SAMPLE CHAPTER

Kubernetes IN ACTION

Marko Lukša

MANNING

Kubernetes in Action

by Marko Lukša

Chapter 3

brief contents

Part 1 Overview

Part 2 Core concepts

PART 3 BEYOND THE BASICS

3

Pods: running containers in Kubernetes

This chapter covers

- Creating, running, and stopping pods
- Organizing pods and other resources with labels
- Performing an operation on all pods with a specific label
- Using namespaces to split pods into non-overlapping groups
- Scheduling pods onto specific types of worker nodes

The previous chapter should have given you a rough picture of the basic components you create in Kubernetes and at least an outline of what they do. Now, we'll start reviewing all types of Kubernetes objects (or *resources*) in greater detail, so you'll understand when, how, and why to use each of them. We'll start with pods, because they're the central, most important, concept in Kubernetes. Everything else either manages, exposes, or is used by pods.

3.1 Introducing pods

You've already learned that a pod is a co-located group of containers and represents the basic building block in Kubernetes. Instead of deploying containers individually, you always deploy and operate on a pod of containers. We're not implying that a pod always includes more than one container—it's common for pods to contain only a single container. The key thing about pods is that when a pod does contain multiple containers, all of them are always run on a single worker node—it never spans multiple worker nodes, as shown in figure 3.1.

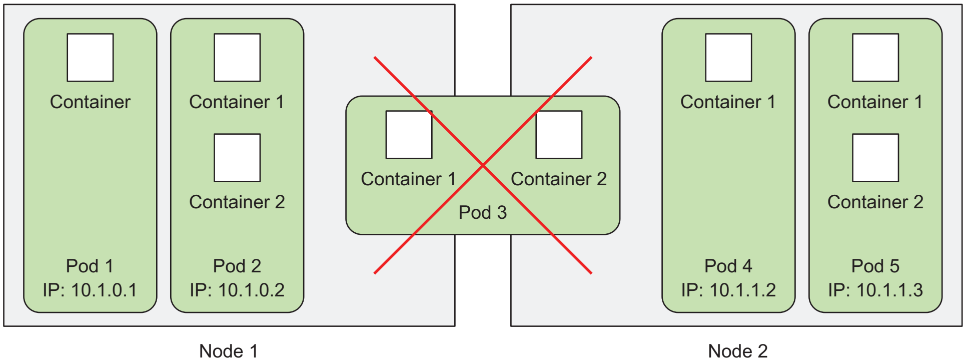

Figure 3.1 All containers of a pod run on the same node. A pod never spans two nodes.

3.1.1 Understanding why we need pods

But why do we even need pods? Why can't we use containers directly? Why would we even need to run multiple containers together? Can't we put all our processes into a single container? We'll answer those questions now.

UNDERSTANDING WHY MULTIPLE CONTAINERS ARE BETTER THAN ONE CONTAINER RUNNING MULTIPLE PROCESSES

Imagine an app consisting of multiple processes that either communicate through *IPC* (Inter-Process Communication) or through locally stored files, which requires them to run on the same machine. Because in Kubernetes you always run processes in containers and each container is much like an isolated machine, you may think it makes sense to run multiple processes in a single container, but you shouldn't do that.

Containers are designed to run only a single process per container (unless the process itself spawns child processes). If you run multiple unrelated processes in a single container, it is your responsibility to keep all those processes running, manage their logs, and so on. For example, you'd have to include a mechanism for automatically restarting individual processes if they crash. Also, all those processes would log to the same standard output, so you'd have a hard time figuring out what process logged what.

Therefore, you need to run each process in its own container. That's how Docker and Kubernetes are meant to be used.

3.1.2 Understanding pods

Because you're not supposed to group multiple processes into a single container, it's obvious you need another higher-level construct that will allow you to bind containers together and manage them as a single unit. This is the reasoning behind pods.

A pod of containers allows you to run closely related processes together and provide them with (almost) the same environment as if they were all running in a single container, while keeping them somewhat isolated. This way, you get the best of both worlds. You can take advantage of all the features containers provide, while at the same time giving the processes the illusion of running together.

Understanding the partial isolation between containers of the same pod

In the previous chapter, you learned that containers are completely isolated from each other, but now you see that you want to isolate groups of containers instead of individual ones. You want containers inside each group to share certain resources, although not all, so that they're not fully isolated. Kubernetes achieves this by configuring Docker to have all containers of a pod share the same set of Linux namespaces instead of each container having its own set.

Because all containers of a pod run under the same Network and UTS namespaces (we're talking about Linux namespaces here), they all share the same hostname and network interfaces. Similarly, all containers of a pod run under the same IPC namespace and can communicate through IPC. In the latest Kubernetes and Docker versions, they can also share the same PID namespace, but that feature isn't enabled by default.

> **NOTE** When containers of the same pod use separate PID namespaces, you only see the container's own processes when running `ps aux` in the container.

But when it comes to the filesystem, things are a little different. Because most of the container's filesystem comes from the container image, by default, the filesystem of each container is fully isolated from other containers. However, it's possible to have them share file directories using a Kubernetes concept called a *Volume*, which we'll talk about in chapter 6.

Understanding how containers share the same IP and port space

One thing to stress here is that because containers in a pod run in the same Network namespace, they share the same IP address and port space. This means processes running in containers of the same pod need to take care not to bind to the same port numbers or they'll run into port conflicts. But this only concerns containers in the same pod. Containers of different pods can never run into port conflicts, because each pod has a separate port space. All the containers in a pod also have the same loopback network interface, so a container can communicate with other containers in the same pod through localhost.

INTRODUCING THE FLAT INTER-POD NETWORK

All pods in a Kubernetes cluster reside in a single flat, shared, network-address space (shown in figure 3.2), which means every pod can access every other pod at the other pod's IP address. No *NAT* (Network Address Translation) gateways exist between them. When two pods send network packets between each other, they'll each see the actual IP address of the other as the source IP in the packet.

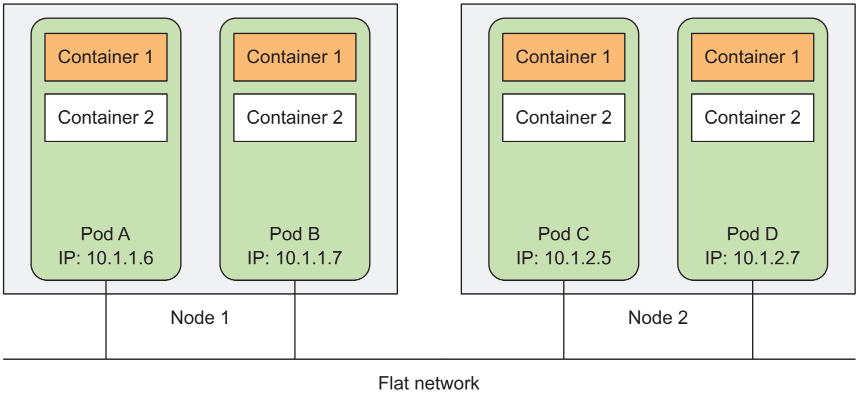

Figure 3.2 Each pod gets a routable IP address and all other pods see the pod under that IP address.

Consequently, communication between pods is always simple. It doesn't matter if two pods are scheduled onto a single or onto different worker nodes; in both cases the containers inside those pods can communicate with each other across the flat NAT-less network, much like computers on a local area network (LAN), regardless of the actual inter-node network topology. Like a computer on a LAN, each pod gets its own IP address and is accessible from all other pods through this network established specifically for pods. This is usually achieved through an additional software-defined network layered on top of the actual network.

To sum up what's been covered in this section: pods are logical hosts and behave much like physical hosts or VMs in the non-container world. Processes running in the same pod are like processes running on the same physical or virtual machine, except that each process is encapsulated in a container.

3.1.3 Organizing containers across pods properly

You should think of pods as separate machines, but where each one hosts only a certain app. Unlike the old days, when we used to cram all sorts of apps onto the same host, we don't do that with pods. Because pods are relatively lightweight, you can have as many as you need without incurring almost any overhead. Instead of stuffing everything into a single pod, you should organize apps into multiple pods, where each one contains only tightly related components or processes.

Having said that, do you think a multi-tier application consisting of a frontend application server and a backend database should be configured as a single pod or as two pods?

Splitting multi-tier apps into multiple pods

Although nothing is stopping you from running both the frontend server and the database in a single pod with two containers, it isn't the most appropriate way. We've said that all containers of the same pod always run co-located, but do the web server and the database really need to run on the same machine? The answer is obviously no, so you don't want to put them into a single pod. But is it wrong to do so regardless? In a way, it is.

If both the frontend and backend are in the same pod, then both will always be run on the same machine. If you have a two-node Kubernetes cluster and only this single pod, you'll only be using a single worker node and not taking advantage of the computational resources (CPU and memory) you have at your disposal on the second node. Splitting the pod into two would allow Kubernetes to schedule the frontend to one node and the backend to the other node, thereby improving the utilization of your infrastructure.

Splitting into multiple pods to enable individual scaling

Another reason why you shouldn't put them both into a single pod is scaling. A pod is also the basic unit of scaling. Kubernetes can't horizontally scale individual containers; instead, it scales whole pods. If your pod consists of a frontend and a backend container, when you scale up the number of instances of the pod to, let's say, two, you end up with two frontend containers and two backend containers.

Usually, frontend components have completely different scaling requirements than the backends, so we tend to scale them individually. Not to mention the fact that backends such as databases are usually much harder to scale compared to (stateless) frontend web servers. If you need to scale a container individually, this is a clear indication that it needs to be deployed in a separate pod.

Understanding when to use multiple containers in a pod

The main reason to put multiple containers into a single pod is when the application consists of one main process and one or more complementary processes, as shown in figure 3.3.

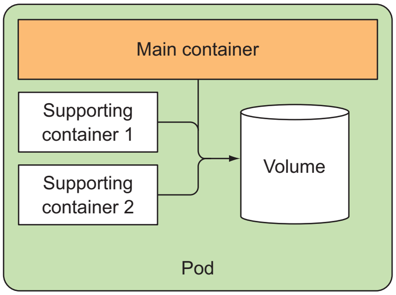

Figure 3.3 Pods should contain tightly coupled containers, usually a main container and containers that support the main one.

For example, the main container in a pod could be a web server that serves files from a certain file directory, while an additional container (a sidecar container) periodically downloads content from an external source and stores it in the web server's directory. In chapter 6 you'll see that you need to use a Kubernetes Volume that you mount into both containers.

Other examples of sidecar containers include log rotators and collectors, data processors, communication adapters, and others.

Deciding when to use multiple containers in a pod

To recap how containers should be grouped into pods—when deciding whether to put two containers into a single pod or into two separate pods, you always need to ask yourself the following questions:

- Do they need to be run together or can they run on different hosts?
- Do they represent a single whole or are they independent components?
- Must they be scaled together or individually?

Basically, you should always gravitate toward running containers in separate pods, unless a specific reason requires them to be part of the same pod. Figure 3.4 will help you memorize this.

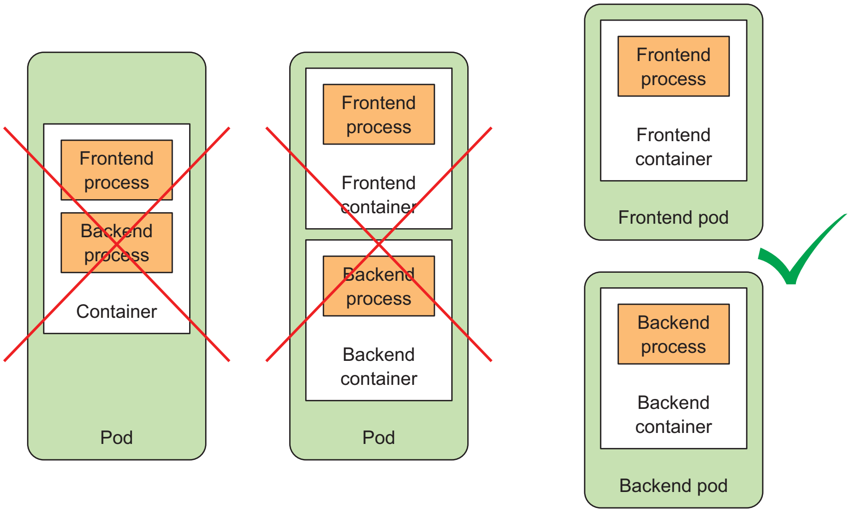

Figure 3.4 A container shouldn't run multiple processes. A pod shouldn't contain multiple containers if they don't need to run on the same machine.

Although pods can contain multiple containers, to keep things simple for now, you'll only be dealing with single-container pods in this chapter. You'll see how multiple containers are used in the same pod later, in chapter 6.

3.2 Creating pods from YAML or JSON descriptors

Pods and other Kubernetes resources are usually created by posting a JSON or YAML manifest to the Kubernetes REST API endpoint. Also, you can use other, simpler ways of creating resources, such as the `kubectl run` command you used in the previous chapter, but they usually only allow you to configure a limited set of properties, not all. Additionally, defining all your Kubernetes objects from YAML files makes it possible to store them in a version control system, with all the benefits it brings.

To configure all aspects of each type of resource, you'll need to know and understand the Kubernetes API object definitions. You'll get to know most of them as you learn about each resource type throughout this book. We won't explain every single property, so you should also refer to the Kubernetes API reference documentation at http://kubernetes.io/docs/reference/ when creating objects.

3.2.1 Examining a YAML descriptor of an existing pod

You already have some existing pods you created in the previous chapter, so let's look at what a YAML definition for one of those pods looks like. You'll use the `kubectl get` command with the `-o yaml` option to get the whole YAML definition of the pod, as shown in the following listing.

Listing 3.1 Full YAML of a deployed pod

```
$ kubectl get po kubia-zxzij -o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubernetes.io/created-by: ...
  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:
  containers:
  - image: luksa/kubia
    imagePullPolicy: IfNotPresent
    name: kubia
    ports:
    - containerPort: 8080
      protocol: TCP
    resources:
      requests:
        cpu: 100m
```

Kubernetes API version used in this YAML descriptor

Type of Kubernetes object/resource

Pod metadata (name, labels, annotations, and so on)

Pod specification/contents (list of pod's containers, volumes, and so on)

```
    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:
  conditions:
  - lastProbeTime: null
    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
```

Pod specification/ contents (list of pod's containers, volumes, and so on)

Detailed status of the pod and its containers

I know this looks complicated, but it becomes simple once you understand the basics and know how to distinguish between the important parts and the minor details. Also, you can take comfort in the fact that when creating a new pod, the YAML you need to write is much shorter, as you'll see later.

INTRODUCING THE MAIN PARTS OF A POD DEFINITION

The pod definition consists of a few parts. First, there's the Kubernetes API version used in the YAML and the type of resource the YAML is describing. Then, three important sections are found in almost all Kubernetes resources:

- *Metadata* includes the name, namespace, labels, and other information about the pod.
- *Spec* contains the actual description of the pod's contents, such as the pod's containers, volumes, and other data.

- *Status* contains the current information about the running pod, such as what condition the pod is in, the description and status of each container, and the pod's internal IP and other basic info.

Listing 3.1 showed a full description of a running pod, including its status. The `status` part contains read-only runtime data that shows the state of the resource at a given moment. When creating a new pod, you never need to provide the `status` part.

The three parts described previously show the typical structure of a Kubernetes API object. As you'll see throughout the book, all other objects have the same anatomy. This makes understanding new objects relatively easy.

Going through all the individual properties in the previous YAML doesn't make much sense, so, instead, let's see what the most basic YAML for creating a pod looks like.

3.2.2 Creating a simple YAML descriptor for a pod

You're going to create a file called kubia-manual.yaml (you can create it in any directory you want), or download the book's code archive, where you'll find the file inside the Chapter03 directory. The following listing shows the entire contents of the file.

Listing 3.2 A basic pod manifest: kubia-manual.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-manual
spec:
  containers:
  - image: luksa/kubia
    name: kubia
    ports:
    - containerPort: 8080
      protocol: TCP
```

Descriptor conforms to version v1 of Kubernetes API

You're describing a pod.

The name of the pod

Container image to create the container from

Name of the container

The port the app is listening on

I'm sure you'll agree this is much simpler than the definition in listing 3.1. Let's examine this descriptor in detail. It conforms to the `v1` version of the Kubernetes API. The type of resource you're describing is a `pod`, with the name `kubia-manual`. The pod consists of a single container based on the `luksa/kubia` image. You've also given a name to the container and indicated that it's listening on port `8080`.

Specifying container ports

Specifying ports in the pod definition is purely informational. Omitting them has no effect on whether clients can connect to the pod through the port or not. If the container is accepting connections through a port bound to the 0.0.0.0 address, other pods can always connect to it, even if the port isn't listed in the pod spec explicitly. But it makes sense to define the ports explicitly so that everyone using your cluster can quickly see what ports each pod exposes. Explicitly defining ports also allows you to assign a name to each port, which can come in handy, as you'll see later in the book.

Using kubectl explain to discover possible API object fields

When preparing a manifest, you can either turn to the Kubernetes reference documentation at http://kubernetes.io/docs/api to see which attributes are supported by each API object, or you can use the `kubectl explain` command.

For example, when creating a pod manifest from scratch, you can start by asking `kubectl` to explain pods:

```
$ 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 prints out the explanation of the object and lists the attributes the object can contain. You can then drill deeper to find out more about each attribute. For example, you can examine the `spec` attribute like this:

```
$ 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 Using kubectl create to create the pod

To create the pod from your YAML file, use the `kubectl create` command:

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

The `kubectl create -f` command is used for creating any resource (not only pods) from a YAML or JSON file.

Retrieving the whole definition of a running pod

After creating the pod, you can ask Kubernetes for the full YAML of the pod. You'll see it's similar to the YAML you saw earlier. You'll learn about the additional fields appearing in the returned definition in the next sections. Go ahead and use the following command to see the full descriptor of the pod:

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

If you're more into JSON, you can also tell `kubectl` to return JSON instead of YAML like this (this works even if you used YAML to create the pod):

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

Seeing your newly created pod in the list of pods

Your pod has been created, but how do you know if it's running? Let's list pods to see their statuses:

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

There's your `kubia-manual` pod. Its status shows that it's running. If you're like me, you'll probably want to confirm that's true by talking to the pod. You'll do that in a minute. First, you'll look at the app's log to check for any errors.

3.2.4 Viewing application logs

Your little Node.js application logs to the process's standard output. Containerized applications usually log to the standard output and standard error stream instead of writing their logs to files. This is to allow users to view logs of different applications in a simple, standard way.

The container runtime (Docker in your case) redirects those streams to files and allows you to get the container's log by running

```
$ docker logs <container id>
```

You could use ssh to log into the node where your pod is running and retrieve its logs with docker logs, but Kubernetes provides an easier way.

RETRIEVING A POD'S LOG WITH KUBECTL LOGS

To see your pod's log (more precisely, the container's log) you run the following command on your local machine (no need to ssh anywhere):

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

You haven't sent any web requests to your Node.js app, so the log only shows a single log statement about the server starting up. As you can see, retrieving logs of an application running in Kubernetes is incredibly simple if the pod only contains a single container.

> **NOTE** Container logs are automatically rotated daily and every time the log file reaches 10MB in size. The kubectl logs command only shows the log entries from the last rotation.

SPECIFYING THE CONTAINER NAME WHEN GETTING LOGS OF A MULTI-CONTAINER POD

If your pod includes multiple containers, you have to explicitly specify the container name by including the -c <container name> option when running kubectl logs. In your kubia-manual pod, you set the container's name to kubia, so if additional containers exist in the pod, you'd have to get its logs like this:

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

Note that you can only retrieve container logs of pods that are still in existence. When a pod is deleted, its logs are also deleted. To make a pod's logs available even after the pod is deleted, you need to set up centralized, cluster-wide logging, which stores all the logs into a central store. Chapter 17 explains how centralized logging works.

3.2.5 Sending requests to the pod

The pod is now running—at least that's what kubectl get and your app's log say. But how do you see it in action? In the previous chapter, you used the kubectl expose command to create a service to gain access to the pod externally. You're not going to do that now, because a whole chapter is dedicated to services, and you have other ways of connecting to a pod for testing and debugging purposes. One of them is through *port forwarding*.

Forwarding a local network port to a port in the pod

When you want to talk to a specific pod without going through a service (for debugging or other reasons), Kubernetes allows you to configure port forwarding to the pod. This is done through the `kubectl port-forward` command. The following command will forward your machine's local port `8888` to port `8080` of your `kubia-manual` pod:

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

The port forwarder is running and you can now connect to your pod through the local port.

Connecting to the pod through the port forwarder

In a different terminal, you can now use `curl` to send an HTTP request to your pod through the `kubectl port-forward` proxy running on `localhost:8888`:

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

Figure 3.5 shows an overly simplified view of what happens when you send the request. In reality, a couple of additional components sit between the `kubectl` process and the pod, but they aren't relevant right now.

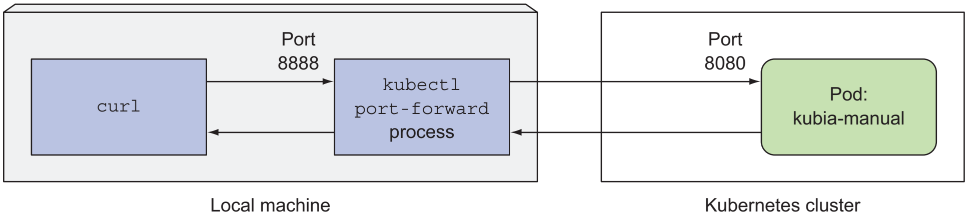

Figure 3.5 A simplified view of what happens when you use `curl` with `kubectl port-forward`

Using port forwarding like this is an effective way to test an individual pod. You'll learn about other similar methods throughout the book.

3.3 Organizing pods with labels

At this point, you have two pods running in your cluster. When deploying actual applications, most users will end up running many more pods. As the number of pods increases, the need for categorizing them into subsets becomes more and more evident.

For example, with microservices architectures, the number of deployed microservices can easily exceed 20 or more. Those components will probably be replicated (multiple copies of the same component will be deployed) and multiple versions or releases (stable, beta, canary, and so on) will run concurrently. This can lead to hundreds of pods in the system. Without a mechanism for organizing them, you end up with a big, incomprehensible mess, such as the one shown in figure 3.6. The figure shows pods of multiple microservices, with several running multiple replicas, and others running different releases of the same microservice.

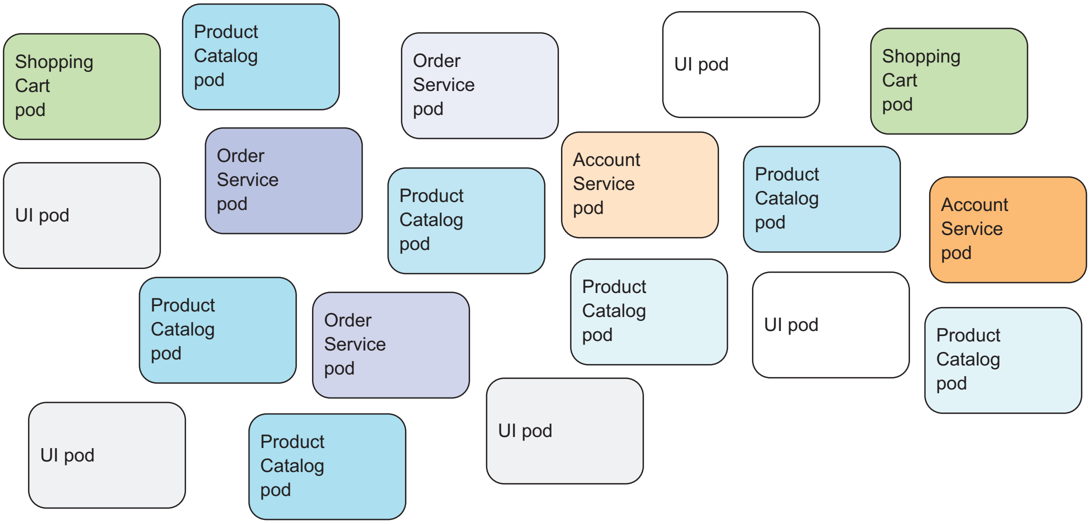

Figure 3.6 Uncategorized pods in a microservices architecture

It's evident you need a way of organizing them into smaller groups based on arbitrary criteria, so every developer and system administrator dealing with your system can easily see which pod is which. And you'll want to operate on every pod belonging to a certain group with a single action instead of having to perform the action for each pod individually.

Organizing pods and all other Kubernetes objects is done through *labels*.

3.3.1 Introducing labels

Labels are a simple, yet incredibly powerful, Kubernetes feature for organizing not only pods, but all other Kubernetes resources. A label is an arbitrary key-value pair you attach to a resource, which is then utilized when selecting resources using *label selectors* (resources are filtered based on whether they include the label specified in the selector). A resource can have more than one label, as long as the keys of those labels are unique within that resource. You usually attach labels to resources when you create them, but you can also add additional labels or even modify the values of existing labels later without having to recreate the resource.

Let's turn back to the microservices example from figure 3.6. By adding labels to those pods, you get a much-better-organized system that everyone can easily make sense of. Each pod is labeled with two labels:

- `app`, which specifies which app, component, or microservice the pod belongs to.
- `rel`, which shows whether the application running in the pod is a stable, beta, or a canary release.

DEFINITION A canary release is when you deploy a new version of an application next to the stable version, and only let a small fraction of users hit the new version to see how it behaves before rolling it out to all users. This prevents bad releases from being exposed to too many users.

By adding these two labels, you've essentially organized your pods into two dimensions (horizontally by app and vertically by release), as shown in figure 3.7.

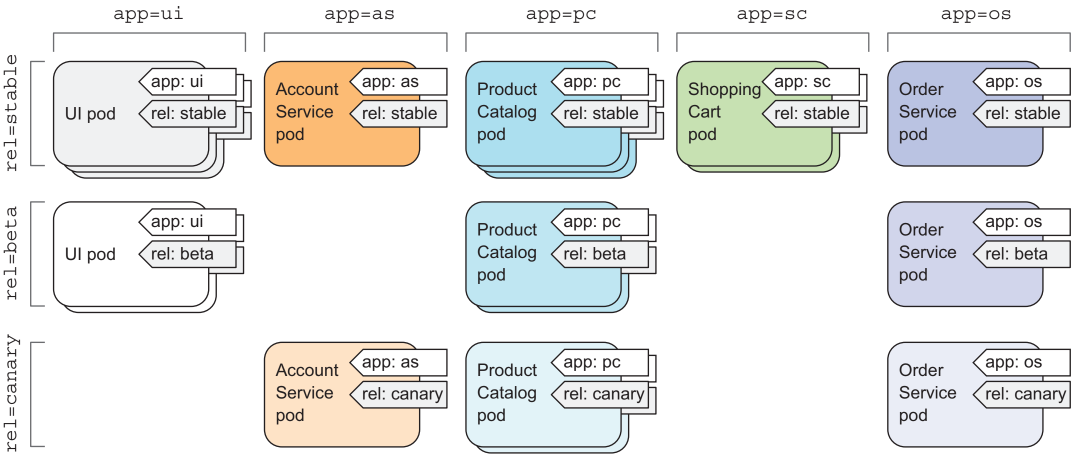

Figure 3.7 Organizing pods in a microservices architecture with pod labels

Every developer or ops person with access to your cluster can now easily see the system's structure and where each pod fits in by looking at the pod's labels.

3.3.2 Specifying labels when creating a pod

Now, you'll see labels in action by creating a new pod with two labels. Create a new file called kubia-manual-with-labels.yaml with the contents of the following listing.

Listing 3.3 A pod with labels: kubia-manual-with-labels.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-manual-v2
  labels:
    creation_method: manual
    env: prod
spec:
  containers:
  - image: luksa/kubia
    name: kubia
    ports:
    - containerPort: 8080
      protocol: TCP
```

Two labels are attached to the pod.

You've included the labels `creation_method=manual` and `env=data.labels` section. You'll create this pod now:

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

The `kubectl get pods` command doesn't list any labels by default, but you can see them by using the `--show-labels` switch:

```
$ 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
```

Instead of listing all labels, if you're only interested in certain labels, you can specify them with the `-L` switch and have each displayed in its own column. List pods again and show the columns for the two labels you've attached to your `kubia-manual-v2` pod:

```
$ 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 Modifying labels of existing pods

Labels can also be added to and modified on existing pods. Because the `kubia-manual` pod was also created manually, let's add the `creation_method=manual` label to it:

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

Now, let's also change the `env=prod` label to `env=debug` on the `kubia-manual-v2` pod, to see how existing labels can be changed.

NOTE You need to use the `--overwrite` option when changing existing labels.

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

List the pods again to see the updated 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>
```

As you can see, attaching labels to resources is trivial, and so is changing them on existing resources. It may not be evident right now, but this is an incredibly powerful feature, as you'll see in the next chapter. But first, let's see what you can do with these labels, in addition to displaying them when listing pods.

3.4 Listing subsets of pods through label selectors

Attaching labels to resources so you can see the labels next to each resource when listing them isn't that interesting. But labels go hand in hand with *label selectors*. Label selectors allow you to select a subset of pods tagged with certain labels and perform an operation on those pods. A label selector is a criterion, which filters resources based on whether they include a certain label with a certain value.

A label selector can select resources based on whether the resource

- Contains (or doesn't contain) a label with a certain key
- Contains a label with a certain key and value
- Contains a label with a certain key, but with a value not equal to the one you specify

3.4.1 Listing pods using a label selector

Let's use label selectors on the pods you've created so far. To see all pods you created manually (you labeled them with `creation_method=manual`), do the following:

```
$ 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
```

To list all pods that include the `env` label, whatever its value is:

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

And those that don't have the `env` label:

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

NOTE Make sure to use single quotes around `!env`, so the bash shell doesn't evaluate the exclamation mark.

Similarly, you could also match pods with the following label selectors:

- `creation_method!=manual` to select pods with the `creation_method` label with any value other than `manual`
- `env in (prod,devel)` to select pods with the `env` label set to either `prod` or `devel`
- `env notin (prod,devel)` to select pods with the `env` label set to any value other than `prod` or `devel`

Turning back to the pods in the microservices-oriented architecture example, you could select all pods that are part of the product catalog microservice by using the `app=pc` label selector (shown in the following figure).

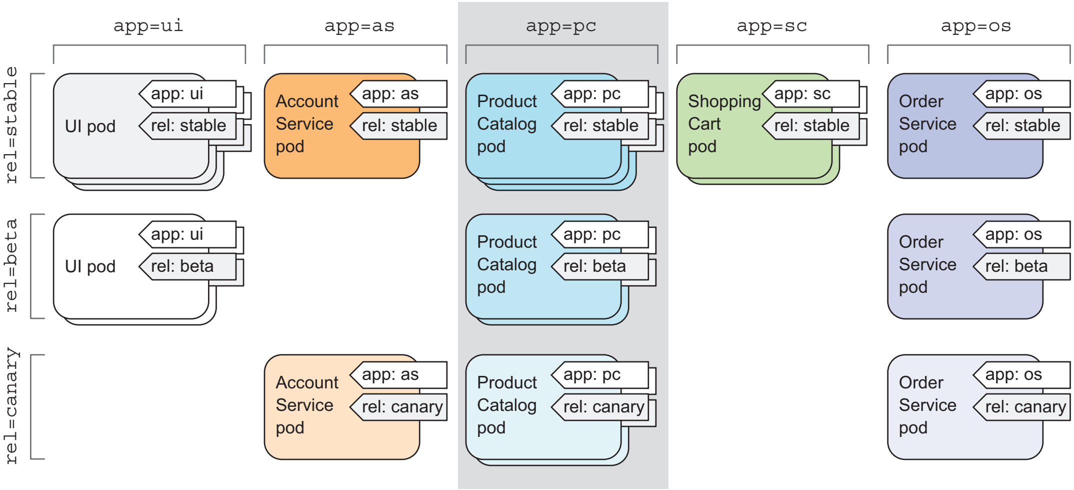

Figure 3.8 Selecting the product catalog microservice pods using the "app=pc" label selector

3.4.2 Using multiple conditions in a label selector

A selector can also include multiple comma-separated criteria. Resources need to match all of them to match the selector. If, for example, you want to select only pods running the beta release of the product catalog microservice, you'd use the following selector: `app=pc,rel=beta` (visualized in figure 3.9).

Label selectors aren't useful only for listing pods, but also for performing actions on a subset of all pods. For example, later in the chapter, you'll see how to use label selectors to delete multiple pods at once. But label selectors aren't used only by `kubectl`. They're also used internally, as you'll see next.

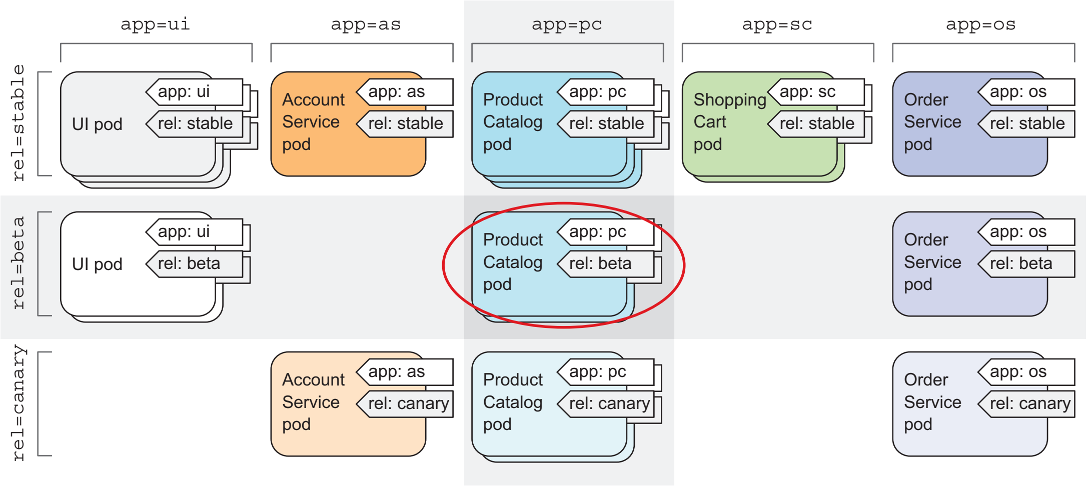

Figure 3.9 Selecting pods with multiple label selectors

3.5 Using labels and selectors to constrain pod scheduling

All the pods you've created so far have been scheduled pretty much randomly across your worker nodes. As I've mentioned in the previous chapter, this is the proper way of working in a Kubernetes cluster. Because Kubernetes exposes all the nodes in the cluster as a single, large deployment platform, it shouldn't matter to you what node a pod is scheduled to. Because each pod gets the exact amount of computational resources it requests (CPU, memory, and so on) and its accessibility from other pods isn't at all affected by the node the pod is scheduled to, usually there shouldn't be any need for you to tell Kubernetes exactly where to schedule your pods.

Certain cases exist, however, where you'll want to have at least a little say in where a pod should be scheduled. A good example is when your hardware infrastructure isn't homogenous. If part of your worker nodes have spinning hard drives, whereas others have SSDs, you may want to schedule certain pods to one group of nodes and the rest to the other. Another example is when you need to schedule pods performing intensive GPU-based computation only to nodes that provide the required GPU acceleration.

You never want to say specifically what node a pod should be scheduled to, because that would couple the application to the infrastructure, whereas the whole idea of Kubernetes is hiding the actual infrastructure from the apps that run on it. But if you want to have a say in where a pod should be scheduled, instead of specifying an exact node, you should describe the node requirements and then let Kubernetes select a node that matches those requirements. This can be done through node labels and node label selectors.

3.5.1 Using labels for categorizing worker nodes

As you learned earlier, pods aren't the only Kubernetes resource type that you can attach a label to. Labels can be attached to any Kubernetes object, including nodes. Usually, when the ops team adds a new node to the cluster, they'll categorize the node by attaching labels specifying the type of hardware the node provides or anything else that may come in handy when scheduling pods.

Let's imagine one of the nodes in your cluster contains a GPU meant to be used for general-purpose GPU computing. You want to add a label to the node showing this feature. You're going to add the label `gpu=true` to one of your nodes (pick one out of the list returned by `kubectl get nodes`):

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

Now you can use a label selector when listing the nodes, like you did before with pods. List only nodes that include the label `gpu=true`:

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

As expected, only one node has this label. You can also try listing all the nodes and tell `kubectl` to display an additional column showing the values of each node's `gpu` label (`kubectl get nodes -L gpu`).

3.5.2 Scheduling pods to specific nodes

Now imagine you want to deploy a new pod that needs a GPU to perform its work. To ask the scheduler to only choose among the nodes that provide a GPU, you'll add a node selector to the pod's YAML. Create a file called kubia-gpu.yaml with the following listing's contents and then use `kubectl create -f kubia-gpu.yaml` to create the pod.

Listing 3.4 Using a label selector to schedule a pod to a specific node: kubia-gpu.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-gpu
spec:
  nodeSelector:
    gpu: "true"
  containers:
  - image: luksa/kubia
    name: kubia
```

nodeSelector tells Kubernetes to deploy this pod only to nodes containing the gpu=true label.

You've added a `nodeSelector` field under the `spec` section. When you create the pod, the scheduler will only choose among the nodes that contain the `gpu=true` label (which is only a single node in your case).

3.5.3 Scheduling to one specific node

Similarly, you could also schedule a pod to an exact node, because each node also has a unique label with the key `kubernetes.io/hostname` and value set to the actual hostname of the node. But setting the `nodeSelector` to a specific node by the hostname label may lead to the pod being unschedulable if the node is offline. You shouldn't think in terms of individual nodes. Always think about logical groups of nodes that satisfy certain criteria specified through label selectors.

This was a quick demonstration of how labels and label selectors work and how they can be used to influence the operation of Kubernetes. The importance and usefulness of label selectors will become even more evident when we talk about Replication-Controllers and Services in the next two chapters.

> **NOTE** Additional ways of influencing which node a pod is scheduled to are covered in chapter 16.

3.6 Annotating pods

In addition to labels, pods and other objects can also contain *annotations*. Annotations are also key-value pairs, so in essence, they're similar to labels, but they aren't meant to hold identifying information. They can't be used to group objects the way labels can. While objects can be selected through label selectors, there's no such thing as an annotation selector.

On the other hand, annotations can hold much larger pieces of information and are primarily meant to be used by tools. Certain annotations are automatically added to objects by Kubernetes, but others are added by users manually.

Annotations are also commonly used when introducing new features to Kubernetes. Usually, alpha and beta versions of new features don't introduce any new fields to API objects. Annotations are used instead of fields, and then once the required API changes have become clear and been agreed upon by the Kubernetes developers, new fields are introduced and the related annotations deprecated.

A great use of annotations is adding descriptions for each pod or other API object, so that everyone using the cluster can quickly look up information about each individual object. For example, an annotation used to specify the name of the person who created the object can make collaboration between everyone working on the cluster much easier.

3.6.1 Looking up an object's annotations

Let's see an example of an annotation that Kubernetes added automatically to the pod you created in the previous chapter. To see the annotations, you'll need to request the full YAML of the pod or use the kubectl describe command. You'll use the first option in the following listing.

Listing 3.5 A pod's annotations

```
$ 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", ...
```

Without going into too many details, as you can see, the kubernetes.io/created-by annotation holds JSON data about the object that created the pod. That's not something you'd want to put into a label. Labels should be short, whereas annotations can contain relatively large blobs of data (up to 256 KB in total).

NOTE The kubernetes.io/created-by annotations was deprecated in version 1.8 and will be removed in 1.9, so you will no longer see it in the YAML.

3.6.2 Adding and modifying annotations

Annotations can obviously be added to pods at creation time, the same way labels can. They can also be added to or modified on existing pods later. The simplest way to add an annotation to an existing object is through the kubectl annotate command.

You'll try adding an annotation to your kubia-manual pod now:

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

You added the annotation mycompany.com/someannotation with the value foo bar. It's a good idea to use this format for annotation keys to prevent key collisions. When different tools or libraries add annotations to objects, they may accidentally override each other's annotations if they don't use unique prefixes like you did here.

You can use kubectl describe to see the annotation you added:

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

3.7 Using namespaces to group resources

Let's turn back to labels for a moment. We've seen how they organize pods and other objects into groups. Because each object can have multiple labels, those groups of objects can overlap. Plus, when working with the cluster (through kubectl for example), if you don't explicitly specify a label selector, you'll always see all objects.

But what about times when you want to split objects into separate, non-overlapping groups? You may want to only operate inside one group at a time. For this and other reasons, Kubernetes also groups objects into namespaces. These aren't the Linux namespaces we talked about in chapter 2, which are used to isolate processes from each other. Kubernetes namespaces provide a scope for objects names. Instead of having all your resources in one single namespace, you can split them into multiple namespaces, which also allows you to use the same resource names multiple times (across different namespaces).

3.7.1 Understanding the need for namespaces

Using multiple namespaces allows you to split complex systems with numerous components into smaller distinct groups. They can also be used for separating resources in a multi-tenant environment, splitting up resources into production, development, and QA environments, or in any other way you may need. Resource names only need to be unique within a namespace. Two different namespaces can contain resources of the same name. But, while most types of resources are namespaced, a few aren't. One of them is the Node resource, which is global and not tied to a single namespace. You'll learn about other cluster-level resources in later chapters.

Let's see how to use namespaces now.

3.7.2 Discovering other namespaces and their pods

First, let's list all namespaces in your cluster:

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

Up to this point, you've operated only in the `default` namespace. When listing resources with the `kubectl get` command, you've never specified the namespace explicitly, so `kubectl` always defaulted to the `default` namespace, showing you only the objects in that namespace. But as you can see from the list, the `kube-public` and the `kube-system` namespaces also exist. Let's look at the pods that belong to the `kube-system` namespace, by telling `kubectl` to list pods in that namespace only:

```
$ 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
```

TIP You can also use `-n` instead of `--namespace`.

You'll learn about these pods later in the book (don't worry if the pods shown here don't match the ones on your system exactly). It's clear from the name of the namespace that these are resources related to the Kubernetes system itself. By having them in this separate namespace, it keeps everything nicely organized. If they were all in the default namespace, mixed in with the resources you create yourself, you'd have a hard time seeing what belongs where, and you might inadvertently delete system resources.

Namespaces enable you to separate resources that don't belong together into nonoverlapping groups. If several users or groups of users are using the same Kubernetes cluster, and they each manage their own distinct set of resources, they should each use their own namespace. This way, they don't need to take any special care not to inadvertently modify or delete the other users' resources and don't need to concern themselves with name conflicts, because namespaces provide a scope for resource names, as has already been mentioned.

Besides isolating resources, namespaces are also used for allowing only certain users access to particular resources and even for limiting the amount of computational resources available to individual users. You'll learn about this in chapters 12 through 14.

3.7.3 Creating a namespace

A namespace is a Kubernetes resource like any other, so you can create it by posting a YAML file to the Kubernetes API server. Let's see how to do this now.

CREATING A NAMESPACE FROM A YAML FILE

First, create a custom-namespace.yaml file with the following listing's contents (you'll find the file in the book's code archive).

Listing 3.6 A YAML definition of a namespace: custom-namespace.yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: custom-namespace
```

This says you're defining a namespace.

This is the name of the namespace.

Now, use `kubectl` to post the file to the Kubernetes API server:

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

CREATING A NAMESPACE WITH KUBECTL CREATE NAMESPACE

Although writing a file like the previous one isn't a big deal, it's still a hassle. Luckily, you can also create namespaces with the dedicated `kubectl create namespace` command, which is quicker than writing a YAML file. By having you create a YAML manifest for the namespace, I wanted to reinforce the idea that everything in Kubernetes has a corresponding API object that you can create, read, update, and delete by posting a YAML manifest to the API server.

You could have created the namespace like this:

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

NOTE Although most objects' names must conform to the naming conventions specified in RFC 1035 (Domain names), which means they may contain only letters, digits, dashes, and dots, namespaces (and a few others) aren't allowed to contain dots.

3.7.4 Managing objects in other namespaces

To create resources in the namespace you've created, either add a `namespace: custom-namespace` entry to the `metadata` section, or specify the namespace when creating the resource with the `kubectl create` command:

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

You now have two pods with the same name (`kubia-manual`). One is in the `default` namespace, and the other is in your `custom-namespace`.

When listing, describing, modifying, or deleting objects in other namespaces, you need to pass the `--namespace` (or `-n`) flag to `kubectl`. If you don't specify the namespace, `kubectl` performs the action in the default namespace configured in the current `kubectl` context. The current context's namespace and the current context itself can be changed through `kubectl config` commands. To learn more about managing `kubectl` contexts, refer to appendix A.

TIP To quickly switch to a different namespace, you can set up the following alias: `alias kcd='kubectl config set-context $(kubectl config current-context) --namespace '`. You can then switch between namespaces using `kcd some-namespace`.

3.7.5 Understanding the isolation provided by namespaces

To wrap up this section about namespaces, let me explain what namespaces don't provide—at least not out of the box. Although namespaces allow you to isolate objects into distinct groups, which allows you to operate only on those belonging to the specified namespace, they don't provide any kind of isolation of running objects.

For example, you may think that when different users deploy pods across different namespaces, those pods are isolated from each other and can't communicate, but that's not necessarily the case. Whether namespaces provide network isolation depends on which networking solution is deployed with Kubernetes. When the solution doesn't provide inter-namespace network isolation, if a pod in namespace `foo` knows the IP address of a pod in namespace bar, there is nothing preventing it from sending traffic, such as HTTP requests, to the other pod.

3.8 Stopping and removing pods

You've created a number of pods, which should all still be running. You have four pods running in the default namespace and one pod in custom-namespace. You're going to stop all of them now, because you don't need them anymore.

3.8.1 Deleting a pod by name

First, delete the kubia-gpu pod by name:

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

By deleting a pod, you're instructing Kubernetes to terminate all the containers that are part of that pod. Kubernetes sends a SIGTERM signal to the process and waits a certain number of seconds (30 by default) for it to shut down gracefully. If it doesn't shut down in time, the process is then killed through SIGKILL. To make sure your processes are always shut down gracefully, they need to handle the SIGTERM signal properly.

> **TIP** You can also delete more than one pod by specifying multiple, space-separated names (for example, kubectl delete po pod1 pod2).

3.8.2 Deleting pods using label selectors

Instead of specifying each pod to delete by name, you'll now use what you've learned about label selectors to stop both the kubia-manual and the kubia-manual-v2 pod. Both pods include the creation_method=manual label, so you can delete them by using a label selector:

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

In the earlier microservices example, where you had tens (or possibly hundreds) of pods, you could, for instance, delete all canary pods at once by specifying the rel=canary label selector (visualized in figure 3.10):

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

3.8.3 Deleting pods by deleting the whole namespace

Okay, back to your real pods. What about the pod in the custom-namespace? You no longer need either the pods in that namespace, or the namespace itself. You can

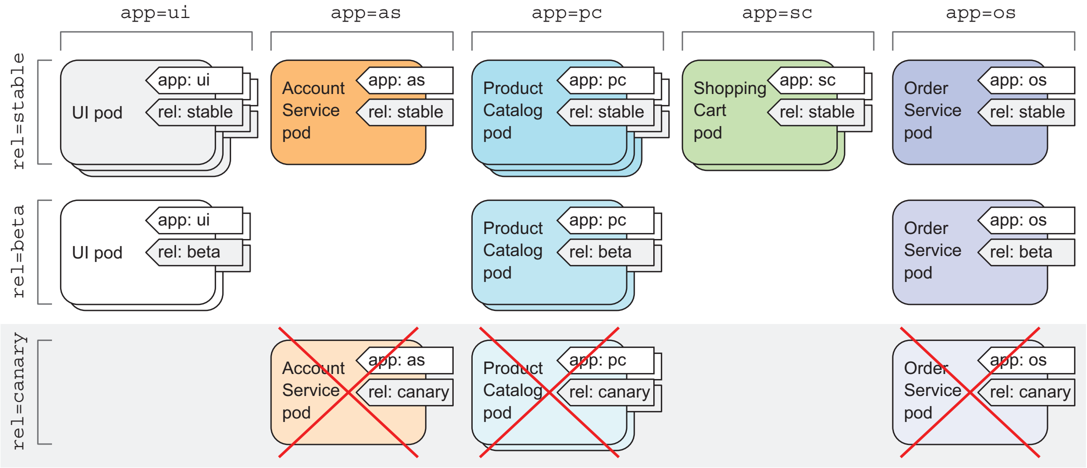

Figure 3.10 Selecting and deleting all canary pods through the `rel=canary` label selector

delete the whole namespace (the pods will be deleted along with the namespace automatically), using the following command:

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

3.8.4 Deleting all pods in a namespace, while keeping the namespace

You've now cleaned up almost everything. But what about the pod you created with the `kubectl run` command in chapter 2? That one is still running:

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

This time, instead of deleting the specific pod, tell Kubernetes to delete all pods in the current namespace by using the `--all` option:

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

Now, double check that no pods were left running:

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

Wait, what!?! The `kubia-zxzij` pod is terminating, but a new pod called `kubia-09as0`, which wasn't there before, has appeared. No matter how many times you delete all pods, a new pod called *kubia-something* will emerge.

You may remember you created your first pod with the `kubectl run` command. In chapter 2, I mentioned that this doesn't create a pod directly, but instead creates a ReplicationController, which then creates the pod. As soon as you delete a pod created by the ReplicationController, it immediately creates a new one. To delete the pod, you also need to delete the ReplicationController.

3.8.5 Deleting (almost) all resources in a namespace

You can delete the ReplicationController and the pods, as well as all the Services you've created, by deleting all resources in the current namespace with a single command:

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

The first `all` in the command specifies that you're deleting resources of all types, and the `--all` option specifies that you're deleting all resource instances instead of specifying them by name (you already used this option when you ran the previous delete command).

> **NOTE** Deleting everything with the `all` keyword doesn't delete absolutely everything. Certain resources (like Secrets, which we'll introduce in chapter 7) are preserved and need to be deleted explicitly.

As it deletes resources, `kubectl` will print the name of every resource it deletes. In the list, you should see the `kubia` ReplicationController and the `kubia-http` Service you created in chapter 2.

> **NOTE** The `kubectl delete all --all` command also deletes the `kubernetes` Service, but it should be recreated automatically in a few moments.

3.9 Summary

After reading this chapter, you should now have a decent knowledge of the central building block in Kubernetes. Every other concept you'll learn about in the next few chapters is directly related to pods.

In this chapter, you've learned

- How to decide whether certain containers should be grouped together in a pod or not.

- Pods can run multiple processes and are similar to physical hosts in the non-container world.
- YAML or JSON descriptors can be written and used to create pods and then examined to see the specification of a pod and its current state.
- Labels and label selectors should be used to organize pods and easily perform operations on multiple pods at once.
- You can use node labels and selectors to schedule pods only to nodes that have certain features.
- Annotations allow attaching larger blobs of data to pods either by people or tools and libraries.
- Namespaces can be used to allow different teams to use the same cluster as though they were using separate Kubernetes clusters.
- How to use the `kubectl explain` command to quickly look up the information on any Kubernetes resource.

In the next chapter, you'll learn about ReplicationControllers and other resources that manage pods.

SOFTWARE DEVELOPMENT/OPERATIONS

Kubernetes IN ACTION

Marko Lukša

Kubernetes is Greek for "helmsman," your guide through unknown waters. The Kubernetes container orchestration system safely manages the structure and flow of a distributed application, organizing containers and services for maximum efficiency. Kubernetes serves as an operating system for your clusters, eliminating the need to factor the underlying network and server infrastructure into your designs.

Kubernetes in Action teaches you to use Kubernetes to deploy container-based distributed applications. You'll start with an overview of Docker and Kubernetes before building your first Kubernetes cluster. You'll gradually expand your initial application, adding features and deepening your knowledge of Kubernetes architecture and operation. As you navigate this comprehensive guide, you'll explore high-value topics like monitoring, tuning, and scaling.

What's Inside

- Kubernetes' internals
- Deploying containers across a cluster
- Securing clusters
- Updating applications with zero downtime

Written for intermediate software developers with little or no familiarity with Docker or container orchestration systems.

Marko Lukša is an engineer at Red Hat working on Kubernetes and OpenShift.

To download their free eBook in PDF, ePub, and Kindle formats,
owners of this book should visit
www.manning.com/books/kubernetes-in-action

Free eBook
SEE INSERT

"Authoritative and exhaustive. In a hands-on style, the author teaches how to manage the complete lifecycle of any distributed and scalable application."
—Antonio Magnaghi, System1

"The best parts are the real-world examples. They don't just apply the concepts, they road test them."
—Paolo Antinori, Red Hat

"An in-depth discussion of Kubernetes and related technologies. A must-have!"
—Al Krinker, USPTO

"The full path to becoming a professional Kubernaut. Fundamental reading."
—Csaba Sári
Chimera Entertainment

ISBN-13: 978-1-61729-372-6
ISBN-10: 1-61729-372-5
55999
9 781617 293726

MANNING

$59.99 / Can $79.99 [INCLUDING eBOOK]