FREE

Exploring Kubernetes

Chapters selected by

Marko Luksa

Exploring Kubernetes

Selected by Marko Lukša

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

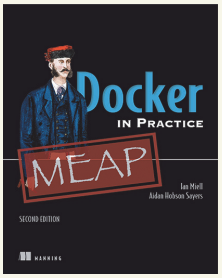
Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

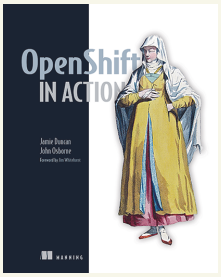
ISBN: 9781617295539
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 21 20 19 18 17 16

brief contents

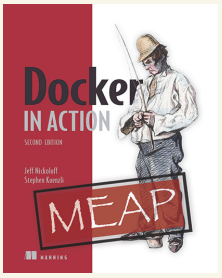
RELATED TITLES



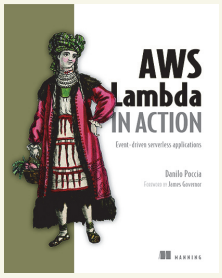
Docker in Practice, Second Edition
by Ian Miell and Aidan Hobson Sayers
ISBN 9781617294808
425 pages
Look Inside



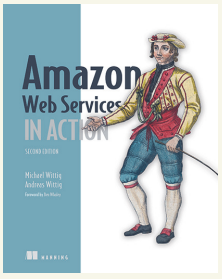
OpenShift in Action
by Jamie Duncan and John Osborne
ISBN 9781617294839
320 pages
Look Inside



Docker in Action, Second Edition
by Jeff Nickoloff and Stephen Kuenzli
ISBN 9781617294761
350 pages
Look Inside

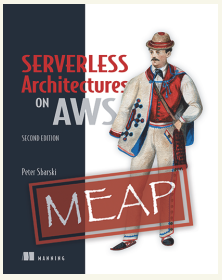


AWS Lambda in Action
by Danilo Poccia
ISBN 9781617293719
384 pages
Look Inside

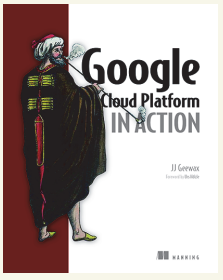


Amazon Web Services in Action, Second Edition
by Michael Wittig and Andreas Wittig
ISBN 9781617295119
528 pages
Look Inside

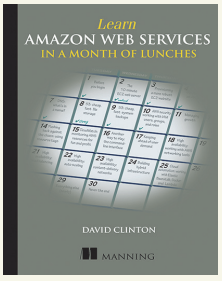
Production-Ready Serverless
by Yan Cui
Duration 10h
Look Inside



Serverless Architectures on AWS, Second Edition
by Peter Sbarski
ISBN 9781617295423
500 pages
Look Inside



Google Cloud Platform in Action
by JJ Geewax
ISBN 9781617293528
632 pages
Look Inside



Learn Amazon Web Services in a Month of Lunches
by David Clinton
ISBN 9781617294440
328 pages
Look Inside

manning

2 First steps with Docker and Kubernetes

This chapter covers

- Creating, running, and sharing a container image with Docker
- Running a single-node Kubernetes cluster locally
- Setting up a Kubernetes cluster on Google Container Engine
- Setting up and using the kubectl command-line client
- Deploying an app on Kubernetes and horizontally scaling it

Before you start learning about Kubernetes concepts in detail, let's see how to create a simple application, package it into a container image, and run it in a managed Kubernetes cluster (in Google Container Engine) or in a local single-node cluster. This should give you a slightly better overview of the whole Kubernetes system and will make it easier to follow the next few chapters, where we'll go over the basic building blocks and concepts in Kubernetes.

2.1 Creating, running, and sharing a container image

As you've already learned in the previous chapter, running applications in Kubernetes requires them to be packaged into container images. We'll do a basic introduction to using Docker in case you haven't used it yet. In the next few sections you'll

1. Install Docker and run your first "Hello world" container
2. Create a trivial Node.js app that you'll later deploy in Kubernetes
3. Package the app into a container image so you can then run it as an isolated container
4. Run a container based on the image
5. Push the image to Docker Hub so that anyone anywhere can run it

2.1.1 Installing Docker and running a Hello World container

First, you'll need to install Docker on your Linux machine. If you don't use Linux, you'll need to start a Linux virtual machine (VM) and run Docker inside that VM. If you're using a Mac or Windows and install Docker per instructions, Docker will set up a VM for you and run the Docker daemon inside that VM. The Docker client executable will be available on your host OS, and will communicate with the daemon inside the VM.

To install Docker, follow the instructions at http://docs.docker.com/engine/installation for your specific operating system. After completing the installation, you can use the Docker client executable to run various Docker commands. For example, you could try pulling and running an existing image from Docker Hub, the public Docker registry, which contains ready-to-use container images for many well-known software packages. One of them is the *busybox* image, which you'll use to run a simple `echo "Hello world"` command.

RUNNING A HELLO WORLD CONTAINER

If you're unfamiliar with *busybox*, it's a single executable that combines many of the standard UNIX command-line tools, such as `echo`, `ls`, `gzip`, and so on. Instead of the busybox image, you could also use any other full-fledged OS container image such as Fedora, Ubuntu, or other similar images, as long as it includes the `echo` executable.

How do you run the busybox image? You don't need to download or install anything. Use the `docker run` command and specify what image to download and run and (optionally) what command to execute, as shown in the following listing.

Listing 2.1 Running a Hello world container with Docker

```
$ docker run busybox echo "Hello world"
Unable to find image 'busybox:latest' locally
latest: Pulling from docker.io/busybox
9a163e0b8d13: Pull complete
fef924a0204a: Pull complete
Digest:
     sha256:97473e34e311e6c1b3f61f2a721d038d1e5eef17d98d1353a513007cf46ca6bd
Status: Downloaded newer image for docker.io/busybox:latest
Hello world
```

This doesn't look that impressive, but when you consider that the whole "app" was downloaded and executed with a single command, without you having to install that app or anything else, you'll agree it's awesome. In your case, the app was a single executable (busybox), but it might as well have been an incredibly complex app with many dependencies. The whole process of setting up and running the app would have been exactly the same. What's probably most important is that the app was executed inside a container, completely isolated from all the other processes running on your machine.

UNDERSTANDING WHAT HAPPENS BEHIND THE SCENES

Figure 2.1 shows exactly what happened when you performed the `docker run` command. First, Docker checked to see if the busybox:latest image was already present on your local machine. It wasn't, so Docker pulled it from the Docker Hub registry at http://docker.io. After the image was downloaded to your machine, Docker created a container from that image and ran the command inside it. The `echo` command printed the text to STDOUT and then the process terminated and the container stopped.

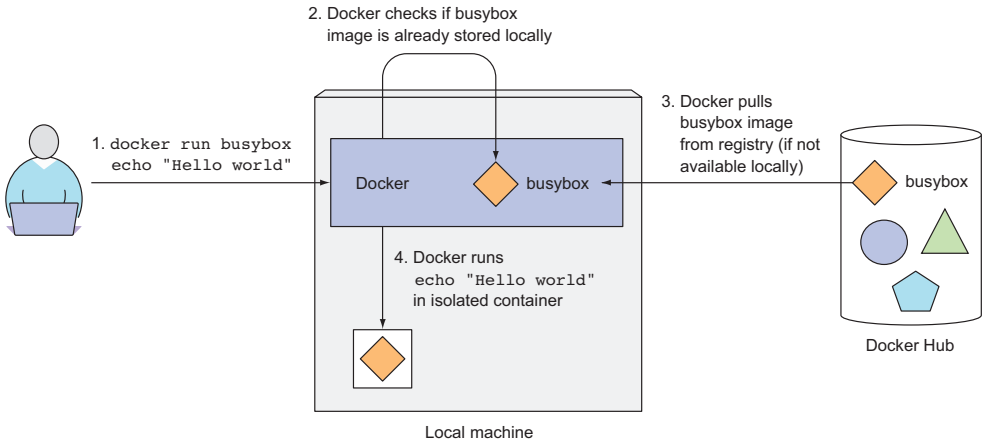


Figure 2.1 Running echo "Hello world" in a container based on the busybox container image.

RUNNING OTHER IMAGES

Running other existing container images is much the same as the way you ran the busybox image. In fact, it's often even simpler, because you usually don't need to specify what command to execute, the way you did in your example (`echo "Hello world"`). The command that should be executed is usually baked into the image itself, but you can override it if you want. After searching or browsing through the publicly available images on http://hub.docker.com or another public registry, you tell Docker to run the image like this:

```
$ docker run <image>
```

VERSIONING CONTAINER IMAGES

All software packages get updated, so more than a single version of a package usually exists. Docker supports having multiple versions or variants of the same image under the same name. Each variant must have a unique tag. When referring to images without explicitly specifying the tag, Docker will assume you're referring to the so-called *latest* tag. To run a different version of the image, you may specify the tag along with the image name like this:

```
$ docker run <image>:<tag>
```

2.1.2 Creating a trivial Node.js app

Now that you have a working Docker setup, you're going to create an app. You'll build a trivial Node.js web application and package it into a container image. The application will accept HTTP requests and return the hostname of the machine it's running in. This way, you'll see that an app running inside a container sees its own hostname and not that of the host machine, even though it's running on the host like any other process. This will be useful later, when you deploy the app on Kubernetes and scale it out (scale it horizontally; that is, run multiple instances of the app). You'll see your HTTP requests hitting different instances of the app.

Your app will consist of a single file called app.js with the contents shown in the following listing.

Listing 2.2 A simple Node.js app: app.js

```
const http = require('http');
const os = require('os');

console.log("Kubia server starting...");

var handler = function(request, response) {
  console.log("Received request from " + request.connection.remoteAddress);
  response.writeHead(200);
  response.end("You've hit " + os.hostname() + "\n");
};

var www = http.createServer(handler);
www.listen(8080);
```

It should be clear what this code does. It starts up an HTTP server on port 8080. The server responds with an HTTP response status code 200 OK and the text `"You've hit <hostname>"` to every request. The request handler also logs the client's IP address to the standard output, which you'll need later.

> **NOTE** The returned hostname is the server's actual hostname, not the one the client sends in the HTTP request's Host header.

You could now download and install Node.js and test your app directly, but this isn't necessary, because you'll use Docker to package the app into a container image and enable

it to be run anywhere without having to download or install anything (except Docker, which does need to be installed on the machine you want to run the image on).

2.1.3 Creating a Dockerfile for the image

To package your app into an image, you first need to create a file called Dockerfile, which is a list of instructions that Docker will perform when building the image. The Dockerfile needs to be in the same directory as the app.js file and should contain the commands shown in the following listing.

Listing 2.3 A Dockerfile for building a container image for your app

```
FROM node:7
ADD app.js /app.js
ENTRYPOINT ["node", "app.js"]
```

The FROM line defines the container image you'll use as a starting point (the base image you're building on top of). In your case, you're using the `node` container image, tag `7`. In the second line, you're adding your app.js file from your local directory into the root directory in the image, under the same name (app.js). Finally, in the third line, you're defining what command should be executed when somebody runs the image. In your case, you're executing the command `node app.js`.

Choosing a base image

You may wonder why we chose this specific image as your base. Because your app is a Node.js app, you need your image to contain the node binary executable to run the app. You could have used any image that contains that binary, or you could have even used a Linux distro base image such as fedora or ubuntu and installed Node.js into the container at image build time. But because the Node.js image is made specifically for running Node.js apps, and you don't require anything else, you'll use that as the base image.

2.1.4 Building the container image

Now that you have your Dockerfile and the app.js file, you have everything you need to build your image. To build it, run the following Docker command:

```
$ docker build -t kubia .
```

Figure 2.2 shows what happens during the build process. You're telling Docker to build an image called "kubia" based on the contents of the current directory (note the dot at the end of the build command). Docker will look for the Dockerfile in the directory and build the image based on the instructions in the file.

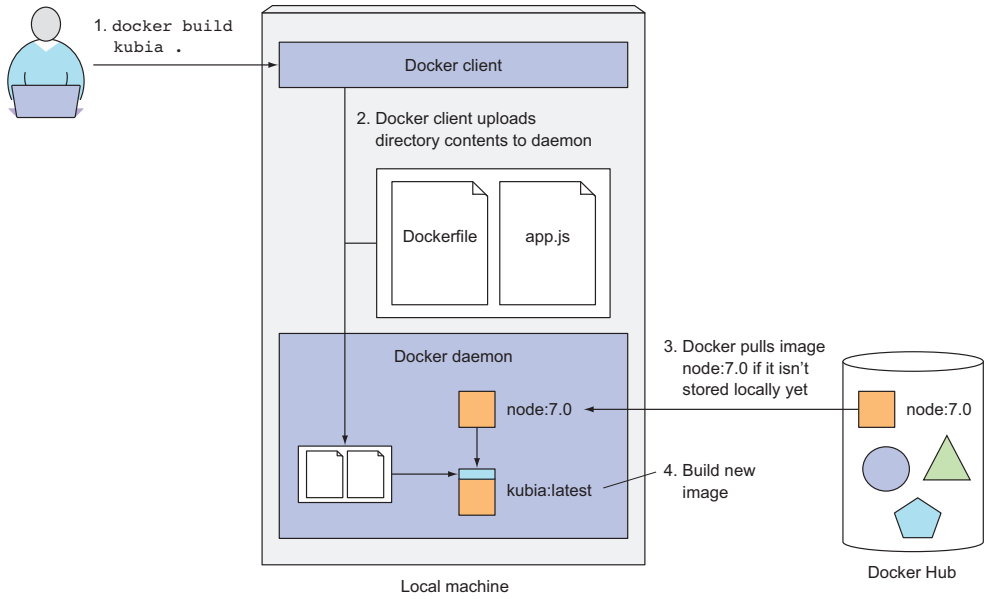


Figure 2.2 Building a new container image from a Dockerfile.

UNDERSTANDING HOW AN IMAGE IS BUILT

The build process isn't performed by the Docker client. Instead, the contents of the whole directory are uploaded to the Docker daemon and the image is built there. The client and daemon don't need to be on the same machine at all. If you're using Docker on a non-Linux OS, the client is on your host OS, but the daemon runs inside a VM. Because all the files in the build directory are uploaded to the daemon, if it contains many large files and the daemon isn't running locally, the upload may take longer.

> **TIP** Don't include any unnecessary files in the build directory, because they'll slow down the build process—especially when the Docker daemon is on a remote machine.

During the build process, Docker will first pull the base image (node:7) from the public image repository (Docker Hub), unless the image has already been pulled and is stored on your machine.

UNDERSTANDING IMAGE LAYERS

An image isn't a single, big, binary blob, but is composed of multiple layers, which you may have already noticed when running the busybox example (there were multiple `Pull complete` lines—one for each layer). Different images may share several layers, which makes storing and transferring images much more efficient. For example, if you create multiple images based on the same base image (such as `node:7` in the example), all the layers comprising the base image will be stored only once. Also, when pulling an image, Docker will download each layer individually. Several layers may already be stored on your machine, so Docker will only download those that aren't.

You may think that each Dockerfile creates only a single new layer, but that's not the case. When building an image, a new layer is created for each individual command in the Dockerfile. During the build of your image, after pulling all the layers of the base image, Docker will create a new layer on top of them and add the app.js file into it. Then it will create yet another layer that will specify the command that should be executed when the image is run. This last layer will then be tagged "kubasia:latest". This is shown in figure 2.3, which also shows how a different image called "other:latest" would use the same layers of the Node.js image as your own image does.

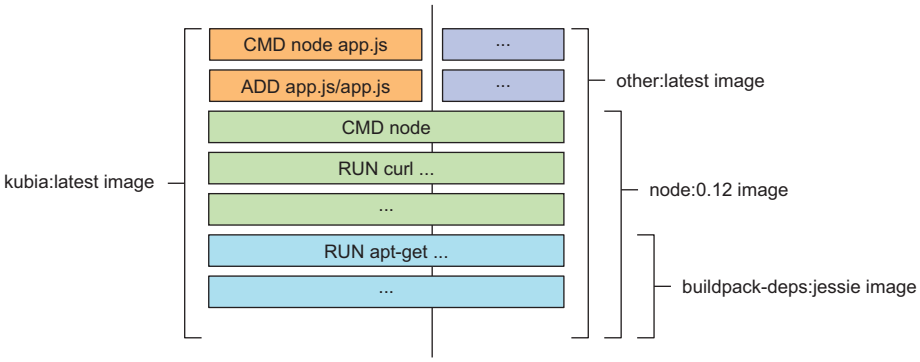


Figure 2.3 Container images are composed of layers that can be shared among different images.

When the build process completes, you have a new image stored locally. You can see it by telling Docker to list all locally stored images, as shown in the following listing.

Listing 2.4 Listing locally stored images

```
$ docker images
REPOSITORY    TAG       IMAGE ID            CREATED              VIRTUAL SIZE
kubia         latest    d30ecc7419e7        1 minute ago         637.1 MB
...
```

COMPARING BUILDING IMAGES WITH A DOCKERFILE VS. MANUALLY

Dockerfiles are the usual way of building container images with Docker, but you could also build the image manually by running a container from an existing image, executing commands in the container, exiting the container, and committing the final state as a new image. This is exactly what happens when you build from a Dockerfile, but it's performed automatically and is repeatable, which allows you to make changes to the Dockerfile and rebuild the image any time, without having to manually retype all the commands again.

2.1.5 Running the container image

You can now run your image with the following command:

```
$ docker run --name kubia-container -p 8080:8080 -d kubia
```

This tells Docker to run a new container called `kubia-container` from the `kubia` image. The container will be detached from the console (`-d` flag), which means it will run in the background. Port 8080 on the local machine will be mapped to port 8080 inside the container (`-p 8080:8080` option), so you can access the app through http://localhost:8080.

If you're not running the Docker daemon on your local machine (if you're using a Mac or Windows, the daemon is running inside a VM), then you'll need to use the hostname or IP of the VM running the daemon instead of localhost. You can look it up through the DOCKER_HOST environment variable.

ACCESSING YOUR APP

Now try to access your application at http://localhost:8080 (be sure to replace localhost with the hostname or IP of the Docker host if necessary):

```
$ curl localhost:8080
You've hit 44d76963e8e1
```

That's the response from your app. Your tiny application is now running inside a container, isolated from everything else. As you can see, it's returning `44d76963e8e1` as its hostname, and not the actual hostname of your host machine. The hexadecimal number is the ID of the Docker container.

LISTING ALL RUNNING CONTAINERS

Let's list all running containers in the following listing, so you can examine the list (I've edited the output to make it more readable—imagine the last two lines as the continuation of the first two).

Listing 2.5 Listing running cotnainers

```
$ docker ps
CONTAINER ID  IMAGE         COMMAND                CREATED         ...
44d76963e8e1  kubia:latest  "/bin/sh -c 'node ap  6 minutes ago   ...

...  STATUS              PORTS                    NAMES
...  Up 6 minutes        0.0.0.0:8080->8080/tcp   kubia-container
```

A single container is running. For each container, Docker prints out its ID and name, the image used to run the container, and the command that's executing inside the container.

GETTING ADDITIONAL INFORMATION ABOUT A CONTAINER

The `docker ps` command only shows the most basic information about the containers. To see additional information, you can use `docker inspect`:

```
$ docker inspect kubia-container
```

Docker will print out a long JSON containing low-level information about the container.

2.1.6 Exploring the inside of a running container

What if you want to see what the environment is like inside the container? Because multiple processes can run inside the same container, you can always run an additional process in it to see what's inside. You can even run a shell, provided that the shell's binary executable is available in the image.

Running a shell inside an existing container

The Node.js image on which you've based your image contains the *bash* shell, so you can run the shell inside the container like this:

```
$ docker exec -it kubia-container bash
```

This will run `bash` inside the existing *kubia-container* container. The `bash` process will have the same Linux namespaces as the main container process. This allows you to explore the container from within and see how Node.js and your app see the system when running inside the container. The `-it` option is shorthand for two options:

- `-i`, which makes sure STDIN is kept open. You need this for entering commands into the shell.
- `-t`, which allocates a pseudo terminal (TTY).

You need both if you want the use the shell like you're used to (if you leave out the first one, you can't type any commands, and if you leave out the second one, the command prompt won't be displayed and some commands will complain about the `TERM` variable not being set).

Exploring the container from within

Let's see how to use the shell in the following listing to see the processes running in the container.

Listing 2.6 Listing processes from inside a container

```
root@44d76963e8e1:/# ps aux
USER  PID %CPU %MEM    VSZ   RSS TTY STAT START TIME COMMAND
root    1  0.0  0.1 676380 16504 ?   Sl   12:31 0:00 node app.js
root   10  0.0  0.0  20216  1924 ?   Ss   12:31 0:00 bash
root   19  0.0  0.0  17492  1136 ?   R+   12:38 0:00 ps aux
```

You see only three processes. You don't see any other processes from the host OS.

Understanding that processes in a container run in the host Operating System

If you now open another terminal and list the processes on the host OS itself, you will, among all other host processes, also see the processes running in the container, as shown in listing 2.7.

NOTE If you're using a Mac or Windows, you'll need to log into the VM where the Docker daemon is running to see these processes.

Listing 2.7 A container's processes run in the host OS

```
$ ps aux | grep app.js
USER  PID %CPU %MEM    VSZ   RSS TTY STAT START TIME COMMAND
root  382  0.0  0.1 676380 16504 ?   Sl   12:31 0:00 node app.js
```

This proves that processes running in the container are running in the host OS. If you have a keen eye, you may have noticed that the processes have different IDs inside the container vs. on the host. The container is using its own PID Linux namespace and has a completely isolated process tree, with its own sequence of numbers.

THE CONTAINER'S FILESYSTEM IS ALSO ISOLATED

Like having an isolated process tree, each container also has an isolated filesystem. Listing the contents of the root directory inside the container will only show the files in the container and will include all the files that are in the image plus any files that are created while the container is running (log files and similar), as shown in the following listing.

Listing 2.8 A container has its own complete filesystem

```
root@44d76963e8e1:/# ls /
app.js  boot  etc   lib    media  opt   root  sbin  sys  usr
bin     dev   home  lib64  mnt    proc  run   srv   tmp  var
```

It contains the app.js file and other system directories that are part of the node:7 base image you're using. To exit the container, you exit the shell with the exit command and you'll be returned to your host machine (like logging out of an ssh session, for example).

> **TIP** Entering a running container like this is useful when debugging an app running in a container. When something's wrong, the first thing you'll want to explore is the actual state of the system your application sees. Keep in mind that an application will not only see its own unique filesystem, but also processes, users, hostname, and network interfaces.

2.1.7 Stopping and removing a container

To stop your app, you tell Docker to stop the `kubia-container` container:

```
$ docker stop kubia-container
```

This will stop the main process running in the container and consequently stop the container, because no other processes are running inside the container. The container itself still exists and you can see it with `docker ps -a`. The `-a` option prints out all the containers, those running and those that have been stopped. To truly remove a container, you need to remove it with the `docker rm` command:

```
$ docker rm kubia-container
```

This deletes the container. All its contents are removed and it can't be started again.

2.1.8 Pushing the image to an image registry

The image you've built so far has only been available on your local machine. To allow you to run it on any other machine, you need to push the image to an external image registry. For the sake of simplicity, you won't set up a private image registry and will instead push the image to Docker Hub (http://hub.docker.com), which is one of the publicly available registries. Other widely used such registries are Quay.io and the Google Container Registry.

Before you do that, you need to re-tag your image according to Docker Hub's rules. Docker Hub will allow you to push an image if the image's repository name starts with your Docker Hub ID. You create your Docker Hub ID by registering at http://hub.docker.com. I'll use my own ID (luksa) in the following examples. Please change every occurrence with your own ID.

TAGGING AN IMAGE UNDER AN ADDITIONAL TAG

Once you know your ID, you're ready to rename your image, currently tagged as "kubia", to "luksa/kubia" (replace luksa with your own Docker Hub ID):

```
$ docker tag kubia luksa/kubia
```

This doesn't rename the tag; it creates an additional tag for the same image. You can confirm this in the following listing by listing the images stored on your system with the `docker images` command.

Listing 2.9 A container image can have multiple tags

```
$ docker images | head
REPOSITORY          TAG       IMAGE ID        CREATED             VIRTUAL SIZE
luksa/kubia         latest    d30ecc7419e7    About an hour ago   654.5 MB
kubia               latest    d30ecc7419e7    About an hour ago   654.5 MB
docker.io/node      7.0       04c0ca2a8dad    2 days ago          654.5 MB
...
```

As you can see, both `kubia` and `luksa/kubia` point to the same Image ID, so they're in fact one single image with two tags.

PUSHING THE IMAGE TO DOCKER HUB

Before you can push the image to Docker Hub, you need to log in under your user ID with the `docker login` command. Once you're logged in, you can finally push the `yourid/kubia` image to Docker Hub like this:

```
$ docker push luksa/kubia
```

RUNNING THE IMAGE ON A DIFFERENT MACHINE

After the push to Docker Hub is complete, the image will be available to everyone. You can now run the image on any machine running Docker by executing the following command:

```
$ docker run -p 8080:8080 -d luksa/kubia
```

It doesn't get much simpler than that. And the best thing about this is that your application will have the exact same environment every time and everywhere it's run. If it ran fine on your machine, it should run as well on every other Linux machine. No need to worry about whether the host machine has Node.js installed or not. In fact, even if it does, your app won't use it, because it will use the one installed inside the image.

2.2 Setting up a Kubernetes cluster

Now that you have your app packaged inside a container image and made available through Docker Hub, you can deploy it in a Kubernetes cluster instead of running it in Docker directly. But first, you need to set up the cluster itself.

Setting up a full-fledged multi-node Kubernetes cluster isn't a simple task, especially if you're not well-versed in Linux and networking administration. A proper Kubernetes install spans multiple physical or virtual machines and requires the networking to be set up properly, so that all the containers running inside the Kubernetes cluster can connect to each other through the same flat networking space.

A long list of methods exists for installing a Kubernetes cluster. These methods are described in detail in the documentation at http://kubernetes.io. We're not going to list all of them here, because the list keeps evolving, but Kubernetes can be run on your local development machine, your own organization's cluster of machines, on cloud providers providing virtual machines (Google Compute Engine, Amazon EC2, Microsoft Azure, and so on) or by using a managed Kubernetes cluster such as Google Container Engine.

In this chapter, we'll cover two simple options for getting your hands on a running Kubernetes cluster. You'll see how to run a single-node Kubernetes cluster on your local machine and how to get access to a hosted cluster running on Google Container Engine (GKE).

A third option, which covers installing a cluster with the *kubeadm* tool, is explained in appendix B. The instructions there show you how to set up a three-node Kubernetes cluster using virtual machines, but I suggest you try it only after reading the first 11 chapters of the book.

Another option is to install Kubernetes on Amazon's AWS (Amazon Web Services). For this, you can look at the *kops* tool, which is built on top of kubeadm mentioned in the previous paragraph, and is available at http://github.com/kubernetes/kops. It helps you deploy production-grade, highly available Kubernetes clusters on AWS and will eventually support other platforms as well (Google Container Engine, VMware, vSphere, and so on).

2.2.1 Running a local single-node Kubernetes cluster with Minikube

The simplest and quickest way to a fully functioning Kubernetes cluster is by using *Minikube*. Minikube is a package that sets up a single-node cluster that's great for both testing Kubernetes and developing apps locally.

Although we can't show certain Kubernetes features related to managing apps on multiple nodes, the single-node cluster should be enough for exploring most topics discussed in this book.

Installing Minikube

Minikube is a single binary that needs to be downloaded and put onto your path. It's available for OSX, Linux, and Windows. To install it, the best place to start is to go to the Minikube repository on GitHub (http://github.com/kubernetes/minikube) and follow the instructions there.

For example, on OSX and Linux, Minikube can be downloaded and set up with a single command. For OSX, this is what the command looks like:

```
$ curl -Lo minikube https://storage.googleapis.com/minikube/releases/
[CA]v0.18.0/minikube-darwin-amd64 && chmod +x minikube && sudo mv minikube
[CA]/usr/local/bin/
```

On Linux, you download a different release (replace "darwin" with "linux" in the URL). On Windows, you can download the file manually, rename it to minikube.exe, and put it onto your path. Minikube runs Kubernetes inside a VM run through either VirtualBox or KVM, so you also need to install one of them before you can start the Minikube cluster.

Starting a Kubernetes cluster with Minikube

Once you have Minikube installed locally, you can immediately start up the Kubernetes cluster with the command in the following listing.

Listing 2.10 Starting a Minikube virtual machine

```
$ minikube start
Starting local Kubernetes cluster...
Starting VM...
SSH-ing files into VM...
...
Kubectl is now configured to use the cluster.
```

Starting the cluster takes more than a minute, so don't interrupt the command before it completes.

Installing the Kubernetes client (kubectl)

To interact with Kubernetes, you also need the *kubectl* CLI client. Again, all you need to do is download it and put it on your path. The latest stable release for OSX, for example, can be downloaded and installed with the following command:

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release
[CA] /$(curl -s https://storage.googleapis.com/kubernetes-release/release
[CA] /stable.txt)/bin/darwin/amd64/kubectl
[CA] && chmod +x kubectl
[CA] && sudo mv kubectl /usr/local/bin/
```

To download kubectl for Linux or Windows, replace `darwin` in the URL with either `linux` or `windows`.

> **NOTE** If you'll be using multiple Kubernetes clusters (for example, both Minikube and GKE), refer to appendix A for information on how to set up and switch between different kubectl contexts.

CHECKING TO SEE THE CLUSTER IS UP AND KUBECTL CAN TALK TO IT

To verify your cluster is working, you can use the `kubectl cluster-info` command in the following listing.

Listing 2.11 Displaying cluster information

```
$ kubectl cluster-info
Kubernetes master is running at https://192.168.99.100:8443
KubeDNS is running at https://192.168.99.100:8443/api/v1/proxy/...
kubernetes-dashboard is running at https://192.168.99.100:8443/api/v1/...
```

This shows the cluster is up. It shows the URLs of the various Kubernetes components, including the API server and the web console.

> **TIP** You can run `minikube ssh` to log into the Minikube VM and explore it from the inside. For example, you may want to see what processes are running on the node.

2.2.2 Using a hosted Kubernetes cluster with Google Container Engine

If you want to explore a full-fledged multi-node Kubernetes cluster instead, you can use a managed Google Container Engine (GKE) Kubernetes cluster. This way, you don't need to manually set up all the cluster nodes and networking, which is usually too much for someone making their first steps with Kubernetes. Using a managed solution such as GKE makes sure you don't end up with a misconfigured, non-working, or partially working cluster.

SETTING UP A GOOGLE CLOUD PROJECT AND DOWNLOADING THE NECESSARY CLIENT BINARIES

Before you can set up a new Kubernetes cluster, you need to set up your GKE environment. Because the process may change, I'm not listing the exact instructions here. To get started, please follow the instructions at https://cloud.google.com/container-engine/docs/before-you-begin.

Roughly, the whole procedure includes

1. Signing up for a Google account, in the unlikely case you don't have one already.
2. Creating a project in the Google Cloud Platform Console.
3. Enabling billing. This does require your credit card info, but Google provides a 12-month free trial. And they're nice enough to not start charging automatically after the free trial is over.)
4. Enabling the Container Engine API.

5 Downloading and installing Google Cloud SDK. (This includes the *gcloud* command-line tool, which you'll need to create a Kubernetes cluster.)
6 Installing the kubectl command-line tool with `gcloud components install kubectl`.

NOTE Certain operations (the one in step 2, for example) may take a few minutes to complete, so relax and grab a coffee in the meantime.

Creating a Kubernetes cluster with three nodes

After completing the installation, you can create a Kubernetes cluster with three worker nodes with the command shown in the following listing.

Listing 2.12 Creating a three-node cluster in GKE

```
$ gcloud container clusters create kubia --num-nodes 3
[CA]--machine-type f1-micro
Creating cluster kubia...done.
Created [https://container.googleapis.com/v1/projects/kubia1-
     1227/zones/europe-west1-d/clusters/kubia].
kubeconfig entry generated for kubia.
NAME   ZONE   MST_VER MASTER_IP     TYPE     NODE_VER NUM_NODES STATUS
kubia  eu-w1d 1.5.3   104.155.92.30 f1-micro 1.5.3    3         RUNNING
```

You should now have a running Kubernetes cluster with three worker nodes as shown in figure 2.4. You're using three nodes to help better demonstrate features that apply to multiple nodes. You can use a smaller number of nodes, if you want.

Getting an overview of your cluster

To give you a basic idea of what your cluster looks like and how to interact with it, see figure 2.4. Each node runs Docker, the Kubelet and the kube-proxy. You'll interact with the cluster through the `kubectl` command line client, which issues REST requests to the Kubernetes API server running on the master node.

Checking if the cluster is up by listing cluster nodes

You'll use the `kubectl` command now to list all the nodes in your cluster, as shown in the following listing.

Listing 2.13 Listing cluster nodes with kubectl

```
$ kubectl get nodes
NAME                       STATUS  AGE  VERSION
gke-kubia-85f6-node-0rrx   Ready   1m    v1.5.3
gke-kubia-85f6-node-heo1   Ready   1m    v1.5.3
gke-kubia-85f6-node-vs9f   Ready   1m    v1.5.3
```

The `kubectl get` command you used lists all kinds of Kubernetes objects. You'll use it constantly, but it usually shows only the most basic information for the listed objects.

TIP You can log into one of the nodes with `gcloud compute ssh <node-name>` to explore what's running on the node.

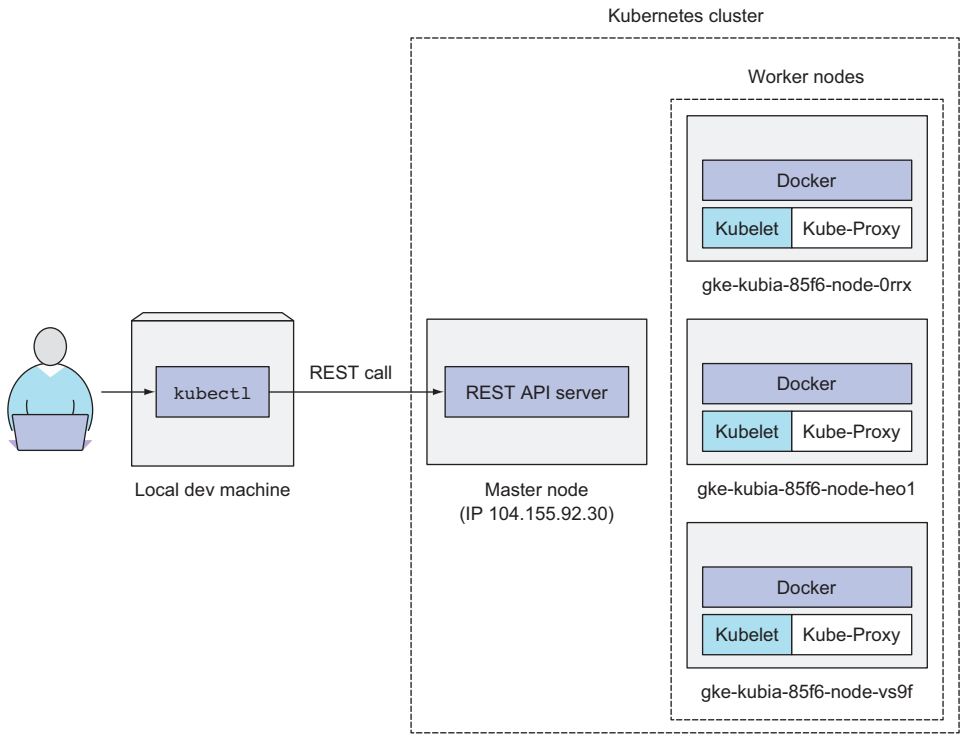


Figure 2.4 How you're interacting with your three-node Kubernetes cluster.

RETRIEVING ADDITIONAL DETAILS OF AN OBJECT

To see more detailed information about an object, you can use the `kubectl describe` command, which shows even more:

```
$ kubectl describe node gke-kubia-85f6-node-0rrx
```

I'm omitting the actual output of the describe command, because it's fairly wide and would be completely unreadable here in the book. The output shows the node's status, its CPU and memory data, system information, containers running on the node, and much more.

In the previous `kubectl describe` example, you specified the name of a specific node, but you could also have performed a simple `kubectl describe node` and it would print out a detailed description of all the nodes.

> **TIP** Running the `describe` and `get` commands without specifying the name of the object comes in handy when only one object of a given type exists, so you don't waste time typing or copy/pasting the object's name.

While we're talking about reducing keystrokes, let me give you additional advice on how to make working with kubectl much easier, before we move on to running your first app in Kubernetes.

2.2.3 Setting up an alias and command-line completion for kubectl

You'll use the kubectl often. You'll soon realize that having to type the full command every time is a real pain. Before you begin, take a minute to make your life easier by setting up an alias and tab completion for `kubectl`.

CREATING AN ALIAS

Throughout the book, I'll always be using the full name of the `kubectl` executable, but you may want to add a short alias such as *k*, so you won't have to type `kubectl` every time. If you haven't used aliases yet, here's how you define one. Add the following line to your `~/.bashrc` or equivalent file:

```
alias k=kubectl
```

> **NOTE** You may already have the `k` executable if you used `gcloud` to set up the cluster.

CONFIGURING TAB COMPLETION FOR KUBECTL

Even with a short alias such as `k`, you'll still need to type way more than you'd like. Luckily, the `kubectl` command can also output shell completion code for both the bash and zsh shell. The nice thing about it is that it doesn't only enable tab completion of command names, but also of the actual object names. For example, instead of having to write the whole node name in one of your previous examples, all you'd need to type is

```
$ kubectl desc<TAB> no<TAB> gke-ku<TAB>
```

To enable tab completion in Bash, you'll first need to install a package called bash-completion and then run the following command (you'll probably also want to add it to `~/.bashrc` or equivalent):

```
$ source <(kubectl completion bash)
```

But there's one caveat. When you run the preceding command, tab completion will only work when you use the full `kubectl` name (it won't work when you use the `k` alias). To fix this, you need to transform the output of the `kubectl completion` command a bit:

```
$ source <(kubectl completion bash | sed s/kubectl/k/g)
```

> **NOTE** Unfortunately, as I'm writing this, shell completion doesn't work for aliases on MacOS. You'll have to use the full `kubectl` command name if you want completion to work.

Now you're all set to start interacting with your cluster without having to type too much. You can finally run your first app on Kubernetes.

2.3 Running your first app on Kubernetes

For your first time, you'll use the simplest possible way of running an app on Kubernetes. Usually, you'd prepare a JSON or YAML manifest, containing a description of all the components you want to deploy, but because we haven't talked about the types of components you can create in Kubernetes yet, you'll use a simple one-line command to get something running.

2.3.1 Deploying your Node.js app

The simplest way to deploy your app is to use the `kubectl run` command, which will create all the necessary components without having to deal with JSON or YAML. This way, we don't need to dive into the structure of each object yet. Try to run the image you created and pushed to Docker Hub earlier. Here's how to run it in Kubernetes:

```
$ kubectl run kubia --image=luksa/kubia --port=8080 --generator=run/v1
replicationcontroller "kubia" created
```

The `--image=luksa/kubia` part obviously specifies the container image you want to run, and the `--port=8080` option tells Kubernetes that your app is listening on port 8080. The last flag (`--generator`) does require an explanation, though. Usually, you won't use it, but you're using it here so Kubernetes creates a *ReplicationController* instead of a *Deployment*. You'll learn what ReplicationControllers are later in the chapter, but we won't talk about Deployments until chapter 9. That's why I don't want `kubectl` to create a Deployment yet.

As the previous command's output shows, a ReplicationController called *kubia* has been created. As already mentioned, we'll see what that is later in the chapter. For now, let's start from the bottom and focus on the container you created (you can assume a container has been created, because you specified a container image in the `run` command).

INTRODUCING PODS

You may be wondering if you can see your container in a list showing all the running containers. Maybe something such as `kubectl get containers`? Well, that's not exactly how Kubernetes works. It doesn't deal with individual containers directly. Instead, it uses the concept of multiple collocated containers. This group of containers is called a *Pod*.

A Pod is a group of one or more tightly related containers that will always run together on the same worker node and in the same Linux namespace(s). Each Pod is like a separate logical machine with its own IP, hostname, processes, and so on, running a single application. The application can be a single process, running in a single container, or it can be a main application process and additional supporting processes, each running in their own container. All the containers in a Pod will appear to be running on the same logical machine, whereas containers in other Pods, even if they're running on the same worker node, will appear to be running on a different one.

To better understand the relationship between containers, Pods, and nodes, examine figure 2.5. As you can see, each Pod has its own IP and contains one or more containers, each running an application process. Pods are spread out across different worker nodes.

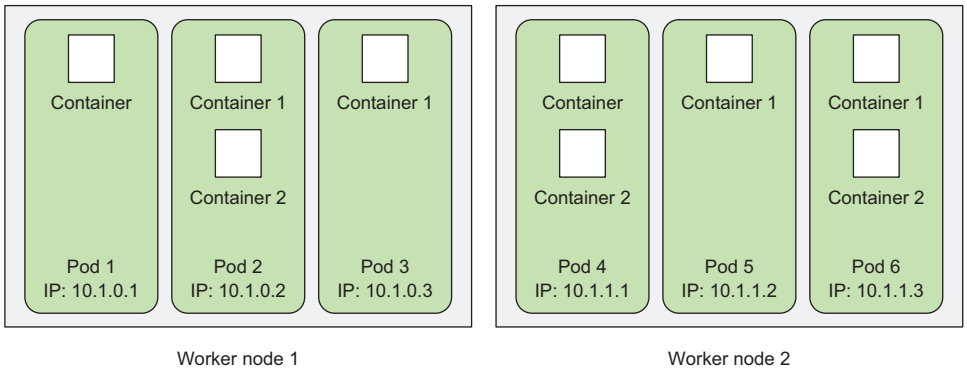


Figure 2.5 The relationship between containers, Pods, and physical worker nodes.

Listing Pods

Because you can't list individual containers, since they're not standalone Kubernetes objects, can you list Pods instead? Yes, you can. Let's see how to tell `kubectl` to list Pods in the following listing.

Listing 2.14 Listing Pods

```
$ kubectl get pods
NAME          READY     STATUS    RESTARTS   AGE
kubia-4jfyf   0/1       Pending   0          1m
```

This is your Pod. Its status is still Pending and the Pod's single container is shown as not ready yet (this is what the `0/1` in the `READY` column means). The reason why the Pod isn't running yet is because the worker node the Pod has been assigned to is downloading the container image before it can run it. When the download is finished, the Pod's container will be created and then the Pod will transition to the Running state, as shown in the following listing.

Listing 2.15 Listing Pods again to see if the Pod's status has changed

```
$ kubectl get pods
NAME          READY     STATUS    RESTARTS   AGE
kubia-4jfyf   1/1       Running   0          5m
```

To see more information about the Pod, you can also use the `kubectl describe pod` command, like you did earlier for one of the worker nodes. If the Pod stays stuck in the *Pending* status, it might be that Kubernetes can't pull the image from the registry. If you're using your own image, make sure it's marked as public on Docker Hub.

To make sure the image can be pulled successfully, try pulling the image manually with the `docker pull` command on another machine.

UNDERSTANDING WHAT HAPPENED BEHIND THE SCENES

To help you visualize what you did, look at figure 2.6. It shows both steps you had to perform to get a container image running inside Kubernetes. First, you built the image and pushed it to Docker Hub. This was necessary because building the image on your local machine only makes it available on your local machine, but you needed to make it accessible to the Docker daemons running on your worker nodes.

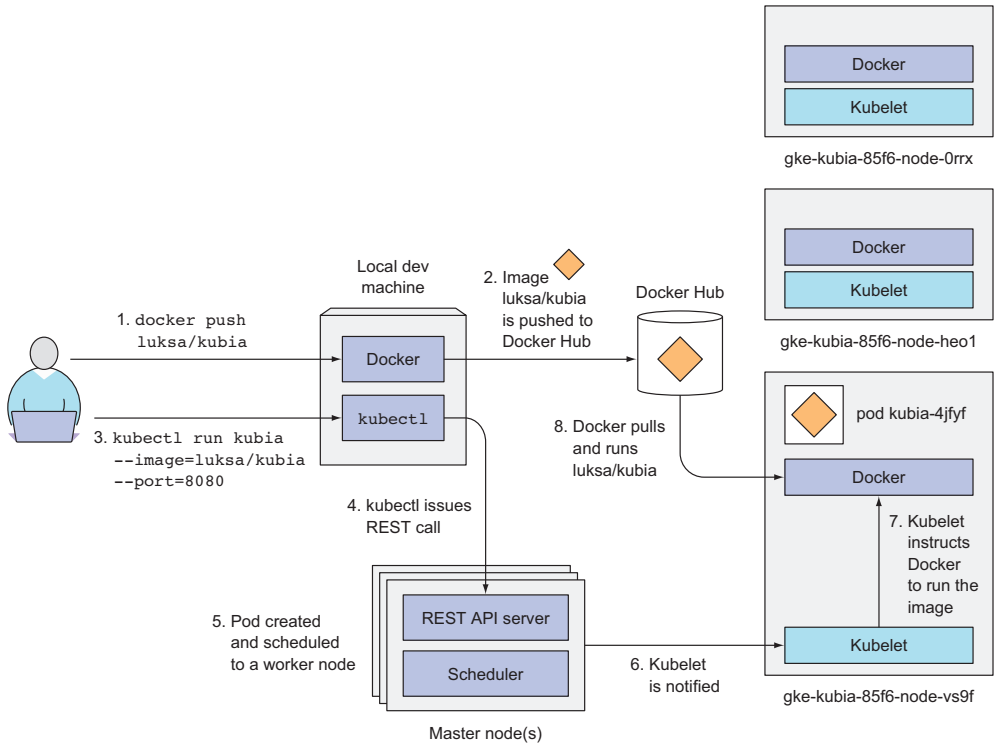


Figure 2.6 Running the luksa/kubia container image in Kubernetes.

When you ran the `kubectl` command, it created a new ReplicationController object in the cluster by performing a REST HTTP request to the Kubernetes API server. The replication controller then created a new Pod, which was then scheduled to one of the worker nodes by the scheduler. The Kubelet on that node saw that the Pod was scheduled to it and instructed Docker to pull the specified image from the registry because the image wasn't available locally. After downloading the image, Docker created and ran the container.

The other two nodes are displayed to show context. They didn't play any role in the process, because the Pod wasn't scheduled to them.

DEFINITION The term *scheduling* means assigning the Pod to a node. The Pod is run immediately, not at a time in the future as the term might lead you to believe.

2.3.2 Accessing your web application

With your Pod running, how do you access it? We mentioned that each Pod gets its own IP address, but this address is internal to the cluster and isn't accessible from outside of it. To make the Pod accessible from the outside, you'll expose it through a *service*. You'll create a special service of type LoadBalancer, because if you create a regular service (a so-called *ClusterIP* service), like the Pod, it would also only be accessible from inside the cluster. By creating a LoadBalancer-type service, an external load balancer will be created and you can connect to the Pod through the load balancer's public IP.

CREATING A SERVICE

To create the service, you'll tell Kubernetes to *expose* the replication controller you created earlier:

```
$ kubectl expose rc kubia --type=LoadBalancer --name kubia-http
service "kubia-http" exposed
```

NOTE We're using the abbreviation "rc" instead of "replicationcontroller." Most resource types have an abbreviation like this so you don't have to type the full name (for example, *po* for Pods, *svc* for services, and so on).

LISTING SERVICES

The expose command's output mentions a service called *kubia-http*. Services are objects like Pods and nodes, so you should see the newly created service by running the `kubectl get services` command, as shown in the following listing.

Listing 2.16 Listing Services

```
$ kubectl get services
NAME         CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
kubernetes   10.3.240.1      <none>        443/TCP          34m
kubia-http   10.3.246.185    <pending>     8080:31348/TCP   4s
```

The list shows two services. Ignore the kubernetes service for now and take a close look at the kubia-http service you created. It doesn't have an external IP address yet, because it takes time for the load balancer to be created by the cloud infrastructure Kubernetes is running on. Once the load balancer is up, the external IP address of the service should be displayed. Let's wait a while and list the services again, as shown in the following listing.

Listing 2.17 Listing Services again to see if an external IP has been assigned

```
$ kubectl get svc

NAME         CLUSTER-IP     EXTERNAL-IP    PORT(S)          AGE
kubernetes   10.3.240.1     <none>         443/TCP          35m
kubia-http   10.3.246.185   104.155.74.57  8080:31348/TCP   1m
```

Aha, there's the external IP. Your service is now accessible at http://104.155.74.57:8080 from anywhere in the world.

> **NOTE** Minikube doesn't support LoadBalancer services yet, so the service will never get an external IP. But you can access the service anyway through its external port. How to do that is described in the next section's tip.

ACCESSING YOUR SERVICE THROUGH ITS EXTERNAL IP

You can now send requests to your Pod through the service's external IP and port:

```
$ curl 104.155.74.57:8080
You've hit kubia-4jfyf
```

Woohoo! Your app is now running somewhere in your three-node Kubernetes cluster (or a single-node cluster if you're using Minikube). If you don't count the steps required to set up the whole cluster, all it took was two simple commands to get your app running and to make it accessible to users across the world.

> **TIP** When using Minikube, you can get the IP and port through which you can access the service by running `minikube service kubia-http`.

If you look closely, you'll see that the app is reporting the name of the Pod as its hostname. As already mentioned, each Pod behaves like a separate independent machine with its own IP address and hostname. Even though the application is running in the worker node's operating system, to the app it appears as though it's running on a completely separate machine dedicated to the app itself—no other processes are running alongside it.

2.3.3 The logical parts of your system

Until now, I've mostly explained the actual physical components of your system. You have three worker nodes, which are VMs running Docker and the Kubelet, and you have a master node that controls the whole system. Honestly, we don't know if a single master node is hosting all the individual components of the Kubernetes control plane or if they're split across multiple nodes. It doesn't really matter, because you're only interacting with the API server, which is accessible at a single endpoint.

Anyway, beside this physical view of the system, there's also a separate, logical view of it. I've already mentioned Pods, ReplicationControllers, and Services. All of them will be explained in the next few chapters, but let's look quickly at how they fit together and what roles they play in your little setup.

Understanding how the replication controller, the Pod, and the service fit together

As I've already explained, you're not creating and working with containers directly. Instead, the basic building block in Kubernetes is the Pod. But, you didn't really create any Pods either, at least not directly. By running the `kubectl run` command you created a replication controller, and this replication controller is what created the actual Pod. To make that Pod accessible from outside the cluster, you told Kubernetes to expose all the Pods managed by that replication controller as a single service. A rough picture of all three elements is presented in figure 2.7.

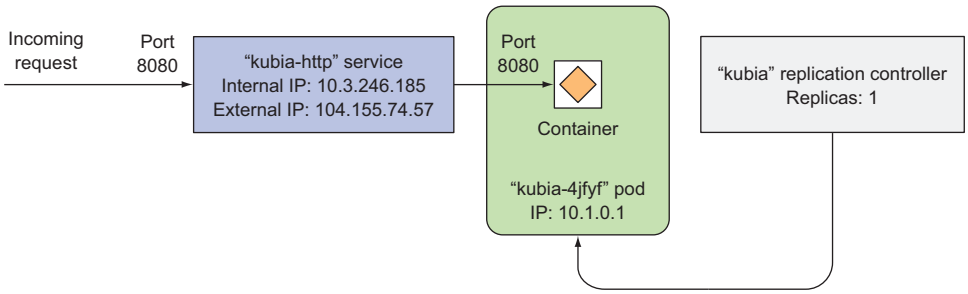


Figure 2.7 Your system consists of a replication controller, a Pod, and a service.

Understanding the Pod and its container

The main and most important component in your system is the Pod. It contains only a single container, but generally a Pod can contain as many containers as you want. Inside the container is your Node.js process, which is bound to port 8080 and is waiting for HTTP requests. The Pod has its own unique private IP address and hostname.

Understanding the role of the replication controller

The next component is the "kubia" replication controller. It makes sure there's always exactly one instance of your Pod running. Generally, replication controllers are used to replicate Pods (that is, create multiple copies of a Pod) and keep them running. In your case, you didn't specify how many Pod replicas you want, so the replication controller created a single one. If your Pod were to disappear for any reason, the replication controller would create a new Pod to replace the missing one.

Understanding why you need a service

The third component of your system is the "kubia-http" service. To understand why you need to have services, you need to learn something about Pods. They're ephemeral. A Pod may disappear at any time, because the node it's running on has failed, because someone deleted the Pod, or because the Pod was evicted from an otherwise healthy node. When any of those occurs, a missing Pod is replaced with a new one by the replication controller, as described previously. This new Pod gets a different IP address from the Pod it's replacing. This is where services come in—to solve the problem of ever-changing Pod IP addresses, as well as exposing multiple Pods at a single constant IP and port pair.

When a service is created, it gets a static IP, which never changes during the lifetime of the service. Instead of connecting to Pods directly, clients should connect to the service through its constant IP address, and the service makes sure one of the Pods receives the connection, regardless of where the Pod is currently running (and what its IP address is).

Services represent a static location for a group of one or more Pods that all provide the same service. Requests coming to the IP and port of the service will be forwarded to the IP and port of one of the Pods that's running at that moment.

2.3.4 Horizontally scaling the application

You now have a running application, monitored and kept running by a replication controller and exposed to the world through a service. Now let's make additional magic happen.

One of the main benefits of using Kubernetes is the simplicity with which you can scale your deployments. Let's see how easy it is to scale up the number of Pods. You'll increase the number of running instances to three.

Your Pod is managed by a ReplicationController. Let's see it with the `kubectl get` command:

```
$ kubectl get replicationcontrollers
NAME        DESIRED     CURRENT     AGE
kubia       1           1           17m
```

> **Listing all the resource types with kubectl get**
>
> You've been constantly using the same basic `kubectl get` command to list things in your cluster. You've used this command to list *nodes*, *Pods*, *services,* and *replication controllers*. You can get a list of all the possible resource types by invoking `kubectl get` without specifying the type. You can then use those types with various `kubectl` commands such as `get`, `describe`, and so on. The list also shows the abbreviations I mentioned earlier.

The list shows a single ReplicationController called *kubia*. The `DESIRED` column shows the number of Pod replicas you want the replication controller to keep, whereas the `CURRENT` column shows the actual number of Pods currently running. In your case, you wanted to have a single replica of the Pod running, and exactly one replica is currently running.

INCREASING THE DESIRED REPLICA COUNT

To scale up the number of replicas of your Pod, you need to change the desired replica count on the replication controller like this:

```
$ kubectl scale rc kubia --replicas=3
replicationcontroller "kubia" scaled
```

You've now told Kubernetes to make sure three instances of your Pod are always running. Notice that you didn't instruct Kubernetes what action to take. You didn't tell it to add two more Pods. You only set the new desired number of instances and let Kubernetes determine what actions it needs to take to achieve the requested state.

This is one of the most fundamental Kubernetes principles. Instead of telling Kubernetes exactly what actions it should perform, you're only declaratively changing the desired state of the system and letting Kubernetes examine the current actual state and reconcile it with the desired state. This is true across all of Kubernetes.

SEEING THE RESULTS OF THE SCALE-OUT

Back to your replica count increase. Let's list the replication controllers again to see the updated replica count:

```
$ kubectl get rc
NAME        DESIRED    CURRENT    READY    AGE
kubia       3          3          2        17m
```

Because the actual number of Pods has already been increased to three (as evident from the `CURRENT` column), listing all the Pods should now show three Pods instead of one:

```
$ kubectl get pods
NAME          READY     STATUS    RESTARTS   AGE
kubia-hczji   1/1       Running   0          7s
kubia-iq9y6   0/1       Pending   0          7s
kubia-4jfyf   1/1       Running   0          18m
```

As you can see, three Pods exist instead of one. Two are already running, one is still pending, but should be ready in a few moments, as soon as the container image is downloaded and the container is started.

As you can see, scaling an application is incredibly simple. Once your app is running in production and a need to scale the app arises, you can add additional instances with a single command without having to install and run additional copies manually.

Keep in mind that the app itself needs to support being scaled horizontally. Kubernetes doesn't magically make your app scalable; it makes it trivial to scale the app up or down.

SEEING REQUESTS HIT ALL THREE PODS WHEN HITTING THE SERVICE

Because you now have multiple instances of your app running, let's see what happens if you hit the service URL again. Will you always hit the same app instance or not?

```
$ curl 104.155.74.57:8080
You've hit kubia-hczji
$ curl 104.155.74.57:8080
You've hit kubia-iq9y6
$ curl 104.155.74.57:8080
You've hit kubia-iq9y6
$ curl 104.155.74.57:8080
You've hit kubia-4jfyf
```

Requests are hitting different Pods randomly. This is what services in Kubernetes do when more than one Pod instance backs them. They act as a load balancer standing in front of multiple Pods. When there's only one Pod, services provide a static address for the single Pod. Whether a service is backed by a single Pod or a group of Pods, those Pods come and go as they're moved around the cluster, which means their IP addresses change, but the service is always there at the same address. This makes it easy for clients to connect to the Pods, regardless of how many exist and how often they change location.

VISUALIZING THE NEW STATE OF YOUR SYSTEM

Let's visualize your system again to see what's changed from before. Figure 2.8 shows the new state of your system. You still have a single service and a single replication controller, but you now have three instances of your Pod, all managed by the replication controller. The service no longer sends all requests to a single Pod, but spreads them across all three Pods as shown in the experiment with `curl` a few moments ago.

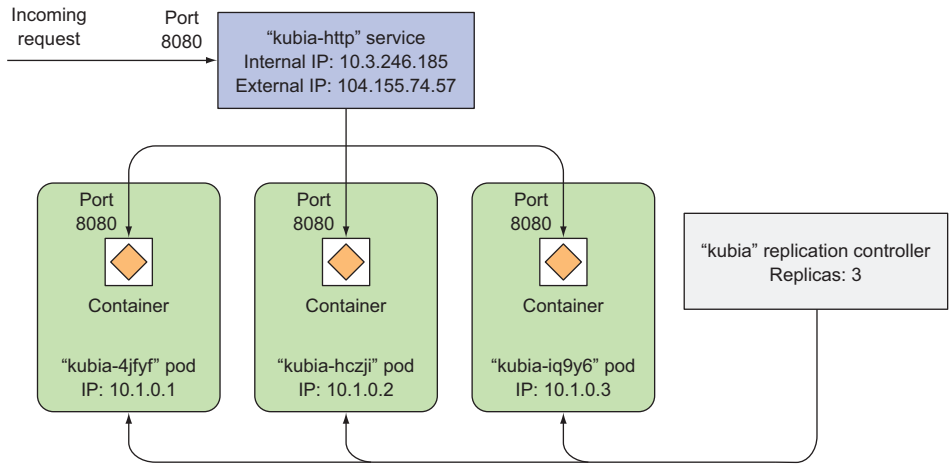


Figure 2.8 Three instances of a Pod managed by the same replication controller and exposed through a single service IP and port.

As an exercise, you can now try spinning up additional instances by increasing the replication controller's replica count even further and then scaling back down.

2.3.5 Examining what nodes your app is running on

You may be wondering what nodes your Pods have been scheduled to. In the Kubernetes world, what node a Pod is running on isn't that important, as long as it gets scheduled to a node that can provide the CPU and memory the Pod needs to run properly.

Regardless of the node they're scheduled to, all the apps running inside containers have the same type of OS environment. Each Pod has its own IP and can talk to

any other Pod, regardless of whether that other Pod is also running on the same node or on another node. Each Pod is provided with the requested amount of computational resources, so whether those resources are provided by one node or another doesn't make any difference.

Displaying the Pod IP and the Pod's node when listing Pods

If you've been paying close attention, you probably noticed that the `kubectl get pods` command doesn't even show any information about the nodes the Pods are scheduled to. This is because it's usually not an important piece of information.

But you can request additional columns to display using the `-o wide` option. When listing Pods, this option shows the Pod's IP and the node the Pod is running on:

```
$ kubectl get pods -o wide
NAME          READY   STATUS    RESTARTS   AGE   IP         NODE
kubia-hczji   1/1     Running   0          7s    10.1.0.2   gke-kubia-85...
```

Inspecting other details of a Pod with kubectl describe

You can also see the node by using the `kubectl describe` command, which shows many other details of the Pod, as shown in the following listing.

Listing 2.18 Describing a Pod with `kubectl describe`

```
$ kubectl describe pod kubia-hczji
Name:         kubia-hczji
Namespace:    default
Node:         gke-kubia-85f6-node-vs9f/10.132.0.3
Start Time:   Fri, 29 Apr 2016 14:12:33 +0200
Labels:       run=kubia
Status:       Running
IP: 10.1.0.2
Controllers: ReplicationController/kubia
Containers:   ...
Conditions:
  Type        Status
  Ready       True
Volumes: ...
Events: ...
```

Here's the node the Pod has been scheduled to.

This shows, among other things, the node the Pod has been scheduled to, the time when it was started, the image(s) it's running, and other useful information.

2.3.6 Introducing the Kubernetes dashboard

Before we wrap up this initial hands-on chapter, let's look at another way of exploring your Kubernetes cluster.

Up to now, you've only been using the kubectl command-line tool. If you're more into graphical web user interfaces, you'll be glad to hear that Kubernetes also comes with a nice (but still evolving) web dashboard.

The dashboard allows you to list all the Pods, replication controllers, services, and other objects deployed in your cluster, as well as to create, modify, and delete them. Figure 2.9 shows the dashboard.

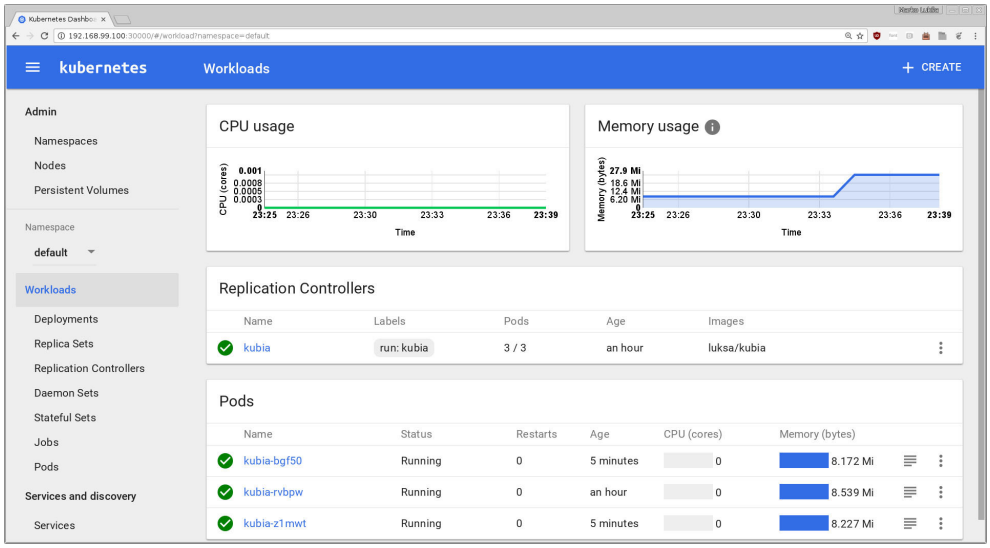


Figure 2.9 **Screenshot of the Kubernetes web-based dashboard.**

Although you won't use the dashboard in this book, you can open it up any time to quickly see a graphical view of what's deployed in your cluster after you create or modify objects through `kubectl`.

ACCESSING THE DASHBOARD WHEN RUNNING KUBERNETES IN GKE

If you're using Google Container Engine, you can find out the URL of the dashboard through the `kubectl cluster-info` command, which we already introduced:

```
$ kubectl cluster-info | grep dashboard
kubernetes-dashboard is running at https://104.155.108.191/api/v1/proxy/
[CA]namespaces/kube-system/services/kubernetes-dashboard
```

If you open this URL in a browser, you're presented with a username and password prompt. You'll find the username and password by running the following command:

```
$ gcloud container clusters describe kubia | grep -E "(username|password):"
  password: 32nENgreEJ632A12
  username: admin
```

The username and password for the dashboard

ACCESSING THE DASHBOARD WHEN USING MINIKUBE

To open the dashboard in your browser when using Minikube to run your Kubernetes cluster, run the following command:

```
$ minikube dashboard
```

The dashboard will open in your default browser. Unlike with GKE, you won't need to enter any credentials to access it.

2.4 Summary

Hopefully, this initial hands-on chapter has shown you that Kubernetes isn't a complicated platform to use, and you're ready to learn in depth about all the things it can provide. After reading this chapter, you should now know how to

- Pull and run any publicly available container image
- Package your apps into container images and make them available to anyone by pushing the images to a remote image registry
- Enter a running container and inspect its environment
- Set up a multi-node Kubernetes cluster on Google Container Engine
- Configure an alias and tab completion for the kubectl command-line tool
- List and inspect nodes, Pods, services, and replication controllers in a Kubernetes cluster
- Run a container in Kubernetes and make it accessible from outside the cluster
- Have a basic sense of how Pods, ReplicationControllers, and Services relate to one another
- Scale an app horizontally by changing the replication controller's replica count
- Access the web-based Kubernetes dashboard on both Minikube and GKE

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 containers 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 collocated 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—a Pod never spans multiple worker nodes, as shown in figure 3.1.

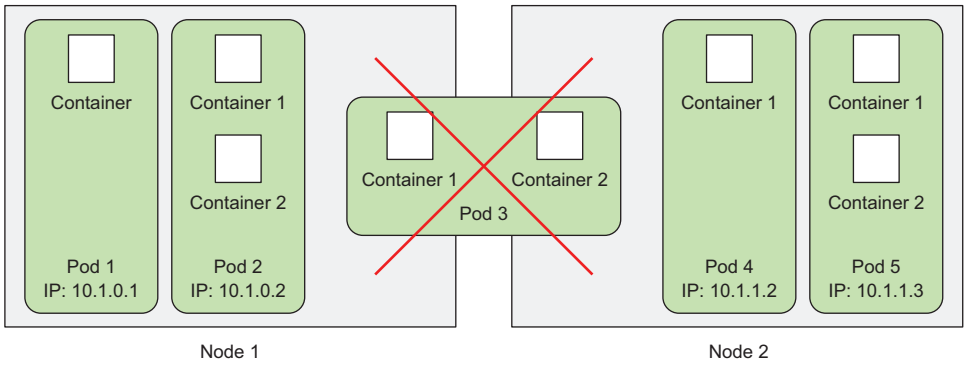


Figure 3.1 **All containers of a Pod run on the same node. A Pod never spans two nodes.**

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 with 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 Linux namespaces instead of each container having its own set of namespaces.

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. They should also share the same PID namespace, but that's currently not the case.

> **NOTE** Because containers of the same Pod currently 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.

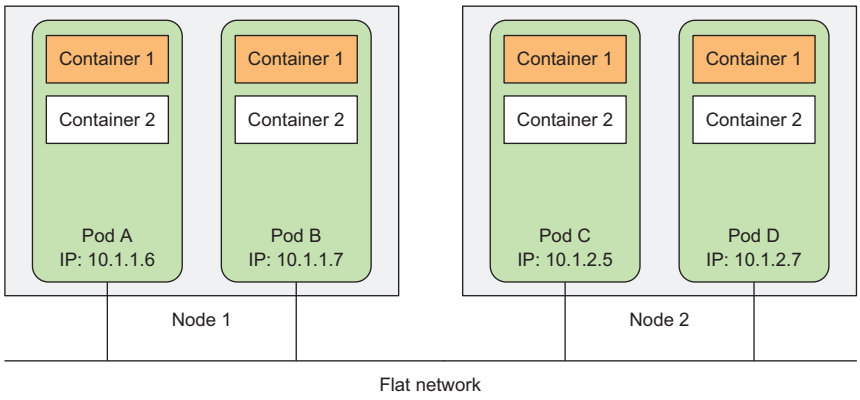


Figure 3.2 Each Pod gets a routable IP address and all other Pods see the Pod under that IP address.

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 collocated, but do the web server and the database really need to run on the same machine? The answer is obviously no, so nothing is forcing you 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 always 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 improve the utilization of your infrastructure.

SPLITTING INTO MULTIPLE PODS BECAUSE OF 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 consisted of a frontend and a backend container, when you'd scale up the number of instances of the Pod to, let's say, two, you'd end up with two frontend containers and two backend containers.

Usually, frontend components have completely different scaling requirements as 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 add 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.

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 so-called *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 in that case and mount it into both containers.

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

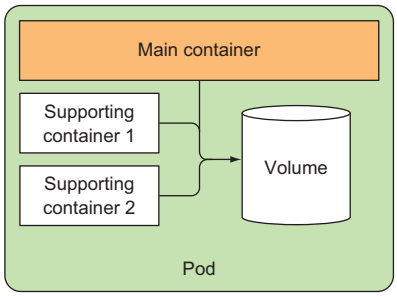


Figure 3.3 Pods should contain tightly coupled containers, usually a main container and containers that support the main one,

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.

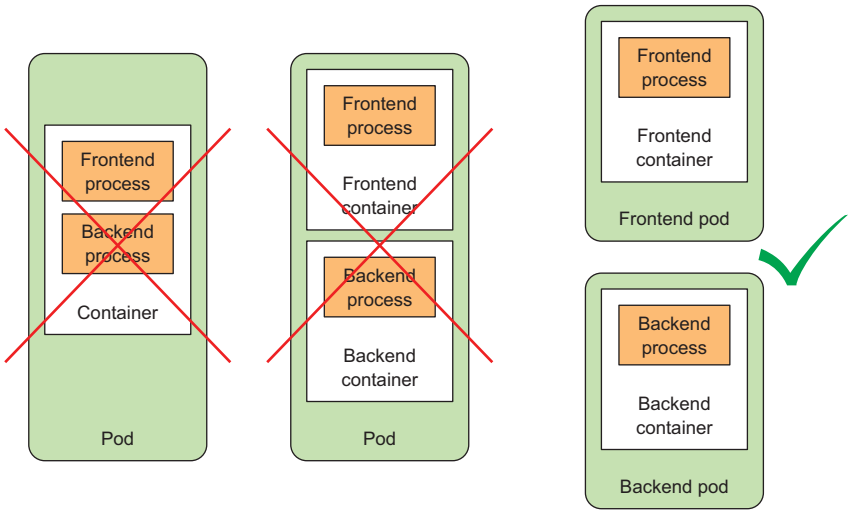


Figure 3.4 A container shouldn't run multiple processes. A Pod shouldn't contain multiple containers if they don't need to run on the same machine.

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                                  <- Kubernetes API version used in this YAML descriptor
kind: Pod                                       <- Type of Kubernetes object/resource
metadata:                                       \
  annotations:                                   |
    kubernetes.io/created-by: ...                |
  creationTimestamp: 2016-03-18T12:37:50Z        |
  generateName: kubia-                           |
  labels:                                        |  Pod metadata
    run: kubia                                   |  (name, labels,
  name: kubia-zxzij                              |  annotations,
  namespace: default                             |  and so on)
  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                                  |  Pod specification/contents
    ports:                                       |  (list of Pod's containers,
    - containerPort: 8080                        |  volumes, and so on)
      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:
  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) *(continued)*

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 at the end, the 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 contains 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 the file in any directory you want), or download the book's code archive and 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 <-- Descriptor conforms to version v1 of Kubernetes API
kind: Pod <-- You're describing a Pod
metadata:
  name: kubia-manual <-- The name of the Pod
spec:
  containers:
  - image: luksa/kubia <-- Container image to create the container from
    name: kubia <-- Name of the container
    ports:
    - containerPort: 8080 <-- The port the app is listening on
      protocol: TCP
```

I'm sure you'll agree this is much simpler compared to 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 (obviously, 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 your 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 get 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. Kubectl logs only shows the log entries in 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.

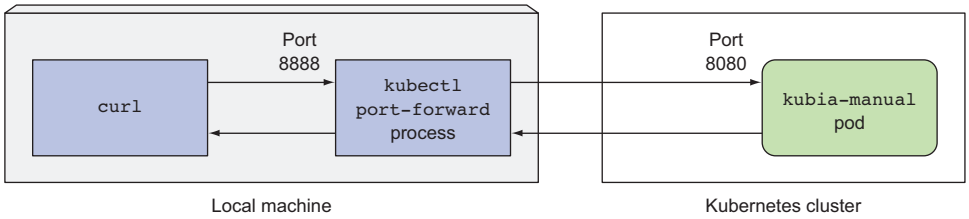


Figure 3.5 A simplified view of what happens when you use curl with kubectl port-forward.

Using port forwarding like this is an effective way to test a specific 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 microservice 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.

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*.

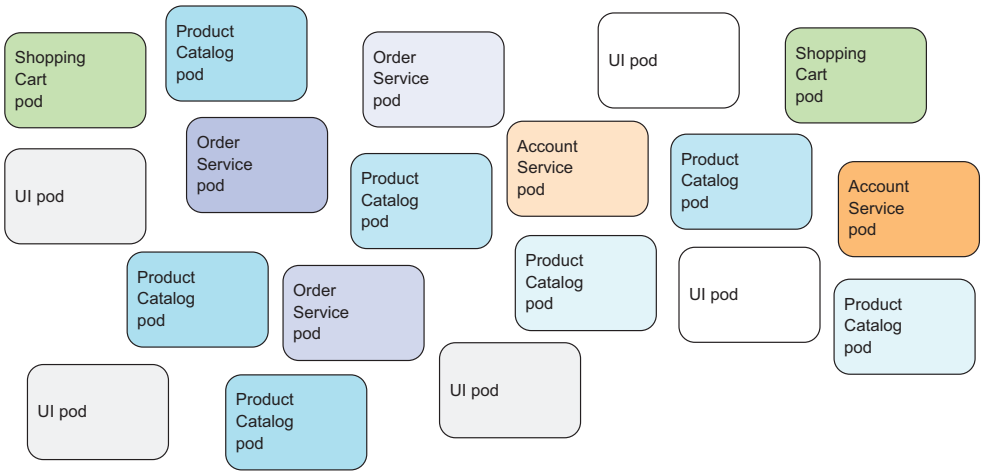


Figure 3.6 **Uncategorized Pods in a microservices architecture.**

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 can attach to a resource and then use it to select only those resources that have that exact label (this is done through *label selectors*). 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 your 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.

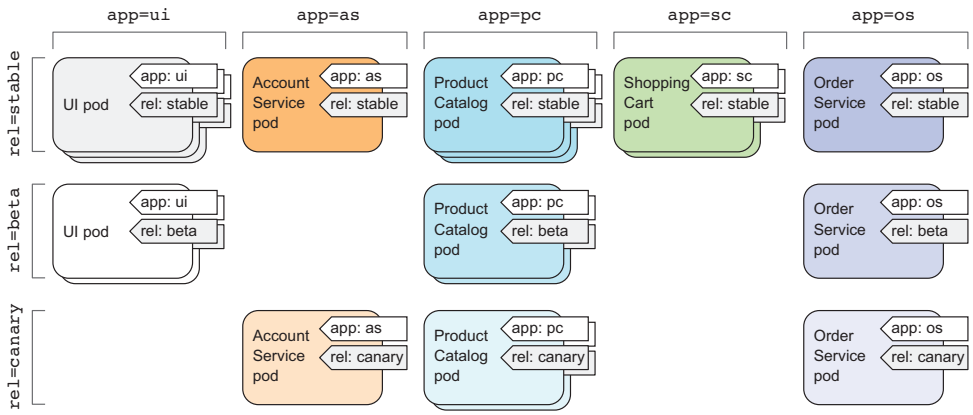


Figure 3.7 Organizing Pods in a microservices architecture with Pod labels.

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=prod` under the `metadata.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 all Pods again and include 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, add the `creation_method=manual` label to it also:

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

Now, change the `env=prod` label to `env=debug` on the *kubia-manual-v2* Pod, to demonstrate how existing labels can also 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 so-called *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 selects 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 has a different value than the one you specify

3.4.1 Listing Pods using a label selector

Let's see label selectors in action on the Pods you created so far. To list 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 `env` label with any value other than `prod` or `devel`

Turning back to the Pods in your microservice-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).

3.4.2 Using multiple conditions in a label selector

A selector can also include multiple comma-separated conditions. 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).

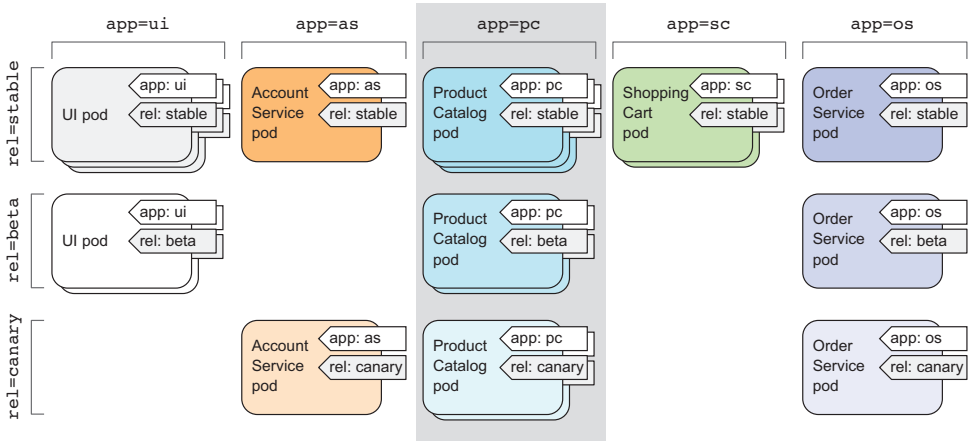


Figure 3.8 **Selecting the product catalog microservice Pods using the "app=pc" label selector.**

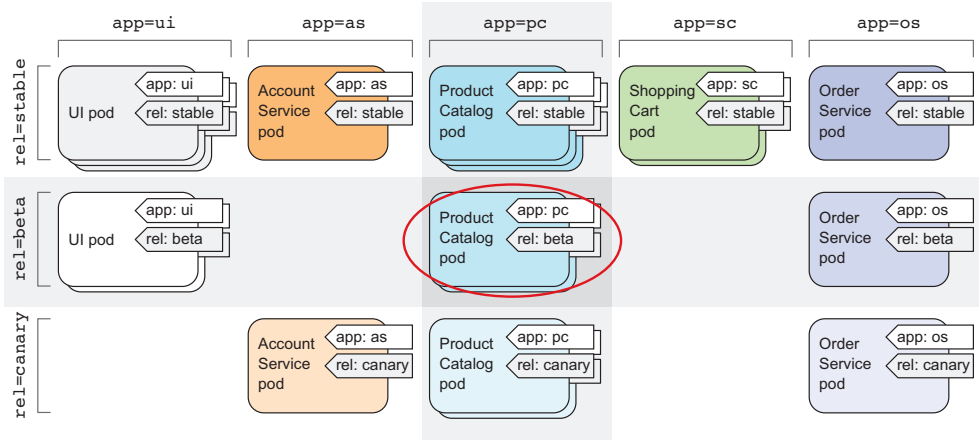


Figure 3.9 **Selecting Pods with multiple label selectors.**

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.

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, normally 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 several Pods to one group of nodes and the others 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 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 is new and contains a GPU meant to be used for so-called 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 requires a GPU to perform its work. To ask the scheduler to only choose among the nodes that provide a proper 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 forces this Pod to deploy 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 a certain criteria.

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, unlike labels, 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 manually added by users.

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 everyone involved, 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",
      "name":"kubia","uid":"3ce076d1-a025-11e6-807b-42010a84015b",
      "apiVersion":"v1","resourceVersion":"86760"}}
...
```

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).

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 instance), you'll always see all objects if you don't explicitly specify a label selector.

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 names. Instead of having all your resources in one single namespace, you can organize 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. You've already learned about nodes and namespaces, and you'll meet a few others along the way.

Let's see how to use namespaces now.

3.7.2 Discovering other namespaces and their Pods

First, 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 the objects inside 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:

```
$ 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 non-overlapping groups. If several users or groups of users are using the same Kubernetes cluster, and they each use 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 only allowing only certain users access to particular resources and even for limiting the amount of computational resources available to individual users.

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 it 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 is an 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` attribute 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 change the default namespace, you can set up the following alias: `alias kcd='kubectl config set-context $(kubectl config current-context) --namespace '`. Then, you can 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. If 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 the 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
```

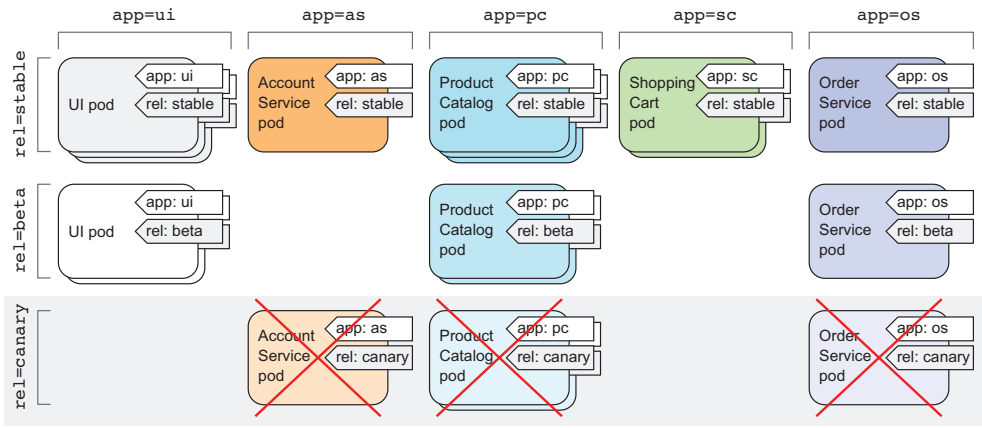


Figure 3.10 Selecting and deleting all canary Pods through the rel=canary label selector.

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 delete the whole namespace (the Pods will be deleted along with the namespace automatically). You're going to delete your custom-namespace now:

```
$ 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 replication controller 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 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 provide 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.

4 Replication and other controllers: deploying managed Pods

This chapter covers

- Keeping Pods healthy
- Running multiple instances of the same Pod
- Automatically rescheduling Pods after a node fails
- Scaling Pods horizontally
- Running system-level Pods on each cluster node
- Running batch jobs
- Scheduling jobs to run periodically or once in the future

As you've learned so far, Pods represent the basic deployable unit in Kubernetes. You know how to create, supervise, and manage them manually. But in real-world use cases, you want your deployments to stay up and running automatically and remain healthy without any manual intervention. To do this, you almost never create Pods directly. Instead, you create other types of resources, such as ReplicationControllers or Deployments, which then create and manage the actual Pods.

When you create unmanaged Pods (such as the ones you created in the previous chapter), a cluster node is selected to run the Pod and then its containers are run on that node. In this chapter, you'll learn that Kubernetes then monitors those containers and automatically restarts them if they fail. But if the whole node fails, the Pods on the node are lost and will not be replaced with new ones, unless those

Pods are managed by the previously mentioned ReplicationControllers or similar. In this chapter, you'll learn how Kubernetes checks if a container is still alive and, if it isn't, restarts it. You'll also learn how to run managed Pods—both those that run indefinitely and those that perform a single task and then stop.

4.1 Keeping Pods healthy

One of the main benefits of using Kubernetes is the ability to give it a list of containers and let it keep those containers running somewhere in the cluster. You do this by creating a Pod, letting Kubernetes pick a worker node, and running the Pod's containers on that node. But what if one of those containers dies? What if all containers of a Pod die?

As soon as a Pod is scheduled to a node, the Kubelet on that node will run its containers and, from then on, keep them running as long as the Pod exists. If the container's main process crashes, the Kubelet will restart the container. If your application has a bug that causes it to crash every once in a while, Kubernetes will restart it automatically, so even without doing anything special in the app itself, running the app in Kubernetes automatically makes it much more robust.

But sometimes apps stop working without their process crashing. For example, a Java app with a memory leak will start throwing OutOfMemoryErrors, but the JVM process will keep running. It would be great to have a way for an app to signal to Kubernetes that it's no longer functioning properly and have Kubernetes restart it.

We've said that a container that crashes is restarted automatically, so maybe you're thinking you could catch these types of errors in the app and exit the process when they occur. You can certainly do that, but it still doesn't solve all your problems.

For example, what about those situations when your app stops responding because it falls into an infinite loop or a deadlock? To make sure applications are restarted in such cases, you must check an application's health from the outside and not depend on the app doing it internally.

4.1.1 Introducing liveness probes

Kubernetes allows checking if a container is still alive through *liveness probes*. You can specify a liveness probe for each container in the Pod specification. Kubernetes will periodically execute the probe and if it fails, restart the container.

> **NOTE** Kubernetes also supports *readiness probes*, which we'll learn about in the next chapter. Be sure not to confuse the two. They're used for two different things.

Kubernetes can probe a container using one of the three types of liveness probes:

- An *HTTP GET* probe performs an HTTP GET request on the container's IP address and a port and path you specify. If the probe receives a response, and the response code doesn't represent an error (in other words, if the HTTP response code is 2xx or 3xx), the probe is considered successful. If the server

returns an error response code or if it doesn't respond at all, the probe is considered a failure and the container will be restarted as a result.

- A *TCP Socket* probe tries to open a TCP connection to the specified port on the container. If the connection is established successfully, the probe is successful. Otherwise, the container is restarted.
- An *Exec* probe executes an arbitrary command inside the container and checks the command's exit status code. If the status code is 0, the probe is successful. All other codes are considered failures.

4.1.2 Creating an HTTP-based liveness probe

Let's see how to add a liveness probe to your Node.js app. Because your sample app is a web app, it makes sense to add a liveness probe that will check whether its web server is serving requests. But because your Node.js app is unlikely to ever fail, you'll need to make the app fail artificially.

To properly demo liveness probes, you'll modify the app slightly and make it return a 500 Internal Server Error HTTP status code for each request after the fifth one—your app will handle the first five client requests properly and then return an error on every subsequent request. Thanks to the liveness probe, it should be restarted when that happens, allowing it to properly handle client requests again.

You can find the code of the new app in the book's code archive (in the folder `Chapter04/kubia-unhealthy`). I've pushed the container image to Docker Hub, so you don't need to build it yourself.

You'll create a new Pod that includes an HTTP GET liveness probe. The following listing shows the YAML for the Pod.

Listing 4.1 Adding a liveness probe to a Pod: kubia-liveness-probe.yaml

```
apiVersion: v1
kind: Pod
metadata:
 name: kubia-liveness
spec:
 containers:
 - image: luksa/kubia-unhealthy
   name: kubia
   livenessProbe:
     httpGet:
       path: /
       port: 8080
```

This is the image containing the (somewhat) broken app.

A liveness probe that will perform an HTTP GET

The path to request in the HTTP request.

The network port the probe should connect to.

The Pod descriptor defines an `httpGet` liveness probe, which tells Kubernetes to periodically perform HTTP GET requests on path `/` on port `8080` to determine if the container is still healthy. These requests start as soon as the container is run.

After five such requests (or actual client requests), your app starts returning HTTP status code 500, which Kubernetes will treat as a probe failure, and will thus restart the container.

4.1.3 Seeing a liveness probe in action

To see what the liveness probe does, try creating the Pod now. After about a minute and a half, the container will be restarted. You can see that when running `kubectl get`:

```
$ kubectl get po kubia-liveness
NAME             READY     STATUS    RESTARTS   AGE
kubia-liveness   1/1       Running   1          2m
```

The `RESTARTS` column shows that the Pod's container has been restarted once (if you wait another minute and a half, it gets restarted again, and then the cycle continues indefinitely).

Obtaining the application log of a crashed container

In the previous chapter, you learned how to print the application's log with `kubectl logs`. If your container is restarted, the `kubectl logs` command will show the log of the current container.

When you want to figure out why the previous container terminated, you'll want to see those logs instead of the current container's logs. This can be done by using the `--previous` option:

```
$ kubectl logs mypod --previous
```

You can see why the container had to be restarted by looking at what `kubectl describe` prints out, as shown in the following listing.

Listing 4.2 A Pod's description after its container is restarted

```
$ kubectl describe po kubia-liveness
Name:           kubia-liveness
...
Containers:
  kubia:
    Container ID:       docker://480986f8
    Image:              luksa/kubia-unhealthy
    Image ID:           docker://sha256:2b208508
    Port:
    State:              Running                          <- The container is currently running.
      Started:          Sun, 14 May 2017 11:41:40 +0200
    Last State:         Terminated                       <- The previous container terminated with an error and exited with code 137.
      Reason:           Error
      Exit Code:        137
      Started:          Mon, 01 Jan 0001 00:00:00 +0000
      Finished:         Sun, 14 May 2017 11:41:38 +0200
    Ready:              True
    Restart Count:      1                                <- The container has been restarted once.
    Liveness:           http-get http://:8080/ delay=0s timeout=1s
                        period=10s #success=1 #failure=3
    ...
Events:
... Killing container with id docker://95246981:pod "kubia-liveness ..."
    container "kubia" is unhealthy, it will be killed and re-created.
```

You can see that the container is currently running, but it previously terminated because of an error. The exit code was `137`, which has a special meaning—it denotes that the process was terminated by an external signal. The number `137` is a sum of two numbers: `128+x`, where `x` is the signal number sent to the process that caused it to terminate. In your case, `x` equals `9`, which is the number of the `SIGKILL` signal, meaning your process was killed forcibly.

The events listed at the bottom show why the container was killed—Kubernetes detected the container was unhealthy, so it killed and re-created it.

NOTE When a container is killed, a completely new container is created—it's not the same container being restarted again.

4.1.4 Configuring additional properties of the liveness probe

You may have noticed that `kubectl describe` also displayed additional information on the liveness probe:

```
Liveness: http-get http://:8080/ delay=0s timeout=1s period=10s #success=1
          [CA]#failure=3
```

Beside the liveness probe options you specified explicitly, you can also see additional properties, such as `delay`, `timeout`, `period`, and so on. The `delay=0s` part shows that the probing begins immediately after the container is started. The `timeout` is set to only 1 second, so the container must return a response in 1 second or the probe is counted as failed. The container is probed every 10 seconds (`period=10s`) and the container is restarted after the probe fails three consecutive times (`#failure=3`).

These additional parameters can be customized when defining the probe. For example, to set the initial delay, add the `initialDelaySeconds` property to the liveness probe as shown in the following listing.

Listing 4.3 A liveness probe with an initial delay: kubia-liveness-probe-initial-delay.yaml

```
livenessProbe:
  httpGet:
    path: /
    port: 8080
  initialDelaySeconds: 15
```

Kubernetes will wait 15 seconds before executing the first probe.

If you don't set the initial delay, the prober will start probing the container as soon as it starts, which usually leads to the probe failing, because the app isn't ready to start receiving requests. If the number of failures exceeds the failure threshold, the container is restarted before it's even able to start responding to requests properly.

TIP Always remember to set an initial delay (according to your app's startup time).

I've seen this on many occasions and users were confused why their container was being restarted. But if they'd used `kubectl describe`, they'd have seen that the con-

tainer terminated with exit code 137 or 143, telling them that the Pod was terminated externally. Additionally, the listing of the Pod's events would show that the container was killed because of a failed liveness probe. If you see this happening at Pod startup, it's because you failed to set `initialDelaySeconds` appropriately.

> **NOTE** Exit code 137 signals that the process was killed by an external signal (exit code is 128 + 9 (SIGKILL). Likewise, exit code 143 corresponds to 128 + 15 (SIGTERM).

4.1.5 Creating good and effective liveness probes

For Pods running in production, you should always define a liveness probe. Without one, Kubernetes has no way of knowing whether your app is still alive or not. As long as the process is still running, Kubernetes will consider the container as being healthy.

What a liveness probe should check

Your simplistic liveness probe simply checks if the server is responding. While this may seem overly simple, even a liveness probe like this does wonders, because it causes the container to be restarted if the web server running within the container stops responding to HTTP requests. Compared to having no liveness probe, this is a major improvement, and may be sufficient in most cases.

But for a better liveness check, you'd configure the probe to perform requests on a specific URL path (`/health`, for example) and have the app perform an internal status check of all the vital components running inside the app to ensure none of them has died or is unresponsive.

> **TIP** Make sure the `/health` HTTP endpoint doesn't require authentication; otherwise the probe will always fail, causing your container to be restarted indefinitely.

Be sure to check only the internals of the app and nothing influenced by an external factor. For example, a frontend web server's liveness probe shouldn't return a failure when the server can't connect to the backend database. If the underlying cause is in the database itself, restarting the web server container will not fix the problem. Because the liveness probe will fail again, you'll end up with the container restarting repeatedly until the database becomes accessible again.

Keeping probes light

Liveness probes shouldn't use too many computational resources and shouldn't take too long to complete. By default, the probes are executed relatively often and are only allowed one second to complete. Having a probe that does heavy lifting can slow down your container considerably. Later in the book, you'll also learn about how to limit CPU time available to a container. The probe's CPU time is counted in the container's CPU time quota, so having a heavyweight liveness probe will reduce the CPU time available to the main application processes.

TIP If you're running a Java app in your container, be sure to use an HTTP GET liveness probe instead of an Exec probe, where you spin up a whole new JVM to get the liveness information. The same goes for any JVM-based or similar applications, whose start-up procedure requires many more computational resources.

DON'T BOTHER IMPLEMENTING RETRY LOOPS IN YOUR PROBES

You've already seen that the failure threshold for the probe is configurable and usually the probe must fail multiple times before the container is killed. But even if you set the failure threshold to 1, Kubernetes will retry the probe a few times before considering it a single failed attempt. Therefore, implementing your own retry loop into the probe is wasted effort.

LIVENESS PROBE WRAP-UP

Okay, you now understand that Kubernetes keeps your containers running by restarting them if they crash or if their liveness probes fail. This job is performed by the Kubelet on the Pod's node—the Kubernetes control plane components running on the master(s) have no part in this process.

But if the node itself crashes, it's the control plane that must create replacement Pods for all the Pods that went down with the node. It doesn't do that for Pods that were created directly. These Pods aren't managed by anything except by the Kubelet, but because the Kubelet runs on each node, when the node fails, the Kubelet is also gone.

To make sure your app is restarted on another node, you need to have the Pod managed by a ReplicationController or similar mechanisms, which we'll discuss in the rest of this chapter.

4.2 Introducing ReplicationControllers

A ReplicationController is a Kubernetes resource that ensures a Pod is always up and running. If the Pod disappears for any reason, such as in the event of a node disappearing from the cluster or because the Pod was evicted from the node, the ReplicationController notices the missing Pod and creates a replacement Pod.

Figure 4.1 shows what happens when a node goes down and takes two Pods with it. Pod A was created directly and is therefore an unmanaged Pod, while Pod B is managed by a ReplicationController. After the node fails, the ReplicationController creates a new Pod (Pod B2) to replace the missing Pod B, whereas Pod A is lost completely—nothing will ever recreate it.

The ReplicationController in the figure manages only a single Pod, but replication controllers, in general, are meant to create and manage multiple copies (replicas) of a Pod, and that's where their name comes from.

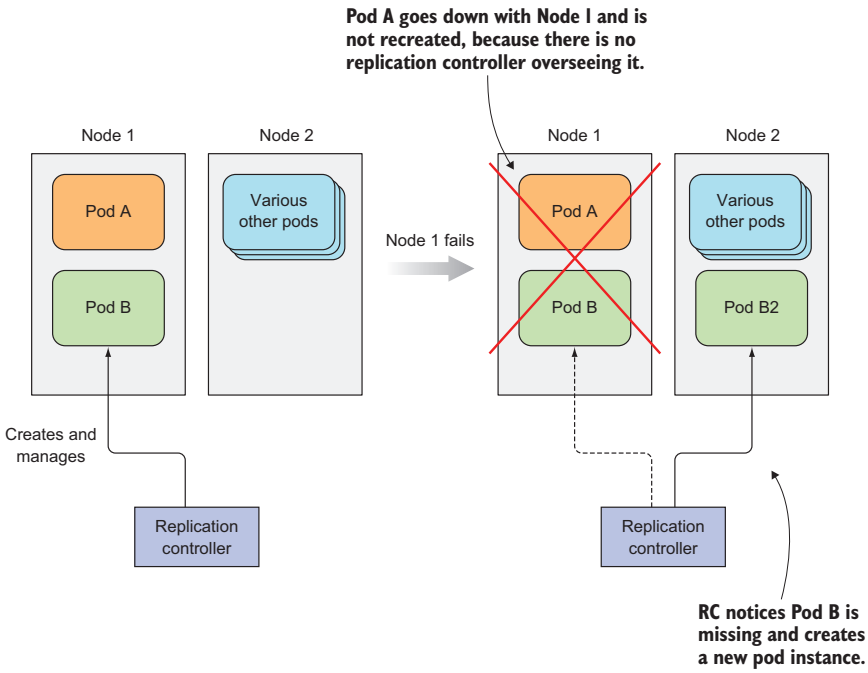


Figure 4.1 When a node fails, only Pods backed by a ReplicationController are recreated.

4.2.1 The operation of a replication controller

A replication controller constantly monitors the list of running Pods and makes sure the actual number of Pods of a "type" always match the desired number. If too few Pods are running, it creates new Pods based on a Pod template that's configured on the replication controller. If too many Pods are running, it removes the excess Pods.

You might be wondering how there can be more than the desired number of running Pods. This can happen for a few reasons:

- Someone creates a Pod of the same type manually.
- Someone changes an existing Pod's "type."
- Someone decreases the desired number of Pods, and so on.

I've used the term Pod "type" a few times. But no such thing exists. Replication controllers don't operate on Pod types, but on sets of Pods that match a certain label selector (you learned about them in the previous chapter).

INTRODUCING THE CONTROLLER'S RECONCILIATION LOOP

A replication controller's job is to make sure that an exact number of Pods always matches its label selector. If it doesn't, the replication controller takes the appropriate action to reconcile the actual with the desired number. The operation of a replication controller is shown in figure 4.2.

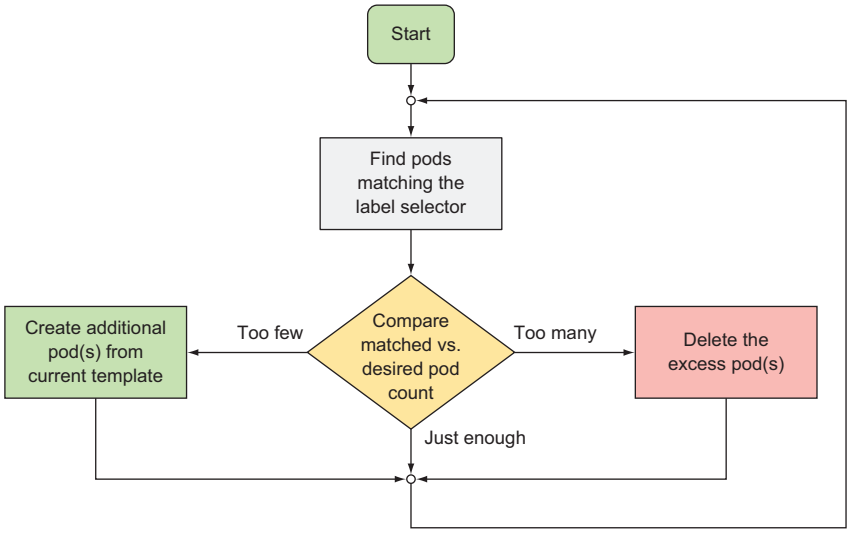


Figure 4.2 **A replication controller's reconciliation loop.**

UNDERSTANDING THE THREE PARTS OF A REPLICATION CONTROLLER

A replication controller has three essential parts (also shown in figure 4.3):

- A *label selector*, which determines what Pods are in the replication controller's scope
- A *replica count*, which specifies the desired number of Pods that should be running
- A *Pod template*, which is used when creating new Pods

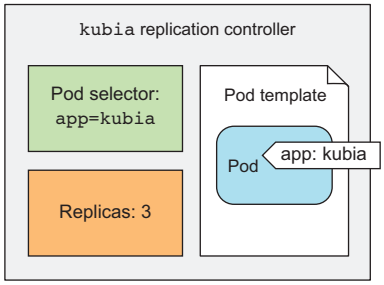


Figure 4.3 **The three key parts of a replication controller (Pod selector, replica count, and Pod template).**

A replication controller's replica count, the label selector, and even the Pod template can all be modified at any time, but only changes to the replica count affect existing Pods.

UNDERSTANDING THE EFFECT OF CHANGING THE CONTROLLER'S LABEL SELECTOR OR POD TEMPLATE

Changes to the label selector and the Pod template have no effect on existing Pods. Changing the label selector makes the existing Pods fall out of the scope of the repli-

cation controller, so the controller stops caring about them. Replication controllers also don't care about the actual "contents" of its Pods (the container images, environment variables, and other things) once a Pod is created. The template therefore only affects new Pods created by this replication controller. The template is used as a cookie cutter to stamp out new Pods.

UNDERSTANDING THE BENEFITS OF USING A REPLICATION CONTROLLER

Like many things in Kubernetes, a replication controller, although an incredibly simple concept, provides or enables the following powerful features:

- It makes sure a Pod (or multiple Pod instances) is always running by starting a new Pod when an existing Pod goes missing.
- When a cluster node fails, it creates replacement Pods for all the Pods that were running on the failed node (those that were under the replication controller's control).
- It enables easy horizontal scaling of Pods—both manual and automatic (see horizontal Pod auto-scaling in chapter 15).

NOTE A Pod instance is never relocated to another node. Instead, the replication controller creates a completely new Pod instance that has no relation to the instance it's replacing.

4.2.2 Creating a replication controller

Let's look at how to create a replication controller and then see how it keeps your Pods running. Like Pods and other Kubernetes resources, you create a replication controller by posting a JSON or YAML descriptor to the Kubernetes API server.

You're going to create a YAML file called kubia-rc.yaml for your replication controller, as shown in the following listing.

Listing 4.4 A YAML definition of a ReplicationController: kubia-rc.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: kubia
spec:
  replicas: 3
  selector:
    app: kubia
  template:
    metadata:
      labels:
        app: kubia
    spec:
      containers:
      - name: kubia
        image: luksa/kubia
        ports:
        - containerPort: 8080
```

This manifest defines a ReplicationController (RC).

The name of this replication controller.

The desired number of Pod instances.

The Pod selector determining what Pods the RC is operating on.

The Pod template for creating new Pods.

When you post it to the API server, Kubernetes will create a new ReplicationController named `kubia`, which will make sure three Pod instances always match the label selector `app=kubia`. When there aren't enough Pods, new Pods will be created from the provided Pod template. The contents of the template are almost identical to the Pod definition you created in the previous chapter.

The Pod labels in the template must obviously match the label selector of the ReplicationController; otherwise the controller would create new Pods indefinitely, because spinning up a new Pod wouldn't bring the actual replica count any closer to the desired number of replicas. To prevent such scenarios, the API server verifies the ReplicationController definition and will not accept it if it's misconfigured.

Not specifying the selector at all is also an option. In that case, it will be configured automatically from the labels in the Pod template.

> **TIP** Don't specify a Pod selector when defining a ReplicationController. Let Kubernetes extract it from the Pod template. This will keep your YAML shorter and simpler.

To create the ReplicationController, use the `kubectl create` command, which you already know:

```
$ kubectl create -f kubia-rc.yaml
replicationcontroller "kubia" created
```

As soon as the ReplicationController is created, it goes to work. Let's see what it does.

4.2.3 Seeing the ReplicationController in action

Because no Pods exist with the `app=kubia` label, the ReplicationController should spin up three new Pods from the Pod template. List the Pods to see if the ReplicationController has done what it's supposed to:

```
$ kubectl get pods
NAME          READY     STATUS              RESTARTS   AGE
kubia-53thy   0/1       ContainerCreating   0          2s
kubia-k0xz6   0/1       ContainerCreating   0          2s
kubia-q3vkg   0/1       ContainerCreating   0          2s
```

Indeed, it has! You wanted three Pods, and it created three Pods. It's now managing those three Pods. Next you'll mess with them a little to see how the ReplicationController responds.

SEEING THE REPLICATIONCONTROLLER RESPOND TO A DELETED POD

First, you'll delete one of the Pods manually to see how the ReplicationController spins up a new one immediately, bringing the number of matching Pods back to three:

```
$ kubectl delete pod kubia-53thy
pod "kubia-53thy" deleted
```

Listing the Pods again shows four of them, because the one you deleted is terminating, and a new Pod has already been created:

```
$ kubectl get pods
NAME          READY     STATUS              RESTARTS   AGE
kubia-53thy   1/1       Terminating         0          3m
kubia-oini2   0/1       ContainerCreating   0          2s
kubia-k0xz6   1/1       Running             0          3m
kubia-q3vkg   1/1       Running             0          3m
```

The ReplicationController has done its job again. It's a nice little helper, isn't it?

GETTING INFORMATION ABOUT A REPLICATION CONTROLLER

Now, let's see what information the `kubectl get` command shows for replication controllers:

```
$ kubectl get rc
NAME      DESIRED   CURRENT   READY     AGE
kubia     3         3         2         3m
```

NOTE We're using `rc` as a shorthand for `replicationcontroller`.

You see three columns showing the desired number of Pods, the actual number of Pods, and how many of them are ready (you'll learn what that means in the next chapter, when we talk about readiness probes).

You can see additional information about your ReplicationController with the `kubectl describe` command, as shown in the following listing.

Listing 4.5 Displaying details of a ReplicationController with kubectl describe

```
$ kubectl describe rc kubia
Name:           kubia
Namespace:      default
Selector:       app=kubia
Labels:         app=kubia
Annotations:    <none>
Replicas:       3 current / 3 desired
Pods Status:    4 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:       app=kubia
  Containers:   ...
  Volumes:      <none>
Events:
From                    Type      Reason            Message
----                    -------   ------            -------
replication-controller  Normal    SuccessfulCreate  Created pod: kubia-53thy
replication-controller  Normal    SuccessfulCreate  Created pod: kubia-k0xz6
replication-controller  Normal    SuccessfulCreate  Created pod: kubia-q3vkg
replication-controller  Normal    SuccessfulCreate  Created pod: kubia-oini2
```

The actual vs. the desired number of Pod instances.

Number of Pod instances per Pod status.

The events related to this ReplicationController.

The current number of replicas matches the desired number, because the controller has already created a new Pod. It shows four running Pods because a Pod that's

terminating is still considered running, although it isn't counted in the current replica count.

The list of events at the bottom shows the actions taken by the ReplicationController—it has created four Pods so far.

UNDERSTANDING EXACTLY WHAT CAUSED THE CONTROLLER TO CREATE A NEW POD

The controller is responding to the deletion of a Pod by creating a new replacement Pod (see figure 4.4). Well, technically, it isn't responding to the deletion itself, but the resulting state—the inadequate number of Pods.

While a ReplicationController is immediately notified about a Pod being deleted (the API server allows clients to watch for changes to resources and resource lists), that's not what causes it to create a replacement Pod. The notification triggers the controller to check the actual number of Pods and react to match it to the desired number.

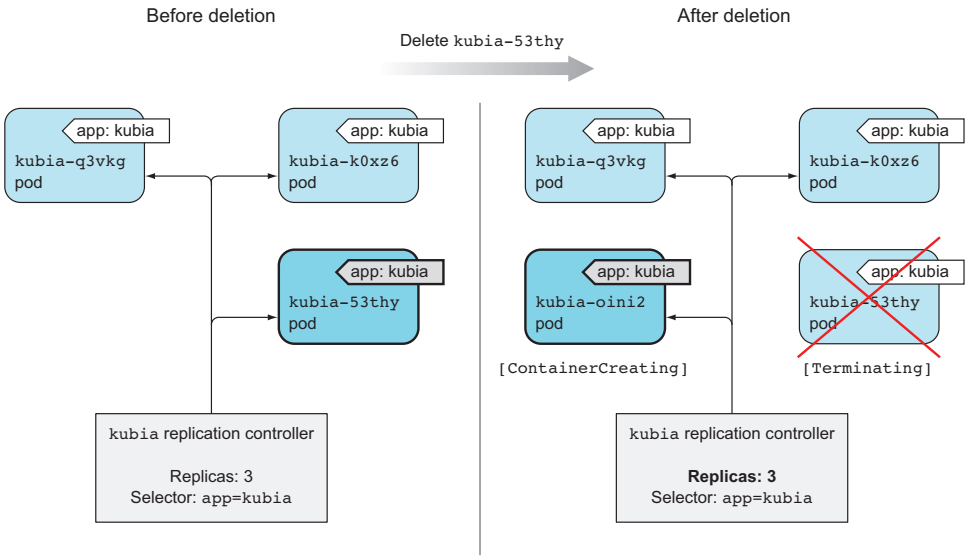


Figure 4.4 If a Pod disappears, the ReplicationController detects too few Pods and creates a new replacement Pod.

RESPONDING TO A NODE FAILURE

Seeing the ReplicationController respond to the manual deletion of a Pod isn't too interesting, so let's look at a better example. If you're using Google Container Engine to run these examples, you have a three-node Kubernetes cluster. You're going to disconnect one of the nodes from the network to simulate a node failure.

> **NOTE** If you're using Minikube, you can't do this exercise, because you only have one node that acts both as a master and a worker node.

If a node fails in the non-Kubernetes world, the ops team would need to migrate the applications running on that node to other machines manually. Kubernetes, on the other hand, does that automatically. Soon after the ReplicationController detects several of its Pods are down, it will spin up new Pods to replace them.

Let's see this in action. You need to ssh into one of the nodes with the `gcloud compute ssh` command and then shut down its network interface with `sudo ifconfig eth0 down`, as shown in the following listing.

NOTE Choose a node that one of your Pods is running on by listing Pods with the `-o wide` option.

Listing 4.6 Simulating a node failure by shutting down its network interface

```
$ gcloud compute ssh gke-kubia-default-pool-b46381f1-zwko
Enter passphrase for key '/home/luksa/.ssh/google_compute_engine':

Welcome to Kubernetes v1.6.4!
...

luksa@gke-kubia-default-pool-b46381f1-zwko ~ $ sudo ifconfig eth0 down
```

When you shut down the network interface, the ssh session will stop responding, so you need to open up another terminal or hard-exit from the ssh session. In the new terminal you can list the nodes to see if Kubernetes has detected that the node is down. This takes a minute or so. Then, the node's status is shown as `NotReady`:

```
$ kubectl get node
NAME                                     STATUS     AGE
gke-kubia-default-pool-b46381f1-opc5     Ready      5h
gke-kubia-default-pool-b46381f1-s8gj     Ready      5h
gke-kubia-default-pool-b46381f1-zwko     NotReady
```

Node isn't ready, because it's disconnected from the network.

If you list the Pods now, you'll still see the same three Pods as before, because Kubernetes waits a while before rescheduling Pods (in case the node is unreachable because of a temporary network glitch or because the Kubelet is restarting). If the node stays unreachable for several minutes, the status of the Pods that were scheduled to that node changes to `Unknown`. At that point, the ReplicationController will immediately spin up a new Pod. You can see this by listing the Pods again:

```
$ kubectl get pods
NAME          READY     STATUS    RESTARTS   AGE
kubia-oini2   1/1       Running   0          10m
kubia-k0xz6   1/1       Running   0          10m
kubia-q3vkg   1/1       Unknown   0          10m
kubia-dmdck   1/1       Running   0          5s
```

This Pod's status is unknown, because its node is unreachable.

This Pod was created five seconds ago.

Looking at the age of the Pods, you see that the `kubia-dmdck` Pod is new. You again have three Pod instances running, which means the ReplicationController has again done its job of bringing the actual state of the system to the desired state.

The same thing happens if a node fails (either breaks down or becomes unreachable). No immediate human intervention is necessary. The system heals itself automatically.

To bring the node back, you need to reset it with the following command:

```
$ gcloud compute instances reset gke-kubia-default-pool-b46381f1-zwko
```

When the node boots up again, its status should return to `Ready`, and the Pod whose status was `Unknown` will be deleted.

4.2.4 Moving Pods in and out of the scope of a replication controller

Pods created by a ReplicationController aren't tied to the replication controller in any way. At any moment, a ReplicationController manages Pods that match its label selector. By changing a Pod's labels, it can be removed from or added to the scope of a ReplicationController. It can even be moved from one ReplicationController to another.

> **TIP** Although a Pod isn't tied to a ReplicationController, the Pod does include a `kubernetes.io/created-by` annotation, which you can use to easily find which ReplicationController a Pod belongs to.

If you make a Pod's labels no longer match a ReplicationController's label selector, the Pod becomes like any other manually created Pod. It's no longer managed by anything. If the node running the Pod fails, the Pod is obviously not rescheduled. But keep in mind that when you change the Pod's labels, the replication controller notices one Pod is missing and spins up a new Pod to replace it.

Let's try this with your Pods. Because your ReplicationController manages Pods that have the `app=kubia` label, to move the Pod out of the ReplicationController's scope, you need to either remove this label or change its value. Adding another label will have no effect, because the ReplicationController only requires the Pod to include all the labels in the label selector and doesn't care if the Pod has any additional labels.

ADDING LABELS TO PODS MANAGED BY A REPLICATIONCONTROLLER

Let's confirm that a ReplicationController doesn't care about adding additional labels to its managed Pods:

```
$ kubectl label pod kubia-dmdck type=special
pod "kubia-dmdck" labeled

$ kubectl get pods --show-labels
NAME          READY   STATUS    RESTARTS   AGE   LABELS
kubia-oini2   1/1     Running   0          11m   app=kubia
kubia-k0xz6   1/1     Running   0          11m   app=kubia
kubia-dmdck   1/1     Running   0          1m    app=kubia,type=special
```

You've added the `type=special` label to the Pod. Listing all Pods still shows the same three Pods as before, because no change occurred as far as the ReplicationController is concerned.

Changing the labels of a managed Pod

Now, you'll change the `app=kubia` label to something else. This will make the Pod no longer match the ReplicationController's label selector, leaving it to only match two Pods. The ReplicationController should therefore start a new Pod to bring the number back to three:

```
$ kubectl label pod kubia-dmdck app=foo --overwrite
pod "kubia-dmdck" labeled
```

The `--overwrite` argument is necessary; otherwise `kubectl` will only print out a warning and won't change the label, to prevent you from inadvertently changing a label's value when your intent is to add a new one.

Listing all the Pods again should now show four Pods:

```
$ kubectl get pods -L app
NAME          READY   STATUS              RESTARTS   AGE   APP
kubia-2qneh   0/1     ContainerCreating   0          2s    kubia
kubia-oini2   1/1     Running             0          20m   kubia
kubia-k0xz6   1/1     Running             0          20m   kubia
kubia-dmdck   1/1     Running             0          10m   foo
```

Newly created Pod to replace the Pod you removed from the scope of the ReplicationController.

Pod no longer managed by the ReplicationController.

NOTE You're using the `-L app` option to only show the `app` label.

There, you now have four Pods altogether: One that isn't managed by your ReplicationController and three that are. Among them is the newly created Pod.

Figure 4.5 illustrates what happened when you changed the Pod's labels so they no longer matched the ReplicationController's Pod selector. You can see your three Pods and your ReplicationController. After you change the Pod's label from `app=kubia` to `app=foo`, the ReplicationController no longer cares about the Pod. Because the controller's replica count is set to 3 and only two Pods match the label selector, the ReplicationController spins up Pod *kubia-2qneh* to bring the number back up to three. Pod *kubia-dmdck* is now completely independent and will keep running until you delete it manually (you can do that now, because you don't need it anymore).

Removing Pods from controllers in practice

Removing a Pod from the scope of the replication controller comes in handy when you want to perform actions on a specific Pod. For example, you might have a bug that causes your Pod to start behaving badly after a specific amount of time or a specific event. If you know a Pod is malfunctioning, you can take it out of the replication controller's scope, let the controller replace it with a new one, and then debug or play with the Pod in any way you want. Once you're done, you delete the Pod.

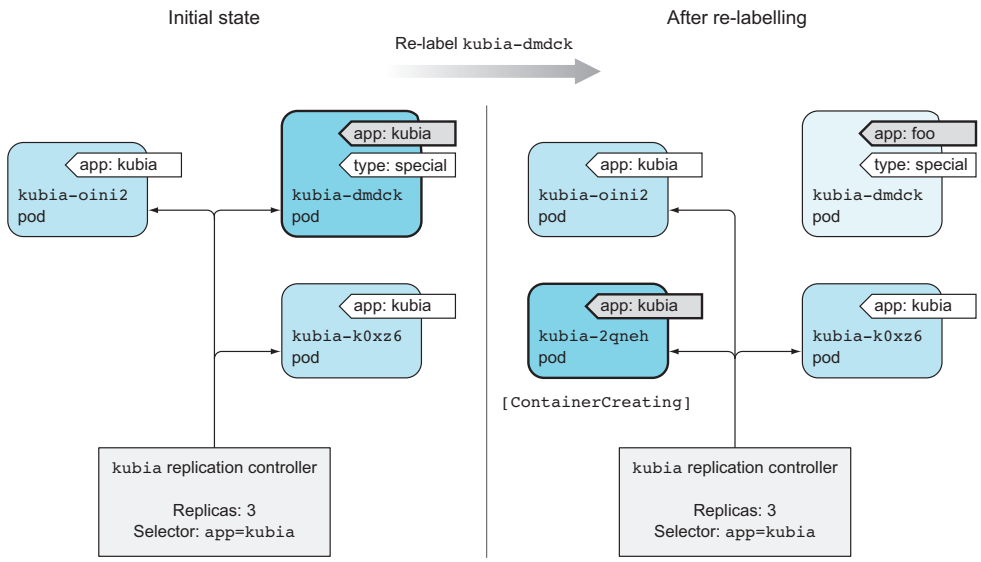


Figure 4.5 Removing a Pod from the scope of a ReplicationController by changing its labels.

CHANGING THE REPLICATIONCONTROLLER'S LABEL SELECTOR

As an exercise to see if you fully understand replication controllers, what do you think would happen if instead of changing the labels of a Pod, you modified the ReplicationController's label selector?

If your answer is that it would make all the Pods fall out of the scope of the replication controller, which would result in it creating three new Pods, you're absolutely right. And it shows that you understand how replication controllers work.

Would you ever need to do that in the real world? I can't think of a good reason why you would, but Kubernetes does allow this and the result is what you'd expect. You're probably never going to change the label selector, but you will want to change the controller's Pod template. Let's take a look at that.

4.2.5 Changing the Pod template

A replication controller's Pod template can be modified at any time. Changing the Pod template is like replacing a cookie cutter with another one. It will only affect the cookies you cut out afterward and will have no effect on the ones you've already cut (see figure 4.6). To modify the old Pods, you'd need to delete them and let the replication controller replace them with new ones based on the new template.

As an exercise, you can try editing the ReplicationController and adding a label to the Pod template. You can edit the ReplicationController with the following command:

```
$ kubectl edit rc kubia
```

This will open the ReplicationController's YAML definition in your default text editor. Find the Pod template section and add an additional label to the metadata. After you

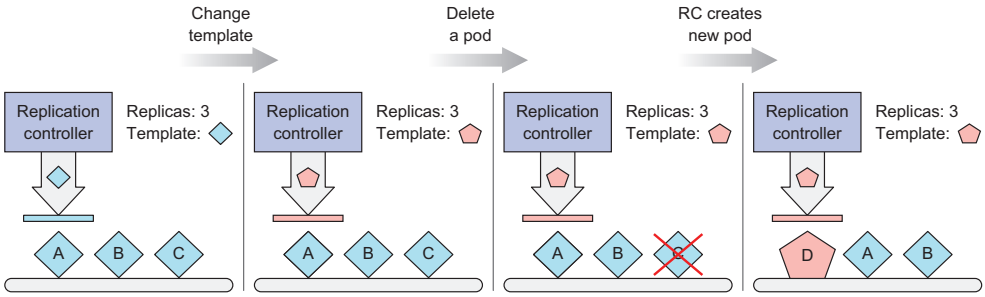


Figure 4.6 Changing a replication controller's Pod template only affects Pods created afterward and has no effect on existing Pods.

save your changes and exit the editor, `kubectl` will update the ReplicationController and print the following message:

```
replicationcontroller "kubia" edited
```

You can now list Pods and their labels again and confirm that they haven't changed. But if you delete the Pods and wait for their replacements to be created, you'll see the new label.

> **Configuring kubectl edit to use a different text editor**
>
> You can tell `kubectl` to use a text editor of your choice by setting the `KUBE_EDITOR` environment variable. For example, if you'd like to use *nano* for editing Kubernetes resources, execute the following command (or put it into your `~/.bashrc` or an equivalent file):
>
> ```
> export KUBE_EDITOR="/usr/bin/nano"
> ```
>
> If the `KUBE_EDITOR` environment variable isn't set, `kubectl edit` falls back to using the default editor, usually configured through the EDITOR environment variable.

Editing a ReplicationController like this to change the container image in the Pod template, deleting the existing Pods, and letting them be replaced with new ones from the new template could be used for upgrading Pods, but you have a much better way to do this. We'll talk about it in chapter 9.

4.2.6 Horizontally scaling Pods

You've seen how replication controllers make sure a specific number of Pod instances is always running. Because it's incredibly simple to change the desired number of replicas, this also means scaling Pods horizontally is trivial.

Scaling the number of Pods up or down is as easy as changing a single attribute of a replication controller. After the change, the replication controller will either see too many Pods exist (when scaling down) and delete part of them, or see too few of them (when scaling up) and create additional Pods.

SCALING UP A REPLICATIONCONTROLLER

Your ReplicationController has been keeping three instances of your Pod running. You're going to scale that number up to 10 now. As you may remember, you've already scaled a ReplicationController in chapter 2. You could use the same command as before:

```
$ kubectl scale rc kubia --replicas=10
```

But you'll do it differently this time.

SCALING A REPLICATIONCONTROLLER BY EDITING ITS DEFINITION

Instead of using the `kubectl scale` command, you're going to scale it in a declarative way by editing the ReplicationController's definition:

```
$ kubectl edit rc kubia
```

When the text editor opens, find the `spec.replicas` field and change its value to `10`, as shown in the following listing.

Listing 4.7 Editing the RC in a text editor by running kubectl edit

```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving
# this file will be reopened with the relevant failures.
apiVersion: v1
kind: ReplicationController
metadata:
  ...
spec:
  replicas: 3
  selector:
    app: kubia
  ...
```

Change the number 3 to number 10 in this line.

When you save the file and close the editor, the ReplicationController is updated and it immediately scales the number of Pods to 10:

```
$ kubectl get rc
NAME      DESIRED   CURRENT   READY     AGE
kubia     10        10        4         21m
```

There you go. If the `kubectl scale` command makes it look as though you're telling Kubernetes exactly what to do, it's now much clearer that you're making a declarative change to the desired state of the ReplicationController and not telling Kubernetes to do something.

SCALING DOWN WITH THE KUBECTL SCALE COMMAND

Now scale back down to 3. You can use the `kubectl scale` command:

```
$ kubectl scale rc kubia --replicas=3
```

All this command does is modify the `spec.replicas` field of the ReplicationController's definition—like when changing it through *kubectl edit*.

UNDERSTANDING THE DECLARATIVE APPROACH TO SCALING

Horizontally scaling Pods in Kubernetes is a matter of stating your desire: "I want to have X number of instances running." You're not telling Kubernetes what or how to do it. You're just specifying the desired state.

This declarative approach makes interacting with a Kubernetes cluster easy. Imagine if you had to manually determine the current number of running instances and then explicitly tell Kubernetes how many additional instances to run. That's more work and is much more error-prone. Changing a simple number is much easier, and in chapter 15, you'll learn that even that can be done by Kubernetes itself if you enable horizontal Pod auto-scaling.

4.2.7 Deleting a replication controller

When you delete a replication controller through `kubectl delete`, the Pods are also deleted. But because Pods created by a replication controller aren't an integral part of the replication controller, and are only managed by it, you can delete only the replication controller and leave the Pods running, as shown in figure 4.7.

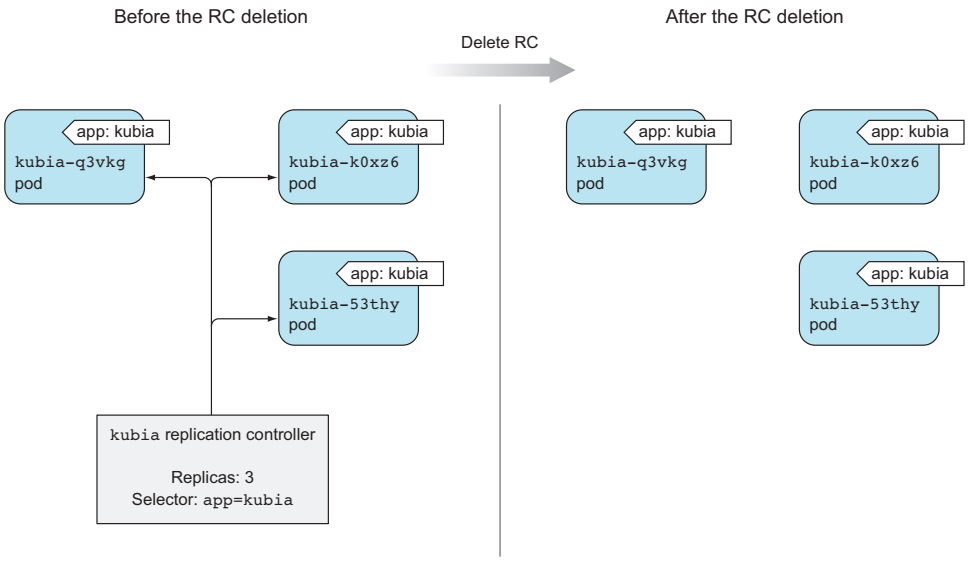


Figure 4.7 Deleting a replication controller with –cascade=false leaves Pods unmanaged.

This may be useful when you initially have a set of Pods managed by a ReplicationController, and then decide to replace the ReplicationController with a ReplicaSet, for example. (You'll learn about them next.) You can do this without affecting the Pods and keep them running without interruption while you replace the ReplicationController that manages them.

When deleting a replication controller with `kubectl delete`, you can keep its Pods running by passing the `--cascade=false` option to the command. Try that now:

```
$ kubectl delete rc kubia --cascade=false
replicationcontroller "kubia" deleted
```

You've deleted the replication controller so the Pods are on their own. They have nothing to manage them. But you can always create a new replication controller with the proper label selector and make them managed again.

4.3 Using ReplicaSets instead of replication controllers

Initially, replication controllers were the only Kubernetes component for replicating Pods and rescheduling them when nodes failed. Later, a similar resource called a ReplicaSet was introduced. It's a new generation of replication controllers and replaces them completely (ReplicationControllers will eventually be deprecated).

You could have started this chapter by creating a ReplicaSet instead of a ReplicationController, but I felt it would be a good idea to start with what was initially available in Kubernetes. Plus, you'll still see replication controllers used in the wild, so it's good for you to know about them. That said, you should always create ReplicaSets instead of ReplicationControllers from now on. They're almost identical, so you shouldn't have any trouble using them instead.

You're not going to create them directly, but instead have them created automatically when you create the higher-level Deployment resource, which you'll learn about in chapter 9. In any case, you should understand ReplicaSets, so let's see how they differ from replication controllers.

4.3.1 Comparing a ReplicaSet to a ReplicationController

A ReplicaSet behaves exactly like a ReplicationController, but it has more expressive Pod selectors. Whereas a ReplicationController's label selector only allows matching Pods that include a certain label, a ReplicaSet's selector also allows matching Pods that lack a certain label or Pods that include a certain label key, regardless of its value.

Also, for example, a single ReplicationController can't match Pods with the label `env=production` and those with the label `env=devel` at the same time. It can only match either Pods with the `env=production` label or Pods with the `env=devel` label. But a single ReplicaSet can match both sets of Pods and treat them as a single group.

Similarly, a ReplicationController can't match Pods based on the presence of a label key, regardless of its value, while a ReplicaSet can. So a ReplicaSet can, for example, match all Pods that include a label with the key `env`, whatever its actual value is (equivalent to `env=*`).

4.3.2 Defining a ReplicaSet

You'e going to create a ReplicaSet now to see how the orphaned Pods that were created by your ReplicationController and then abandoned earlier can now be adopted

by a ReplicaSet. First, you'll rewrite your ReplicationController into a ReplicaSet by creating a new file called kubia-replicaset.yaml with the contents in the following listing.

Listing 4.8 A YAML definition of a ReplicaSet: kubia-replicaset.yaml

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
 name: kubia
spec:
 replicas: 3
 selector:
   matchLabels:
     app: kubia
 template:
   metadata:
     labels:
       app: kubia
   spec:
     containers:
     - name: kubia
       image: luksa/kubia
```

ReplicaSets aren't part of the v1 API, but belong to the extensions API group and version v1beta.

You're using the simpler matchLabels selector here, which is similar to a ReplicationController's selector.

The template is the same as in the ReplicationController.

The first thing to note is that ReplicaSets aren't part of the v1 API, so you need to ensure you specify the proper `apiVersion` when creating the resource. You're creating a resource of type `ReplicaSet` which has much the same contents as the ReplicationController you created earlier.

The only difference is in the selector. Instead of listing labels the Pods need to have directly under the `selector` property, you're specifying them under `selector.matchLabels`. This is the simpler (and less expressive) way of defining label selectors in a ReplicaSet. Later, you'll look at the more expressive option, as well.

About the API version attribute

This is your first opportunity to see that the `apiVersion` property specifies two things:

- The API group (which is `extensions` in this case)
- The actual API version (`v1beta`)

You'll see throughout the book that certain Kubernetes resources are in what's called the *core* API group, which doesn't need to be specified in the `apiVersion` field (you just specify the version—for example, you've been using `apiVersion: v1` when defining Pod resources). Other resources, which were introduced in later Kubernetes versions, are categorized into several API groups. Look at the inside of the book's covers to see all resources and their respective API groups.

Because you still have three Pods matching the `app=kubia` selector running from earlier, creating this ReplicaSet will not cause any new Pods to be created. The ReplicaSet will take those existing three Pods under its wing.

4.3.3 Creating and examining a ReplicaSet

Create the ReplicaSet from the YAML file with the `kubectl create` command. After that, you can examine the ReplicaSet with `kubectl get` and `kubectl describe`:

```
$ kubectl get rs
NAME      DESIRED   CURRENT   READY     AGE
kubia     3         3         3         3s
```

TIP Use rs shorthand, which stands for replicaset.

```
$ kubectl describe rs
Name:           kubia
Namespace:      default
Selector:       app=kubia
Labels:         app=kubia
Annotations:    <none>
Replicas:       3 current / 3 desired
Pods Status:    3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:       app=kubia
  Containers:   ...
  Volumes:      <none>
Events:         <none>
```

As you can see, the ReplicaSet isn't any different from a ReplicationController. It's showing it has three replicas matching the selector. If you list all the Pods, you'll see they're still the same three Pods you had before. The ReplicaSet didn't create any new ones.

4.3.4 Using the ReplicaSet's more expressive label selectors

The main improvements of ReplicaSets over ReplicationControllers are their more expressive label selectors. You intentionally used the simpler `matchLabels` selector in the first ReplicaSet example to show that ReplicaSets are no different from ReplicationControllers. Now, you'll rewrite the selector to use the more powerful `matchExpressions` property, as shown in the following listing.

Listing 4.9 A matchExpressions selector: kubia-replicaset-matchexpressions.yaml

```
selector:
  matchExpressions:
    - key: app
      operator: In
      values:
        - kubia
```

This selector requires the Pod to contain a label with the "app" key.

The label's value must be "kubia".

NOTE Only the selector is shown. You'll find the whole ReplicaSet definition in the book's code archive.

You can add additional expressions to the selector. As in the example, each expression must contain a `key`, an `operator`, and possibly (depending on the operator) a list of `values`. You'll see four valid operators:

- `In`–Label's value must match one of the specified `values`.
- `NotIn`–Label's value must not match any of the specified `values`.
- `Exists`–Pod must include a label with the specified key (the label's value isn't important). The `values` property in the `matchExpressions` rule must not be specified.
- `DoesNotExist`–Pod must not include a label with the specified key. The `values` property must not be specified.

If you specify multiple expressions, all those expressions must evaluate to true for the selector to match a Pod. If you specify both `matchLabels` and `matchExpressions`, all the labels must match and all the expressions must evaluate to true for the Pod to match the selector.

4.3.5 Wrapping up ReplicaSets

This was a quick introduction to ReplicaSets as an alternative to replication controllers. Remember, always use them instead of replication controllers, but you may still find replication controllers in other people's deployments.

Now, delete the ReplicaSet to clean up your cluster a little. You can delete the ReplicaSet the same way you'd delete a replication controller:

```
$ kubectl delete rs kubia
replicaset "kubia" deleted
```

Deleting the ReplicaSet should delete all the Pods. List the Pods to confirm that's the case.

4.4 Running exactly one Pod on each node with DaemonSets

Both ReplicationControllers and ReplicaSets are used for running a specific number of Pods deployed anywhere in the Kubernetes cluster. But certain cases exist when you want a Pod to run on each and every node in the cluster (and each node needs to run exactly one instance of the Pod, as shown in figure 4.8).

You'll find infrastructure-related Pods that usually perform system-level operations. For example, you'd want to run a log collector and a resource monitor on each node. Another good example is Kubernetes' own kube-proxy process, which needs to run on every node to make services work.

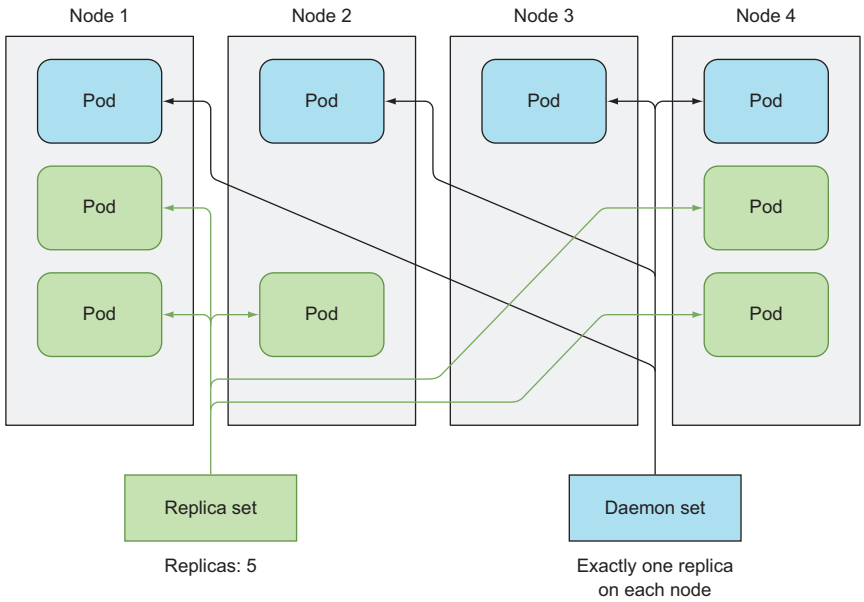


Figure 4.8 DaemonSets run only a single Pod on each node, whereas ReplicaSets scatter Pods around the whole cluster.

Outside of Kubernetes, such processes would usually be started through system init scripts or the systemd daemon during node boot up. On Kubernetes nodes, you can still use systemd to run your system processes, but then you can't take advantage of all the features Kubernetes provides.

4.4.1 Using a DaemonSet for running a Pod on every node

To run a Pod on all cluster nodes, create a *DaemonSet*, which is much like a ReplicationController or a ReplicaSet, except that Pods created by a DaemonSet already have a target node specified and skip the Kubernetes Scheduler, and aren't randomly scattered around the cluster.

A DaemonSet makes sure it creates as many Pods as there are nodes and deploys each one on its own node, as shown in figure 4.8.

Whereas a ReplicaSet (or ReplicationController) makes sure that a desired number of Pod replicas are running in the cluster, a DaemonSet doesn't even have any notion of a desired replica count. Its job is to ensure that a Pod matching its Pod selector is running on each node.

If the node goes down, the DaemonSet doesn't cause the Pod to be created elsewhere. But if a new node is added to the cluster, the DaemonSet immediately deploys a new Pod instance to it. It also does the same if someone inadvertently deletes one of the Pods, leaving the node without the DaemonSet's Pod. Like a ReplicaSet, a DaemonSet creates the Pod from the Pod template configured in its definition.

4.4.2 Using a DaemonSet to run Pods only on certain nodes

A DaemonSet deploys Pods on all nodes in the cluster, unless you specify that the Pods should only run on a subset of all the nodes. This is done by specifying the `nodeSelector` property in the Pod template, which is part of the DaemonSet definition (similar to the Pod template in a ReplicaSet or ReplicationController).

You've already used node selectors to deploy a Pod onto specific nodes in chapter 3. A node selector in a DaemonSet is similar—it defines the nodes the DaemonSet must deploy its Pods to.

> **NOTE** Later in the book, you'll learn that nodes can be made unschedulable, preventing Pods from being scheduled to them. A DaemonSets will deploy Pods even to such nodes, because the unschedulable attribute is only used by the scheduler, whereas Pods managed by a DaemonSet bypass the scheduler completely. This is usually desirable, because DaemonSets are meant to run system services, which usually need to run even on unschedulable nodes.

Explaining DaemonSets with an example

Let's imagine having a daemon called *ssd-monitor* that needs to run on all nodes that contain a Solid-State Drive (SSD). You'll create a DaemonSet that runs this daemon on all nodes that are marked as having an SSD. The cluster administrators have added the `disk=ssd` label to all such nodes, so you'll create the DaemonSet with a node selector that only selects nodes with that label, as shown in figure 4.9.

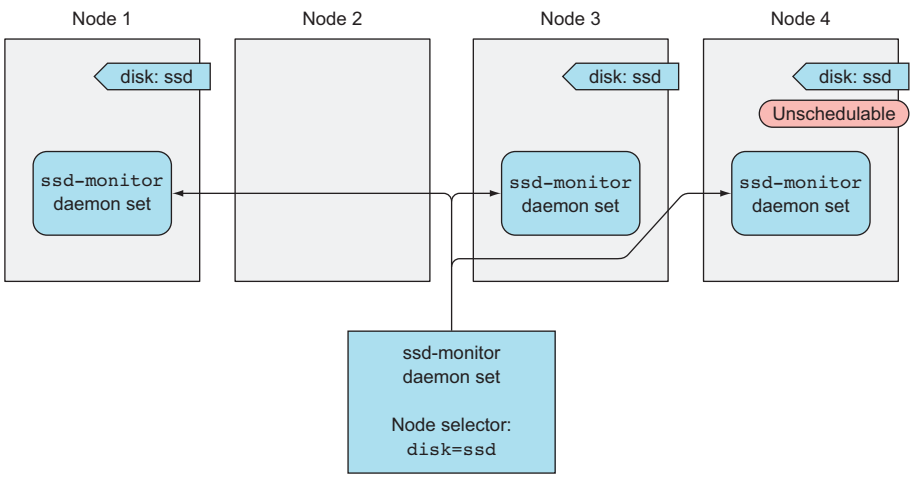


Figure 4.9 Using a DaemonSet with a node selector to deploy system Pods only on certain nodes.

Creating a DaemonSet YAML definition

You'll create a DaemonSet that runs a mock ssd-monitor process, which prints "SSD OK" to the standard output every five seconds. I've already prepared the mock container image and pushed it to Docker Hub, so you can use it instead of building your own.

Create the YAML for the DaemonSet, as shown in the following listing.

Listing 4.10 A YAML for a DaemonSet: ssd-monitor-daemonset.yaml

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
 name: ssd-monitor
spec:
 template:
   metadata:
     labels:
       app: ssd-monitor
   spec:
     nodeSelector:
       disk: ssd
     containers:
     - name: main
       image: luksa/ssd-monitor
```

DaemonSets are in the extensions API group, version v1beta1.

You're not specifying a Pod selector (it will be created based on the labels in the Pod template).

The Pod template includes a node selector, which selects nodes with the disk=ssd label.

You're defining a DaemonSet that will run a Pod with a single container based on the luksa/ssd-monitor container image. An instance of this Pod will be created for each node that has the `disk=ssd` label, even if the node is marked as unschedulable (as shown previously in figure 4.9).

CREATING THE DAEMONSET

You'll create the DaemonSet like you always create resources from a YAML file:

```
$ kubectl create -f ssd-monitor-daemonset.yaml
daemonset "ssd-monitor" created
```

Let's see the created DaemonSet:

```
$ kubectl get ds
NAME          DESIRED  CURRENT  READY  UP-TO-DATE  AVAILABLE  NODE-SELECTOR
ssd-monitor   0        0        0      0           0          disk=ssd
```

Those zeroes look strange. Didn't the DaemonSet deploy any Pods? List the Pods:

```
$ kubectl get po
No resources found.
```

What? Where are the Pods? Any idea what's going on? Yes, you forgot to label your nodes with the `disk=ssd` label. No problem—you can do that now. The DaemonSet should detect that the nodes' labels have changed and deploy the Pod to all nodes with a matching label. Let's see if that's true.

ADDING THE REQUIRED LABEL TO YOUR NODE(S)

Regardless if you're using Minikube, GKE, or another multi-node cluster, you'll need to list the nodes first, because you'll need to know the node's name when labelling it:

```
$ kubectl get node
NAME       STATUS    AGE       VERSION
minikube   Ready     4d        v1.6.0
```

Now, add the `disk=ssd` label to one of your nodes like this:

```
$ kubectl label node minikube disk=ssd
node "minikube" labeled
```

> **NOTE** Replace *minikube* with the name of one of your nodes if you're not using Minikube.

The DaemonSet should have created one Pod now. Let's see:

```
$ kubectl get po
NAME                READY     STATUS    RESTARTS   AGE
ssd-monitor-hgxwq   1/1       Running   0          35s
```

Okay; so far so good. If you have multiple nodes and you add the same label to further nodes, you'll see the DaemonSet spin up Pods for each of them.

REMOVING THE REQUIRED LABEL FROM THE NODE

Now, imagine you've made a mistake and have mislabeled one of the nodes. It has a spinning disk drive, not an SSD. What happens if you change the node's label?

```
$ kubectl label node minikube disk=hdd --overwrite
node "minikube" labeled
```

Let's see if the change has any effect on the Pod that was running on that node:

```
$ kubectl get po
NAME                READY     STATUS        RESTARTS   AGE
ssd-monitor-hgxwq   1/1       Terminating   0          4m
```

The Pod is being terminated. But you knew that was going to happen, right? This wraps up your exploration of DaemonSets, so you may want to delete it. If you still have any other daemon Pods running, you'll see that deleting the DaemonSet deletes those Pods as well.

4.5 Running Pods that perform a single completable task

Up to now, we've only talked about Pods than need to run continuously. You'll have cases where you only want to run a task that terminates after completing its work. Replication controllers, ReplicaSets, and DaemonSets run continuous tasks that are never considered completed. Processes in such Pods are restarted when they exit. But in a completable task, after its process terminates, it should not be restarted again.

4.5.1 Introducing the Job resource

Kubernetes includes support for this through the *Job* resource, which is similar to the other resources we've discussed in this chapter, but it allows you to run a Pod whose container isn't restarted when the process running inside finishes successfully. Once it does, the Pod is considered complete.

In the event of a node failure, the Pods running as jobs on that node will be rescheduled to other nodes the way ReplicaSet Pods are. In the event of a failure of the process itself (when the process returns an error exit code), the job can be configured to either restart the container or not.

Figure 4.10 shows how a Pod created by a Job is rescheduled to a new node if the node it was initially scheduled to fails. The figure also shows both a managed Pod, which isn't rescheduled, and a Pod backed by a ReplicaSet, which is.

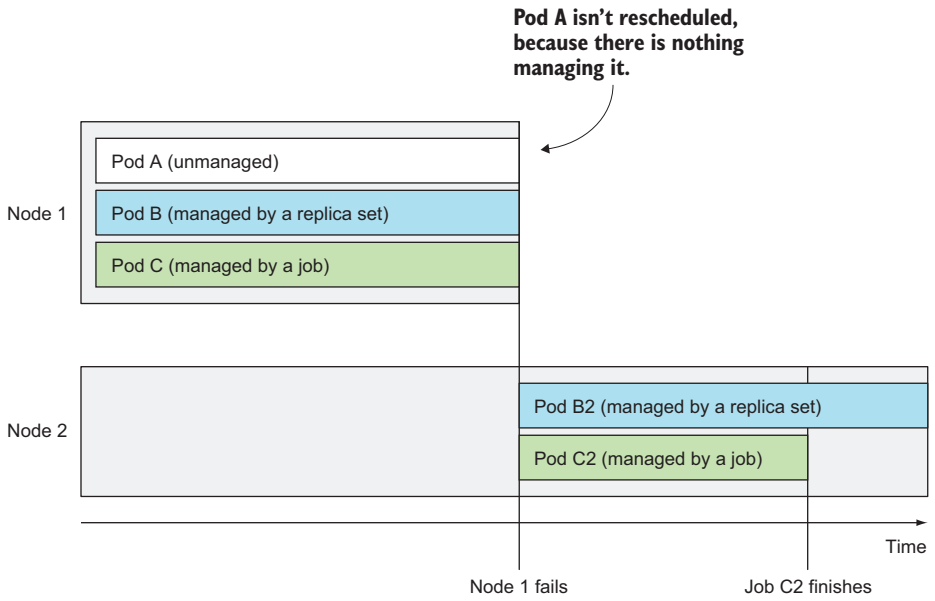


Figure 4.10 Pods backed by jobs are rescheduled until they finish successfully.

For example, Jobs are useful for ad hoc tasks, where it's crucial that the task finishes properly. You could run the task in an unmanaged Pod and wait for it to finish, but in the event of a node failing or the Pod being evicted from the node while it is performing its task, you'd need to manually recreate it. Doing this manually doesn't make sense—especially if the job takes hours to complete.

An example of such a job would be if you had data stored somewhere and you needed to transform and export it somewhere. You're going to emulate this by running a container image built on top of the busybox image, which invokes the sleep command for two minutes. I've already built the image and pushed it to Docker Hub, but you can peek into its Dockerfile in the book's code archive.

4.5.2 Defining a job

Create the job definition in the following listing.

Listing 4.11 A YAML definition of a Job: exporter.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: batch-job
spec:
  template:
    metadata:
      labels:
        app: batch-job
    spec:
      restartPolicy: OnFailure
      containers:
      - name: main
        image: luksa/batch-job
```

Jobs are in the batch API group, version vl.

You're not specifying a Pod selector (it will be created based on the labels in the Pod template).

Jobs can't use the default restart policy, which is Always.

Jobs are part of the `batch` API group and `v1` API version. The YAML is defining a resource of type `Job` that will run the `luksa/batch-job` image, which runs a process that runs for exactly 120 seconds and then exits.

In a Pod's specification, you can specify what Kubernetes should do when the processes running in the container finish. This is done through the `restartPolicy` Pod spec property, which defaults to `Always`. Job Pods can't use the default policy, because they're not meant to run indefinitely. Therefore, you need to explicitly set the restart policy to either `OnFailure` or `Never`. This setting is what prevents the container from being restarted when it finishes (not the fact that the Pod is being managed by a Job resource).

4.5.3 Seeing a Job run a Pod

After you create this Job with the `kubectl create` command, you should see it start up a Pod immediately:

```
$ kubectl get jobs
NAME        DESIRED   SUCCESSFUL   AGE
batch-job   1         0            2s

$ kubectl get po
NAME              READY     STATUS    RESTARTS   AGE
batch-job-28qf4   1/1       Running   0          4s
```

After the two minutes have passed, the Pod will no longer show up in the Pod list and the job will be marked as completed. By default, completed Pods aren't shown when you list Pods, unless you use the `--show-all` (or simply `-a`) switch:

```
$ kubectl get po -a
NAME              READY     STATUS      RESTARTS   AGE
batch-job-28qf4   0/1       Completed   0          2m
```

The reason the Pod isn't deleted when it completes is to allow you to examine its logs; for example:

```
$ kubectl logs batch-job-28qf4
Fri Apr 29 09:58:22 UTC 2016 Batch job starting
Fri Apr 29 10:00:22 UTC 2016 Finished succesfully
```

The Pod will be deleted when you delete it or the Job that created it. Before you do that, let's look at the Job resource again:

```
$ kubectl get job
NAME        DESIRED   SUCCESSFUL   AGE
batch-job   1         1            9m
```

The job is shown as having completed successfully. But why is that piece of information shown as a number instead of as `yes` or `true`? And what does the `DESIRED` column indicate?

4.5.4 Running multiple Pod instances in a Job

Jobs may be configured to create more than one Pod instance and run them in parallel or sequentially. This is done by setting the `completions` and the `parallelism` properties in the job spec.

RUNNING JOB PODS SEQUENTIALLY

If you need a job to run more than once, you set `completions` to how many times you want the job's Pod to run. The following listing shows an example.

Listing 4.12 A job requiring multiple completions: multi-completion-batch-job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: multi-completion-batch-job
spec:
  completions: 5
  template:
    <template is the same as in listing 4.11>
```

Setting completions to 5 makes this job run five Pods sequentially.

This job will run five Pods one after the other. It initially creates one Pod, and when the Pod's container finishes, it creates the second Pod, and so on, until five Pods complete successfully. If one of the Pods fails, the Job creates a new Pod, so the Job may create more than five Pods overall.

Running job Pods in parallel

Instead of running single job Pods one after the other, you can also make the job run multiple Pods in parallel. You specify how many Pods are allowed to run in parallel with the `parallelism` job spec property, as shown in the following listing.

Listing 4.13 Running job Pods in parallel: multi-completion-parallel-batch-job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: multi-completion-batch-job
spec:
  completions: 5
  parallelism: 2
  template:
    <same as in listing 4.11>
```

This job must ensure five Pods complete successfully.

Up to two Pods can run in parallel.

By setting `parallelism` to 2, the job creates two Pods and runs them in parallel:

```
$ kubectl get po
NAME                                READY   STATUS      RESTARTS   AGE
multi-completion-batch-job-lmmnk    1/1     Running     0          21s
multi-completion-batch-job-qx4nq    1/1     Running     0          21s
```

As soon as one of them finishes, the job will run the next Pod, until five Pods finish successfully.

Scaling a Job

You can even change a Job's `parallelism` property while the Job is running. This is similar to scaling a ReplicaSet or ReplicationController, and can be done with the `kubectl scale` command:

```
$ kubectl scale job multi-completion-batch-job --replicas 3
job "multi-completion-batch-job" scaled
```

Because you've increased `parallelism` from 2 to 3, another Pod is immediately spun up, so three Pods are now running.

4.5.5 Limiting the time allowed for a job Pod to complete

We need to discuss one final thing about jobs. How long should the job wait for a Pod to finish? What if the Pod gets stuck and can't finish at all (or it can't finish fast enough)?

A Pod's time can be limited by setting the `activeDeadlineSeconds` property in the Pod spec. If the Pod runs longer than that, the system will try to terminate it and will mark the Job as failed.

4.6 Scheduling jobs to run periodically or once in the future

Job resources run their Pods immediately when you create the job. But many batch jobs need to be run at a specific time in the future or repeatedly in the specified interval. In Linux- and UNIX-like operating systems, these jobs are better known as cron jobs. Kubernetes supports them, too.

A cron job in Kubernetes is configured by creating a CronJob resource. The schedule for running the job is specified in the well-known cron format, so if you're familiar with regular cron jobs, you'll understand Kubernetes' CronJobs in a matter of seconds.

At the configured time, Kubernetes will create a Job resource according to the Job template configured in the CronJob object. When the Job resource is created, one or more Pod replicas will be created and started according to the Job's Pod template, as you learned in the previous section. There's nothing more to it.

Let's look at how to create CronJobs.

NOTE In the current Kubernetes version (1.6), CronJobs are an alpha feature and may change. Refer to the Kubernetes API reference and other documentation to see if they have.

4.6.1 Creating a CronJob

Imagine you need to run the batch job from your previous example every 15 minutes. To do that, create a CronJob resource with the following specification.

Listing 4.14 YAML for a CronJob resource: cronjob.yaml

```
apiVersion: batch/v2alpha1
kind: CronJob
metadata:
  name: batch-job-every-fifteen-minutes
spec:
  schedule: "0,15,30,45 * * * *"
  jobTemplate:
    spec:
      template:
        metadata:
          labels:
            app: periodic-batch-job
        spec:
          restartPolicy: OnFailure
          containers:
          - name: main
            image: luksa/batch-job
```

API group is batch, version is v2alpha1.

This job should run at the 0, 15, 30 and 45 minutes of every hour, every day.

The template for the Job resources that will be created by this CronJob.

As you can see, it's not too complicated. You've specified a schedule and a template from which the Job objects will be created.

NOTE CronJobs are currently not available in GKE, unless you create an alpha cluster. Refer to the GKE documentation on how to do that. You can create a CronJob in Minikube.

CONFIGURING THE SCHEDULE

If you're unfamiliar with the cron schedule format, you'll find great tutorials and explanations online, but as a quick introduction, from left to right, the schedule contains the following five entries:

- Minute, Hour, Day of month, Month, and Day of week.

In the example, you want to run the job every 15 minutes, so the schedule needs to be `"0,15,30,45 * * * *"`, which means at the 0, 15, 30 and 45 minutes mark of every hour (first asterisk), of every day of the month (second asterisk), of every month (third asterisk) and on every day of the week (fourth asterisk).

If, instead, you wanted it to run every 30 minutes, but only on the first day of the month, you'd set the schedule to `"0,30 * 1 * *"`, and if you want it to run at 3AM every Sunday, you'd set it to `"0 3 * * 0"` (the last zero stands for Sunday).

CONFIGURING THE JOB TEMPLATE

A CronJob creates Job resources from the `jobTemplate` property configured in the CronJob spec, so refer to section Running Pods that perform a single completable task for more information on how to configure it.

4.6.2 Understanding how scheduled jobs are run

Job resources will be created from the CronJob resource at approximately the scheduled time. The Job then creates the Pods.

It may happen that the Job or Pod is created and run relatively late. You may have a hard requirement for the job to not be started too far over the scheduled time. In that case, you can specify a deadline by specifying the `startingDeadlineSeconds` field in the CronJob specification as shown in the following listing.

Listing 4.15 Specifying a startingDeadlineSeconds for a CronJob

```
apiVersion: batch/v2alpha1
kind: CronJob
spec:
  schedule: "0,15,30,45 * * * *"
  startingDeadlineSeconds: 15
  ...
```

At the latest, the Pod must start running at 15 seconds past the scheduled time.

In the example in listing 4.15, one of the times the job is supposed to run is 10:30:00. If it doesn't start by 10:30:15 for whatever reason, the job will not run and will be shown as Failed.

In normal circumstances, a CronJob always creates only a single job for each execution configured in the schedule, but it may happen that two jobs are created at the same time, or none at all. To combat the first problem, your jobs should be idempotent (running them multiple times instead of once shouldn't lead to bad results). For

the second problem, make sure that the next job run performs any work that should have been done by the previous (missed) run.

4.7 Summary

You've now learned how to keep Pods running and having them rescheduled in the event of node failures. You should now know that

- You can specify a liveness probe to have Kubernetes restart your container as soon as it's no longer healthy (where the app defines what's considered healthy).
- Pods shouldn't be created directly, because they will not be re-created if they're deleted by mistake, if the node they're running on fails, or if they're evicted from the node.
- Replication controllers always keep the desired number of Pod replicas running.
- Scaling Pods horizontally is as easy as changing the desired replica count on a replication controller.
- Pods aren't owned by the replication controllers and can be moved between them if necessary.
- A replication controller creates new Pods from a Pod template. Changing the template has no effect on existing Pods.
- Replication controllers will eventually be replaced by ReplicaSets and deployments, which provide the same functionality, but with additional powerful features.
- Replication controllers and ReplicaSets schedule Pods to random cluster nodes, whereas DaemonSets make sure every node runs a single instance of a Pod defined in the DaemonSet.
- Pods that perform a batch task should be created through a Kubernetes Job resource, not directly or through a ReplicationController or similar object.
- Jobs that need to run sometime in the future can be created through CronJob resources.

5 Services: enabling clients to discover and talk to Pods

This chapter covers

- Creating Services to expose a group of Pods at a single address
- Discovering services in the cluster
- Exposing services to external clients
- Connecting to external services from inside the cluster
- Controlling whether a Pod is ready to be part of the service or not
- Troubleshooting services

You've learned about Pods and how to deploy them through replica sets and other resources to ensure they keep running. Although certain Pods can do their work independently of an external stimulus, many applications these days are meant to respond to external requests. For example, in the case of microservices, Pods will usually respond to HTTP requests coming either from other Pods inside the cluster or from clients outside the cluster.

Pods need a way of finding other Pods if they want to consume the services they provide. Unlike in the non-Kubernetes world, where a sysadmin would configure each client app by specifying the exact IP address or hostname of the server

providing the service in the client's configuration files, doing the same in Kubernetes wouldn't work, because

- *Pods are ephemeral*—They may come and go at any time, whether it's because a Pod is removed from a node to make room for other Pods, because someone scaled down the number of Pods, or because a cluster node has failed.
- *Kubernetes assigns an IP address to a Pod after the Pod has been scheduled to a node and before it's started*—Clients thus can't know the IP address of the server Pod up front.
- *Horizontal scaling means multiple Pods may provide the same service*—Each of those Pods has its own IP address. Clients shouldn't care how many Pods are backing the service and what their IPs are. They shouldn't have to keep a list of all the individual IPs of Pods. Instead, all those Pods should be accessible through a single IP address.

To solve these problems, Kubernetes also provides another resource type—*Services*—that we'll discuss in this chapter.

5.1 Introducing services

A Kubernetes service is a resource you create to get a single, constant point of entry to a group of Pods providing the same service. Each service has an IP address and port that never change while the service exists. Clients can open connections to that IP and port, and those connections are then routed to one of the Pods backing that service. This way, clients of a service don't need to know the location of individual Pods providing the service, allowing those Pods to be moved around the cluster at any time.

EXPLAINING SERVICES WITH AN EXAMPLE

Let's revisit the example where you have a frontend web server and a backend database server. There may be multiple Pods that all act as the frontend, but there may only be a single backend database Pod. You need to solve two problems to make the system function:

- External clients need to connect to the frontend Pods without caring if there's only a single web server or hundreds.
- The frontend Pods need to connect to the backend database. Because the database runs inside a Pod, it may be moved around the cluster over time, causing its IP address to change. You don't want to reconfigure the frontend Pods every time the backend database is moved.

By creating a service for the frontend Pods and configuring it to be accessible from outside the cluster, you expose a single, constant IP address through which external clients can connect to the Pods. Similarly, by also creating a service for the backend Pod, you create a stable address for the backend Pod. The service address doesn't

change even if the Pod's IP address changes. Additionally, by creating the service, you also enable the frontend Pods to easily find the backend service by its name through either environment variables or DNS. All the components of your system (the two services, the two sets of Pods backing those services, and the interdependencies between them) are shown in figure 5.1.

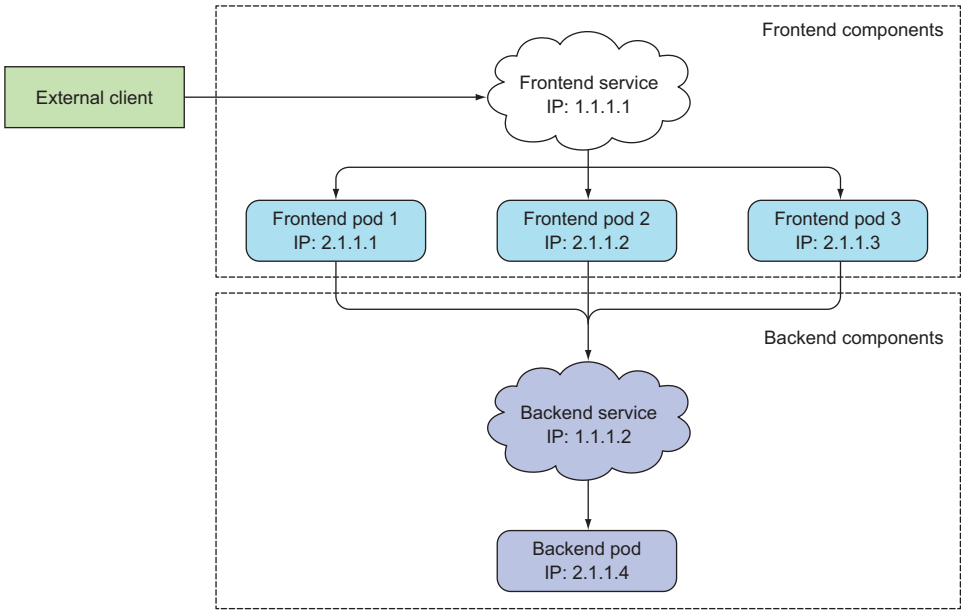


Figure 5.1 **Both internal and external clients usually connect to Pods through services.**

You now understand the basic idea behind Services. Now, let's dig deeper by first seeing how they can be created.

5.1.1 Creating services

As you've seen, because of horizontal scaling, a service can be backed by more than one Pod. Connections to the service are load-balanced across all the backing Pods. But how exactly do you define which Pods are part of the service and which aren't?

You probably remember label selectors and how they're used in ReplicationControllers and other Pod controllers to specify which Pods belong to the same set. The same mechanism is used by services in the same way, as you can see in figure 5.2.

In the previous chapter, you created a ReplicationController which then ran three instances of the Pod containing the Node.js app. Create the replication controller again and verify three Pod instances are up and running. After that, you'll create a Service for those three Pods.

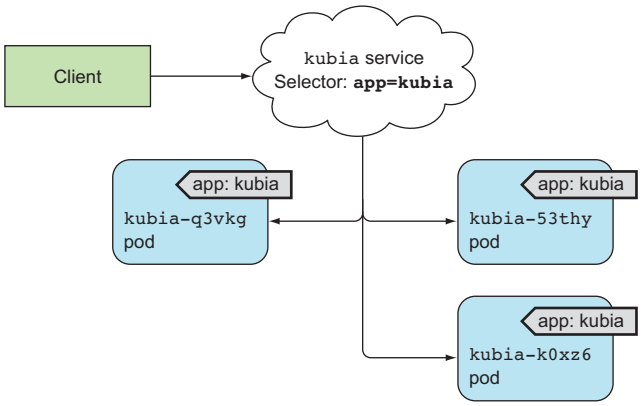


Figure 5.2 Label selectors determine which Pods belong to the Service.

CREATING A SERVICE THROUGH KUBECTL EXPOSE

The easiest way to create a service is through `kubectl expose`, which you've already used in chapter 2 to expose the replication controller you created earlier. The `expose` command created a service with the same Pod selector as the one used by the replication controller, thereby exposing all its Pods through a single IP address and port.

Now, instead of using the expose command, you'll create a service manually by posting a YAML to the Kubernetes API server.

CREATING A SERVICE THROUGH A YAML DESCRIPTOR

Create a file called kubia-svc.yaml with the following listing's contents.

Listing 5.1 A definition of a service: kubia-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
 name: kubia
spec:
 ports:
 - port: 80
   targetPort: 8080
 selector:
   app: kubia
```

The port this service will be available on.

The container port the service will forward to.

All Pods with the app=kubia label will be part of this service.

You're defining a service called `kubia`, which will accept connections on port `80` and route each connection to port `8080` of one of the Pods matching the `app=kubia` label selector.

Go ahead and create the service by posting the file using `kubectl create`.

Examining your new service

After posting the YAML, you can list services in your namespace and see that an internal cluster IP has been assigned to your service:

```
$ kubectl get svc
NAME         CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
kubernetes   10.111.240.1     <none>        443/TCP   30d
kubia        10.111.249.153   <none>        80/TCP    6m
```

Here's your service.

The list shows that the IP address assigned to your service is 10.111.249.153. Because this is the cluster IP, it's only accessible from inside the cluster. The primary purpose of services is exposing groups of Pods to other Pods in the cluster. You'll usually also want to expose several services externally. You'll see how to do this later. For now, let's use your service from inside the cluster and see what it does.

Testing a service from within the cluster

You can send requests to services from within the cluster in a few ways:

- The obvious way is to create a Pod that will perform the request and log the response. You can then examine the Pod's log to see what the service's response was.
- You can `ssh` into one of the Kubernetes nodes and use the `curl` command.
- You can execute the `curl` command inside one of your existing Pods through the `kubectl exec` command.

Let's go for the last option, so you also learn how to run commands in existing Pods.

Remotely executing commands in running containers

The `kubectl exec` command allows you to remotely run arbitrary commands inside an existing container of a Pod. This comes in handy when you want to examine the contents, state, and/or environment of a container. List the Pods with the `kubectl get pods` command and choose one as your target for the `exec` command (in the following example, I've chosen the *kubia-7nog1* Pod as the target). You'll also need to obtain the cluster IP of your service (using `kubectl get svc`, for example). When running the following commands yourself, be sure to replace the Pod name and the service IP with your own:

```
$ kubectl exec kubia-7nog1 -- curl -s http://10.111.249.153
You've hit kubia-gzwli
```

If you've used `ssh` to execute commands on a remote system before, you'll recognize that `kubectl exec` isn't much different.

Why the double dash?

The double dash (`--`) in the command signals the end of command options for `kubectl`. Everything after the double dash is the command that should be executed inside the Pod. Using the double dash isn't necessary if the command has no arguments that start with a dash. But in your case, if you don't use the double dash there, the `-s` option would be interpreted as an option for `kubectl exec` and would result in the following strange and highly misleading error:

```
$ kubectl exec kubia-7nog1 curl -s http://10.111.249.153
The connection to the server 10.111.249.153 was refused - did you
specify the right host or port?
```

This has nothing to do with your service refusing the connection. It's because `kubectl` is not able to connect to an API server at 10.111.249.153 (the `-s` option is used to tell `kubectl` to connect to a different API server than the default).

Let's go over what you did. Figure 5.3 shows the sequence of events. You instructed Kubernetes to execute the `curl` command inside the single container of one of your Pods. Curl sent an HTTP request to the service IP, which is backed by three Pods. The Kubernetes service proxy intercepted the connection, selected a random Pod among the three Pods, and forwarded the request to it. Node.js running inside that Pod then handled the request and returned an HTTP response with the Pod's name. Curl then printed the response to the standard output, which was intercepted and printed to the standard output on your local machine by `kubectl`.

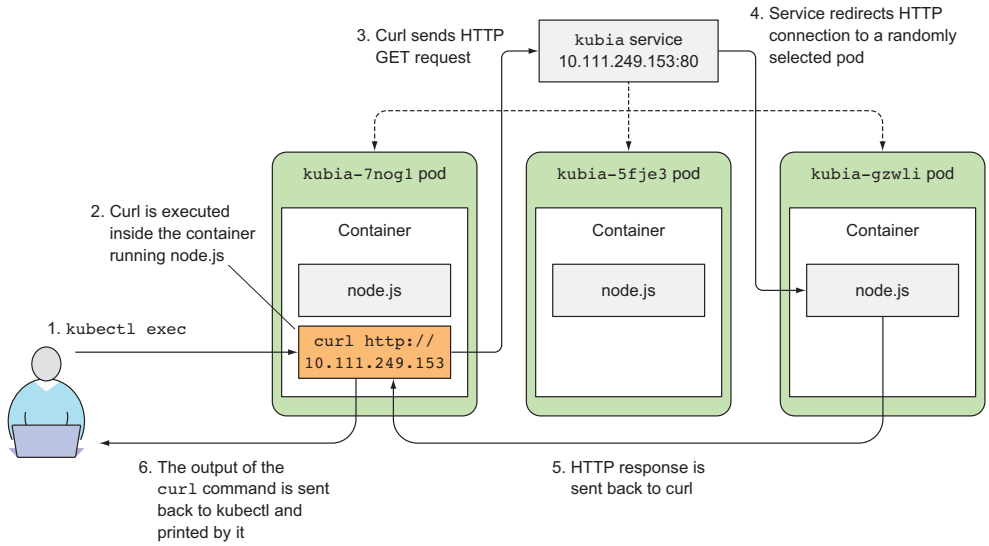


Figure 5.3 Using kubectl exec to run curl in one of the Pods and test out a connection to the service.

In the previous example, you executed the `curl` command as a separate process, but inside the Pod's main container. This isn't much different from the actual main process in the container talking to the service.

CONFIGURING SESSION AFFINITY ON THE SERVICE

If you execute the same command a few more times, you should hit a different Pod with every invocation, because the service proxy normally forwards each connection to a randomly selected backing Pod, even if the connections are being opened by the same client.

If, on the other hand, you want all requests made by a certain client to be redirected to the same Pod every time, you can set the service's `sessionAffinity` property to `ClientIP` (instead of `None`, which is the default), as shown in the following listing.

Listing 5.2 A example of a service with ClientIP session affinity configured

```
apiVersion: v1
kind: Service
spec:
  sessionAffinity: ClientIP
  ...
```

This makes the service proxy redirect all requests coming from the same client IP to the same Pod. As an exercise, you can create an additional service with session affinity set to `ClientIP` and try sending requests to it.

Kubernetes supports only two types of service session affinity: `None` and `ClientIP`. You may be surprised it doesn't have a cookie-based session affinity option, but you need to understand that Kubernetes services operate on a lower level than HTTP. Services forward TCP and UDP packets and don't care about the payload they carry. Because cookies are a construct of the HTTP protocol, services don't know about them, which explains why session affinity cannot be based on cookies.

EXPOSING MULTIPLE PORTS IN THE SAME SERVICE

Your service exposes only a single port, but services can also support multiple ports. For example, if your Pods listened on two ports—let's say 8080 for HTTP and 8443 for HTTPS—you could use a single service to forward both port 80 and 443 to the Pod's ports 8080 and 8443. You don't need to create two different services in such cases. Using a single, multi-port service exposes all the service's ports through a single cluster IP.

The spec for a multi-port service looks like the following listing.

Listing 5.3 Specifying multiple ports in a service definition

```
apiVersion: v1
kind: Service
metadata:
  name: kubia
spec:
  ports:
```

```
  - port: 80                    Port 80 is mapped to the Pods' port 8080.
    targetPort: 8080
  - port: 443                   Port 443 is mapped to Pods' port 8443.
    targetPort: 8443
  selector:                     The label selector always applies to the whole service.
    app: kubia
```

NOTE The label selector applies to the service as a whole—it can't be configured for each port individually. If you want different ports to map to different subsets of Pods, you need to create two services.

Because your kubia Pods don't listen on multiple ports, creating a multi-port service and a multi-port Pod is left as an exercise to you.

USING NAMED PORTS

In all these examples, you've referred to the target port by its number, but you can also give a name to each Pod's port and refer to it by name in the service spec. This makes the service spec slightly clearer, especially if the port numbers aren't well-known.

For example, suppose your Pod defines names for its ports as shown in the following listing.

Listing 5.4 Specifying port names in a Pod definition

```
kind: Pod
spec:
  containers:
  - name: kubia
    ports:
    - name: http                Container's port 8080 is called http.
      containerPort: 8080
    - name: https               Port 8443 is called https.
      containerPort: 8443
```

You can then refer to those ports by name in the service spec, as shown in the following listing.

Listing 5.5 Referring to named ports in a service

```
apiVersion: v1
kind: Service
spec:
 ports:
 - port: 80                     Port 80 is mapped to the container's port called http.
   targetPort: http
 - port: 443                    Port 443 is mapped to the container's port, whose name is https.
   targetPort: https
```

But why should you even bother with naming ports? The biggest benefit of doing so is that it enables you to change port numbers later without having to change the service spec. Your Pod currently uses port 8080 for http, but what if you later decide you'd like to move that to port 80?

If you're using named ports, all you need to do is change the port number in the Pod spec (while keeping the port's name unchanged). As you spin up Pods with the new ports, client connections will be forwarded to the appropriate port numbers, depending on the Pod receiving the connection (port 8080 on old Pods and port 80 on the new ones).

5.1.2 Discovering services

By creating a service, you now have a single and stable IP address and port that you can hit to access your Pods. This address will remain constant throughout the whole lifetime of the service. Pods behind this service may come and go, their IPs may change, their number can go up or down, but they'll always be accessible through the service's single IP address.

But how do the client Pods know the IP and port of a service? Do you need to create the service first, then manually look up its IP address and pass the IP to the configuration options of the client Pod? Not really. Kubernetes also provides ways for client Pods to discover a service's IP and port.

DISCOVERING SERVICES THROUGH ENVIRONMENT VARIABLES

When a Pod is started, Kubernetes initializes a set of environment variables for each service that exists at that moment. If you create the service before creating the client Pods, processes in those Pods can look up the IP address and port of the service by inspecting their environment variables.

Let's see what those environment variables look like by examining the environment of one of your running Pods. You've already learned that you can use the `kubectl exec` command to run a command in the Pod, but because you created the service only after your Pods had been created, the environment variables for the service couldn't have been set yet.

Before you can see environment variables for your service, you first need to delete all the Pods and let the replication controller create new ones. You may remember you can delete all Pods without having to specify their names like this:

```
$ kubectl delete po --all
pod "kubia-7nog1" deleted
pod "kubia-bf50t" deleted
pod "kubia-gzwli" deleted
```

Now you can list the new Pods (I'm sure you know how to do that) and pick one as your target for the `kubectl exec` command. Once you've selected your target Pod, you can list environment variables by running the `env` command inside the container, as shown in the following listing.

Listing 5.6 Environment variables in a container

```
$ kubectl exec kubia-3inly env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=kubia-3inly
KUBERNETES_SERVICE_HOST=10.111.240.1
KUBERNETES_SERVICE_PORT=443
...
KUBIA_SERVICE_HOST=10.111.249.153
KUBIA_SERVICE_PORT=80
...
```

Here's the cluster IP of the service.

And here's the port the service is available on.

Two services are defined in your cluster: the kubernetes and the kubia service (you saw this earlier with the `kubectl get svc` command); consequently, two sets of service-related environment variables are in the list. Among the variables that pertain to the kubia service you created, you'll see the `KUBIA_SERVICE_HOST` and the `KUBIA_SERVICE_PORT` environment variables, which hold the IP address and port of the kubia service, respectively.

Turning back to the frontend-backend example we started this chapter with, when you have a frontend Pod that requires the use of a backend database server Pod, you can expose the backend Pod through a service called *backend-database* and then have the frontend Pod look up its IP address and port through the environment variables `BACKEND_DATABASE_SERVICE_HOST` and `BACKEND_DATABASE_SERVICE_PORT`.

> **NOTE** Dashes in the service name are converted to underscores and all letters are upper-cased when the service name is used as the prefix in the environment variable's name.

Environment variables are one way of looking up the IP and port of a service, but isn't this usually the domain of DNS? Why doesn't Kubernetes include a DNS server and allow you to look up service IPs through DNS instead? As it turns out, it does!

DISCOVERING SERVICES THROUGH DNS

Remember in chapter 3 when you listed Pods in the kube-system namespace? One of the Pods was called kube-dns. The kube-system namespace also includes a corresponding service with the same name.

As the name suggests, the Pod is a DNS server, which all other Pods running in the cluster are automatically configured to use (Kubernetes modifies each container's `/etc/resolv.conf` file). Any DNS query performed by a process running in a Pod will be handled by Kubernetes' own DNS server, which knows all the services running in your system.

> **NOTE** Whether a Pod uses the internal DNS server or not is configurable through the `dnsPolicy` property in each Pod's spec.

Each service gets its own DNS entry in the internal DNS server and client Pods that know the name of the service can access it through its fully qualified domain name (FQDN) without having to look up environment variables.

CONNECTING TO THE SERVICE THROUGH ITS FQDN

To revisit the frontend-backend example, a frontend Pod can connect to the backend-database service by opening a connection to the following FQDN:

```
backend-database.default.svc.cluster.local
```

`backend-database` corresponds to the service name, `default` stands for the namespace the service is defined in, and `svc.cluster.local` is a configurable cluster domain suffix used in all cluster local service names.

> **NOTE** The client must still know the service's port number. If the service is using a standard port (for example, 80 for HTTP or 5432 for Postgres), that shouldn't be a problem. If not, the client can get the port number from the environment variable.

Connecting to a service can be even simpler than that. You can omit the svc.cluster.local suffix or even use backend-database without appending the namespace, if the frontend Pod is in the same namespace as the database Pod. That's incredibly simple, right?

Let's try this. You'll try to access the service through its FQDN instead of its IP. Again, you'll need to do that inside an existing Pod. You already know how to use `kubectl exec` to run a single command in a Pod's container, but this time, instead of running the `curl` command directly, you'll run the bash shell instead, so you can then run multiple commands in the container. This is similar to what you did in chapter 2 when you entered the container you ran with Docker by using the `docker exec -it bash` command.

RUNNING A SHELL IN A POD'S CONTAINER

You can use the `kubectl exec` command to run bash (or any other shell) inside a Pod's container. This way you're free to explore the container as long as you want, without having to perform a `kubectl exec` for every command you want to run.

> **NOTE** The shell's binary executable must be available in the container image for this to work.

To use the shell properly, you need to pass the `-it` option to `kubectl exec`:

```
$ kubectl exec -it kubia-3inly bash
root@kubia-3inly:/#
```

You're now inside the container. You can use the `curl` command to access the kubia service in any of the following ways:

```
root@kubia-3inly:/# curl http://kubia.default.svc.cluster.local
You've hit kubia-5asi2

root@kubia-3inly:/# curl http://kubia.default
You've hit kubia-3inly

root@kubia-3inly:/# curl http://kubia
You've hit kubia-8awf3
```

You can hit your service by using the service's name as the hostname in the requested URL. You can omit the namespace and the svc.cluster.local suffix because of how the DNS resolver inside each Pod's container is configured. Look at the /etc/resolv.conf file in the container and you'll understand:

```
root@kubia-3inly:/# cat /etc/resolv.conf
search default.svc.cluster.local svc.cluster.local cluster.local ...
```

UNDERSTANDING WHY YOU CAN'T PING A SERVICE IP

One last thing before we move on. You know how to create services now, so you'll soon create your own. But what if, for whatever reason, you can't access your service.

You'll probably try to figure out what's wrong by entering an existing Pod and trying to access the service like you did in the last example. Then, if you still can't access the service with a simple curl command, maybe you'll try to ping the service IP to see if it's up. Try to do this now, because you're already inside the container:

```
root@kubia-3inly:/# ping kubia
PING kubia.default.svc.cluster.local (10.111.249.153): 56 data bytes
^C--- kubia.default.svc.cluster.local ping statistics ---
54 packets transmitted, 0 packets received, 100% packet loss
```

Hmm. Curl-ing the service works, but pinging it doesn't. This is because the service's cluster IP is a virtual IP, and only has meaning when combined with the service port. We'll explain what this means and how services work in chapter 11. I wanted to mention this here because it's the first thing users do when they try to debug a broken service and it catches most of them off guard.

5.2 Connecting to services living outside the cluster

Up to now, we've talked about services backed by one or more Pods running inside the cluster. But cases exist when you'd like to expose external services through the Kubernetes services feature. Instead of having the service redirect connections to Pods in the cluster, you want it to redirect to external IP(s) and port(s).

This allows you to take advantage of both service load balancing and service discovery. Client Pods running in the cluster can connect to the external service like they connect to internal services.

5.2.1 Introducing service endpoints

Before going into how to do this, let me first shed more light on services. Services don't link to Pods directly. Instead, a resource sits in between—the *Endpoints* resource. You may have already noticed endpoints if you used the `kubectl describe` command on your service, as shown in the following listing.

Listing 5.7 Full details of a service displayed with kubectl describe

```
$ kubectl describe svc kubia
Name:              kubia
Namespace:         default
Labels:            <none>
Selector:          app=kubia
Type:              ClusterIP
IP:                10.111.249.153
Port:              <unset> 80/TCP
Endpoints:         10.108.1.4:8080,10.108.2.5:8080,10.108.2.6:8080
Session Affinity:  None
No events.
```

The service's Pod selector is used to create the list of Endpoints.

The list of Pod IPs and ports that represent the endpoints of this service.

An Endpoints resource (yes, plural) is a list of IP addresses and ports exposing a service. The Endpoints resource is like any other Kubernetes resource, so you can display its basic info with `kubectl get`:

```
$ kubectl get endpoints kubia
NAME    ENDPOINTS                                          AGE
kubia   10.108.1.4:8080,10.108.2.5:8080,10.108.2.6:8080    1h
```

Although the Pod selector is defined in the service spec, it's not used directly when redirecting incoming connections. Instead, the selector is used to build a list of IPs and ports, which are then stored in the Endpoints resource. When a client connects to a service, the service proxy selects one of those IP and port pairs and redirects the incoming connection to the server listening at that location.

5.2.2 Manually configuring service endpoints

You may have probably realized this already, but having the service's endpoints decoupled from the service allows them to be configured and updated manually.

If you create a service without a Pod selector, Kubernetes won't even create the Endpoints resource (after all, without a selector, it can't know which Pods to include in the service). It's up to you to create the Endpoints resource and specify the list of Endpoints for the service.

To create a service with manually managed endpoints, you need to create both a Service and an Endpoints resource.

CREATING A SERVICE WITHOUT A SELECTOR

You'll first create the YAML for the service itself, as shown in the following listing.

Listing 5.8 A service without a Pod selector: external-service.yaml

```
apiVersion: v1
kind: Service
metadata:
 name: external-service
spec:
 ports:
 - port: 80
```

The name of the service must match the name of the Endpoints object (see next listing).

This service has no selector defined.

You're creating a service called `external-service` that will accept incoming connections on port 80. You didn't define a Pod selector for the service.

CREATING AN ENDPOINTS RESOURCE FOR A SERVICE WITHOUT A SELECTOR

The endpoints are a separate resource and not an attribute of a service. Because you created the service without a selector, the corresponding Endpoints resource hasn't been created automatically, so it's up to you to create it. The following listing shows its YAML manifest.

Listing 5.9 A manually created Endpoints resource: external-service-endpoints.yaml

```
apiVersion: v1
kind: Endpoints
metadata:
 name: external-service
subsets:
 - addresses:
   - ip: 11.11.11.11
   - ip: 22.22.22.22
   ports:
   - port: 80
```

The name of the Endpoints object must match the name of the service (see previous listing).

The IPs of the Endpoints that the service will forward connections to.

The target port of the Endpoints.

The Endpoints object needs to have the same name as the service and contain the list of target IP addresses and ports for the service. After both the service and the Endpoints resources are posted to the server, the service is ready to be used like any regular service with a Pod selector. Containers created after the service is created will include the environment variables for the service, and all connections to the service will be load balanced between the service's Endpoints.

Figure 5.4 shows three Pods connecting to the service with the external Endpoints.

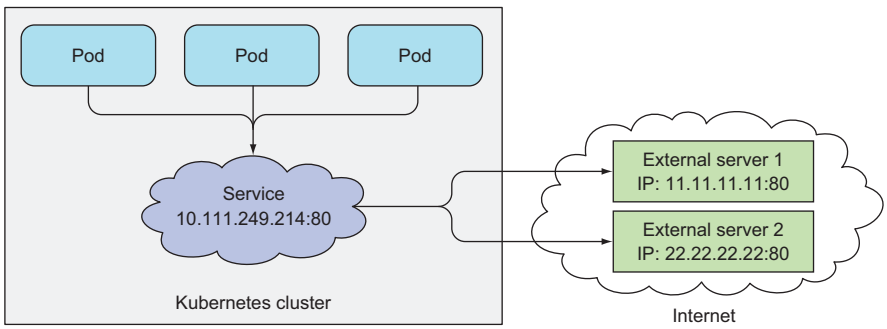


Figure 5.4 Pods consuming a service with two external Endpoints.

If you later decide to migrate the external service to Pods running inside Kubernetes, you can add a selector to the service, thereby making its Endpoints managed automatically. The same is also true in reverse—by removing the selector from a Service, Kubernetes stops updating its Endpoints. This means a service IP address can remain constant while the actual implementation of the service can be changed.

5.2.3 Creating an alias for an external service

Instead of exposing an external service internally by manually configuring the service's Endpoints, a simpler method allows you to refer to an external service by its fully qualified domain name (FQDN).

CREATING AN EXTERNALNAME SERVICE

To create a service that serves as an alias for an external service, you create a service with the type set to `ExternalName`. For example, let's imagine there's a public API available at api.somecompany.com. You can define a service that points to it as shown in the following listing.

Listing 5.10 An ExternalName-type service: external-service-externalname.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: external-service
spec:
  type: ExternalName
  externalName: someapi.somecompany.com
  ports:
  - port: 80
```

Service type is set to ExternalName.

The fully qualified domain name of the actual service.

After the service is created, Pods can connect to the external service through the external-service.default.svc.cluster.local domain name (or even external-service) instead of using the service's actual FQDN. This hides the actual service name and its location from Pods consuming the service, allowing you to modify the service definition and point it to a different service any time later, by changing the `externalName` attribute or by changing the type back to `ClusterIP` and creating an Endpoints object for the service—either manually or by specifying a label selector on the service and having it created automatically.

ExternalName services are implemented solely at the DNS level—a simple CNAME DNS record is created for the service. Therefore, clients connecting to the service will connect to the external service directly, bypassing the service proxy completely. For this reason, these types of services don't even get a cluster IP.

> **NOTE** A CNAME records points to a fully qualified domain name instead of a numeric IP address.

5.3 Exposing services to external clients

Up to now, we've only talked about how services can be consumed by Pods from inside the cluster. But you'll also want to expose certain services, such as frontend webservers, to the outside, so external clients can access them, as depicted in figure 5.5.
You have a few ways to make a service accessible externally:

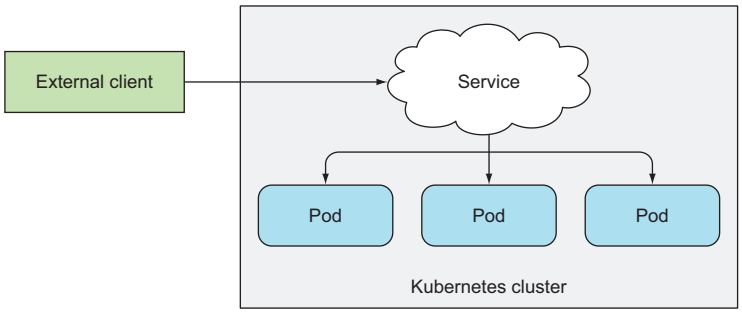


Figure 5.5 Exposing a service to external clients.

- *Setting the service type to NodePort*—For a NodePort service, each cluster node opens a port on the node itself (hence the name) and redirects traffic received on that port to the underlying service. The service isn't only accessible at the internal cluster IP and port, but also through a specific port on all nodes.
- *Setting the service type to LoadBalancer, an extension of the NodePort type*—This makes the service accessible through a dedicated load balancer, provisioned from the (cloud) infrastructure Kubernetes is running on. The load balancer redirects traffic to the node port across all the nodes. Clients connect to the service through the load balancer's IP.
- *Creating an Ingress resource, which is a radically different mechanism for exposing multiple services through a single IP address*—It operates at the HTTP level (network layer 7) and can thus offer more features than layer 4 services can. We'll explain Ingress resources in section Exposing services externally through an Ingress resource.

5.3.1 Using a NodePort service

The first method of exposing a set of Pods to external clients is by creating a service and setting its type to NodePort. By creating a NodePort service, you make Kubernetes reserve a port on all its nodes (the same port number is used across all of them) and forward incoming connections to the Pods that are part of the service.

This is similar to a regular service (their actual type is ClusterIP), but a NodePort service can be accessed not only through the service's internal cluster IP, but also through any node's IP and the reserved node port.

This will make more sense when you try interacting with a NodePort service.

CREATING A NODEPORT SERVICE

You'll now create a NodePort service to see how you can use it. The following listing shows the YAML for the service.

Listing 5.11 A NodePort service definition: kubia-svc-nodeport.yaml

```
apiVersion: v1
kind: Service
metadata:
 name: kubia-nodeport
spec:
 type: NodePort
 ports:
 - port: 80
   targetPort: 8080
   nodePort: 30123
 selector:
   app: kubia
```

Set the service type to NodePort.

This is the port of the service's internal cluster IP.

This is the target port of the backing Pods.

The service will be accessible through port 30123 of each of your cluster nodes.

You set the type to NodePort and specify the node port this service should be bound to across all cluster nodes. Specifying the port isn't mandatory; Kubernetes will choose a random port if you omit it.

> **NOTE** If you're using GKE, `kubectl` will print out a warning about having to configure firewall rules. We'll see how to do that in a few moments.

Examining our NodePort service

Let's see the basic information of your service to learn more about it:

```
$ kubectl get svc kubia-nodeport
NAME             CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubia-nodeport   10.111.254.223   <nodes>       80:30123/TCP   2m
```

Look at the `EXTERNAL-IP` column. It shows `<nodes>`, indicating the service is accessible through the IP address of any cluster node. The `PORT(S)` column shows both the internal port of the cluster IP (`80`) and the node port (`30123`). The service is accessible at the following addresses:

- `10.11.254.223:80`,
- `<1st node's IP>:30123`,
- `<2nd node's IP>:30123`, and so on.

Figure 5.6 shows your service exposed on port 30123 of both of your cluster nodes (if you're running this on GKE; Minikube only has a single node, but the principle is the same). An incoming connection to one of those ports will be redirected to a randomly selected Pod, but not necessarily the one running on the node the connection is being made to.

A connection received on port 30123 of the first node might be forwarded either to the Pod running on the first node or to one of the Pods running on the second node.

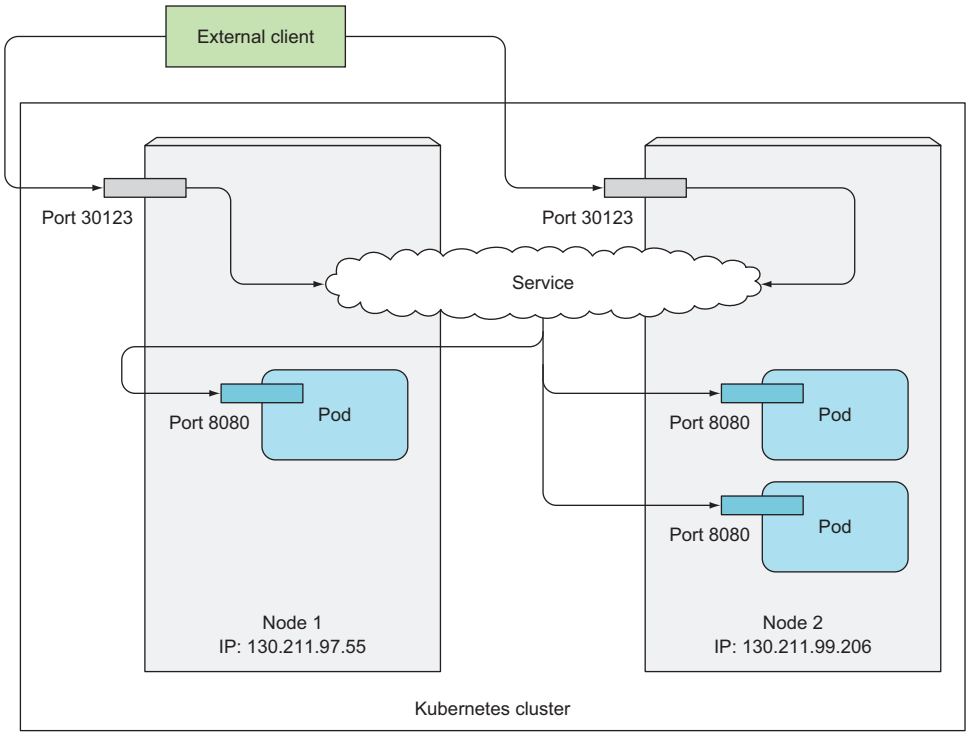


Figure 5.6 An external client connecting to a NodePort service either through Node 1 or 2.

CHANGING FIREWALL RULES TO LET EXTERNAL CLIENTS ACCESS OUR NODEPORT SERVICE

As I've mentioned previously, before you can access your service through the node port, you need to configure the Google Cloud Platform's firewalls to allow external connections to your nodes on that port. You'll do this now:

```
$ gcloud compute firewall-rules create kubia-svc-rule --allow=tcp:30123
Created [https://www.googleapis.com/compute/v1/projects/kubia-
    1295/global/firewalls/kubia-svc-rule].
NAME            NETWORK  SRC_RANGES  RULES      SRC_TAGS  TARGET_TAGS
kubia-svc-rule  default  0.0.0.0/0   tcp:30123
```

You can access your service through port 30123 of one of the node's IPs. But you need to figure out the IP of at least one of them. Refer to the sidebar on how to do that.

Using JSONPath to get the IPs of all your nodes

You can find the IP in the JSON or YAML descriptors of the nodes. But instead of sifting through the relatively large JSON, tell `kubectl` to print out only the node IP instead of the whole service definition:

```
$ kubectl get nodes -o jsonpath='{.items[*].status.
[CA] addresses[?(@.type=="ExternalIP")].address}'
130.211.97.55 130.211.99.206
```

You're telling `kubectl` to only output the information you want by specifying a JSONPath. You're probably familiar with XPath and how it's used with XML. JSONPath is basically XPath for JSON. The JSONPath in the previous example instructs `kubectl` to do the following:

- Go through all the elements in the `items` attribute.
- For each element, enter the `status` attribute.
- Filter elements of the `addresses` attribute, taking only those that have the `type` attribute set to `ExternalIP`.
- Finally, print the `address` attribute of the filtered elements.

To learn more about how to use JSONPath with `kubectl`, refer to the documentation at http://kubernetes.io/docs/user-guide/jsonpath.

Once you know the IPs of your nodes, you can try accessing your service through them:

```
$ curl http://130.211.97.55:30123
You've hit kubia-ym8or
$ curl http://130.211.99.206:30123
You've hit kubia-xueq1
```

TIP When using Minikube, you can easily access your NodePort services through your browser by running `minikube service <service-name> [-n <namespace>]`.

As you can see, your Pods are now accessible to the whole Internet through port 30123 on any of your nodes. It doesn't matter what node a client sends the request to. But if you only point your clients to the first node, when that node fails, your clients can't access the service anymore. That's why it makes sense to put a load balancer in front of the nodes to make sure you're spreading requests across all healthy nodes and never sending them to a node that's offline at that moment.

If your Kubernetes cluster supports it (which is mostly true when Kubernetes is deployed on cloud infrastructure), the load balancer can be provisioned automatically by creating a LoadBalancer instead of a NodePort service. We'll look at this next.

5.3.2 Exposing a service through an external load balancer

Kubernetes clusters running on cloud providers usually support the automatic provision of a load balancer from the cloud infrastructure. All you need to do is set the

service's type to LoadBalancer instead of NodePort. The load balancer has its own unique, publicly accessible IP address and will redirect all connections to your service. You can access your service through the load balancer's IP address.

If Kubernetes is running in an environment that doesn't support LoadBalancer services, the load balancer will not be provisioned, but the service will still work like a NodePort service. That's because a LoadBalancer service is an extension of a NodePort service. You'll run this example on Google Container Engine, which supports LoadBalancer services. Minikube doesn't, at least not as of this writing.

CREATING A LOADBALANCER SERVICE

To create a service with a load balancer in front, create the service from the following YAML manifest, as shown in the following listing.

Listing 5.12 A LoadBalancer-type service: kubia-svc-loadbalancer.yaml

```
apiVersion: v1
kind: Service
metadata:
 name: kubia-loadbalancer
spec:
 type: LoadBalancer
 ports:
 - port: 80
   targetPort: 8080
 selector:
   app: kubia
```

This type of service obtains a load balancer from the infrastructure hosting the Kubernetes cluster.

The service type is set to `LoadBalancer` instead of `NodePort`. You're not specifying a specific node port, although you could (you're letting Kubernetes choose one instead).

CONNECTING TO THE SERVICE THROUGH THE LOAD BALANCER

After you create the service, it takes time for the cloud infrastructure to create the load balancer and store its IP address in the Service object. Once it does that, the IP address will be listed as the external IP address of your service:

```
$ kubectl get svc kubia-loadbalancer
NAME                 CLUSTER-IP       EXTERNAL-IP      PORT(S)          AGE
kubia-loadbalancer   10.111.241.153   130.211.53.173   80:32143/TCP     1m
```

In this case, the load balancer is available at IP 130.211.53.173, so you can now access the service at that IP address:

```
$ curl http://130.211.53.173
You've hit kubia-xueq1
```

Success! As you may have noticed, this time you didn't need to mess with firewalls the way you had to before with the NodePort service.

Session affinity and web browsers

Because your service is now exposed externally, you may try accessing it with your web browser. You'll see something that may strike you as odd—the browser will hit the exact same Pod every time. Did the service's session affinity change in the meantime? With `kubectl explain`, you can double-check that the service's session affinity is still set to `None`, so why don't different browser requests hit different Pods, as is the case when using `curl`?

Let me explain what's happening. The browser is using *keep-alive* connections and sends all its requests through a single connection, whereas `curl` opens a new connection every time. Services work at the connection level, so when a connection to a service is first opened, a random Pod is selected and then all network packets belonging to that connection are all sent to that single Pod. Even if session affinity is set to `None`, users will mostly hit the same Pod (at least while the connection is kept open).

See figure 5.7 to see how HTTP requests are delivered to the Pod. External clients (`curl` in your case) connect to port 80 of the load balancer and get routed to the implicitly assigned node port on one of the nodes. From there, the connection is forwarded to one of the Pod instances.

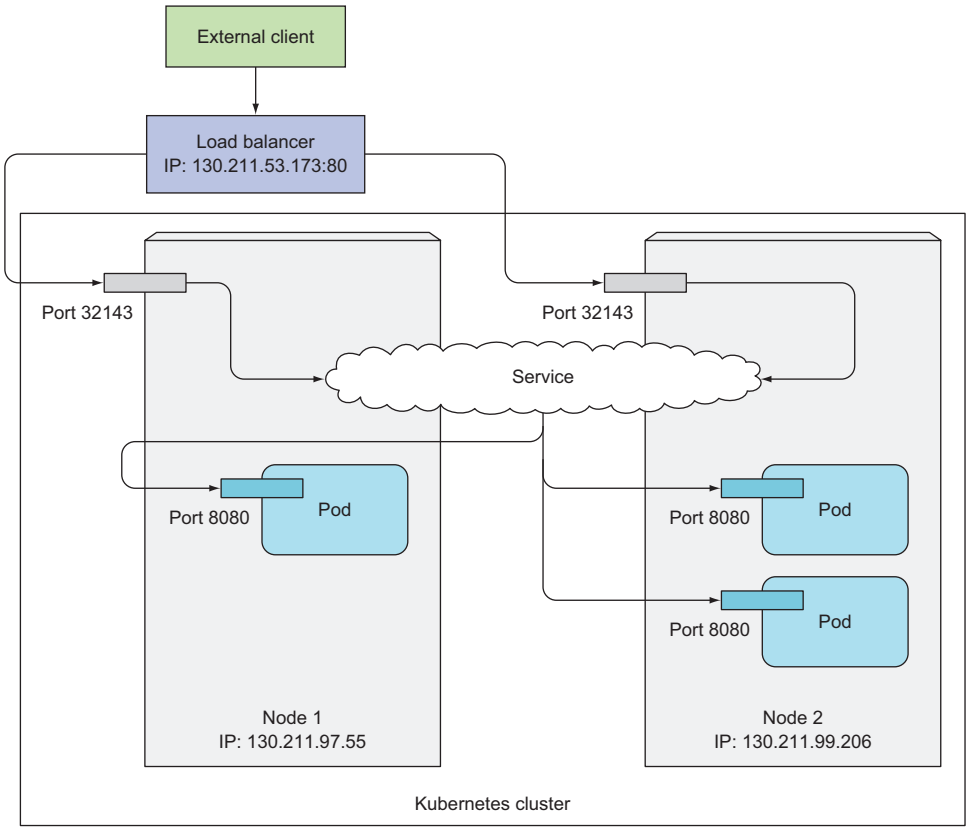


Figure 5.7 An external client connecting to a LoadBalancer service.

As already mentioned, a LoadBalancer-type service is a NodePort service with an additional infrastructure-provided load balancer. If you use `kubectl describe` to see additional info about the service, you'll see that a node port has been selected for the service. If you were to open the firewall for this port, the way you did in the previous section about NodePort services, you could access the service through the node IPs as well.

> **TIP** If you're using Minikube, even though the load balancer will never be provisioned, you can still access the service through the node port (at the Minikube VM's IP address).

5.3.3 Understanding the peculiarities of external connections

You must be aware of several things related to externally-originating connections to services.

UNDERSTANDING AND PREVENTING UNNECESSARY NETWORK HOPS

When an external client connects to a service through the node port (this also includes cases when it goes through the load balancer first), the randomly chosen Pod may or may not be running on the same node that received the connection. An additional network hop is required to reach the Pod, but this may not always be desirable.

You can prevent this additional hop by configuring the service to redirect external traffic only to Pods running on the node that received the connection. This is done by adding the following annotation to the service's metadata section:

```
metadata:
  annotations:
    service.beta.kubernetes.io/external-traffic: OnlyLocal
```

If a service includes this annotation and an external connection is opened through the service's node port, the service proxy will choose a locally running Pod. If no local Pods exist, the connection will hang (it won't be forwarded to a random global Pod, the way connections are when not using the annotation). You therefore need to ensure the load balancer forwards connections only to nodes that have at least one such Pod.

Using this annotation also has other drawbacks. Normally, connections are spread evenly across all the Pods, but when using this annotation, that's no longer the case.

Imagine having two nodes and three Pods. Let's say node A runs one Pod and node B runs the other two. If the load balancer spreads connections evenly across the two nodes, the Pod on node A will receive 50% of all connections, but the two Pods on node B will only receive 25% each, as shown in figure 5.8.

BEING AWARE OF THE NON-PRESERVATION OF THE CLIENT'S IP

Usually, when clients inside the cluster connect to a service, the Pods backing the service can obtain the client's IP address. But when the connection is opened through a node port, the packets' source IP is changed, because Source Network Address Translation (SNAT) is performed on the packets.

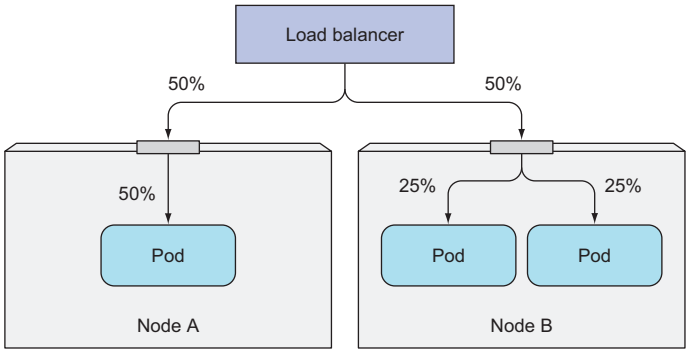


Figure 5.8 A Service annotated as OnlyLocal may lead to uneven load distribution across Pods.

The backing Pod can't see the actual client's IP, which may be a problem for some applications that need to know the client's IP. In the case of a web server, for example, this means the access log won't show the browser's IP.

The `OnlyLocal` annotation described in the previous section affects the preservation of the client's IP, because there's no additional hop between the node receiving the connection and the node hosting the target Pod.

5.4 Exposing services externally through an Ingress resource

You've now seen two ways of exposing a service to clients outside the cluster, but another method exists—creating a *Ingress* resource.

> **DEFINITION** *Ingress* (noun)—The act of going in or entering; the right to enter; a means or place of entering; entryway.

Let me first explain why you need another way to access Kubernetes services from the outside.

Understanding why Ingresses are needed

One important reason is that each LoadBalancer service requires its own load balancer with its own public IP address, whereas an Ingress only requires one, even when providing access to dozens of services. When a client sends an HTTP request to the Ingress, the host and path in the request determine which Service the request is forwarded to, as shown in figure 5.9.

Ingresses operate at the application layer of the network stack (HTTP) and can provide features such as cookie-based session affinity and the like, which Services can't.

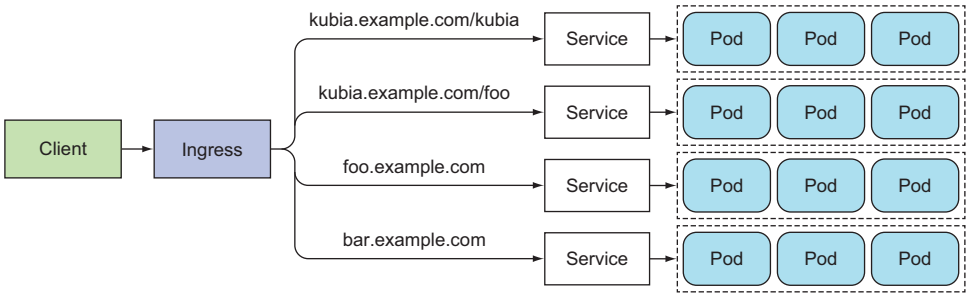


Figure 5.9 **Multiple Services can be exposed through a single Ingress.**

UNDERSTANDING THAT AN INGRESS CONTROLLER IS REQUIRED

Before we go into the features an Ingress object provides, let me emphasize that to make Ingress resources work, an Ingress controller needs to be running in the cluster. Different Kubernetes environments use different implementations of the controller, but several don't provide a default controller at all.

For example, Google Container Engine uses Google Cloud Platform's own HTTP load-balancing features to provide the Ingress functionality. Initially, Minikube didn't provide a controller out of the box, but it now includes an add-on that can be enabled to let you try out the Ingress functionality. Follow the instructions in the sidebar to ensure it's enabled.

Enabling the Ingress add-on in Minikube

If you're using Minikube to run the examples in this book, you'll need to ensure the Ingress add-on is enabled. You can check whether it is by listing all the add-ons:

```
$ minikube addons list
- default-storageclass: enabled
- kube-dns: enabled
- heapster: disabled
- ingress: disabled
- registry-creds: disabled
- addon-manager: enabled
- dashboard: enabled
```

The Ingress add-on isn't enabled.

You'll learn about what these add-ons are throughout the book, but it should be pretty clear what the dashboard and the kube-dns add-ons do. Enable the Ingress add-on so you can see Ingresses in action:

```
$ minikube addons enable ingress
ingress was successfully enabled
```

This should have spun up an Ingress controller as another Pod. Most likely, the controller Pod will be in the kube-system namespace, but not necessarily, so list all the running Pods across all namespaces by using the `--all-namespaces` option:

```
$ kubectl get po --all-namespaces
NAMESPACE     NAME                                READY   STATUS    RESTARTS AGE
default       kubia-rsv5m                         1/1     Running   0        13h
default       kubia-fe4ad                         1/1     Running   0        13h
default       kubia-ke823                         1/1     Running   0        13h
kube-system   default-http-backend-5wb0h           1/1     Running   0        18m
kube-system   kube-addon-manager-minikube         1/1     Running   3        6d
kube-system   kube-dns-v20-101vq                  3/3     Running   9        6d
kube-system   kubernetes-dashboard-jxd91          1/1     Running   3        6d
kube-system   nginx-ingress-controller-gdts0      1/1     Running   0        18m
```

At the bottom of the output, you see the Ingress controller Pod. The name suggests that nginx (an open-source HTTP server and reverse proxy) is used to provide the Ingress functionality.

TIP The `--all-namespaces` option mentioned in the sidebar is handy when you don't know what namespace your Pod (or other type of resource) is in, or if you want to list resources across all namespaces.

5.4.1 Creating an Ingress resource

You've confirmed there's an Ingress controller running in your cluster, so you can now create an Ingress resource. The following listing shows what the YAML manifest for the Ingress looks like.

Listing 5.13 An Ingress resource definition: kubia-ingress.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: kubia
spec:
  rules:
  - host: kubia.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: kubia-nodeport
          servicePort: 80
```

This Ingress maps the kubia.example.com domain name to your service.

All requests will be sent to port 80 of the kubia-nodeport service.

This defines an Ingress with a single rule, which makes sure all HTTP requests to the Ingress controller, in which the host `kubia.example.com` is requested, will be sent to the `kubia-nodeport` service on port `80`.

NOTE Ingress controllers on cloud providers (in GKE, for example) require the Ingress to point to a NodePort service. But that's not a requirement of Kubernetes itself.

5.4.2 Accessing the service through the Ingress

To access your service through http://kubia.example.com, you'll need to make sure the domain name resolves to the IP of the Ingress controller.

OBTAINING THE IP ADDRESS OF THE INGRESS

To look up the IP, you need to list Ingresses:

```
$ kubectl get ingresses
NAME      HOSTS               ADDRESS          PORTS     AGE
kubia     kubia.example.com   192.168.99.100   80        29m
```

> **NOTE** When running on cloud providers, the address may take time to appear, because the Ingress controller provisions a load balancer behind the scenes.

The IP is shown in the `ADDRESS` column.

ENSURING THE HOST CONFIGURED IN THE INGRESS POINTS TO THE INGRESS' IP ADDRESS

Once you know the IP, you can then either configure your DNS servers to resolve kubia.example.com to that IP or you can add the following line to `/etc/hosts` (or `C:\windows\system32\drivers\etc\hosts` on Windows):

```
192.168.99.100    kubia.example.com
```

ACCESSING PODS THROUGH THE INGRESS

Everything is now set up, so you can access the service at http://kubia.example.com (using a browser or `curl`):

```
$ curl http://kubia.example.com
You've hit kubia-ke823
```

You've successfully accessed the service through an Ingress. Let's take a better look at how that unfolded.

UNDERSTANDING HOW INGRESSES WORK

Figure 5.10 shows how the client connected to one of the Pods through the Ingress controller. The client first performed a DNS lookup for kubia.example.com, and the DNS server (or the local operating system) returned the IP of the Ingress controller. The client then sent an HTTP request to the Ingress controller and specified `kubia.example.com` in the `Host` header. From that header, the controller determined which Service the client is trying to access, looked up the Pod IPs through the Endpoints object associated with the Service, and forwarded the client's request to one of the Pods.

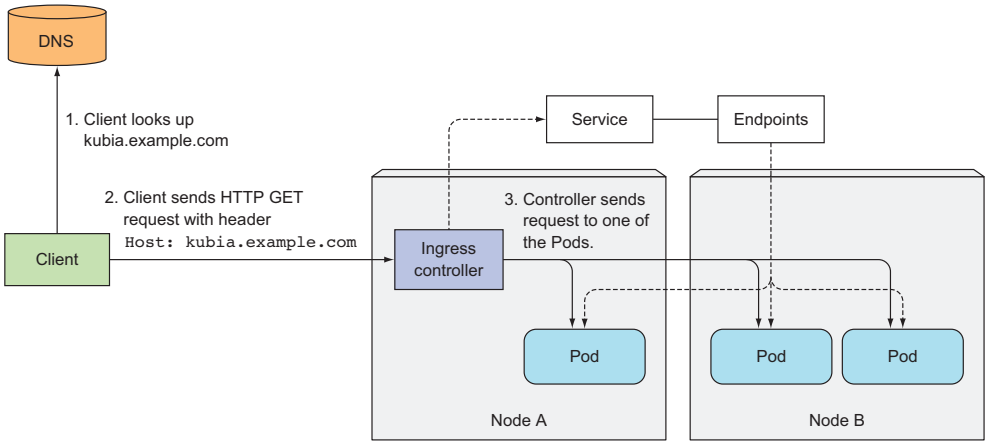


Figure 5.10 **Accessing Pods through an Ingress.**

As you can see, the Ingress controller didn't forward the request to the Service. It only used it to select a Pod. Most, if not all, controllers work like this.

5.4.3 Exposing multiple services through the same Ingress

If you look at the Ingress spec closely, you'll see that both `rules` and `paths` are arrays, so they can contain multiple items. An Ingress can map multiple hosts and paths to multiple Services, as you'll see next. Let's focus on `paths` first.

Mapping different services to different paths of the same host

You can map multiple paths on the same host to different services, as shown in the following listing.

Listing 5.14 Ingress exposing multiple services on same host, but different paths

```
...
  - host: kubia.example.com
    http:
      paths:
      - path: /kubia
        backend:
          serviceName: kubia
          servicePort: 80
      - path: /foo
        backend:
          serviceName: bar
          servicePort: 80
```

Requests to kubia.example.com/kubia will be routed to the kubia service.

Requests to kubia.example.com/bar will be routed to the bar service.

In this case, requests will be sent to two different services, depending on the path in the requested URL. Clients can therefore reach two different services through a single IP address (that of the Ingress controller).

MAPPING DIFFERENT SERVICES TO DIFFERENT HOSTS

Similarly, you can use an Ingress to map to different services based on the host in the HTTP request instead of (only) the path, as shown in the next listing.

Listing 5.15 Ingress exposing multiple services on different hosts

```
spec:
  rules:
  - host: foo.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: foo
          servicePort: 80
  - host: bar.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: bar
          servicePort: 80
```

Requests for foo.example.com will be routed to service foo.

Requests for bar.example.com will be routed to service bar.

Requests received by the controller will be forwarded to either service foo or bar, depending on the `Host` header in the request (exactly how virtual hosts are handled in web servers). DNS needs to point both the foo.example.com and the bar.example.com domain names to the Ingress controller's IP address.

5.4.4 Configuring Ingress to handle TLS traffic

You've seen how an Ingress forwards HTTP traffic. But what about HTTPS? Let's take a quick look at how to configure Ingress to support TLS.

CREATING A TLS CERTIFICATE FOR THE INGRESS

When a client opens a TLS connection to an Ingress controller, the controller terminates the TLS connection. The communication between the client and the controller is encrypted, whereas the communication between the controller and the backend Pod isn't. The application running in the Pod doesn't need to support TLS. For example, if the Pod runs a web server, it can accept only HTTP traffic and let the Ingress controller take care of everything related to TLS. To enable the controller to do that, you need to attach a certificate and a private key to the Ingress. The two need to be stored in a Kubernetes resource called a *Secret*, which is then referenced in the Ingress manifest. We'll explain secrets in detail in chapter 7. For now, you'll create the secret without paying too much attention to it.

First, you need to create the private key and certificate:

```
$ openssl genrsa -out tls.key 2048
$ openssl req -new -x509 -key tls.key -out tls.cert -days 360 -subj
[CA] /CN=kubia.example.com
```

Then you create the secret from the two files like this:

```
$ kubectl create secret tls tls-secret --cert=tls.cert --key=tls.key
secret "tls-secret" created
```

Signing certificates through the CertificateSigningRequest resource

Instead of signing the certificate ourselves, you can get the certificate signed by creating a CertificateSigningRequest (CSR) resource. Users or their applications can create a regular certificate request, put it into a CSR, and then either a human operator or an automated process can approve the request like this:

```
$ kubectl certificate approve <name of the CSR>
```

The signed certificate can then be retrieved from the CSR's `status.certificate` field.

Note that a certificate signer component must be running in the cluster; otherwise creating CertificateSigningRequest and approving or denying them won't have any effect.

The private key and the certificate are now stored in the secret called tls-secret. Now, you can update your Ingress object so it will also accept HTTPS requests for kubia.example.com. The Ingress manifest should now look like the following listing.

Listing 5.16 Ingress handling TLS traffic: kubia-ingress-tls.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: kubia
spec:
  tls:
  - hosts:
    - kubia.example.com
    secretName: tls-secret
  rules:
  - host: kubia.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: kubia-nodeport
          servicePort: 80
```

The whole TLS configuration is under this attribute.

TLS connections will be accepted for the kubia.example.com hostname.

The private key and the certificate should be obtained from the tls-secret you created previously.

TIP Instead of deleting the Ingress and re-creating it from the new file, you can invoke `kubectl apply -f kubia-ingress-tls.yaml`, which updates the Ingress resource with what's specified in the file.

You can now use HTTPS to access your service through the Ingress:

```
$ curl -k -v https://kubia.example.com/kubia
* About to connect() to kubia.example.com port 443 (#0)
...
* Server certificate:
* subject: CN=kubia.example.com
...
> GET /kubia HTTP/1.1
> ...
You've hit kubia-xueq1
```

The command's output shows the response from the app, as well as the server certificate you configured the Ingress with.

> **NOTE** Support for Ingress features varies between the different Ingress controller implementations, so check the implementation specific documentation to know what's supported and what isn't.

Ingresses are a relatively new Kubernetes feature, so you can expect to see many improvements and new features in the future. Although they currently support only L7 (HTTP/HTTPS) load balancing, support for L4 load balancing is also planned.

5.5 Signaling when a Pod is ready to accept connections

There's one more thing we need to cover regarding both services and Ingresses. You've already learned that Pods are included as endpoints of a service if their labels match the service's Pod selector. As soon as a new Pod with proper labels is created, it becomes part of the service and requests start to be redirected to the Pod. But what if the Pod isn't ready to start serving requests immediately?

The Pod may need time to load either configuration or data, or it may need to perform a warm-up procedure to prevent the first user request from taking too long and affecting the user experience. In such cases you don't want the Pod to start receiving requests immediately, especially if Pod instances that are already running can process requests properly and quickly. It makes sense to not forward requests to a Pod that's in the process of starting up until it's fully ready.

5.5.1 Introducing readiness probes

In the previous chapter you learned about liveness probes and how they help keep your apps healthy by ensuring unhealthy containers are restarted automatically. Similar to liveness probes, Kubernetes allows you to also define a *readiness probe* for your Pod.

The readiness probe is invoked periodically and determines whether the specific Pod should receive client requests or not. When a container's readiness probe returns `success`, it's signaling that the container is ready to accept requests.

This notion of being ready is obviously something that's specific to each container. Kubernetes can merely check if the app running in the container responds to a simple `GET /` request or it can hit a specific URL path, which cases the app to perform a

whole list of checks to determine if it's ready. Such a detailed readiness probe, which takes the app's specifics into account, is the app developer's responsibility.

TYPES OF READINESS PROBES

Like liveness probes, three types of readiness probes exist:

- An *Exec* probe, where a process is executed. The container's status is determined by the process' exit status code.
- An *HTTP GET* probe, which sends an HTTP GET request to the container and the HTTP status code of the response determines whether the container is ready or not.
- A *TCP Socket* probe, which opens a TCP connection to a specified port of the container. If the connection is established, the container is considered ready.

UNDERSTANDING THE OPERATION OF READINESS PROBES

When a container is started, Kubernetes can be configured to wait for a configurable amount of time to pass before performing the first readiness check. After that, it invokes the probe periodically and acts based on the result of the readiness probe. If a Pod reports that it's not ready, it's removed from the service. If the Pod then becomes ready again, it's re-added.

Unlike liveness probes, if a container fails the readiness check, it won't be killed or restarted. This is an important distinction between liveness and readiness probes. Liveness probes keep Pods healthy by killing off unhealthy containers and replacing them with new, healthy ones, whereas readiness probes make sure that only Pods that are ready to serve requests receive them. This is mostly necessary during container start up, but it's also useful later after the container has been running for a while.

As you can see in figure 5.11, if a Pod's readiness probe fails, the Pod is removed from the Endpoints object. Clients connecting to the Service will not be redirected to the Pod. The effect is the same as when the Pod doesn't match the Service's label selector at all.

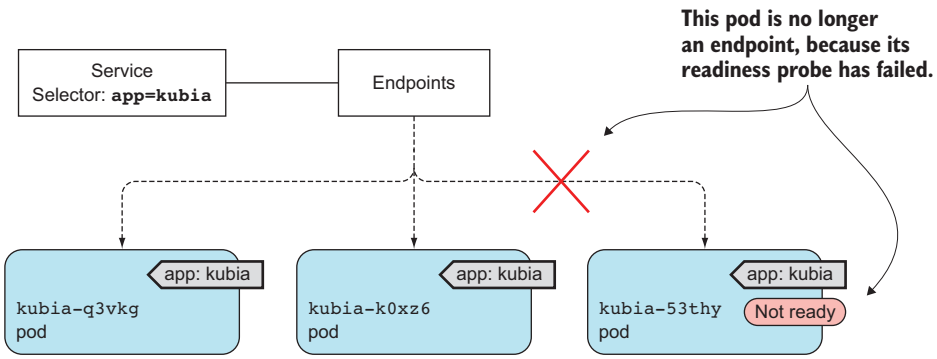


Figure 5.11 A Pod whose readiness probe fails is removed as an endpoint of a Service.

UNDERSTANDING WHY READINESS PROBES ARE IMPORTANT

Imagine that a group of Pods (for example, Pods running application servers) depends on a service provided by another Pod (a backend database, for example). If at any point one of the frontend Pods experiences connectivity problems and can't reach the database anymore, it may be wise for its readiness probe to signal to Kubernetes that the Pod isn't ready to serve any requests at that time. If other Pod instances aren't experiencing the same type of connectivity issues, they can serve requests normally. A readiness probe makes sure clients only talk to those healthy Pods and never notice there anything's wrong with the system.

5.5.2 Adding a readiness probe to a Pod

Next you'll add a readiness probe to your existing Pods by modifying the replication controller's Pod template.

ADDING A READINESS PROBE TO THE POD TEMPLATE

You'll use the `kubectl edit` command to add the probe to the Pod template in your existing ReplicationController:

```
$ kubectl edit rc kubia
```

When the replication controller's YAML opens in the text editor, find the container specification in the Pod template and add the following readiness probe definition to the first container under `spec.template.spec.containers`. The YAML should look like the following listing.

Listing 5.17 RC creating a Pod with a readiness probe: kubia-rc-readinessprobe.yaml

```
apiVersion: v1
kind: ReplicationController
...
spec:
  ...
  template:
    ...
    spec:
      containers:
      - name: kubia
        image: luksa/kubia
        readinessProbe:
          exec:
            command:
            - ls
            - /var/ready
        ...
```

A readinessProbe may be defined for each container in the Pod.

The readiness probe will periodically perform the command `ls /var/ready` inside the container. The `ls` command returns exit code zero if the file exists, or a non-zero exit code otherwise. If the file exists, the readiness probe will succeed; otherwise, it will fail.

The reason you're defining such a strange readiness probe is to change its result by creating or removing the file in question. The file doesn't exist yet, so all the Pods should now report not being ready, right? Well, not exactly. As you may remember from the previous chapter, changing a replication controller's Pod template has no effect on existing Pods.

In other words, all your existing Pods still have no readiness probe defined. You can see this by listing the Pods with `kubectl get pods` and looking at the `READY` column. You need to delete the Pods and have them re-created by the replication controller. The new Pods will fail the readiness check and won't be included as endpoints of the service until you create the `/var/ready` file in each of them.

OBSERVING AND MODIFYING THE PODS' READINESS STATUS

List the Pods again and inspect whether they're ready or not:

```
$ kubectl get po
NAME          READY     STATUS    RESTARTS   AGE
kubia-2r1qb   0/1       Running   0          1m
kubia-3rax1   0/1       Running   0          1m
kubia-3yw4s   0/1       Running   0          1m
```

The `READY` column shows that none of the containers are ready. Now make the readiness probe of one of them start returning success by creating the `/var/ready` file, whose existence makes your mock readiness probe succeed:

```
$ kubectl exec kubia-2r1qb -- touch /var/ready
```

You've used the `kubectl exec` command to execute the `touch` command inside the container of the `kubia-2r1qb` Pod. The `touch` command creates the file if it doesn't yet exist. The Pod's readiness probe command should now exit with status code 0, which means the probe is successful, and the Pod should now be shown as ready. Let's see if it is:

```
$ kubectl get po kubia-2r1qb
NAME          READY     STATUS    RESTARTS   AGE
kubia-2r1qb   0/1       Running   0          2m
```

The Pod still isn't ready. Is there something wrong or is this the expected result? Take a more detailed look at the Pod with `kubectl describe`. The output should contain the following line:

```
Readiness: exec [ls /var/ready] delay=0s timeout=1s period=10s #success=1
     [CA] #failure=3
```

The readiness probe is checked periodically—every 10 seconds by default. The Pod isn't ready because the readiness probe hasn't been invoked yet. But in ten seconds at the latest, the Pod should become ready and its IP should be listed as the only endpoint of the service (run `kubectl get endpoints kubia-loadbalancer` to confirm).

HITTING THE SERVICE WITH THE SINGLE READY POD

You can now hit the service URL a few times to see that each and every request is redirected to this one Pod.

```
$ curl http://130.211.53.173
You've hit kubia-2r1qb
$ curl http://130.211.53.173
You've hit kubia-2r1qb
...
$ curl http://130.211.53.173
You've hit kubia-2r1qb
```

Even though there are three Pods running, only a single Pod is reporting as being ready and is therefore the only Pod receiving requests. If you now delete the file, Pod will be removed from the service again.

5.5.3 Understanding what real-world readiness probes should do

This mock readiness probe is useful only for demonstrating what readiness probes do. In the real world, the readiness probe should switch return success or failure depending on whether the app can (and wants to) receive client requests or not.

Manually removing Pods from services should be performed by either deleting the Pod or changing the Pod's labels instead of manually flipping a switch in the probe.

> **TIP** If you want to add or remove a Pod from a Service manually, add `enabled=true` as a label to your Pods and to the label selector of your Service. Remove the label when you want to remove the Pod from the Service.

ALWAYS DEFINE A READINESS PROBE

Before we conclude this section, there are two final notes about readiness probes that I need to emphasize. First, if you don't add a readiness probe to your Pods, they'll become service endpoints almost immediately. If your application takes too long to start listening for incoming connections, client requests hitting the service will be forwarded to the Pod while it's still starting up and not ready to accept incoming connections. Clients will therefore see "Connection refused" types of errors.

> **TIP** You should always define a readiness probe, even if it's as simple as sending an HTTP request to the base URL.

DON'T INCLUDE POD SHUTDOWN LOGIC INTO YOUR READINESS PROBES

The other thing I need to mention applies to the other end of the Pod's life (Pod shutdown) and is also related to clients experiencing connection errors.

When a Pod is being shut down, the app running within usually stops accepting connections as soon as it receives the termination signal. Because of this, you might think of writing your readiness probe to start failing as soon as the shutdown procedure is initiated, ensuring the Pod is removed from all services it's part of. But that's not necessary, because Kubernetes removes the Pod from all services as soon as you delete the Pod.

5.6 Using a headless service for discovering individual Pods

You've seen how services can be used to provide a stable IP address allowing clients to connect to Pods (or other Endpoints) backing each service. Each connection to the service is forwarded to one randomly selected backing Pod. But what if the client needs to connect to all of those Pods? What if the backing Pods themselves need to each connect to all the other backing Pods? Connecting through the service clearly isn't the way to do this. What is?

For a client to connect to all Pods, it needs to figure out the individual IPs of all those Pods. One option is to have the client call the Kubernetes API server and get the list of Pods and their IP addresses through an API call, but because you should always strive to keep your apps Kubernetes-agnostic, using the API server isn't ideal.

Luckily, Kubernetes allows clients to discover Pod IPs through DNS lookups. Usually, when you perform a DNS lookup for a service, the DNS server returns a single IP—the service's cluster IP. But if you tell Kubernetes you don't need a cluster IP for your service (you do this by setting the `clusterIP` field to `None` in the service specification), the DNS server will then return the actual Pod IPs instead of the single service IP.

Instead of returning a single DNS A record, the DNS server will return multiple A records for the service, each pointing to the IP of an individual Pod backing the service at that moment. Clients can therefore do a simple DNS A record lookup and get the IPs of all the Pods that are part of the service. The client can then use that information to connect to one, many, or all of them.

5.6.1 Creating a headless service

Creating a headless service is like creating any other service, except that you set the service's `clusterIP` property to `None`. In this case, Kubernetes won't assign a cluster IP to the Service.

You'll create a headless service called kubia-headless now. The following listing shows its definition.

Listing 5.18 A headless service: kubia-svc-headless.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: kubia-headless
spec:
  clusterIP: None
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: kubia
```

This makes the service headless.

After you create the service with `kubectl create`, you can inspect it with `kubectl get` and `kubectl describe`. You'll see it has no cluster IP and its Endpoints include (part of) the Pods matching its Pod selector. I say "part of" because your Pods contain a

readiness probe, so only Pods that are ready will be listed as endpoints of the service. Before continuing, please make sure at least two Pods report being ready, by creating the `/var/ready` file, as in the previous example:

```
$ kubectl exec <pod name> -- touch /var/ready
```

5.6.2 Discovering Pods through DNS

With your Pods ready, you can now try performing a DNS lookup to see if you get the actual Pod IPs or not. You'll need to perform the lookup from inside one of the Pods. Unfortunately, your kubia container image doesn't include the nslookup (or the dig) binary, so you can't use it to perform the DNS lookup.

All you're trying to do is perform a DNS lookup from inside a Pod running in the cluster. Why not run a new Pod based on an image that contains the binaries you need? To perform DNS-related actions, you can use the tutum/dnsutils container image, which is available on Docker Hub and contains both the nslookup and the dig binaries. To run the Pod, you can go through the whole process of creating a YAML manifest for it and passing it to `kubectl create`, but that's too much work, right? Luckily, there's a faster way.

RUNNING A POD WITHOUT WRITING A YAML MANIFEST

In chapter 1, you already created Pods without writing a YAML manifest by using the `kubectl run` command. But this time you want to create a single Pod—you don't want to create a ReplicationController. And you don't need to:

```
$ kubectl run dnsutils --image=tutum/dnsutils --generator=run-pod/v1
[CA]--command -- sleep infinity
pod "dnsutils" created
```

The trick is in the `--generator=run-pod/v1` option, which tells `kubectl` to create the Pod directly, without any kind of ReplicationController or similar behind it.

UNDERSTANDING DNS A RECORDS RETURNED FOR A HEADLESS SERVICE

Use the newly created Pod to perform a DNS lookup:

```
$ kubectl exec dnsutils nslookup kubia-headless
...
Name:    kubia-headless.default.svc.cluster.local
Address: 10.108.1.4
Name:    kubia-headless.default.svc.cluster.local
Address: 10.108.2.5
```

The DNS server returns two different IPs for the kubia-headless.default.svc.cluster.local FQDN. Those are the IPs of the two Pods that are reporting being ready. You can confirm this by listing Pods with `kubectl get pods -o wide`, which shows the Pods' IPs.

This is different from what DNS returns for regular (non-headless) services, such as for your kubia service, where the returned IP is the service's cluster IP:

```
$ kubectl exec dnsutils nslookup kubia
...
Name:    kubia.default.svc.cluster.local
Address: 10.111.249.153
```

Although headless services allow you to obtain the actual IPs of Pods and may seem different from regular services, but they aren't that different. Even with a headless service, clients can connect to its Pods by connecting to the service's DNS name, as they can with regular services. But with headless services, because DNS returns the Pods' IPs, clients connect directly to the Pods, instead of through the service proxy.

NOTE A headless services still provides load balancing across Pods, but through the DNS round-robin mechanism.

5.6.3 Discovering all Pods—even those that aren't ready

We've only discussed Pods that are ready become endpoints of services. But sometimes you want to use the service discovery mechanism to find all Pods matching the service's label selector, even those that aren't ready.

Luckily, you don't have to resort to querying the API server. You can use the DNS lookup mechanism to find even those unready Pods. To tell Kubernetes you want all Pods added to a service, regardless of the Pod's readiness status, you must add the following annotation to the service:

```
kind: Service
metadata:
  annotations:
    service.alpha.kubernetes.io/tolerate-unready-endpoints: "true"
```

WARNING As the annotation name suggests, as I'm writing this, this an alpha feature. When this feature graduates to beta, the annotation will be renamed (the word *alpha* will be replaced with *beta*) and then ultimately the annotation will be replaced with a regular field under the service's `spec` property. Check the documentation for details.

5.7 Troubleshooting services

Services are a crucial Kubernetes concept and the source of frustration for many developers. I've seen many developers trying to figure out why they can't connect to their Pods through the service IP or FQDN. For this reason, a short look at how to troubleshoot services is in order.

When you're unable to access your Pods through the service, you should start by checking and keeping in mind the following things:

- First, make sure you're connecting to the service's cluster IP from within the cluster, not from the outside.

- Don't bother pinging the service IP to figure out if the service is accessible (remember, the service's cluster IP is a virtual IP and pinging it will never work).
- If you've defined a readiness probe, make sure it's succeeding; otherwise the Pod won't be part of the service.
- To confirm that a Pod is part of the service, examine the Endpoints object with `kubectl get endpoints`.
- If you're trying to access the service through its FQDN or a part of it (for example, *myservice.mynamespace.svc.cluster.local* or *myservice.mynamespace*) and it doesn't work, see if you can access it using its cluster IP instead of the FQDN.
- Check whether you're connecting to the port exposed by the service and not the target port.
- Try connecting to the Pod IP directly to confirm your Pod is accepting connections on the correct port.
- If you can't even access your app through the Pod's IP, make sure your app isn't only binding to localhost.

This should help you resolve most of your service-related problems. You'll learn much more about how services work in chapter 11. By understanding exactly how they're implemented, you can do an even better job of troubleshooting them.

5.8 Summary

In this chapter, you've learned how to create Kubernetes services to expose actual services in your application system, regardless of the number of Pod instances that are providing each service. You've learned how Kubernetes

- Exposes multiple Pods that match a certain label selector under a single, stable IP address and port
- Makes services accessible from inside the cluster by default, but allows you to make the service accessible from outside the cluster either by setting its type to NodePort or LoadBalancer
- Enables Pods to discover services and their IP addresses and ports by looking up environment variables
- Allows discovery of and communication with services residing outside the cluster by creating a service without a selector and an associated Endpoints resource
- Provides a DNS CNAME alias for external services with the ExternalName service type
- Exposes multiple HTTP services through a single Ingress (consuming a single IP).
- Uses a Pod container's readiness probe to determine whether a Pod should or shouldn't be included as a service endpoint
- Enables discovery of Pod IPs through DNS when you define a headless service.

Along with getting a better understanding of services, you've also learned how to

- Troubleshoot them
- Modify firewall rules in Google Container/Compute Engine
- Execute commands in Pod containers through `kubectl exec`
- Run a bash shell in an existing Pod's container
- Modify Kubernetes resources through the `kubectl apply` command
- Run (only) a Pod with `kubectl run --generator=run-pod/v1`

index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

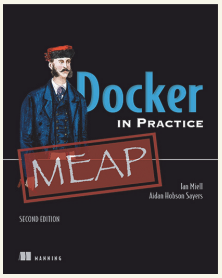
U

V

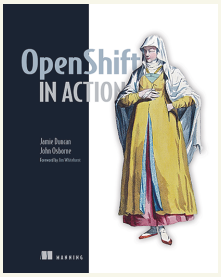
W

Y

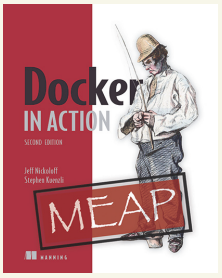
RELATED TITLES



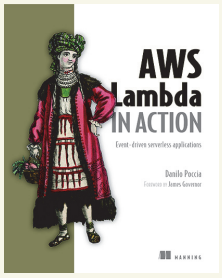
Docker in Practice, Second Edition
by Ian Miell and Aidan Hobson Sayers
ISBN 9781617294808
425 pages
Look Inside



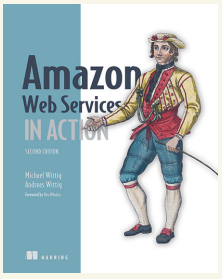
OpenShift in Action
by Jamie Duncan and John Osborne
ISBN 9781617294839
320 pages
Look Inside



Docker in Action, Second Edition
by Jeff Nickoloff and Stephen Kuenzli
ISBN 9781617294761
350 pages
Look Inside

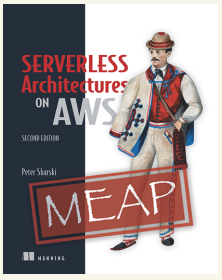


AWS Lambda in Action
by Danilo Poccia
ISBN 9781617293719
384 pages
Look Inside

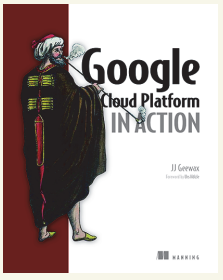


Amazon Web Services in Action, Second Edition
by Michael Wittig and Andreas Wittig
ISBN 9781617295119
528 pages
Look Inside

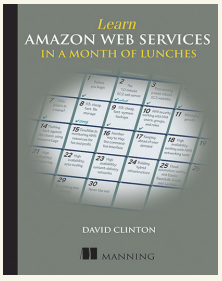
Production-Ready Serverless
by Yan Cui
Duration 10h
Look Inside



Serverless Architectures on AWS, Second Edition
by Peter Sbarski
ISBN 9781617295423
500 pages
Look Inside



Google Cloud Platform in Action
by JJ Geewax
ISBN 9781617293528
632 pages
Look Inside



Learn Amazon Web Services in a Month of Lunches
by David Clinton
ISBN 9781617294440
328 pages
Look Inside

manning