3 Pods: Container in Kubernetes ausführen

Der Inhalt dieses Kapitels

- Pods erstellen, ausführen und anhalten
- Pods und andere Ressourcen mit Labels ordnen
- Eine Operation an allen Pods mit einem gegebenen Label durchführen
- Pods mithilfe von Namespaces in nicht überlappende Gruppen aufteilen
- Container bestimmten Arten von Arbeitsknoten zuweisen

Im vorherigen Kapitel haben Sie einen groben Überblick über die grundlegenden Komponenten erhalten, die Sie in Kubernetes erstellen, und zumindest in Umrissen gesehen, was sie tun. Jetzt wollen wir alle Arten von Kubernetes-Objekten (oder *Ressourcen*) ausführlich darstellen, damit Sie lernen, wann, wie und warum Sie sie jeweils einsetzen müssen. Dabei beginnen wir mit den *Pods*, denn dies sind die zentralen, wichtigsten Objekte in Kubernetes. Alle anderen dienen dazu, die Pods zu verwalten oder zu exponieren, oder werden von ihnen verwendet.

3.1 Einführung in Pods

Wie wir bereits gesehen haben, ist ein Pod eine Gruppe von Containern, die sich auf demselben Knoten befinden, und stellt den Grundbaustein von Kubernetes dar. Wir stellen nicht einzelne Container bereit, sondern immer Pods. Das heißt nicht, dass Pods grundsätzlich mehr als einen Container enthalten. Sehr häufig schließt ein Pod nur einen einzigen Container ein. Wenn ein Pod aber tatsächlich über mehrere Container verfügt, dann werden diese immer auf einem einzigen Arbeitsknoten ausgeführt. Ein Pod überspannt niemals mehrere Knoten, wie Bild 3.1 zeigt.

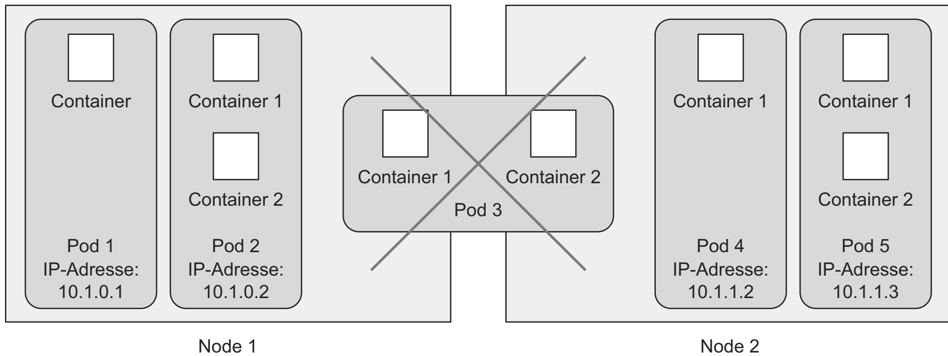

Bild 3.1 Alle Container eines Pods werden auf demselben Knoten ausgeführt. Ein Pod überspannt niemals mehrere Knoten.

3.1.1 Wozu benötigen wir Pods?

Aber wozu brauchen wir überhaupt Pods? Warum können wir die Container nicht einfach direkt verwenden? Warum müssen wir überhaupt mehrere Container nebeneinander verwenden? Können wir nicht einfach alle Prozesse in einen einzigen Container packen? Diese Fragen wollen wir im Folgenden beantworten.

Warum mehrere Container besser sind als ein Container mit mehreren Prozessen

Stellen Sie sich eine Anwendung aus mehreren Prozessen vor, die über IPC *(Inter-Process Communication)* oder lokal gespeicherte Dateien miteinander kommunizieren. Das macht es erforderlich, dass sie auf demselben Computer laufen. Da Prozesse in Kubernetes immer in Containern ausgeführt werden und jeder Container sehr wahrscheinlich auf einem eigenen Rechner läuft, scheint es sinnvoll zu sein, mehrere Prozesse in einem einzigen Container auszuführen. Trotzdem sollten Sie das nicht tun.

Container sind zur Ausführung eines einzigen Prozesses ausgelegt (abgesehen davon, dass der Prozess eigene Kindprozesse erzeugen kann). Wenn Sie mehrere nicht zusammengehörige Prozesse in einem einzigen Container ausführen, müssen Sie selbst dafür sorgen, dass all diese Prozesse laufen, ihre Protokolle verwalten usw. Beispielsweise müssten Sie einen Mechanismus hinzufügen, damit einzelne Prozesse automatisch neu gestartet werden, wenn sie abstürzen. Des Weiteren würden alle Prozesse ihre Protokolle in denselben Standardausgang schreiben, sodass es ziemlich schwer wäre herauszufinden, welcher Prozess was protokolliert hat.

Daher müssen Sie jeden Prozess in seinem eigenen Container ausführen. Das ist die vorgesehene Methode zur Nutzung von Docker und Kubernetes.

3.1.2 Grundlagen von Pods

Da es nicht vorgesehen ist, in einem Container mehrere Prozesse unterzubringen, brauchen wir offensichtlich einen Mechanismus, um mehrere Container aneinander zu binden und als eine Einheit zu behandeln. Das ist die Begründung für die Verwendung von Pods.

Ein Containerpod ermöglicht es, mehrere eng miteinander verbundene Prozesse gemeinsam auszuführen und mit (fast) derselben Umgebung zu versehen, als ob sie alle in einem einzigen Container ausgeführt werden, und sie gleichzeitig bis zu einem gewissen Grad voneinander zu isolieren. Dadurch können wir sowohl die Vorteile von Containern nutzen als auch den Prozessen vorgaukeln, dass sie zusammen ausgeführt werden.

Teilisolierung zwischen den Containern in einem Pod

Im vorherigen Kapitel haben wir gesagt, dass Container vollständig voneinander isoliert sind. In Wirklichkeit jedoch wollen wir nicht einzelne Container, sondern Gruppen von Containern voneinander isolieren. Die Container innerhalb einer Gruppe sollen sich einige Ressourcen teilen können, allerdings nicht alle. Das bedeutet, dass sie nicht vollständig isoliert sind. Kubernetes erreicht dies durch eine Konfiguration von Docker, sodass sich alle Container in einem Pod dieselben Linux-Namespaces teilen, anstatt jeweils ihre eigenen zu verwenden.

Da alle Container in einem Pod in denselben *Netzwerk-* und *UTS*-Namespaces ausgeführt werden (wir sprechen hier über Linux-Namespaces), teilen sie sich auch alle denselben Hostnamen und dieselben Netzwerkschnittstellen. Außerdem befinden sich alle Container eines Pods im selben *IPC*-Namespace und können daher über IPC miteinander kommunizieren. Sie sollten auch denselben *PID*-Namespace verwenden, allerdings ist dieses Merkmal nicht standardmäßig aktiviert.

HINWEIS: Da die Container eines Pods zurzeit verschiedene PID-Namespaces verwenden, können Sie die eigenen Prozesse eines Containers nur einsehen, wenn Sie `ps aux` im Container ausführen.

Beim Dateisystem sieht die Sache jedoch anders aus. Da der Großteil des Container-Dateisystems von dem Containerimage stammt, ist das Dateisystem jedes Containers standardmäßig vollständig von den Dateisystemen anderer Container getrennt. Mithilfe von *Volumes*, einer weiteren Art von Kubernetes-Ressource, können sie jedoch Dateiverzeichnisse gemeinsam nutzen. Darüber werden wir in Kapitel 6 sprechen.

Gemeinsame Nutzung des IP-Adress- und Portraums

Da die Container eines Pods alle im selben Netzwerk-Namespace ausgeführt werden, haben sie auch einen gemeinsamen IP-Adress- und Portraum. Das heißt, dass die Prozesse in den Containern eines Pods darauf achten müssen, sich nicht an dieselben Portnummern zu binden, da es sonst zu Konflikten kommen kann. Das betrifft aber nur Container innerhalb eines Pods. Zwischen Containern in verschiedenen Pods kann es keine Portkonflikte geben, da jeder Pod seinen eigenen Portraum aufweist. Alle Container in einem Pod haben außerdem dieselbe Loopback-Netzwerkschnittstelle, sodass sie über `localhost` miteinander kommunizieren können.

Das lineare Podnetzwerk

Alle Pods in einem Kubernetes-Cluster befinden sich in einem einzigen gemeinsam genutzten, linearen Netzwerkadressraum (siehe Bild 3.2). Dadurch kann jeder Pod jeden anderen Pod über dessen IP-Adresse erreichen. Zwischen ihnen bestehen keine NAT-Gateways (Network Address Translation). Wenn zwei Pods untereinander Netzwerkpakete austauschen, sehen sie jeweils die IP-Adresse des anderen als die Quell-IP-Adresse des Pakets.

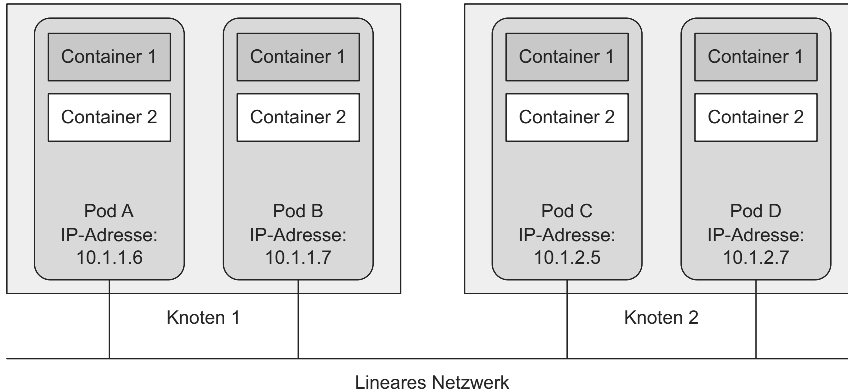

Bild 3.2 Jeder Pod erhält eine routingfähige IP-Adresse, unter der ihn die anderen Pods sehen.

Die Kommunikation zwischen den Pods ist infolgedessen sehr einfach. Unabhängig davon, ob zwei Pods auf demselben oder auf verschiedenen Arbeitsknoten ausgeführt werden, können die in ihnen enthaltenen Pods über das lineare Nicht-NAT-Netzwerk miteinander kommunizieren. Das ist damit vergleichbar, wie Computer in einem LAN unabhängig von der eigentlichen Netzwerktopologie miteinander kommunizieren. Ebenso wie solche Rechner hat jeder Pod seine eigene IP-Adresse und ist für die anderen Pods durch dieses eigens für die Pods eingerichtete Netzwerk erreichbar. Das wird gewöhnlich durch ein zusätzliches, in der Software definiertes Netzwerk erreicht, das auf das tatsächliche Netzwerk geschichtet wird.

Um den Inhalt dieses Abschnittes zusammenzufassen: Pods sind logische Hosts und verhalten sich sehr stark wie physische Hosts oder VMs. Prozesse innerhalb eines Pods ähneln den Prozessen, die auf ein und demselben physischen oder virtuellen Rechner ausgeführt werden, abgesehen davon, dass jeder Prozess in einem Container gekapselt ist.

3.1.3 Container auf Pods verteilen

Sie können sich Pods also wie eigenständige Computer vorstellen, die allerdings jeweils nur eine einzige Anwendung ausführen. Früher haben wir alle möglichen Arten von Anwendungen auf demselben Host untergebracht, aber bei Pods machen wir das nicht mehr. Da Pods recht schlank sind, können Sie so viele davon einrichten, wie Sie brauchen, ohne dabei nennenswerten Mehraufwand zu verursachen. Anstatt alles in einen einzigen Pod zu stopfen, verteilen Sie die Anwendungen auf mehrere Pods, sodass jeder nur eng miteinander in Beziehung stehende Komponenten oder Prozesse enthält.

Was glauben Sie – sollte eine mehrschichtige Anwendung, die aus einem Front-End-Anwendungsserver und einer Back-End-Datenbank besteht, in einem einzigen Pod oder in zwei Pods bereitgestellt werden?

Mehrschichtige Anwendungen auf mehrere Pods aufteilen

Es gibt zwar nichts, was Sie daran hindert, sowohl den Front-End-Server als auch die Datenbank in einem einzigen Pod mit zwei Containern auszuführen, allerdings ist es nicht die beste Vorgehensweise. Wir haben bereits gesagt, dass alle Container eines Pods immer auf demselben Knoten ausgeführt werden, aber ist es wirklich erforderlich, dass der Webserver und die Datenbank auf demselben Computer laufen? Das ist offensichtlich nicht der Fall, weshalb es nicht notwendig ist, beide in demselben Pod unterzubringen. Aber ist es auch falsch? In gewissem Sinne ist es das tatsächlich.

Wenn sich Front-End und Back-End im selben Pod befinden, laufen sie immer auf demselben Computer. Bei einem Kubernetes-Cluster mit zwei Knoten, in dem es nur diesen einen Pod gibt, wird immer nur ein einziger Arbeitsknoten verwendet. Die Rechenressourcen (CPU und Arbeitsspeicher), die auf dem zweiten Knoten zur Verfügung stehen, werden dagegen nicht genutzt. Durch die Aufteilung des Pods in zwei kann Kubernetes das Front-End auf dem einen Knoten und das Back-End auf dem anderen ausführen und dadurch die Infrastruktur besser ausnutzen.

Aufteilung aus Gründen der Skalierbarkeit

Ein weiterer Grund dafür, die beiden Schichten der Anwendung nicht in denselben Pod zu stellen, ist die Skalierung. Ein Pod ist auch das Grundelement für die Skalierung. Kubernetes kann nicht einzelne Container horizontal skalieren, sondern nur ganze Pods. Wenn Sie das Front-End und das Back-End in demselben Pod unterbringen und die Anzahl der Instanzen des Pods beispielsweise auf zwei heraufsetzen, dann haben Sie zwei Front-End- und zwei Back-End-Container.

Gewöhnlich aber besteht für Front-End-Komponenten ganz anderer Skalierungsbedarf als für Back-Ends, weshalb wir sie einzeln skalieren. Außerdem lassen sich Back-Ends wie Datenbanken gewöhnlich viel schwerer skalieren als (zustandslose) Front-End-Webserver. Wenn Sie also einen Container einzeln skalieren müssen, ist das ein deutliches Indiz dafür, dass Sie ihn in einem eigenen Pod bereitstellen sollten.

Wann Sie mehrere Container in einem Pod unterbringen sollten

Ein wichtiger Grund, um mehrere Container in einem einzigen Pod unterzubringen, liegt vor allem dann vor, wenn die Anwendung aus dem Hauptprozess und einem oder mehreren ergänzenden Prozessen besteht (siehe Bild 3.3).

Beispielsweise kann es sich bei dem Hauptcontainer in einem Pod um einen Webserver handeln, der einfach Dateien aus einem bestimmten Dateiverzeichnis zur Verfügung stellt, während ein zusätzlicher Container (auch *Sidecar* genannt, also wörtlich „Beiwagen“) regelmäßig Inhalte von einer externen Quelle herunterlädt und im Verzeichnis des Webservers speichert. In Kapitel 6 sehen wir uns an, dass wir in einem solchen Fall ein Kubernetes-*Volume* benötigen und in beiden Containern einhängen müssen.

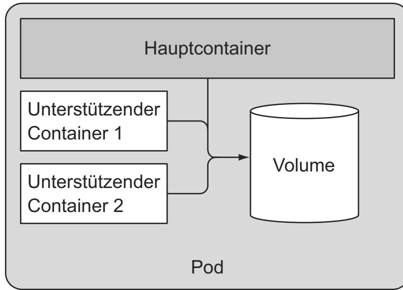

Bild 3.3 Pods sollten nur eng gekoppelte Container enthalten (gewöhnlich einen Hauptcontainer und solche, die ihn unterstützen).

Weitere Beispiele für Sidecar-Container sind Protokollrotatoren und -sammler, Datenverarbeiter, Kommunikationsadapter u. Ä.

Die Entscheidung treffen

Um zu entscheiden, ob Sie zwei Container in einen einzigen Pod stellen oder auf zwei verschiedene Pods aufteilen sollten, müssen Sie sich jeweils die folgenden Fragen stellen:

- Müssen die beiden Container zusammen ausgeführt werden oder können sie auch auf verschiedenen Hosts laufen?
- Bilden Sie ein Ganzes oder handelt es sich bei ihnen um unabhängige Komponenten?
- Müssen sie gemeinsam oder einzeln skaliert werden?

Im Allgemeinen sollten Sie Container eher in separaten Pods ausführen, sofern es keinen besonderen Grund dafür gibt, sie in einem gemeinsamen Pod unterzubringen. Bild 3.4 dient als Gedächtnisstütze.

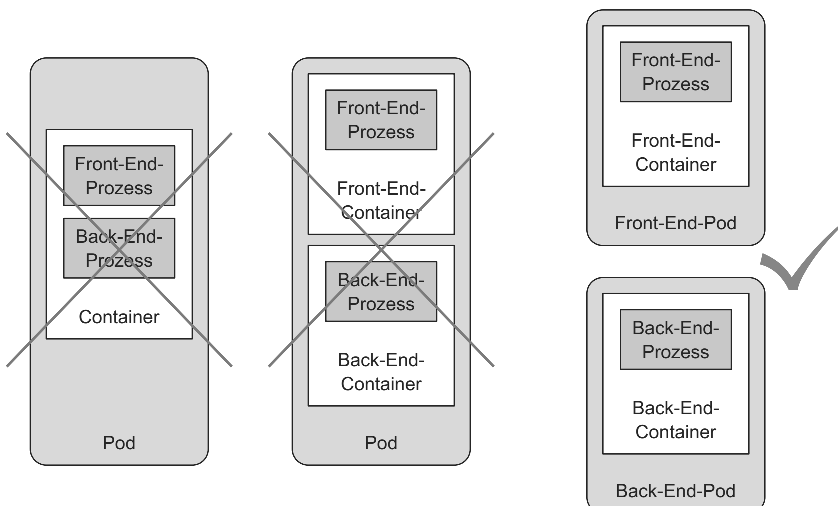

Bild 3.4 Ein Container sollte nicht mehrere Prozesse beherbergen, und ein Pod sollte nicht mehrere Container enthalten, sofern sie nicht auf demselben Computer ausgeführt werden müssen.

Pods können zwar mehrere Container enthalten, doch der Einfachheit halber wollen wir uns in diesem Kapitel nur mit Ein-Container-Pods beschäftigen. In Kapitel 6 sehen wir uns dann an, wie mehrere Container in einem Pod verwendet werden.

3.2 Pods aus YAML- und JSON-Deskriptoren erstellen

Pods und andere Kubernetes-Ressourcen erstellen Sie gewöhnlich dadurch, dass Sie ein JSON- oder YAML-Manifest am Endpunkt der REST-API von Kubernetes bereitstellen. Es gibt zwar einfachere Methoden, um solche Ressourcen anzulegen, etwa den im letzten Kapitel besprochenen Befehl `kubectl run`, allerdings können Sie dabei gewöhnlich immer nur eine Auswahl der Eigenschaften einstellen und nicht alle. Außerdem ist es bei der Definition aller Kubernetes-Objekte mit YAML-Dateien möglich, sie in einem Versionsverwaltungssystem zu speichern und all dessen Vorteile zu nutzen.

Um sämtliche Aspekte aller Arten von Ressourcen konfigurieren zu können, müssen Sie die Objektdefinitionen der Kubernetes-API kennen. Die meisten davon besprechen wir in diesem Buch zusammen mit den jeweiligen Ressourcentypen. Wir werden jedoch nicht jede einzelne Eigenschaft erklären, weshalb Sie sich zum Erstellen von Objekten auch die Dokumentation zur Kubernetes-API auf *http://kubernetes.io/docs/reference/* ansehen sollten.

3.2.1 Den YAML-Deskriptor eines bestehenden Pods untersuchen

Im vorherigen Kapitel haben wir bereits einige Pods erstellt, sodass wir uns nun ansehen können, wie die YAML-Definitionen dafür aussehen. Um diese Definition vollständig abzurufen, verwenden wir den Befehl `kubectl get` mit der Option `-o yaml`:

Listing 3.1 Vollständige YAML-Definition eines bereitgestellten Pods

```
$ kubectl get po kubia-zxzij -o yaml
apiVersion: v1   < Die Version der in diesem YAML-Deskriptor verwendeten Kubernetes-API
kind: Pod   < Typ des Kubernetes-Objekts/der Kubernetes-Ressource
metadata:                                                    | Metadaten des Pods (Name,
  annotations:                                               | Labels,
    kubernetes.io/created-by: ...                            | Anmerkungen usw.)
  creationTimestamp: 2016-03-18T12:37:50Z                    |
  generateName: kubia-                                       |
  labels:                                                    |
    run: kubia                                               |
  name: kubia-zxzij                                          |
  namespace: default                                         |
  resourceVersion: "294"                                     |
  selfLink: /api/v1/namespaces/default/pods/kubia-zxzij      |
  uid: 3a564dc0-ed06-11e5-ba3b-42010af00004                  |
```

```
spec:                                              | Spezifikation/Inhalt des Pods
  containers:                                      | (Liste der Container, Volumes
  - image: luksa/kubia                             | etc. im Pod)
    imagePullPolicy: IfNotPresent                  |
    name: kubia                                    |
    ports:                                         |
    - containerPort: 8080                          |
      protocol: TCP                                |
    resources:                                     |
      requests:                                    |
        cpu: 100m                                  |
    terminationMessagePath: /dev/termination-log   |
    volumeMounts:                                  |
    - mountPath: /var/run/secrets/k8s.io/servacc   |
      name: default-token-kvcqa                    |
      readOnly: true                               |
  dnsPolicy: ClusterFirst                          |
  nodeName: gke-kubia-e8fe08b8-node-txje           |
  restartPolicy: Always                            |
  serviceAccount: default                          |
  serviceAccountName: default                      |
  terminationGracePeriodSeconds: 30                |
  volumes:                                         |
  - name: default-token-kvcqa                      |
    secret:                                        |
      secretName: default-token-kvcqa              |
status:                                               | Ausführliche Angaben zum
  conditions:                                         | Zustand des Pods und seiner
  - lastProbeTime: null                               | Container
    lastTransitionTime: null                          |
    status: "True"                                    |
    type: Ready                                       |
  containerStatuses:                                  |
  - containerID: docker://f0276994322d247ba...        |
    image: luksa/kubia                                |
    imageID: docker://4c325bcc6b40c110226b89fe...     |
    lastState: {}                                     |
    name: kubia                                       |
    ready: true                                       |
    restartCount: 0                                   |
    state:                                            |
      running:                                        |
        startedAt: 2016-03-18T12:46:05Z               |
  hostIP: 10.132.0.4                                  |
  phase: Running                                      |
  podIP: 10.0.2.3                                     |
  startTime: 2016-03-18T12:44:32Z                     |
```

Ich gebe zu, dass das kompliziert aussieht, allerdings wird es verständlich, wenn Sie die Grundlagen kennen und zwischen den wichtigen Teilen und den Kleinigkeiten unterscheiden können. Außerdem ist der YAML-Code, den Sie schreiben müssen, um einen neuen Pod anzulegen, viel kürzer, wie wir noch sehen werden.

Die Hauptteile der Poddefinition

Die Poddefinition besteht aus einigen wenigen Teilen. Als Erstes werden die in YAML verwendete Version der Kubernetes-API und der Typ der beschriebenen Ressource angegeben. Darauf folgen drei wichtige Abschnitte, die in fast allen Kubernetes-Ressourcen zu finden sind:

- `metadata` enthält den Namen, den Namespace, Labels sowie weitere Informationen über den Pod.
- `spec` enthält die Beschreibung der Podinhalte, also der Container, Volumes und anderen Daten.
- `status` gibt aktuelle Informationen über den laufenden Pod, darunter den Zustand des Pods, die Zustände der einzelnen Container und schließlich auch die interne IP-Adresse und andere grundlegenden Angaben.

Listing 3.1 zeigt die vollständige Beschreibung eines laufenden Pods einschließlich seines Status. Der Statusabschnitt enthält nicht schreibbare Laufzeitdaten über den augenblicklichen Zustand der Ressource. Wenn Sie einen neuen Pod erstellen, geben Sie diesen Teil niemals an.

Die drei zuvor beschriebenen Teile zeigen die typische Struktur eines Kubernetes-API-Objekts. Wie Sie in diesem Buch noch sehen werden, haben alle Objekte den gleichen Aufbau. Das erleichtert es, sich mit neuen Objekten vertraut zu machen.

Es wäre nicht sehr sinnvoll, sämtliche Eigenschaften der vorstehenden YAML-Beschreibung nacheinander durchzugehen. Stattdessen wollen wir uns den grundlegenden YAML-Code zum Erstellen eines Pods ansehen.

3.2.2 Einen einfachen YAML-Deskriptor für einen Pod schreiben

Als Nächstes erstellen wir (in einem beliebigen Verzeichnis) die Datei *kubia-manual.yaml*. Sie können auch einfach das Codearchiv zu diesem Buch herunterladen, wo Sie die Datei im Verzeichnis *Chapter03* finden. Diese Datei hat folgenden Inhalt:

Listing 3.2 Ein grundlegendes Podmanifest: *kubia-manual.yaml*

```
apiVersion: v1   < Der Deskriptor folgt Version 1 der Kubernetes-API
kind: Pod   < Wir beschreiben einen Pod
metadata:
  name: kubia-manual   < Der Name des Pods
spec:
  containers:
  - image: luksa/kubia  < Das Containerimage, aus dem der Container erzeugt wird
    name: kubia   < Der Name des Containers
    ports:
    - containerPort: 8080   < Der Port, an dem die Anwendung lauscht
      protocol: TCP
```

Dies ist viel einfacher als die Definition in Listing 3.1. Sehen wir uns diesen Deskriptor im Einzelnen an. Er folgt der Version 1 der Kubernetes-API, und bei der beschriebenen Ressource handelt es sich um einen Pod namens `kubia-manual`. Dieser Pod besteht aus einem

einzelnen Container auf der Grundlage des Images `luksa/kubia`. Des Weiteren haben wir dem Container einen Namen verliehen und angegeben, dass er auf Port 8080 lauscht.

Containerports angeben

Die Angabe der Ports in der Poddefinition erfolgt rein aus informativen Gründen. Sie wegzulassen, hat keine Auswirkung darauf, ob der Pod Verbindung über diesen Port aufnehmen kann oder nicht. Wenn der Container Verbindungen über einen an die Adresse 0.0.0.0 gebundenen Port annimmt, können auch andere Pods Verbindung damit aufnehmen, selbst wenn dieser Port in ihrer Podspezifikation nicht ausdrücklich genannt wird. Es ist jedoch sinnvoll, die Ports ausdrücklich anzugeben, damit jeder, der den Cluster nutzt, auf einen Blick sehen kann, welche Ports die einzelnen Pods offenlegen. Durch die ausdrückliche Angabe der Ports können Sie ihnen auch Namen zuweisen, was sehr praktisch ist, wie wir weiter hinten in diesem Buch noch sehen werden.

Mögliche API-Objektfelder mit kubectl explain finden

Beim Schreiben eines Manifestes können Sie sich nach der Kubernetes-Dokumentation auf *kubernetes.io/docs/api* richten, um zu sehen, welche Attribute bei den verschiedenen Arten von API-Objekten angegeben werden können. Sie können dazu aber auch den Befehl `kubectl explain` verwenden.

Wenn Sie beispielsweise ein Podmanifest von Grund auf erstellen, können Sie sich von `kubectl` als Erstes eine Erklärung über Pods geben lassen:

```
$ kubectl explain pods
DESCRIPTION:
Pod is a collection of containers that can run on a host. This resource
is created by clients and scheduled onto hosts.

FIELDS:
  kind <string>
    Kind is a string value representing the REST resource this object
    represents...

  metadata <Object>
    Standard object's metadata...

  spec <Object>
    Specification of the desired behavior of the pod...

  status <Object>
    Most recently observed status of the pod. This data may not be up to
    date...
```

Kubectl gibt eine Erklärung des Objekts und eine Liste der Attribute aus, die es enthalten kann. Anschließend können Sie sich mehr über die einzelnen Attribute anzeigen lassen. Beispielsweise können wir wie folgt das Attribut spec untersuchen:

```
$ kubectl explain pod.spec
RESOURCE: spec <Object>

DESCRIPTION:
  Specification of the desired behavior of the pod...
  PodSpec is a description of a pod.

FIELDS:
  hostPID <boolean>
    Use the host's pid namespace. Optional: Default to false.

  ...

  volumes <[]Object>
    List of volumes that can be mounted by containers belonging to the
    pod.

  Containers <[]Object> -required-
    List of containers belonging to the pod. Containers cannot currently
    Be added or removed. There must be at least one container in a Pod.
    Cannot be updated. More info:
    http://releases.k8s.io/release-1.4/docs/user-guide/containers.md
```

3.2.3 Einen Pod mit kubectl create erstellen

Um aus unserer YAML-Datei nun einen Pod zu erstellen, verwenden wir den Befehl `kubectl create`:

```
$ kubectl create -f kubia-manual.yaml
Pod "kubia-manual" created
```

Der Befehl `kubectl create -f` dient dazu, beliebige Ressourcen (nicht nur Pods) aus einer YAML- oder JSON-Datei zu erstellen.

Die komplette Definition eines laufenden Pods abrufen

Nachdem Sie den Pod erstellt haben, können Sie Kubernetes nach seiner kompletten YAML-Definition fragen, um zu sehen, ob sie der zuvor betrachteten YAML-Definition ähnlich sieht. Was es mit den zusätzlichen Feldern in der zurückgegebenen Definition auf sich hat, besprechen wir in den folgenden Abschnitten. Zum Abrufen des vollständigen Deskriptors verwenden Sie folgenden Befehl:

```
$ kubectl get po kubia-manual -o yaml
```

Wenn Sie lieber mit JSON arbeiten, können Sie `kubectl` wie folgt auch anweisen, den JSON- statt des YAML-Deskriptors abzurufen (was auch dann funktioniert, wenn Sie den Pod mit YAML erstellt haben):

```
$ kubectl get po kubia-manual -o json
```

Der neue Pod in der Podliste

Wir haben jetzt zwar einen neuen Pod angelegt, aber woher wissen wir, dass er auch ausgeführt wird? Dazu rufen wir die Liste der Pods ab, um ihren jeweiligen Status einzusehen:

```
$ kubectl get pods
NAME           READY  STATUS   RESTARTS  AGE
kubia-manual   1/1    Running  0         32s
kubia-zxzij    1/1    Running  0         1d
```

Da haben wir unseren Pod `kubia-manual`! Laut Statusangabe wird er ausgeführt. Wenn Sie genauso veranlagt sind wie ich, möchten Sie sich vielleicht auch vergewissern, dass das wirklich stimmt, indem Sie mit dem Pod kommunizieren. Das werden wir in Kürze tun. Zunächst aber sehen wir uns das Protokoll unserer Anwendung an, um zu prüfen, ob irgendwelche Fehler aufgetreten sind.

3.2.4 Anwendungsprotokolle anzeigen

Unsere kleine Node.js-Anwendung schreibt ihre Protokolle an den Standardausgang des Prozesses. Containeranwendungen verwenden für ihre Protokolle gewöhnlich den Standardausgang und den Standardfehlerstream, anstatt sie in Dateien zu schreiben. Dadurch können die Benutzer die Protokolle der verschiedenen Anwendungen auf einfache, standardisierte Weise einsehen.

Die Containerlaufzeitumgebung (in unserem Fall Docker) leitet diese Streams in Dateien um, sodass sich das Protokoll des Containers mit folgendem Befehl abrufen lässt:

```
$ docker logs <Container-ID>
```

Sie könnten sich also mit `ssh` an dem Knoten anmelden, auf dem der Pod ausgeführt wird, und seine Protokolle mit `docker logs` abrufen, aber Kubernetes bietet eine einfachere Möglichkeit.

Das Protokoll eines Pods mit kubectl logs abrufen

Um an das Protokoll des Pods (genauer gesagt, des Containers) zu kommen, müssen wir einfach den folgenden Befehl auf dem lokalen Computer ausführen (ohne dass irgendeine `ssh`-Anmeldung erforderlich wäre):

```
$ kubectl logs kubia-manual
Kubia server starting...
```

Da wir noch keine Webanforderungen an unsere Node.js-Anwendung gesandt haben, zeigt das Protokoll nur den einen Eintrag über den Start des Servers an. Wie Sie sehen, ist es unglaublich einfach, die Protokolle einer in Kubernetes laufenden Anwendung abzurufen, wenn der Pod nur einen einzigen Container enthält.

HINWEIS: Containerprotokolle werden nach einem Tag sowie nach Erreichen der Protokolldateigröße von 10 MB zyklisch wiederverwertet. Kubect-Protokolle zeigen nur die Protokolleinträge des letzten Zyklus an.

Protokolle von Mehr-Container-Pods unter Angabe des Containernamens abrufen

Enthält der Pod mehrere Container, müssen wir beim Ausführen von `kubectl logs` mit der Option `-c <Containername>` ausdrücklich den Container angeben. Der Container in `kubia-manual` heißt `kubia`. Wenn es noch weitere Container in dem Pod gäbe, müssen wir die Protokolle von `kubia` wie folgt abrufen:

```
$ kubectl logs kubia-manual -c kubia
Kubia server starting...
```

Beachten Sie, dass Sie nur die Containerprotokolle von Pods abrufen können, die noch existieren. Wenn ein Pod gelöscht ist, dann gibt es auch seine Protokolle nicht mehr. Um die Protokolle eines Pods auch nach dessen Löschung noch verfügbar zu haben, müssen Sie eine zentrale, clusterweite Protokollierung einrichten, bei der alle Protokolle in einem zentralen Speicherbereich abgelegt werden. Wie das funktioniert, erfahren Sie in Kapitel 17.

3.2.5 Anforderungen an den Pod senden

Der Pod wird jetzt ausgefüllt – zumindest behaupten das `kubectl get` und das Protokoll unserer Anwendung. Aber wie können wir ihn uns im Einsatz ansehen? Im vorherigen Kapitel haben wir mit dem Befehl `kubectl expose` einen Dienst erstellt, um von außen Einsicht in den Pod zu nehmen. Da wir noch ein ganzes Kapitel haben werden, das Diensten gewidmet ist, wollen wir das hier nicht tun. Es gibt auch noch andere Möglichkeiten, um zum Testen und Debuggen Verbindung mit einem Pod aufzunehmen. Eine davon bietet die *Portweiterleitung*.

Einen lokalen Netzwerkport zu einem Port im Pod weiterleiten

Wenn Sie mit einem bestimmten Pod kommunizieren wollen, ohne den Weg über den Dienst zu nehmen (etwa für das Debugging oder aus anderen Gründen), können Sie eine Portweiterleitung zu dem Pod einrichten. Das erledigen Sie mit `kubectl port-forward`. Der folgende Befehl leitet den lokalen Port 8888 Ihres Computers zu unserem Pod `kubia-manual` weiter:

```
$ kubectl port-forward kubia-manual 8888:8080
... Forwarding from 127.0.0.1:8888 -> 8080
... Forwarding from [::1]:8888 -> 8080
```

Die Portweiterleitung läuft, sodass wir nun über den lokalen Port Verbindung mit unserem Pod aufnehmen können.

Über die Portweiterleitung Verbindung mit dem Pod aufnehmen

In einem anderen Terminal können wir jetzt mit `curl` eine HTTP-Anforderung über den Proxy `kubectl port-forward` auf `localhost:8888` an unseren Pod senden:

```
$ curl localhost:8888
You've hit kubia-manual
```

Bild 3.5 zeigt eine stark vereinfachte Ansicht dessen, was beim Senden dieser Anforderung geschieht. In Wirklichkeit gibt es noch einige weitere Komponenten zwischen dem `kubectl`-Prozess und dem Pod, aber sie sind für unser aktuelles Thema nicht von Bedeutung.

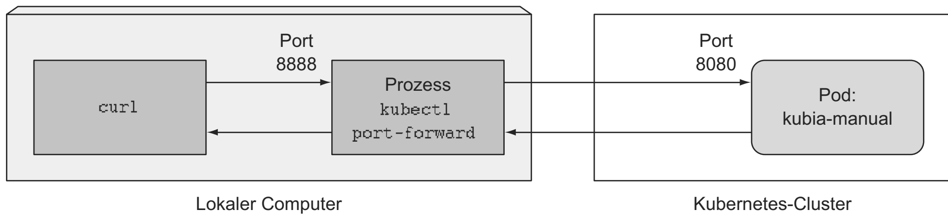

Bild 3.5 Eine stark vereinfachte Ansicht der Vorgänge bei der Verwendung von `curl` mit `kubectl port forward`

Die Verwendung der Portweiterleitung auf diese Weise ist eine wirkungsvolle Methode, um einen einzelnen Pod zu testen. In diesem Buch werden wir noch ähnliche Vorgehensweisen kennenlernen.

3.3 Pods mithilfe von Labels ordnen

Zurzeit laufen in unserem Cluster zwei Pods. Bei der Bereitstellung echter Anwendungen sind es jedoch meistens viel mehr. Je größer die Anzahl von Pods, umso wichtiger wird es, sie in Kategorien zu ordnen.

Betrachten Sie als Beispiel eine Microservice-Architektur. Dort kann die Zahl der bereitgestellten Microservices sehr schnell zwanzig oder mehr betragen. Diese Komponenten sind dann gewöhnlich auch noch repliziert (es werden mehrere Exemplare derselben Komponente bereitgestellt), und es können verschiedene Versionen (stabil, Beta, Canary usw.) gleichzeitig ausgeführt werden. Das kann dazu führen, dass sich Hunderte von Pods in unserem System tummeln. Ohne einen Ordnungsmechanismus haben wir am Ende ein riesiges, unübersichtliches Durcheinander wie in Bild 3.6. Dieses Bild zeigt die Pods mehrerer Microservices, von denen einige mehrere Replikate und einige auch unterschiedliche Versionen aufweisen.

Offensichtlich benötigen wir irgendeine Möglichkeit, um die Pods aufgrund selbst gewählter Kriterien in kleinere Gruppen zu gliedern, damit alle Entwickler und Systemadministratoren, die mit unserem System arbeiten müssen, sofort sehen können, welcher Pod wozu da ist. Außerdem wollen wir Operationen für alle Pods einer bestimmten Gruppe in einer einzigen Aktion durchführen können, anstatt diese Aktion auf jeden Pod einzeln anwenden zu müssen.

Eine solche Gliederung von Pods und anderen Kubernetes-Objekten erfolgt mithilfe von *Labels*.

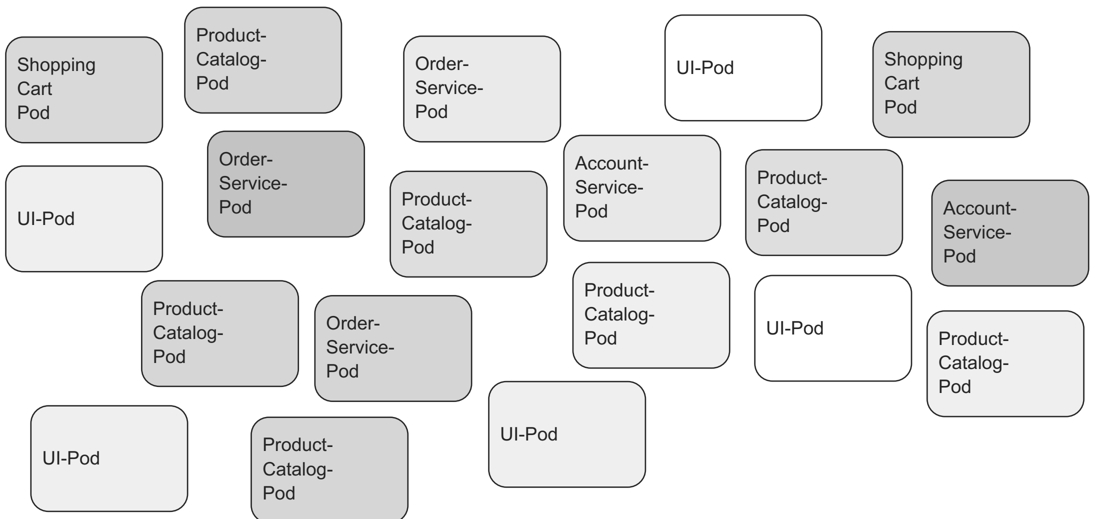

Bild 3.6 Nicht kategorisierte Pods in einer Microservice-Architektur

3.3.1 Einführung in Labels

Labels sind einfache und dort sehr vielseitige Kubernetes-Vorkehrungen, um nicht nur Pods, sondern auch alle anderen Kubernetes-Ressourcen zu ordnen. Ein Label ist ein willkürliches Schlüssel-Wert-Paar, das Sie einer Ressource zuweisen. Dieses Label wird später zur Auswahl von Ressourcen mithilfe von *Labelselektoren* herangezogen. Dabei werden die Ressourcen danach gefiltert, ob sie das in dem Selektor angegebene Label aufweisen. Eine Ressource kann mehrere Labels tragen, sofern die Schlüssel dieser Labels innerhalb der Ressource eindeutig sind. Gewöhnlich weisen Sie einer Ressource Labels zu, wenn Sie sie erstellen, aber Sie können später auch weitere Labels vergeben und sogar die Werte vorhandener Labels ändern, ohne die Ressource neu anlegen zu müssen.

Kehren wir wieder zu unserem Microservice-Beispiel aus Bild 3.6 zurück. Durch das Hinzufügen von Labels zu diesen Pods erhalten wir ein viel übersichtlicheres System, das für jeden leicht überschaubar ist. Jeder Pod wird dabei mit zwei Labels versehen:

- `app`, um anzugeben, zu welcher Anwendung oder Komponente oder welchem Microservice der Pod gehört
- `rel`, um anzugeben, ob es sich bei der Anwendung in dem Pod um ein stabiles, ein Beta- oder ein Canary-Release handelt

Definition

Bei einem Canary-Release stellen Sie eine neue Version einer Anwendung parallel zu einer stabilen Version bereit und sorgen dafür, dass nur ein Bruchteil der Benutzer zu dieser Version geleitet wird. Dadurch können Sie das

> Verhalten dieser Version prüfen, bevor Sie sie für alle Benutzer bereitstellen. Diese Vorgehensweise verhindert, dass eine schlechte Version zu viele Benutzer beeinträchtigt.

Durch Hinzufügen dieser beiden Labels ordnen wir unsere Pods nach zwei Dimensionen (horizontal nach Anwendung und vertikal nach Release), wie Sie in Bild 3.7 sehen.

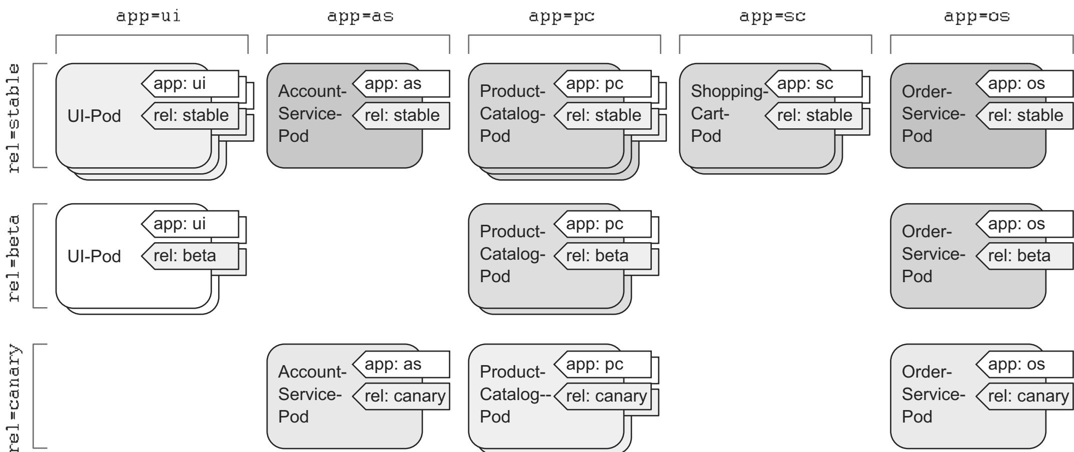

Bild 3.7 Pods in einer Microservice-Architektur mithilfe von Labels ordnen

Alle Mitglieder der Entwicklungs- und Betriebsteams, die Zugriff auf unseren Cluster haben, können jetzt leicht die Struktur des Systems erkennen und durch einen Blick auf die Labels sehen, wohin die einzelnen Pods gehören.

3.3.2 Labels beim Erstellen eines Pods angeben

Sehen wir uns das jetzt in der Praxis an, indem wir einen neuen Pod mit zwei Labels erstellen. Legen Sie dazu eine neue Datei namens *kubia-manual-with-labels.yaml* mit dem folgenden Inhalt an:

Listing 3.3 Ein Pod mit Labels: *kubia-manual-with-labels.yaml*

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-manual-v2
  labels:
    creation_method: manual   | Der Pod ist mit zwei Labels versehen
    env: prod                 |
spec:
  containers:
  - image: luksa/kubia
    name: kubia
```

```
    ports:
    - containerPort: 8080
      protocol: TCP
```

Im Abschnitt `metadata.labels` haben wir die beiden Labels `creation_method=manual` und `env=prod` angegeben. Erstellen wir nun den Pod:

```
$ kubectl create -f kubia-manual-with-labels.yaml
pod "kubia-manual-v2" created
```

Der Befehl `kubectl get pods` führt standardmäßig keine Labels auf. Sie können sie aber anzeigen lassen, indem Sie den Schalter `--show-labels` verwenden:

```
$ kubectl get po --show-labels
NAME             READY  STATUS   RESTARTS  AGE  LABELS
kubia-manual     1/1    Running  0         16m  <none>
kubia-manual-v2  1/1    Running  0         2m   creat_method=manual,env=prod
kubia-zxzij      1/1    Running  0         1d   run=kubia
```

Wenn Sie nicht alle Labels anzeigen lassen wollen, nur an einzelnen interessiert sind, können Sie sie mit dem Schalter `-L` angeben. Sie werden dann jeweils in ihrer eigenen Spalte angezeigt. Im nächsten Beispiel listen wir erneut alle Pods auf, fügen aber Spalten für die beiden an `kubia-manual-v2` angehängten Labels hinzu:

```
$ kubectl get po -L creation_method,env
NAME             READY  STATUS    RESTARTS  AGE  CREATION_METHOD  ENV
kubia-manual     1/1    Running   0         16m  <none>           <none>
kubia-manual-v2  1/1    Running   0         2m   manual           prod
kubia-zxzij      1/1    Running   0         1d   <none>           <none>
```

3.3.3 Labels vorhandener Pods ändern

Sie können auch Labels zu bestehenden Pods hinzufügen und deren Labels ändern. Da wir `kubia-manual` ebenfalls manuell erstellt haben, wollen wir ihm das Label `creation_method=manuel` hinzufügen:

```
$ kubectl label po kubia-manual creation_method=manual
pod "kubia-manual" labeled
```

Als Nächstes ändern wir das Label `env=prod` von `kubia-manual-v2` in `env=debug`, um zu zeigen, wie vorhandene Labels bearbeitet werden können.

HINWEIS: Beim Ändern vorhandener Labels müssen wir die Option `--overwrite` verwenden.

```
$ kubectl label po kubia-manual-v2 env=debug --overwrite
Pod "kubia-manual-v2" labeled
```

Wenn wir uns jetzt erneut die Liste der Pods anzeigen lassen, sehen wir die geänderten Labels:

```
$ kubectl get po -L creation_method,env
NAME              READY  STATUS   RESTARTS  AGE  CREATION_METHOD  ENV
kubia-manual      1/1    Running  0         16m  manual           <none>
kubia-manual-v2   1/1    Running  0         2m   manual           debug
kubia-zxzij       1/1    Running  0         1d   <none>           <none>
```

Wie Sie sehen, ist es ganz einfach, Labels an Ressourcen anzuhängen und zu ändern. Es mag jetzt noch nicht deutlich sein, aber dies ist eine enorm hilfreiche Möglichkeit, wie wir im nächsten Kapitel noch sehen werden. Zunächst aber wollen wir uns anschauen, was wir mit diesen Labels anstellen können, außer sie in der Podliste anzeigen zu lassen.

3.4 Eine Auswahl der Pods mithilfe von Labelselektoren auflisten

Ressourcen mit Labels zu versehen, nur um die Labels in der Ressourcenliste sehen zu können, wäre nicht sehr sinnvoll. Eine sehr praktische Verwendung haben Labels jedoch im Zusammenhang mit sogenannten *Labelselektoren*. Damit können Sie die Pods auswählen, die über ein bestimmtes Label verfügen, und eine Operation gesammelt an diesen Pods ausführen. Ein Labelselektor ist ein Kriterium, nach dem die Ressourcen gefiltert werden. Dabei ist es möglich, die Ressourcen je nachdem auszuwählen, ob sie:

- über ein bestimmtes Label (mit einem bestimmten Schlüssel) verfügen oder nicht.
- über ein Label mit einem bestimmten Schlüssel und Wert verfügen.
- über ein Label mit einem bestimmten Schlüssel, aber einem anderen als dem angegebenen Wert verfügen.

3.4.1 Pods anhand eines Labelselektors auflisten

Sehen wir uns nun die Verwendung von Labelselektoren an dem Beispiel unserer bisher erstellten Pods an. Um alle manuell erstellten Pods aufzulisten (also diejenigen mit dem Label `creation_method=manual`), gehen wir wie folgt vor:

```
$ kubectl get po -l creation_method=manual
NAME              READY  STATUS   RESTARTS  AGE
kubia-manual      1/1    Running  0         51m
kubia-manual-v2   1/1    Running  0         37m
```

Um alle Pods mit dem Label `env` unabhängig von dessen Wert aufzulisten, verwenden wir folgenden Befehl:

```
$ kubectl get po -l env
NAME              READY   STATUS    RESTARTS   AGE
kubia-manual-v2   1/1     Running   0          37m
```

Wir können auch diejenigen auflisten, die nicht über das Label `env` verfügen:

```
$ kubectl get po -l '!env'
NAME              READY   STATUS    RESTARTS   AGE
kubia-manual      1/1     Running   0          51m
kubia-zxzij       1/1     Running   0          10d
```

HINWEIS: Sie müssen `!env` in einfache Anführungszeichen stellen, damit die Bash-Shell nicht versucht, das Ausrufezeichen auszuwerten.

Wir können Pods auch mit den folgenden Labelselektoren auswählen:

- `creation_method!=manual` findet alle Pods, bei denen das Label `creation_method` einen anderen Wert als `manual` hat.
- `env in (prod,devel)` findet alle Pods, bei denen das Label `env` auf `prod` oder `devel` gesetzt ist.
- `env notin (prod,devel)` findet alle Pods, bei denen das Label `env` einen anderen Wert als `prod` oder `devel` hat.

In unserem Microservice-Beispiel können wir mit dem Labelselektor `app=pc` alle Pods auswählen, die zum Microservice für den Produktkatalog gehören (siehe Bild 3.8).

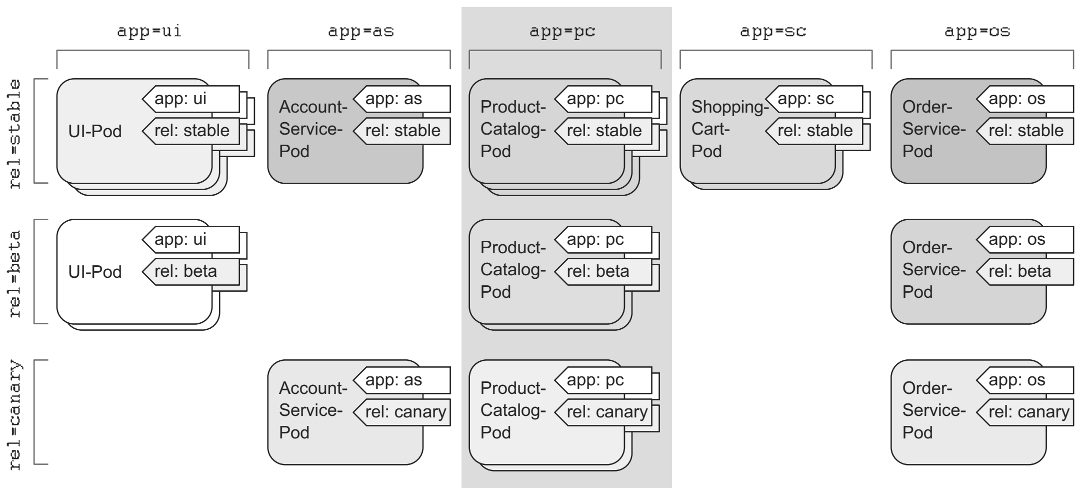

Bild 3.8 Auswahl der Pods für den Microservice des Produktkatalogs mithilfe des Labelselektors `app=pc`

3.4.2 Labelselektoren mit mehreren Bedingungen

Ein Selektor kann auch mehrere, durch Kommata getrennte Bedingungen enthalten. Um von dem Selektor ausgewählt zu werden, müssen die Ressourcen alle diese Bedingungen erfüllen. Wenn wir beispielsweise nur die Pods auswählen wollen, die das Beta-Release des Microservices für den Produktkatalog ausführen, können wir den Selektor `app=pc,rel=beta` verwenden (siehe Bild 3.9).

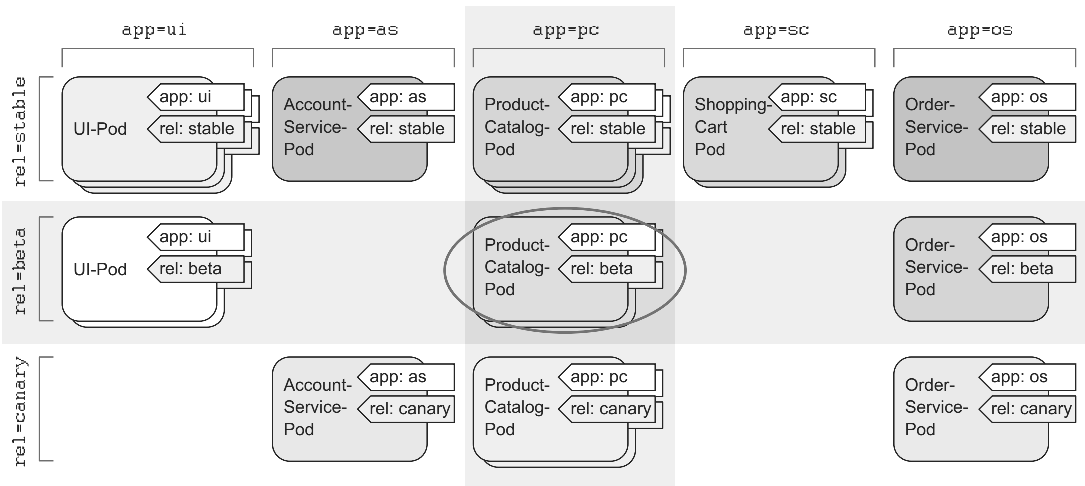

Bild 3.9 Auswählen von Pods mithilfe mehrerer Labelselektoren

Labelselektoren können wir nicht nur verwenden, um Pods für die Anzeige zu filtern, sondern auch, um Aktionen auf eine Auswahl von Pods anzuwenden. Weiter hinten in diesem Kapitel schauen wir uns beispielsweise an, wie wir damit mehrere Pods auf einmal löschen. Außerdem werden Labelselektoren nicht nur von `kubectl` genutzt, sondern auch intern verwendet, wie wir im nächsten Abschnitt sehen.

3.5 Die Podzuweisung mithilfe von Labels und Selektoren einschränken

Alle bisher von uns erstellten Pods wurden zufällig über die Arbeitsknoten verteilt. Wie ich bereits im vorherigen Kapitel gesagt habe, ist das die übliche Vorgehensweise in einem Kubernetes-Cluster. Da Kubernetes alle Knoten eines Clusters als eine riesige, geschlossene Bereitstellungsplattform exponiert, spielt es keine Rolle, welchem Knoten ein Pod zugewiesen ist. Jeder Pod bekommt genau die Menge an Rechenressourcen (CPU, Arbeitsspeicher usw.), die er angefordert hat, und auch der Zugang anderer Pods auf ihn hängt nicht davon ab, auf welchem Knoten er ausgeführt wird. Daher sollte es normalerweise nicht erforderlich sein, Kubernetes genau anzuweisen, wo es Ihre Pods zuweisen soll.

Es gibt jedoch besondere Fälle, in denen Sie zumindest ein gewisses Mitspracherecht bei dieser Zuweisung haben wollen. Ein gutes Beispiel dafür ist ein System mit inhomogener Hardware-Infrastruktur. Wenn einige der Arbeitsknoten über HDD- und andere über SSD-Laufwerke verfügen, dann kann es sein, dass Sie einige Pods lieber auf Knoten der einen Gruppe und andere lieber auf denen der anderen ausführen lassen möchten. Um ein weiteres Beispiel zu geben: Es kann auch sein, dass Sie Pods, die intensive GPU-Berechnungen vornehmen, nur Knoten zuweisen lassen möchten, die die erforderliche GPU-Beschleunigung bieten.

Sie sollten natürlich niemals genau den Knoten angeben, auf dem Sie einen Pod ausführen lassen wollen, denn dadurch würde die Anwendung an die Infrastruktur gekoppelt, was im Gegensatz zu dem Prinzip von Kubernetes steht, die Infrastruktur vor den Anwendungen zu verbergen, die darauf laufen. Wenn Sie Einfluss darauf nehmen wollen, wo ein Pod ausgeführt werden soll, geben Sie nicht konkret einen Knoten an, sondern beschreiben die Anforderungen an den Knoten und überlassen Kubernetes die Auswahl anhand dieser Anforderungen. Dazu verwenden Sie *Knotenlabels* und *Knotenlabelselektoren*.

3.5.1 Labels zur Klassifizierung von Arbeitsknoten

Wie Sie bereits wissen, sind Pods nicht die einzigen Kubernetes-Ressourcen, denen wir Labels anhängen können. Alle Kubernetes-Objekte können Labels tragen, auch Knoten. Wenn das Betriebsteam dem Cluster einen neuen Knoten hinzufügt, klassifiziert es diesen Knoten gewöhnlich, indem es ihm Labels mitgibt, die die Art der Hardware oder irgendwelche anderen Informationen angeben, die für die Zuweisung der Pods von Bedeutung sind.

Nehmen wir beispielsweise an, unser Cluster enthält einen Knoten mit einem Grafikprozessor (GPU). Um dieses Merkmal herauszustellen, fügen wir ihm wie folgt das Label `gpu=true` hinzu (wählen Sie dazu einfach einen der Knoten auf der von `kubectl get nodes` zurückgegebenen Liste aus):

```
$ kubectl label node gke-kubia-85f6-node-0rrx gpu=true
node "gke-kubia-85f6-node-0rrx" labeled
```

Jetzt können wir beim Auflisten der Knoten wie zuvor bei den Pods einen Labelselektor angeben, um nur die Knoten anzuzeigen, die über das Label `gpu=true` verfügen:

```
$ kubectl get nodes -l gpu=true
NAME                      STATUS  AGE
gke-kubia-85f6-node-0rrx  Ready   1d
```

Wie erwartet, gibt es nur einen Knoten mit diesem Label. Sie können auch versuchen, alle Knoten aufzuführen und `kubectl` anzuweisen, eine zusätzliche Spalte mit den Werten der `gpu`-Label hinzuzufügen (`kubectl get nodes -L gpu`).

3.5.2 Pods bestimmten Knoten zuweisen

Nehmen wir an, Sie wollen einen neuen Pod bereitstellen, der zur Erfüllung seiner Aufgaben auf einen Grafikprozessor zurückgreifen muss. Um den Scheduler anzuweisen, seine Auswahl auf die Knoten einzuschränken, die einen richtigen Grafikprozessor mitbringen, fügen wir der YAML-Definition des Pods einen Knotenselektor hinzu. Legen Sie dazu die Datei *kubia-gpu.yml* mit dem folgenden Inhalt an und erstellen Sie den Pod anschließend mit `kubectl create -f kubia-gpu.yaml`:

Listing 3.4 Einen Pod mithilfe eines Labelselektors einem bestimmten Knoten zuweisen: *kubia-gpu.yaml*

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-gpu
spec:
  nodeSelector:    | Dieser Knotenselektor weist Kubernetes an, diesen Pod nur
    gpu: "true"    | auf Knoten mit dem Label gpu=true bereitzustellen
  containers:
  - image: luksa/kubia
    name: kubia
```

Wir haben hier einfach im Abschnitt `spec` das Feld `nodeSelection` hinzugefügt. Wenn wir den Pod erstellen, trifft der Scheduler seine Auswahl nur unter Knoten mit dem Label `gpu=true` (das in unserem Beispiel nur ein einziger Knoten hat).

3.5.3 Zuweisung zu einem einzelnen Knoten

Da jeder Knoten über ein eindeutiges Label mit dem Schlüssel `kubernetes.io/hostname` und dem Hostnamen als Wert verfügt, können wir einen Pod auch einem einzelnen Knoten zuweisen. Wenn wir `nodeSelector` auf ein bestimmtes Hostnamenlabel setzen, kann das aber dazu führen, dass der Pod bei einem Ausfall des Knotens nicht zugewiesen werden kann. Richten Sie Ihre Überlegungen niemals auf einzelne Knoten, sondern auf logische Gruppen von Knoten, die von Knotenselektoren vorgegebene Kriterien erfüllen.

In diesem Abschnitt haben Sie einen Überblick darüber erhalten, was Labels sind, wie Labelselektoren funktionieren und wie Sie damit den Betrieb von Kubernetes beeinflussen können. Die Wichtigkeit und Nützlichkeit von Labelselektoren wird noch deutlicher, wenn wir in den nächsten beiden Kapiteln über Replikationscontroller und Dienste sprechen.

HINWEIS: Weitere Möglichkeiten, um zu beeinflussen, welchen Knoten ein Pod zugewiesen wird, lernen Sie in Kapitel 16 kennen.

3.6 Pods mit Anmerkungen versehen

Neben Labels können Pods und andere Objekte auch *Anmerkungen (Annotations)* enthalten. Dies sind ebenfalls Schlüssel-Wert-Paare, weshalb Sie ähnlich wie Labels verwendet werden. Allerdings machen sie im Gegensatz zu Labels keine Angaben, die zur Identifizierung verwendet werden können. Es ist nicht möglich, Objekte mithilfe von Anmerkungen zu gruppieren. Anmerkungsselektoren, mit denen Sie Objekte auf ähnliche Weise auswählen könnten wie mit Labelselektoren, gibt es nicht.

Dafür können Anmerkungen aber viel umfangreichere Informationen enthalten. Sie sind hauptsächlich als Werkzeuge gedacht. Manche Anmerkungen fügt Kubernetes automatisch zu Objekten hinzu, während andere manuell von den Benutzern angegeben werden müssen.

Anmerkungen sind auch bei der Einführung neuer Funktionen in Kubernetes gebräuchlich. Gewöhnlich werden in den Alpha- und Beta-Versionen neuer Merkmale keine neuen Felder zu API-Objekten hinzugefügt, sondern Anmerkungen. Wenn die erforderlichen Änderungen an der API geklärt und von allen Beteiligten abgesegnet wurden, werden die neuen Felder eingeführt und die entsprechenden Anmerkungen entfernt.

Besonders sinnvoll sind Anmerkungen, um den Pods oder anderen API-Objekten Beschreibungen hinzuzufügen, sodass jeder, der den Cluster nutzt, rasch Informationen über die einzelnen Objekte nachschlagen kann. Beispielsweise kann eine Anmerkung den Namen der Person angeben, die das Objekt erstellt hat, was die Zusammenarbeit aller Personen, die mit dem Cluster zu tun haben, stark vereinfacht.

3.6.1 Die Anmerkungen zu einem Objekt einsehen

Um uns ein Beispiel der Art von Anmerkungen anzusehen, die Kubernetes automatisch zu einem Pod hinzufügt, können wir die vollständige YAML-Definition des Pods abrufen oder den Befehl `kubectl describe` verwenden. Hier versuchen wir es zunächst mit der ersten Möglichkeit:

Listing 3.5 Die Anmerkungen zu einem Pod

```
$ kubectl get po kubia-zxzij -o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubernetes.io/created-by: |
      {"kind":"SerializedReference", "apiVersion":"v1",
      "reference":{"kind":"ReplicationController", "namespace":"default", ...
```

Ohne allzu sehr ins Detail zu gehen, können Sie schon erkennen, dass die Anmerkung `kubernetes.io/created-by` JSON-Daten über das Objekt enthält, das den Pod erstellt hat. Solche Angaben wollen Sie bestimmt nicht in ein Label stellen! Labels sollen kurz sein, wohingegen Anmerkungen relativ große Datenmengen enthalten können (insgesamt bis zu 256 KB).

HINWEIS: Die Anmerkung `kubernetes.io/created-by` gilt in Version 1.8 schon als veraltet und unerwünscht und wird in Version 1.9 entfernt, weshalb Sie sie in YAML-Definitionen nicht mehr sehen werden.

3.6.2 Anmerkungen hinzufügen und ändern

Ebenso wie Labels können Anmerkungen zu Pods hinzugefügt werden, während diese erstellt werden. Es ist aber auch möglich, sie später hinzuzufügen und bereits vorhandene Anmerkungen zu ändern. Die einfachste Möglichkeit, um einem Objekt nachträglich eine Anmerkung hinzuzufügen, bietet der Befehl `kubectl annotate`.

Versuchen wir, unseren Pod `kubia-manual` um eine Anmerkung zu ergänzen:

```
$ kubectl annotate pod kubia-manual mycompany.com/someannotation="foo bar"
pod "kubia-manual" annotated
```

Hier haben wir die Anmerkung `mycompany.com/someannotation` mit dem Wert `"foo bar"` hinzugefügt. Um Konflikte zu vermeiden, sollten Sie für Anmerkungsschlüssel das hier gezeigte Format mit eindeutigen Präfixen verwenden. Wenn mehrere Werkzeuge oder Bibliotheken Anmerkungen zu Objekten machen, besteht sonst die Gefahr, dass sie gegenseitig ihre Anmerkungen überschreiben.

Um die hinzugefügte Anmerkung einzusehen, können wir `kubectl describe` versuchen:

```
$ kubectl describe pod kubia-manual | grep Annotations
...
Annotations: mycompany.com/someannotation=foo bar
...
```

3.7 Ressourcen mithilfe von Namespaces gruppieren

Kehren wir noch einmal zu den Labels zurück. Wir haben schon gesehen, wie wir Pods und andere Objekte gruppieren können. Da jedes Objekt aber mehrere Labels tragen kann, ist es möglich, dass sich Objektgruppen überlappen. Außerdem werden bei der Arbeit mit dem Cluster (z. B. mithilfe von `kubectl`) immer alle Objekte angezeigt, solange sie nicht ausdrücklich einen Labelselektor angeben.

Was aber tun Sie, wenn Sie Objekte in voneinander getrennte, nicht überlappende Gruppen aufteilen möchten, etwa um nur mit einer einzigen Gruppe zu arbeiten? Für diese und andere Zwecke gruppiert Kubernetes Objekte auch mithilfe von *Namespaces*. Dabei handelt es sich nicht um die Linux-Namespaces aus Kapitel 2, die dazu dienen, Prozesse voneinander zu isolieren. Kubernetes-Namespaces richten einfach einen Gültigkeitsbereich für

Namen ein. Anstatt alle Ressourcen in einem einzigen Namespace unterzubringen, können Sie sie auf mehrere aufteilen, sodass Sie einzelne Ressourcennamen mehrfach (nämlich in verschiedenen Namespaces) verwenden können.

3.7.1 Der Bedarf für Namespaces

Durch die Verwendung mehrerer Namespaces können Sie komplizierte Systeme mit zahlreichen Bestandteilen in kleine, scharf umrissene Gruppen zerlegen. Damit ist es beispielsweise möglich, Ressourcen in einer Mehrbenutzerumgebung aufzuteilen, z. B. in *Produktion*, *Entwicklung* und *Qualitätssicherung* oder auf jede andere sinnvolle Weise. Ressourcennamen müssen nur innerhalb eines Namespaces eindeutig sein, wohingegen in zwei verschiedenen Namespaces Ressourcen desselben Namens auftreten dürfen. Die meisten Arten von Ressourcen können einem Namespace zugeordnet werden, doch bei einigen wenigen ist das nicht der Fall. Ein Beispiel dafür sind Knoten, die global gelten und nicht an einen einzelnen Namespace gebunden sind. Weitere clusterweit gültige Ressourcen werden Sie noch in späteren Kapiteln kennenlernen.

Sehen wir uns nun an, wie Namespaces verwendet werden.

3.7.2 Andere Namespaces und die zugehörigen Pods finden

Als Erstes listen wir alle Namespaces in unserem Cluster auf:

```
$ kubectl get ns
NAME         LABELS   STATUS   AGE
default      <none>   Active   1h
kube-public  <none>   Active   1h
kube-system  <none>   Active   1h
```

Bis jetzt haben wir uns immer nur im *Standardnamespace* (`default`) bewegt. Beim Auflisten von Ressourcen mit dem Befehl `kubectl get` haben wir niemals ausdrücklich einen Namespace angegeben, weshalb `kubectl` immer auf den Standardnamespace zurückgegriffen und uns die darin enthaltenen Objekte angezeigt hat. Wie diese Liste zeigt, gibt es jedoch auch die Namespaces `kube-public` und `kube-system`. Wir können `kubectl` wie folgt anweisen, ausschließlich die Pods im Namespace `kube-system` anzuzeigen:

```
$ kubectl get po --namespace kube-system
NAME                               READY  STATUS   RESTARTS  AGE
fluentd-cloud-kubia-e8fe-node-txje 1/1    Running  0         1h
heapster-v11-fz1ge                 1/1    Running  0         1h
kube-dns-v9-p8a4t                  0/4    Pending  0         1h
kube-ui-v4-kdlai                   1/1    Running  0         1h
l7-lb-controller-v0.5.2-bue96      2/2    Running  92        1h
```

TIPP: Statt `--namespace` können Sie auch `-n` schreiben.

Mit diesen Pods werden wir uns weiter hinten in diesem Buch noch beschäftigen. (Machen Sie sich auch keine Sorgen, wenn die hier gezeigten Pods nicht genau die gleichen sind, die Sie auf Ihrem System sehen.) Die Bezeichnung des Namespaces macht deutlich, dass diese Ressourcen zum Kubernetes-System selbst gehören. Dadurch, dass sie in einem eigenen Namespace untergebracht werden, ist alles sauber geordnet. Würden sie dagegen zusammen mit den von uns selbst erstellten Ressourcen im Standardnamespace stehen, wäre es ziemlich schwierig herauszufinden, was wozu gehört. Dabei bestünde auch die Gefahr, versehentlich Systemressourcen zu löschen.

Namespaces ermöglichen es uns, Ressourcen, die nicht zusammengehören, in nicht überlappende Gruppen aufzuteilen. Wenn mehrere Benutzer oder Benutzergruppen denselben Kubernetes-Cluster verwenden und dabei jeweils ihren eigenen Satz von Ressourcen nutzen, sollten sie auch alle ihre eigenen Namespaces haben. Dadurch besteht keine Gefahr mehr, dass sie versehentlich die Ressourcen der anderen Benutzer löschen, und sie müssen sich auch keine Gedanken mehr über Namenskonflikte machen, da die Namen ihrer Ressourcen wie bereits erwähnt nur jeweils in ihrem eigenen Namespace gültig sind.

Neben der Trennung von Ressourcen bieten Namespaces auch die Möglichkeit, den Zugriff auf manche Ressourcen auf bestimmte Benutzer einzuschränken oder die Menge der Rechenressourcen zu begrenzen, die einzelnen Benutzern zur Verfügung stehen. Mehr darüber erfahren Sie in Kapitel 12 bis 14.

3.7.3 Namespaces erstellen

Ein Namespace ist eine Kubernetes-Ressource wie jede andere, weshalb Sie sie dadurch anlegen können, dass Sie eine YAML-Datei auf den Kubernetes-API-Server stellen. Das wollen wir im Folgenden tun.

Einen Namespace mithilfe einer YAML-Datei erstellen

Erstellen Sie als Erstes die Datei *custom-namespace.yaml* mit dem folgenden Inhalt (Sie finden sie auch im Codearchiv zu diesem Buch):

Listing 3.6 YAML-Definition eines Namespaces: *custom-namespace.yaml*

```
apiVersion: v1
kind: Namespace   < Dies besagt, dass wir einen Namespace erstellen
metadata:
  name: custom-namespace   < Der Name des Namespaces
```

Anschließend platzieren Sie die Datei mithilfe von `kubectl` auf dem Kubernetes-API-Server:

```
$ kubectl create -f custom-namespace.yaml
namespace "custom-namespace" created
```

Einen Namespace mithilfe von kubectl create erstellen

Es ist zwar nicht weiter schwierig, eine Datei wie die zuvor gezeigte zu schreiben, aber trotzdem umständlich. Glücklicherweise gibt es mit `kubectl create namespace` auch einen

eigenen Befehl zum Erstellen von Namespaces, womit sich der Vorgang viel schneller erledigen lässt als mit einer YAML-Datei. Wir haben trotzdem die andere Vorgehensweise gewählt, um noch einmal deutlich zu machen, dass alle Dinge in Kubernetes API-Objekte sind, die Sie erstellen, lesen, ändern und löschen können, indem Sie ein YAML-Manifest auf den API-Server stellen.

Allerdings hätten wir den Namespace auch einfach wie folgt anlegen können:

```
$ kubectl create namespace custom-namespace
namespace "custom-namespace" created
```

HINWEIS: Die meisten Objekte müssen der in RFC 1035 aufgeführten Namenskonvention (Domänennamen) folgen, weshalb sie nur Buchstaben, Ziffern, Bindestriche und Punkte enthalten dürfen. Allerdings sind in den Namen von Namespaces (und einiger weniger anderer Ressourcen) auch keine Punkte zulässig.

3.7.4 Objekte in anderen Namespaces verwalten

Um Ressourcen in dem neuen Namespace zu erstellen, können wir Ihrer YAML-Definition das Attribut `namespace: custom-namespace` zum Feld `metadata` hinzufügen bzw. im Befehl `kubectl create` den Namespace angeben:

```
$ kubectl create -f kubia-manual.yaml -n custom-namespace
Pod "kubia-manual" created
```

Jetzt gibt es zwei Pods mit dem Namen `kubia-manual`, einen im Standardnamespace und einen in `custom-namespace`.

Um Objekte in anderen Namespaces aufzulisten, zu beschreiben, zu ändern oder zu löschen, müssen wir `kubectl` das Flag `--namespace` (oder `-n`) übergeben. Ohne die Angabe des Namespaces führt `kubectl` die Aktion in dem Standardnamespace aus, der im aktuellen `kubectl`-Kontext eingerichtet ist. Den Namespace des aktuellen Kontextes und den aktuellen Kontext selbst können Sie mithilfe der `kubectl config`-Befehle ändern. Mehr darüber erfahren Sie in Anhang A.

TIPP: Um den Standardnamespace schnell zu ändern, können Sie den Alias `alias kcd='kubectl config set-context $(kubectl config current-context) --namespace '` einrichten. Anschließend können Sie mit kcd <Namespace> zwischen den Namensräumen umschalten.

3.7.5 Die Trennung der Namespaces

Zum Abschluss dieses Abschnitts über Namespaces möchte ich noch erklären, was Namespaces nicht können, zumindest nicht für sich allein. Mit Namespaces können Sie Objekte in scharf abgegrenzte Gruppen aufteilen, was es Ihnen ermöglicht, Operationen gezielt auf die Mitglieder eines Namespaces anzuwenden, aber eine Isolierung laufender Objekte lässt sich damit nicht erreichen.

Nehmen wir an, verschiedene Benutzer stellen Pods jeweils in eigenen Namespaces bereit. Sind diese Pods dann voneinander isoliert und können nicht miteinander kommunizieren? Nicht notwendigerweise. Ob Namespaces eine Isolierung im Netzwerk bewirken, hängt von der verwendeten Netzwerklösung ab. Wenn diese Lösung keine Netzwerkisolierung zwischen Namespaces bietet, kann ein Pod im Namespace `foo`, der die IP-Adresse eines Pods im Namespace `bar` kennt, Datenverkehr wie beispielsweise HTTP-Anforderungen an diesen anderen Pod senden.

3.8 Pods stoppen und entfernen

Die Pods, die wir bis jetzt erstellt haben – vier im Standardnamespace und einer in `custom-namespace` –, sollten noch alle laufen. Da wir sie nicht mehr benötigen, wollen wir sie alle stoppen.

3.8.1 Pods unter Angabe des Namens löschen

Als Erstes löschen wir den Pod `kubia-gpu` unter Angabe seines Namens:

```
$ kubectl delete po kubia-gpu
pod "kubia-gpu" deleted
```

Dadurch weisen wir Kubernetes an, alle Container zu beenden, die zu dem Pod gehören. Kubernetes sendet das Signal `SIGTERM` an den Prozess und wartet eine gewisse Zeit lang (in der Standardeinstellung 30 Sekunden) darauf, dass er sauber heruntergefahren wird. Geschieht das nicht, wird der Prozess mit `SIGKILL` zwangsweise beendet. Damit die Prozesse immer sauber heruntergefahren werden, müssen daher sie das Signal `SIGTERM` korrekt handhaben.

TIPP: Sie können auch mehrere Pods auf einmal löschen, indem Sie einfach mehrere Namen durch Leerzeichen getrennt angeben (z. B. `kubectl delete po pod1 pod2`).

3.8.2 Pods mithilfe von Labelselektoren löschen

Anstatt jeden Pod, den wir löschen wollen, namentlich anzugeben, können wir auch Labelselektoren nutzen, um sowohl `kubia-manual` als auch `kubia-manual-v2` zu stoppen. Beide Pods haben das Label `creation_method=manual`, sodass wir sie wie folgt auf einen Streich löschen können:

```
$ kubectl delete po -l creation_method=manual
pod "kubia-manual" deleted
pod "kubia-manual-v2" deleted
```

In unserem Microservice-Beispiel mit Dutzenden (oder möglicherweise sogar Hunderten) von Pods können wir etwa alle Canary-Pods auf einmal löschen, indem wir den Labelselektor `rel=canary` angeben (siehe Bild 3.10):

```
$ kubectl delete po -l rel=canary
```

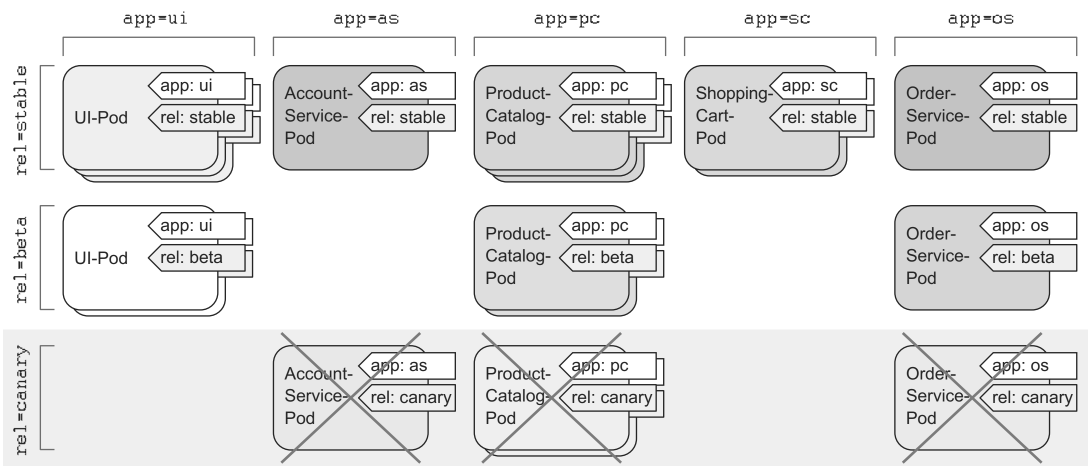

Bild 3.10 Auswählen und Löschen aller Canary-Pods mit dem Labelselektor `rel=canary`

3.8.3 Pods durch Entfernen eines ganzen Namespaces löschen

Aber nun zurück zu unserem Übungsbeispiel! Was machen wir mit dem Pod in `custom-namespace`? Da wir weder die Pods in diesem Namespace noch den Namespace selbst benötigen, können wir ihn einfach komplett löschen (wobei automatisch alle darin enthaltenen Pods entfernt werden):

```
$ kubectl delete ns custom-namespace
namespace "custom-namespace" deleted
```

3.8.4 Alle Pods in einem Namespace löschen und den Namespace erhalten

Wir haben jetzt schon fast komplett aufgeräumt. Allerdings wird der Pod, den wir in Kapitel 2 mit dem Befehl `kubectl run` erstellt haben, nach wie vor ausgeführt:

```
$ kubectl get pods
NAME         READY  STATUS   RESTARTS  AGE
kubia-zxzij  1/1    Running  0         1d
```

Anstatt gezielt diesen einen Pod zu löschen, weisen wir Kubernetes hier mit der Option `--all` an, alle Pods im aktuellen Namespace zu entfernen:

```
$ kubectl delete po --all
pod "kubia-zxzij" deleted
```

Nun wollen wir uns vergewissern, dass auch wirklich keine Pods mehr laufen:

```
$ kubectl get pods
NAME         READY  STATUS       RESTARTS  AGE
kubia-09as0  1/1    Running      0         1d
kubia-zxzij  1/1    Terminating  0         1d
```

Halt, was ist denn hier los? Wir sehen zwar, dass `kubia-zxzij` beendet wird, aber da ist ja plötzlich ein neuer Pod namens `kubia-09as0` zu sehen, der vorher nicht da war. Wie oft wir auch immer alle Pods löschen, es wird stets ein neuer Pod namens `kubia-irgendwas` auftauchen.

Denken Sie daran zurück, wie wir unseren ersten Pod mit dem Befehl `kubectl run` angelegt haben. In Kapitel 2 habe ich erwähnt, dass dadurch nicht unmittelbar ein Pod, sondern ein Replikationscontroller erstellt wird, der wiederum den Pod anlegt. Wenn wir einen Pod löschen, der von einem Replikationscontroller erstellt wurde, dann legt der Controller sofort einen neuen Pod an. Um den Pod zu entfernen, müssen wir den Replikationscontroller löschen.

3.8.5 (Fast) alle Ressourcen in einem Namespace löschen

Wir können den Replikationscontroller und alle Pods sowie alle Dienste mit einem einzigen Befehl löschen, indem wir alle Ressourcen aus dem aktuellen Namespace entfernen:

```
$ kubectl delete all --all
pod "kubia-09as0" deleted
replicationcontroller "kubia" deleted
service "kubernetes" deleted
service "kubia-http" deleted
```

Das erste `all` in diesem Befehl gibt an, dass wir alle Arten von Ressourcen löschen wollen, und mit der Option `--all` sorgen wir dafür, dass sämtliche Instanzen der Ressourcen entfernt werden, ohne dass wir sie namentlich aufführen müssen. (Diese Option haben wir schon im vorherigen `delete`-Befehl verwendet.)

HINWEIS: Auch bei der Verwendung des Schlüsselworts `all` wird nicht alles gelöscht. Einige Ressourcen (z. B. die in Kapitel 7 beschriebenen *Geheimnisse*) bleiben dabei erhalten und müssen ausdrücklich gelöscht werden.

Während des Löschvorgangs gibt `kubectl` den Namen jeder Ressource aus, die gerade entfernt wird. In der Liste sehen Sie den Replikationscontroller `kubia` sowie den Dienst `kubia-http`, die wir beide in Kapitel 2 erstellt haben.

HINWEIS: Beim Ausführen des Befehls `kubectl delete all --all` wird auch der Dienst `kubernetes` gelöscht, allerdings wird er nach wenigen Augenblicken automatisch neu erstellt.

3.9 Zusammenfassung

Nach der Lektüre dieses Kapitels sollten Sie sich jetzt mit den wichtigsten Bausteinen von Kubernetes auskennen. Jedes andere Element, das wir in den folgenden Kapiteln kennenlernen werden, steht in direktem Zusammenhang mit Pods.

In diesem Kapitel haben Sie Folgendes gelernt:

- Sie wissen, wann Sie mehrere Container in einem Pod zusammenfassen sollten und wann nicht.
- Pods können mehrere Prozesse ausführen und ähneln physischen Hosts.
- Sie können YAML- und JSON-Deskriptoren schreiben, um Pods zu erstellen, sich ihre Spezifikation anzusehen und ihren aktuellen Zustand zu untersuchen.
- Sie können Labels und Labelselektoren verwenden, um Pods zu ordnen und Operationen auf mehrere Pods auf einmal anzuwenden.
- Sie können Knotenlabels verwenden, um Pods nur Knoten zuzuweisen, die bestimmte Merkmale aufweisen.
- Mithilfe von Anmerkungen lassen sich umfangreichere Informationen zu Pods hinzufügen. Solche Anmerkungen können von Personen, aber auch von Tools und Bibliotheken erstellt werden.
- Namespaces ermöglichen es, dass mehrere Teams denselben Cluster so nutzen, als würden sie jeweils mit ihrem eigenen Cluster arbeiten.
- Mit dem Befehl `kubectl explain` können Sie Informationen über jegliche Kubernetes-Ressourcen nachschlagen.

Im nächsten Kapitel lernen Sie Replikationscontroller und andere Ressourcen zur Verwaltung von Pods kennen.