konstrukte begegnen. Ihr erster Pod kommt ohne solche Feinheiten aus. Um ihn anzulegen, brauchen Sie irgendwo auf Ihrer lokalen Maschine eine Datei (am besten in einem eigenen Ordner), die Sie zum Beispiel pod.yml nennen (auf den Namen der Datei kommt es Kubernetes nicht an). Darin folgende Zeilen:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-first-nginx
  labels:
    app: my-nginx
spec:
  containers:
    - name: nginx
      image: nginx:alpine
      ports:
        - containerPort: 80
```

Dieser Schnipsel enthält die Definition des Pods mit dem Namen `my-first-nginx`. Unterhalb von `spec:` bekommt er seine Eigenschaften zugewiesen: Die Liste der Container enthält nur einen einzigen Eintrag, erzeugt aus dem Abbild `nginx:alpine` (wie auch Docker nutzt die Kubernetes-Distribution k3s den Docker-Hub als Standard-Registry für Abbilder, Sie können aber auch jede andere Registry vor den Namen schreiben). Insgesamt werden in dieser Definition gleich mehrere Namen vergeben; der Pod selbst braucht einen nach außen einzigartigen Namen, festgelegt als Teil der `metadata`. Hier wird dem Pod auch direkt ein Label angeheftet – das kommt später zum Einsatz, um diesen Pod zu identifizieren. Der Name eines Containers (hier `nginx`) muss nur innerhalb des Pods einzigartig sein.

Navigieren Sie auf der Kommandozeile in den Ordner mit der Datei namens pod.yml und weisen Kubectl an, den Schnipsel in den Cluster zu bringen:

```
kubectl -f pod.yml apply
```

Wenn die Verbindung zum Cluster erfolgreich war, sollte Kubectl vermelden, dass ein Objekt angelegt wurde. Führen Sie den Befehl zum Test direkt noch einmal aus, meldet die Software, dass es nichts zu ändern gab. Dieselbe Datei könnten Sie jetzt von jedem anderen Rechner, auf dem die Zugangsdaten liegen, in den Cluster schieben, das Kubernetes-API würde immer prüfen, ob es einen Pod dieses Namens schon gibt und bei Bedarf dessen Attribute ändern. Um auch mal eine Änderung gesehen zu haben, ändern Sie den Namen des Images zu `nginx:1.23-alpine` und schicken den Änderungswunsch per obenstehendem Apply-Befehl in den Cluster. Einen Befehl, der wie `docker compose up` und `down` funktioniert, gibt es in Kubernetes nicht, die Änderung wird sofort umgesetzt. Um zu sehen, ob Ihr Pod wirklich läuft, brauchen Sie einen weiteren Unterbefehl von `kubectl get` (`kubectl get nodes` haben Sie bereits kennengelernt):

```
kubectl get pods
```

Auch das Entfernen von Pods ist keine schwarze Magie und die Syntax ist einleuchtend:

```
kubectl delete pod my-first-nginx
```

Nachdem Sie den letzten Befehl ausprobiert haben, bringen Sie den Pod mit dem obenstehenden Apply-Befehl einfach wieder zurück in den Cluster. Solche Operationen laufen in Kubernetes alle etwas fixer ab als das Starten und Stoppen in Docker.

Der Nginx-Container läuft jetzt, von außen sehen Sie davon aber nichts, die Webseite ist noch nirgends veröffentlicht. Auch wenn die letzten beiden Zeilen in der Definition vermuten lassen, dass hier Port 80 auf der externen Netzwerkkarte freigegeben wurde – dem ist nicht so, der Port ist bisher nur auf Container-Ebene geöffnet.

Damit ein Port auf der externen Netzwerkkarte abgehört wird, brauchen Sie ein weiteres Kubernetes-Objekt. In der Kubernetes-Welt sind alle Dinge, die man in den Cluster befördert, Objekte. Das Objekt vom Typ Pod (`kind: Pod`) haben Sie bereits kennengelernt, im zweiten Schritt brauchen Sie einen Service, also ein Objekt vom `kind: Service`. Für dessen Definition können Sie entweder eine neue Datei anlegen oder in der bestehenden Datei unterhalb der Pod-Definition mit

der Zeile `---` ein weiteres Objekt einleiten. Diese beiden Optionen stehen Ihnen grundsätzlich zur Verfügung und es ist Ihre Entscheidung, wie Sie Ihre Dateien zuschneiden (anders als bei Docker-Compose). Die Definition für den Service sieht wie folgt aus:

```
---
apiVersion: v1
kind: Service
metadata:
  name: my-service
  namespace: default
spec:
  type: NodePort
  selector:
    app: my-nginx
  ports:
    - protocol: TCP
      nodePort: 30000
      port: 80
```

Wenn diese YAML-Epen Sie abschrecken, hier ein paar Worte zur Einordnung: Im Alltag muss man solche Konstrukte nicht auswendig in den Editor tippen. Ziemlich schnell hat man alle wesentlichen Objekte für die eigenen Anwendungsfälle zusammengebaut und kann sich ab dann an eigenem YAML bedienen. Außerdem sind alle Objekte nach demselben Schema aufgebaut, das man schnell verinnerlicht. Unter `metadata:` liegen Attribute, die das Objekt nach außen beschreiben und auffindbar machen – das braucht man immer, wenn Objekte sich auf andere Objekte beziehen. Neben dem Pflichtattribut `name` kann man hier `labels` und `annotations` zur Wiedererkennung anheften.

Unterhalb von `spec:` wird das Objekt selbst ausgestaltet. Dieser Service ist vom Typ `NodePort`, eine eher selten genutzte (für den Einstieg aber ganz nützliche) Spielart, die Ports der externen Netzwerkkarte weitergibt. Später werden Sie Services vor allem dafür einsetzen, einen Pod innerhalb des Clusters erreichbar zu machen, damit ein API-Pod zum Beispiel seine Datenbank erreichen kann.

Im `selector:` findet eine Verknüpfung von Objekten statt: Der Service soll nach einem Pod suchen, der das Kriterium `app: my-nginx` erfüllt. Gesucht wird dabei innerhalb der Labels. Wenn Sie sich die Pod-Definition noch einmal ansehen, erkennen Sie, dass genau dieses Attribut am Pod gesetzt ist.

Das ist im Vergleich zu Docker eine zusätzliche Ebene der Abstraktion – in der Docker-Compose-Datei reicht es aus, unterhalb von `ports:` einen Port der Netzwerkkarte mit einem Container zu verknüpfen, etwa wie folgt:

```
# Docker-Schnipsel zum Vergleich
web:
  image: nginx:latest
  ports:
  -80:80
```

Sinnvoll ist die zusätzliche Komplexität durchaus: So ist es möglich, mehreren Pods das Label `app: my-nginx` zu geben. Die Netzwerkschicht in Kubernetes würde eingehende Anfragen nacheinander auf all diese verteilen. Ohne viel Arbeit haben Sie einen internen Load-Balancer. Im nächsten Abschnitt erfahren Sie, wie Sie einen Pod klonen und je eine Instanz auf jeden Ihrer Server im Cluster legen.

Vorher ist der letzte Abschnitt `ports:` im Service erklärungsbedürftig. Darin wird die Verbindung zur Außenwelt hergestellt. TCP-Verkehr, der auf den externen Netzwerkkarten auf Port 30000 ankommt, wird zu Port 80 der Pods geschickt. Kleinere Ports als 30000 kann man nicht mit einem `NodePort` belegen, gedacht ist ein solcher Service nicht für die fertige Anwendung, sondern mehr für Experimente und Admin-Hintertüren. Später lernen Sie einen besseren Weg für eingehenden Verkehr kennen.

Speichern Sie die Service-Definition ab und schieben sie mit Kubectl in den Cluster. Der sollte vermelden, dass er ein Objekt angelegt hat, danach erreichen Sie eine Nginx-Beispielseite auf Port 30000 auf allen drei externen IP-Adressen Ihres Clusters. Dabei spielt es auch keine Rolle, auf welchem Node der Pod gestartet wurde, der Verkehr kommt dank interner Cluster-Magie immer an. Sie könnten jetzt zum Beispiel einen externen Load-Balancer vor den gesamten Cluster hängen und die Anfragen verteilen.

Um herauszufinden, auf welchem Node Ihr Pod gestartet wurde, können Sie Kubectl nutzen:

```
kubectl get pods -o wide
```

Skalieren, bitte

Für diesen einzelnen Pod hätten Sie sich den ganzen Aufwand bis hier sparen können, denn noch läuft ja nur eine Instanz. Ausfallsicher wird Ihr Konstrukt erst, wenn eine Nginx-Kopie auf jedem der drei Nodes läuft. Dafür gibt es eine weitere Abstraktionsschicht: das Deployment. Das enthält die Schablone (`template`) für einen Pod, die Kubernetes beliebig oft anwenden kann. In der folgenden YAML-Definition sehen Sie ein Deployment für den obigen Pod – und das meiste sieht vertraut aus:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-first-nginx-deployment
  labels:
    app: my-nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-nginx
  template:
    metadata:
      labels:
        app: my-nginx
```

```
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Im `metadata`-Abschnitt bekommt das Deployment wie gewohnt Name und Label, damit es selbst später auffindbar ist. Unterhalb von `spec:` dann die zentrale Information im Attribut `replicas`: Drei Kopien sollen entstehen. Es folgt ein `selector`, der wie der eines Service funktioniert und nach einem anderen Objekt sucht. In diesem speziellen Fall muss man aber innerhalb desselben Objekts dafür sorgen, dass dieses Suchkriterium erfüllt wird: Unterhalb von `template:` folgt die Blaupause für die Pods, die alle das geforderte Label `app: my-nginx` angeheftet bekommen.

Um jetzt vom Einzel-Pod auf das Deployment umzustellen, löschen Sie zuerst den Pod:

```
kubectl delete pod my-first-nginx
```

Entfernen Sie seine Definition aus der YAML-Datei und ersetzen ihn durch den obigen Deployment-Abschnitt, die Definition des Service bleibt unberührt. Dann schieben Sie die Änderungen in den Cluster. Der Befehl `kubectl get pods -o wide` zeigt den Erfolg: Es liegen drei Pods mit einer Zufallszeichenkette im Namen im Cluster. Im Browser sehen Sie von der Redundanz noch nichts, weil alle Nginx-Container dieselbe Seite ausliefern – die Anfragen werden aber schon auf alle drei Pods verteilt. Glauben Sie nicht? Dann ändern Sie das Image zu:

```
image: containous/whoami
```

Dieser Container wurde genau für solche Tests erfunden und zeigt in der ersten Zeile der Ausgabe den Namen seines Pods an. Mit dem Apply-Befehl tauschen Sie das Abbild aus — dann müssen Sie ihrem Browser nur noch das Cachen abgewöhnen (oder Curl benutzen) und mehrere Anfragen absetzen, um die Lastverteilung des Kubernetes-Service bei der Arbeit zu sehen. Sie sollten nacheinander Antworten Ihrer drei Pods bekommen. Wenn Ihnen drei Pods nicht mehr reichen, gibt es gleich zwei Wege, das Deployment zu skalieren. Entweder ändern Sie Zeile `replicas: 3` oder Sie setzen einen eigenen Befehl dafür ab:

```
kubectl scale deployment/my-first-nginx-deployment --replicas=5
```

Letztere Strategie hat aber ihre Nachteile: Wenn Sie oder ein Admin-Kollege später die YAML-Datei per Apply-Befehl ins Cluster schieben, wird die Änderung überschrieben und wieder der Wert für `replicas` aus der Datei benutzt.

> Jedes Objekt, das man aus einer YAML-Definition in den Cluster bringt, reichert Kubernetes mit
>
> Jedes Objekt, das man aus einer YAML-Definition in den Cluster bringt, reichert Kubernetes mit weiteren Attributen an. Um die anzuzeigen, greift man zu Kubectl auf der Kommandozeile oder einem grafischen Werkzeug wie Lens.

Ein Deployment ist verhältnismäßig hartnäckig. Kubernetes wird sich immer wieder darum kümmern, dass die geforderte Anzahl Pods existiert. Das können Sie ausprobieren, indem Sie sich die existierenden Pods anzeigen lassen, einen Namen herauskopieren und diesen mit `kubectl delete pod` entfernen. Schneller als Sie schauen können, hat Kubernetes den Bestand wieder aufgefüllt.

> Wenn die Anforderungen größer werden: Ein Kubernetes-Deployment ist eine Schablone für Pc
>
> Wenn die Anforderungen größer werden: Ein Kubernetes-Deployment ist eine Schablone für Pods – Sie bestimmen, wie viele Kopien für Ihre Anwendung sinnvoll sind.

Der Reihe nach

Mit Deployments kennen Sie jetzt eines der mächtigsten Werkzeuge für reibungsarme Softwareverteilung. Aber noch ist Ihr erstes Deployment nicht perfekt eingerichtet. Im Idealfall sollten Sie in der Lage sein, ein neues Abbild in den Cluster zu bringen, ohne dass die Nutzer einen Ausfall bemerken. Keine einzige HTTP-Anfrage darf verloren gehen. Die Zeiten, in denen Sie Ihren Nutzern ein längeres Wartungsfenster ankündigen müssen, weil Sie ein Update ausrollen, sind damit vorbei.

Bisher klappt das aber nicht: Wenn Sie in der aktuellen Konfiguration den Namen des Images wechseln und die Änderung übernehmen, wird Kubernetes alle alten Pods abreißen und neue Pods hinstellen. Weil Nginx sehr schnell einsatzbereit ist, kann es sein, dass man keinen Ausfall bemerkt – aber die wenigsten Container sind so schnell startklar wie ein nackter Nginx, manch ein Container braucht Minuten, bis er sich berappelt hat. Was Sie in den meisten Deployments nutzen wollen, ist das `RollingUpdate`. Ist das aktiv, wird Kubernetes die neuen Pods nacheinander hoch- und die alten runterfahren, sodass jederzeit eine definierte Anzahl Pods einsatzbereit ist. Die Funktion aktivieren Sie mit folgendem Schnipsel unterhalb von `spec:` des Deployments (nicht des Templates):

```
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 50%
      maxSurge: 1
[...]
```

Die letzten beiden Angaben sind optional: `maxUnavailable` gibt an, wie viele Pods parallel im nicht einsatzbereiten Zustand sein dürfen. `maxSurge` legt fest, wie viele Pods über `replicas` hinaus existieren dürfen. Das ist erst dann nötig, wenn man ressourcenhungrige Pods nutzt und verhindern muss, dass zu viele davon Prozessor und RAM überbelasten. Die Angaben für beide Einstellungen dürfen absolut oder relativ sein.

Mit der oben beschriebenen Definition räumt Kubernetes maximal 50 Prozent der Pods weg, rundet aber immer ab, sodass von dreien nur ein Pod verschwindet. Dann legt das System zwei neue Pods

an (bis zu vier gleichzeitige sind durch `maxSurge` ja erlaubt). Um dieses Schauspiel zu sehen, bauen Sie zunächst den Schnipsel in Ihr Deployment ein und ändern das Image (zum Beispiel wieder auf `nginx:alpine`). Bevor Sie die Änderungen anwenden, öffnen Sie ein zweites Kommandozeilenfenster und führen darin folgenden Befehl aus:

```
kubectl get pods -w
```

Der Parameter `-w` aktiviert den Watch-Modus, Sie sehen Änderungen damit in Echtzeit. Wenden Sie dann im anderen Fenster die Änderungen an. Im Schnelldurchlauf sehen Sie, wie Kubernetes Pods auf- und abbaut.

Die Zeiten von Wartungsfenstern sind mit RollingUpdate vorbei.

Perfekt ist der fliegende Wechsel aber noch nicht. Die Pods werden gestartet und sofort für einsatzbereit betrachtet – für Perfektion müssen Sie zwei weitere Konzepte kennenlernen: LivenessProbes und ReadinessProbes. Mit diesen kann Kubernetes durch regelmäßige Prüfung herausfinden, ob ein Pod einsatzbereit ist. Die beiden Funktionen arbeiten gut im Zusammenspiel, gehören aber zu den am häufigsten missverstandenen Kubernetes-Funktionen. Beide werden unterhalb von `spec:` an einem Container definiert und beschreiben einen Test, der wiederholt ausgeführt wird. Das kann ein Kommandozeilenbefehl im Container selbst sein. Bei allen Containern, die irgendwie übers Netzwerk erreichbar sind, kann man einen TCP-Port anfragen lassen. Container, die HTTP anbieten, prüft man mit einer spezialisierten HTTP-Prüfung. In der Praxis ist das sehr häufig der geeignete Weg und auch die Nginx-Container kann man so prüfen.

Zunächst zur ReadinessProbe. Ihre Aufgabe ist festzustellen, ob ein Container bereit ist, Anfragen anzunehmen. Das Ergebnis dieser Prüfung nutzt ein vorgeschalteter Service bei der Entscheidung, welchem Pod er Anfragen zustellt. Schlägt die ReadinessProbe fehl, bekommt der Pod solange keine Anfragen, bis sie erfolgreich ist. Ein neu gestarteter Pod muss sich immer erst beweisen, bevor er Verkehr zugestellt bekommt. In Kombination mit `rollingUpdate` stellt das sicher, dass wirklich alle Anfragen ankommen. Der kleine Nginx-Container ist schnell mit einer ReadinessProbe versehen:

```
    spec:
      containers:
        - name: nginx
          image: nginx:alpine
          ports:
            - containerPort: 80
          readinessProbe:
            httpGet:
              path: /index.html
              port: 80
            periodSeconds: 10
```

Alle 10 Sekunden wird Kubernetes mit dieser Konfiguration versuchen, die Seite index.html zu öffnen. Kommt ein Statuscode zwischen 200 und 299 (in diesem Fall 200) zurück, wird der Container als empfangsbereit eingestuft und mit Anfragen von außen belästigt.

In einer echten Anwendung sollten Sie nicht die Startseite index.html für die Prüfung einsetzen – die kann ja sehr groß werden. Bauen Sie lieber einen eigenen Endpunkt wie /ready für diesen Zweck, der nur „ok“ zurückgibt. Viele fertige Container sind bereits für den Betrieb mit LivenessProbes vorbereitet und in der Dokumentation findet sich der passende Pfad. Wenn Sie wissen, dass ein Container überdurchschnittlich lange beim Start braucht, können Sie die Ausführung der ersten LivenessProbe mit der Angabe `initialDelaySeconds` (auf gleicher Höhe mit `periodSeconds`) auch verzögern.

Die zweite Prüfung braucht man eher in seltenen Fällen: Die LivenessProbe kann Container aufspüren, die sich in einen Status manövriert haben, aus dem sie aus eigener Kraft nicht mehr herauskommen. Ganz selten kann es zum Beispiel mal vorkommen, dass eine Software zwar läuft (der Prozess also nicht mit einer Fehlermeldung und einem Fehlercode aussteigt), aber keine Aufträge mehr verarbeitet. Gibt es einen Endpunkt, der dann einen Fehler zurückgibt, bemerkt das eine LivenessProbe, Kubernetes löscht den Pod und ersetzt ihn. Viele fertige Images haben einen solchen Endpunkt ebenfalls eingebaut. In der YAML-Datei würden Sie eine LivenessProbe wie eine ReadinessProbe definieren. Für den Einstieg brauchen Sie aber keine LivenessProbe, schon gar nicht für einen statischen Webserver wie Nginx. In echten Anwendungen, die Sie selbst programmieren, lohnt es sich aber, etwas Denkarbeit in die Gestaltung eines geeigneten Endpunkts wie /live zu stecken.

Niemals darf eine LivenessProbe von einem anderen Pod abhängig sein.

Aber Achtung: Niemals, wirklich niemals darf eine LivenessProbe so konstruiert sein, dass sie von einem anderen Pod abhängig ist – damit haben sich vor Ihnen schon andere sehr unschöne Domino-Effekte im Cluster gebaut. Stellen Sie sich vor, Sie haben eine Datenbank im Cluster installiert. Außerdem diverse Pods mit Microservices, also Anwendungen, die auf diese Datenbank zugreifen und zum Beispiel ein API anbieten. Für die LivenessProbes haben Sie eigens den Endpunkt /live gebaut, der zum Test etwas aus der Datenbank holt und im Erfolgsfall „ok“ meldet. Fällt jetzt die Datenbank kurz aus, werden schlagartig alle Pods in den Fehlerzustand wechseln, weil ihre LivenessProbe scheitert. Kubernetes wird sie alle ausmustern und neue Pods starten. Im Kleinen geht das fix, im großen Cluster richtet das minutenlanges Chaos an und die Anwendung wird länger unerreichbar. Wenn Sie eigene Endpunkte für LivenessProbes programmieren, achten Sie immer darauf, dass der Endpunkt wirklich nur dann einen Fehler zurückgibt, wenn ein Neustart des Pods das Problem lösen kann.

Eine Frage des Stils

Bisher haben Sie Kubernetes ausschließlich mit Kubectl-Aufrufen und lokalen YAML-Dateien bedient. Diese Herangehensweise bezeichnet man als imperativ, weil sie aus einer Reihe von Befehlen besteht. Was am Ende im Cluster läuft, kann man nur erschließen, wenn man alle ausgeführten Befehle in der richtigen Reihenfolge kennt. Und das ist auch das größte Problem am imperativen Administrieren: Besonders in Teams mit mehreren Admins weiß schnell niemand mehr, wer wann welche YAML-Dateien von seinem Computer per Kubectl ins Cluster geschoben hat – vorbei ist es mit der schönen Reproduzierbarkeit und der Cluster wird zur Objekthalde. `kubectl apply` kann immer nur Objekte anlegen oder bearbeiten, nicht aber alte Objekte löschen. Daher ist

das Administrieren per Kubectl noch nicht der Königsweg, sondern nur der Einstieg und ein Mittel für Reparaturen und Experimente. Erstrebenswert ist stattdessen der deklarative Stil. Statt den Cluster mit Befehlen wie „Erzeuge einen neuen Pod“, „Lösche einen alten Pod“ und „Ändere eine Einstellung in einem anderen Pod“ zu versorgen, sagt man „Ich hätte gern den folgenden Zustand.“ Was zum Erreichen dieses Zustands nötig ist, muss und soll kein Mensch entscheiden, der darf sich darauf verlassen können, dass die Maschine sich darum kümmert.

> Mit dem Image containous/whoami sehen Sie, welcher Pod die Anfrage bearbeitet hat.

Mit dem Image containous/whoami sehen Sie, welcher Pod die Anfrage bearbeitet hat.

Das aktuell populärste (aber nicht das einzige) Werkzeug, um das von Haus aus imperative Kubernetes-API deklarativ zu nutzen, heißt Helm (helm.sh). In der Selbstbeschreibung bezeichnen die Entwickler ihr Kommandozeilenprogramm als Paketmanager für Kubernetes. Das greift aber zu kurz, Helm enthält zusätzlich eine Templating-Engine, die Werte aus Konfigurationsdateien in YAML-Gerüste einbaut und macht ganz nebenbei deklaratives Arbeiten möglich.

Im dritten Teil dieser Reihe, der in einer der nächsten Ausgaben erscheint, erfahren Sie, wie Sie fertiges YAML für Helm einpacken, mit Parametern versehen und sich einem deklarativen Stil annähern. Als Beispiel dient eine Anwendung mit mehreren Pods, die miteinander kommunizieren müssen. Wenn Sie bis dahin das Wissen aus diesem Teil anwenden wollen, versuchen Sie doch einmal, eine containerisierte Anwendung, die Sie aktuell mit Docker nutzen, in einen Pod zu verpacken und mit einer LivenessProbe auszustatten. *(jam@ct.de)*

Dokumentation und Beispiele: ct.de/ynfw

Revision #1
Created 2026-09-26 13:20:53 UTC by ralf
Updated 2026-09-26 13:20:53 UTC by ralf