12 The data center as an OS with Docker

This chapter covers

- How to use the official Docker solution for orchestration
- The different ways Mesos can be used to manage Docker containers
- Two heavyweights in the Docker orchestration ecosystem, Kubernetes and OpenShift

If you look back at figure 11.1 in the previous chapter, we're now going to continue moving down the branches of the tree and on to tools that take away some of the detail to increase productivity. Most of these are designed with larger deployments across multiple machines in mind, but there's no reason you can't use them on one machine.

As for the last chapter, we recommend trying to come up with a scenario for each tool, to clarify possible use cases in your environment. We'll continue to give examples along the way as starting points.

12.1 Multi-host Docker

The best process for moving Docker containers to target machines and starting them up is a matter of much debate in the Docker world. A number of well-known companies have created their own ways of doing things and have released them to the world. You can benefit massively from this, if you can decide what tools to use.

This is a fast moving topic—we've seen the birth and death of multiple orchestration tools for Docker, and we recommend caution when considering whether to move over to a brand-new tool. As a result, we've tried to select tools with significant stability or momentum (or both).

TECHNIQUE 87 A seamless Docker cluster with swarm mode

It's great having complete control over your cluster, but sometimes the micromanagement isn't necessary. In fact, if you have a number of applications with no complex requirements, you can take full advantage of the Docker promise of being able to run anywhere—there's no reason you shouldn't be able to throw containers at a cluster and let the cluster decide where to run them.

Swarm mode could be useful for a research lab if the lab were able to split up a computationally intensive problem into bite-size chunks. This would allow them to very easily run their problem on a cluster of machines.

PROBLEM

You have a number of hosts with Docker installed, and you want to be able to start containers without needing to micromanage where they'll run.

SOLUTION

Use swarm mode for Docker, a feature Docker itself has built in to tackle orchestration.

Swarm mode for Docker is the official solution from Docker Inc. to treat a cluster of hosts as a single Docker daemon and deploy services to them. It has a command line quite similar to one you're familiar with from `docker run`. Swarm mode evolved from an official Docker tool that you'd use alongside Docker, and it was integrated into the Docker daemon itself. If you see old references to "Docker Swarm" anywhere, they may be referring to the older tool.

A Docker swarm consists of a number of nodes. Each node may be a manager or a worker, and these roles are flexible and can be changed in the swarm at any time. A manager coordinates the deployment of services to available nodes, whereas workers will only run containers. By default, managers are available to run containers as well, but you'll see how to alter that as well.

When the manager is started, it initializes some state for the swarm and then listens for incoming connections from additional nodes to add to the swarm.

> **NOTE** All versions of Docker used in a swarm must be at least 1.12.0. Ideally you should try to keep all versions exactly the same, or you may encounter issues due to version incompatibilities.

First, let's create a new swarm:

```
h1 $ ip addr show | grep 'inet ' | grep -v 'lo$\|docker0$' # get external IP
    inet 192.168.11.67/23 brd 192.168.11.255 scope global eth0
h1 $ docker swarm init --advertise-addr 192.168.11.67
Swarm initialized: current node (i5vtd3romfl9jg9g4bxtg0kis) is now a
manager.

To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-4blo74l0m2bu5p8synq3w4239vxr1pyoa29cgkrjonx0tuid68
➥ -dhl9o1b62vrhhi0m817r6sxp2 \
    192.168.11.67:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and
follow the instructions.
```

This has created a new swarm and set up the Docker daemon of the host `h1` to be a manager.

You can now inspect your newly created swarm:

```
h1 $ docker info
[...]
Swarm: active
 NodeID: i5vtd3romfl9jg9g4bxtg0kis
 Is Manager: true
 ClusterID: sg6sfmsa96nir1fbwcf939us1
 Managers: 1
 Nodes: 1
 Orchestration:
  Task History Retention Limit: 5
 Raft:
  Snapshot Interval: 10000
  Number of Old Snapshots to Retain: 0
  Heartbeat Tick: 1
  Election Tick: 3
 Dispatcher:
  Heartbeat Period: 5 seconds
 CA Configuration:
  Expiry Duration: 3 months
 Node Address: 192.168.11.67
 Manager Addresses:
  192.168.11.67:2377
[...]
h1 $ docker node ls
$ docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
i5vtd3romfl9jg9g4bxtg0kis *  h1        Ready   Active        Leader
```

You can now make a Docker daemon on a different host join as a worker by running the command specified after the manager started:

```
h2 $ docker swarm join \
    --token SWMTKN-1-4blo74l0m2bu5p8synq3w4239vxr1pyoa29cgkrjonx0tuid68
➥ -dhl9o1b62vrhhi0m817r6sxp2 \
    192.168.11.67:2377
This node joined a swarm as a worker.
```

`h2` has now been added to our cluster as a worker. Running `docker info` on either host will reveal that the `Nodes` count has gone up to 2, and `docker node ls` will list both nodes.

Finally, let's start a container. In swarm mode, this is referred to as deploying a service, because there are additional features that don't make sense with a container. Before deploying the service, we'll mark the manager as having availability `drain`—by default, all managers are available to run containers, but in this technique we want to demonstrate remote machine scheduling capabilities, so we'll constrain things to avoid the manager. Drain will cause any containers already on the node to be redeployed elsewhere, and no new services will be scheduled on the node.

```
h1 $ docker node update --availability drain i5vtd3romfl9jg9g4bxtg0kis
h1 $ docker service create --name server -d -p 8000:8000 ubuntu:14.04 \
       python3 -m http.server 8000
vp0fj8p9khzh72eheoye0y4bn
h1 $ docker service ls
ID            NAME    MODE        REPLICAS  IMAGE         PORTS
vp0fj8p9khzh  server  replicated  1/1       ubuntu:14.04  *:8000->8000/tcp
```

There are a few things to note here. The most important is that the swarm has automatically selected a machine to start the container on—if you had multiple workers, the manager would choose one based on load balancing. You probably also recognize some of the arguments to `docker service create` as familiar from `docker run`—a number of arguments are shared, but it's worth reading the documentation. For example, the `--volume` argument to `docker run` has a different format in the `--mount` argument that you should read the documentation for.

It's now time to check and see if our service is up and running:

```
h1 $ docker service ps server
ID            NAME      IMAGE         NODE  DESIRED STATE  CURRENT STATE
➥          ERROR  PORTS
mixc9w3frple  server.1  ubuntu:14.04  h2    Running        Running 4
minutes ago
h1 $ docker node inspect --pretty h2 | grep Addr
 Address:               192.168.11.50
h1 $ curl -sSL 192.168.11.50:8000 | head -n4
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
➥ "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ascii">
```

Swarm mode has a piece of additional functionality it enables by default, called the *routing mesh*. This allows each node in the swarm to appear as if it can serve requests for all services within the swarm that have published ports—any incoming connections are forwarded to an appropriate node.

For example, if you go back on the `h1` manager node again (which we know isn't running the service, because it has availability `drain`), it will still respond on port 8000 to any requests:

```
h1 $ curl -sSL localhost:8000 | head -n4
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
➥ "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ascii">
```

This can be particularly useful for a simple kind of service discovery—as long as you know the address of one node, you can access all your services very easily.

Once you're finished with the swarm, you can shut down all services and delete the cluster.

```
$ docker service rm server
server
$ docker swarm leave
Error response from daemon: You are attempting to leave the swarm on a >
node that is participating as a manager. Removing the last manager erases >
all current state of the swarm. Use `--force` to ignore this message.
$ docker swarm leave --force
Node left the swarm.
```

As you can see here, swarm mode will warn you if you're shutting down the last manager in a node, because all information on the swarm will be lost. You can override this warning with `--force`. You'll need to run `docker swarm leave` on all worker nodes as well.

DISCUSSION

This has been a brief introduction to swarm mode in Docker, and there's a lot we haven't covered here. For example, you may have noticed that the help text after we initialized the swarm mentioned the ability to connect additional masters to the swarm—this is useful for resilience. Additional subjects of interest are built-in pieces of functionality that store service configuration information (as you did with etcd in technique 74), using constraints to guide placement of containers, and information on how to upgrade containers with rollbacks on failure. We recommend you refer to the official documentation at https://docs.docker.com/engine/swarm/ for more information.

TECHNIQUE 88 Using a Kubernetes cluster

You've now seen two extremes in approaches to orchestration—the conservative approach of Helios and the much more free-form approach of Docker swarm. But some users and companies will expect a little more sophistication from their tooling.

This need for customizable orchestration can be fulfilled by many options, but there are a few that are used and discussed more than the others. In one case, that's undoubtedly partially due to the company behind it, but one would hope that Google knows how to build orchestration software.

PROBLEM

You want to manage Docker services across hosts.

SOLUTION

Use Kubernetes and its powerful abstractions to manage your fleet of containers.

Kubernetes, a tool created by Google, is for companies that prefer to have clear guidance and best practices on how to arrange applications and state relationships between them. It allows you to use specially designed tools to manage a dynamic infrastructure based on a specified structure.

Before we get going with Kubernetes, let's take a quick look at Kubernetes' high-level architecture in figure 12.1.

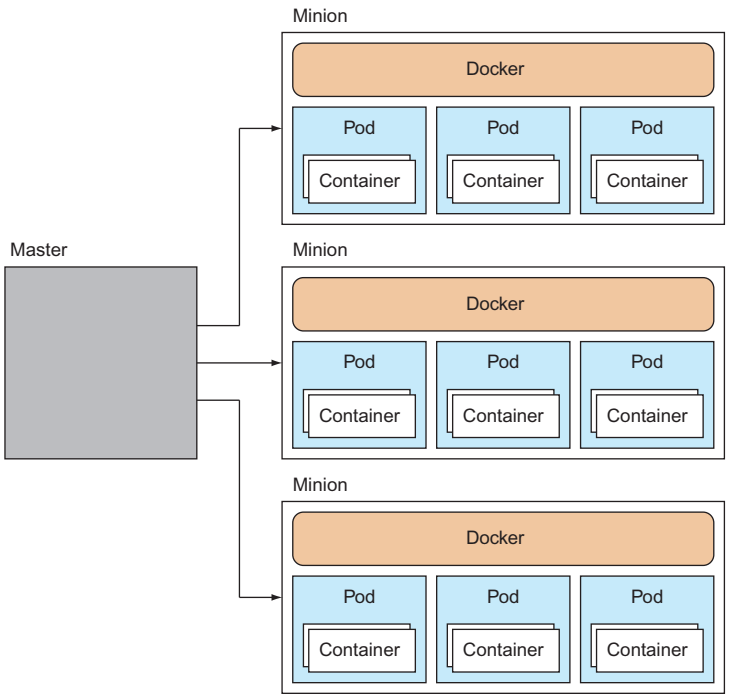


Figure 12.1 Kubernetes high-level view

Kubernetes has a master-minion architecture. Master nodes are responsible for receiving orders about what should be run on the cluster and orchestrating its resources. Each minion has Docker installed on it, along with a *kubelet* service, which manages the pods (sets of containers) running on each node. Information about the cluster is maintained in etcd, a distributed key/value data store (see technique 74), and this is the cluster's source of truth.

> **TIP** We'll go over it again later in this technique, so don't worry about it too much now, but a *pod* is a grouping of related containers. The concept exists to facilitate the management and maintenance of Docker containers.

The end goal of Kubernetes is to make running your containers at scale a simple matter of declaring what you want and letting Kubernetes ensure the cluster meets your needs. In this technique you'll see how to scale a simple service to a given size by running one command.

> **NOTE** Kubernetes was originally developed by Google as a means for managing containers at scale. Google has been running containers for over a decade at scale, and it decided to develop this container orchestration system when Docker became popular. Kubernetes builds on the lessons learned from Google's extensive experience. Kubernetes is also known as "K8s."

A full treatment of Kubernetes' installation, setup, and features is a big and fast-changing topic that's beyond the scope of this book (and no doubt a book in itself, before too long). Here we're going to focus on Kubernetes' core concepts and set up a simple service so you can get a feel for it.

Installing Kubernetes

You can either install Kubernetes directly on your host via Minikube, which will give you a single-minion cluster, or use Vagrant to install a multi-minion cluster managed with VMs. In this technique we'll focus on the first option—the latter is best achieved with research to identify the correct option for the latest version of Kubernetes.

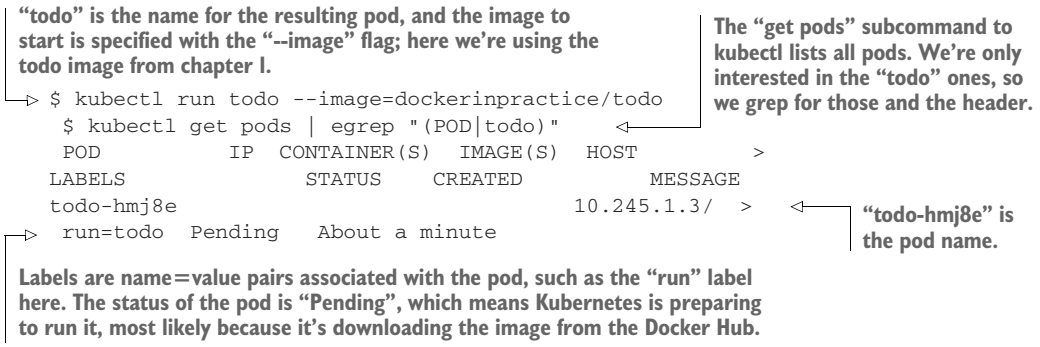
The recommended approach for getting started locally with Kubernetes is to install a single-minion cluster on your host by following the official documentation for Minikube at https://kubernetes.io/docs/tasks/tools/install-minikube/.

Minikube is a specialized tool from the Kubernetes project created to ease the process of local development, but it's currently a bit limited. If you want to stretch yourself a bit more, we recommend searching for a guide to setting up a multi-node Kubernetes cluster with Vagrant—this process tends to change with the Kubernetes version, so we won't give specific advice here (though, at time of writing, we found https://github.com/Yolean/kubeadm-vagrant to be a reasonable starting point).

Once you have Kubernetes installed, you can follow along from here. The following output will be based on a multi-node cluster. We're going to start by creating a single container and using Kubernetes to scale it up.

SCALING A SINGLE CONTAINER

The command used to manage Kubernetes is `kubectl`. In this case you're going to use the `run` subcommand to run a given image as a container within a pod.



Kubernetes picks a pod name by taking the name from the `run` command (`todo` in the preceding example), adding a dash and adding a random string. This ensures it doesn't clash with other pod names.

After waiting a few minutes for the todo image to download, you'll eventually see that its status has changed to "Running":

```
$ kubectl get pods | egrep "(POD|todo)"
POD          IP           CONTAINER(S)    IMAGE(S)                >
HOST                      LABELS               STATUS    CREATED          MESSAGE
todo-hmj8e  10.246.1.3                                             >
10.245.1.3/10.245.1.3  run=todo  Running   4 minutes
                          todo           dockerinpractice/todo  >
                                              Running   About a minute
```

This time the IP, CONTAINER(S), and IMAGE(S) columns are populated. The IP column gives the address of the pod (in this case `10.246.1.3`), and the container column has one row per container in the pod (in this case we have only one, `todo`).

You can test that the container (todo) is indeed up and running and serving requests by hitting the IP address and port directly:

```
$ wget -qO- 10.246.1.3:8000
<html manifest="/todo.appcache">
[...]
```

At this point we haven't seen much difference from running a Docker container directly. To get your first taste of Kubernetes, you can scale up this service by running a `resize` command:

```
$ kubectl resize --replicas=3 replicationController todo
resized
```

This command tells Kubernetes that you want the todo replication controller to ensure that there are three instances of the todo app running across the cluster.

> **TIP** A replication controller is a Kubernetes service that ensures that the right number of pods is running across the cluster.

You can check that the additional instances of the todo app have been started with the `kubectl get pods` command:

```
$ kubectl get pods | egrep "(POD|todo)"
POD          IP            CONTAINER(S)   IMAGE(S)                >
HOST                       LABELS              STATUS    CREATED         MESSAGE
todo-2ip3n   10.246.2.2                                           >
10.245.1.4/10.245.1.4   run=todo  Running   10 minutes
                         todo           dockerinpractice/todo   >
                                            Running   8 minutes
todo-4os5b   10.246.1.3                                           >
10.245.1.3/10.245.1.3   run=todo  Running   2 minutes
                         todo           dockerinpractice/todo   >
                                            Running   48 seconds
todo-cuggp   10.246.2.3                                           >
10.245.1.4/10.245.1.4   run=todo  Running   2 minutes
                         todo           dockerinpractice/todo   >
                                            Running   2 minutes
```

Kubernetes has taken the `resize` instruction and the todo replication controller and ensured that the right number of pods is started up. Notice that it placed two on one host (10.245.1.4) and one on another (10.245.1.3). This is because Kubernetes' default scheduler has an algorithm that spreads pods across nodes by default.

> **TIP** A scheduler is a piece of software that decides where and when items of work should be run. For example, the Linux kernel has a scheduler that decides what task should be run next. Schedulers range from the stupidly simple to the incredibly complex.

You've started to see how Kubernetes can make managing containers easier across multiple hosts. Next we'll dive into the core Kubernetes concept of pods.

USING PODS

A *pod* is a collection of containers that are designed to work together in some way and that share resources.

Each pod gets its own IP address and shares the same volumes and network port range. Because a pod's containers share a localhost, the containers can rely on the different services being available and visible wherever they're deployed.

Figure 12.2 illustrates this with two containers that share a volume. In the figure, container 1 might be a web server that reads data files from the shared volume, which is in turn updated by container 2. Both containers are therefore stateless; state is stored in the shared volume.

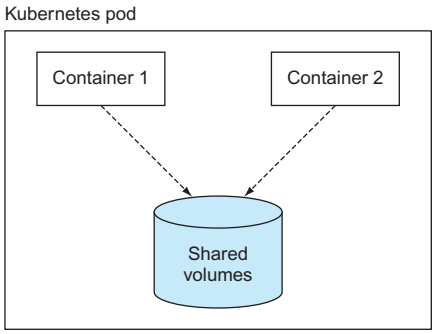


Figure 12.2 A two-container pod

This design of separated responsibilities facilitates a microservices approach by allowing you to manage each part of your service separately. You can upgrade one container within a pod without needing to be concerned with the others.

The following pod specification defines a complex pod with one container that writes random data (`simplewriter`) to a file every 5 seconds, and another container that reads from the same file. The file is shared via a volume (`pod-disk`).

Listing 12.1 complexpod.json

```
{
  "id": "complexpod",
  "kind": "Pod",
  "apiVersion": "v1beta1",
  "desiredState": {
    "manifest": {
      "version": "v1beta1",
      "id": "complexpod",
      "containers": [{
        "name": "simplereader",
        "image": "dockerinpractice/simplereader",
        "volumeMounts": [{
          "mountPath": "/data",
          "name": "pod-disk"
        }]
      },{
        "name": "simplewriter",
        "image": "dockerinpractice/simplewriter",
        "volumeMounts": [{
          "mountPath": "/data",
          "name": "pod-disk"
        }]
      }],
      "volumes": [{
        "name": "pod-disk",
        "emptydir": {}
      }]
    }
  }
}
```

Gives the entity a name

Specifies the type of object this is

Specifies to Kubernetes the version the JSON is targeting

The meat of the pod's specification is in the "desiredState" and "manifest" attributes.

Gives the entity a name

Details of the containers in the pod are stored in this JSON array.

Volume mount points are specified for each container.

The volume mount name refers to the name in the pod manifest's "volumes" definition.

Each container has a name for reference, and the Docker image is specified in the "image" attribute.

The mount path is the path to the volume mounted on the filesystem of the container. This could be set to a different location for each container.

Volume mount points are specified for each container.

The mount path is the path to the volume mounted on the filesystem of the container. This could be set to a different location for each container.

The volume mount name refers to the name in the pod manifest's "volumes" definition.

The "volumes" attribute defines the volumes created for this pod.

The name of the volume is referred to in the previous "volumeMounts" entries.

A temporary directory that shares a pod's lifetime

To load this pod specification, create a file with the preceding listing and run this command:

```
$ kubectl create -f complexpod.json
pods/complexpod
```

After waiting a minute for the images to download, you'll see the log output of the container by running `kubectl log` and specifying first the pod and then the container you're interested in.

```
$ kubectl log complexpod simplereader
2015-08-04T21:03:36.535014550Z '? U
[2015-08-04T21:03:41.537370907Z] h(^3eSk4y
[2015-08-04T21:03:41.537370907Z] CM(@
[2015-08-04T21:03:46.542871125Z] qm>5
[2015-08-04T21:03:46.542871125Z] {Vv_
[2015-08-04T21:03:51.552111956Z] KH+74    f
[2015-08-04T21:03:56.556372427Z] j?p+!\
```

DISCUSSION

We've only scratched the surface of Kubernetes' capabilities and potential here, but this should give you a sense of what can be done with it and how it can make orchestrating Docker containers simpler.

The next technique looks at directly taking advantage of some more features of Kubernetes. Kubernetes is also used behind the scenes as an orchestration engine by OpenShift in techniques 90 and 99.

TECHNIQUE 89 Accessing the Kubernetes API from within a pod

Often it's possible for pods to operate completely independently from each other, not even knowing that they're running as part of a Kubernetes cluster. But Kubernetes does provide a rich API, and giving containers access to this opens the door to introspection and adaptive behavior, as well as the ability for containers to manage the Kubernetes cluster themselves.

PROBLEM

You want to access the Kubernetes API from within a pod.

SOLUTION

Use `curl` to access the Kubernetes API from within a container in a pod, using authorization information made available to the container.

This is one of the shorter techniques in the book, but it contains a lot to unpack. This is one of the reasons it's a useful technique to study. Among other things, we'll cover

- The `kubectl` command
- Starting Kubernetes pods
- Accessing Kubernetes pods
- A Kubernetes anti-pattern

- Bearer tokens
- Kubernetes secrets
- The Kubernetes "downwards API"

NO KUBERNETES CLUSTER?

If you don't have access to a Kubernetes cluster, you have a few options. There are many cloud providers that offer pay-as-you-go Kubernetes clusters. For the fewest dependencies, though, we recommend using Minikube (mentioned in the last technique), which doesn't require a credit card.

For information on how to install Minikube, see the documentation at https://kubernetes.io/docs/tasks/tools/install-minikube/.

CREATING A POD

First you're going to create a container within the fresh `ubuntu` pod using the `kubectl` command, and then you'll access a shell within that container on the command line. (`kubectl run` currently imposes a 1-1 relationship between pods and containers, though pods are more flexible than this in general.)

Listing 12.2 Creating and setting up a container

The kubectl command using the -ti flag, naming the pod "ubuntu", using the by-now familiar ubuntu:16.04 image, and telling Kubernetes not to restart once the pod/container has exited

Kubectl helpfully tells you that your terminal may not show you the prompt unless you press Enter.

```
$ kubectl run -it ubuntu --image=ubuntu:16.04 --restart=Never
 If you don't see a command prompt, try pressing enter.
 root@ubuntu:/# apt-get update -y && apt-get install -y curl
 [...]
root@ubuntu:/
```

This is the prompt from within the container that you'll see if you press Enter, and we're updating the container's package system and installing curl.

Once the install is complete, the prompt is returned.

You're now in the container created by the `kubectl` command, and you've ensured that curl is installed.

WARNING Accessing and modifying a pod from a shell is considered a Kubernetes anti-pattern. We use it here to demonstrate what is possible from within a pod, rather than how pods should be used.

Listing 12.3 Access the Kubernetes API from a pod

Uses the curl command to access the Kubernetes API. The -k flag allows curl to work without certificates being deployed on the client, and the HTTP method used to talk to the API is specified as GET by the -X flag.

The -H flag adds an HTTP header to the request. This is an authentication token discussed shortly.

```
root@ubuntu:/# $ curl -k -X GET \
    -H "Authorization: Bearer \
    $(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" <3> \
   https://${KUBERNETES_PORT_443_TCP_ADDR}:${KUBERNETES_SERVICE_PORT_HTTPS}
  {
```

The URL to contact is constructed from environment variables available within the pod.

```
  "paths": [
     "/api",
    "/api/v1",
    "/apis",
    "/apis/apps",
    "/apis/apps/v1beta1",
    "/apis/authentication.k8s.io",
    "/apis/authentication.k8s.io/v1",
    "/apis/authentication.k8s.io/v1beta1",
    "/apis/authorization.k8s.io",
    "/apis/authorization.k8s.io/v1",
    "/apis/authorization.k8s.io/v1beta1",
    "/apis/autoscaling",
    "/apis/autoscaling/v1",
    "/apis/autoscaling/v2alpha1",
    "/apis/batch",
    "/apis/batch/v1",
    "/apis/batch/v2alpha1",
    "/apis/certificates.k8s.io",
    "/apis/certificates.k8s.io/v1beta1",
    "/apis/extensions",
    "/apis/extensions/v1beta1",
    "/apis/policy",
    "/apis/policy/v1beta1",
    "/apis/rbac.authorization.k8s.io",
    "/apis/rbac.authorization.k8s.io/v1alpha1",
    "/apis/rbac.authorization.k8s.io/v1beta1",
    "/apis/settings.k8s.io",
    "/apis/settings.k8s.io/v1alpha1",
    "/apis/storage.k8s.io",
    "/apis/storage.k8s.io/v1",
    "/apis/storage.k8s.io/v1beta1",
    "/healthz",
    "/healthz/ping",
    "/healthz/poststarthook/bootstrap-controller",
    "/healthz/poststarthook/ca-registration",
    "/healthz/poststarthook/extensions/third-party-resources",
    "/logs",
    "/metrics",
    "/swaggerapi/",
    "/ui/",
    "/version"
  ]
}
root@ubuntu:/# curl -k -X GET -H "Authorization: Bearer $(cat
➥ /var/run/secrets/kubernetes.io/serviceaccount/token)"
➥ https://${KUBERNETES_PORT_443_TCP_ADDR}:
➥ ${KUBERNETES_SERVICE_ORT_HTTPS}/version
 {
  "major": "1",
   "minor": "6",
  "gitVersion": "v1.6.4",
  "gitCommit": "d6f433224538d4f9ca2f7ae19b252e6fcb66a3ae",
  "gitTreeState": "dirty",
  "buildDate": "2017-06-22T04:31:09Z",
```

The default response for the API is to list the paths it offers for consumption.

Another request is made, this time to the /version path.

The response to the /version request is to specify the version of Kubernetes that's running.

```
  "goVersion": "go1.7.5",
  "compiler": "gc",
  "platform": "linux/amd64"
}
```

The preceding listing covered a lot of new material, but we hope it gives a flavor of what can be done within Kubernetes pods dynamically, without any setup.

The key point to take from this listing is that information is made available to users within the pod, allowing the pod to make contact with the Kubernetes API. These items of information are collectively called the "downward API." At present, the downward API consists of two classes of data: environment variables, and files exposed to the pod.

A file is used in the preceding example to provide an authentication token to the Kubernetes API. This token is made available in the file /var/run/secrets/kubernetes.io/serviceaccount/token. In listing 12.3, this file is run through `cat`, and the output of the `cat` command is supplied as part of the `Authorization:` HTTP header. This header specifies that the authorization used is of the `Bearer` type, and the bearer token is the output of `cat`, so the `-H` argument to `curl` is as follows:

```
-H "Authorization: Bearer
➥ $(cat /var/run/secrets/kubernetes.io/serviceaccount/token)"
```

> **NOTE** *Bearer tokens* are an authentication method that requires only that a specified token is given—no identification is required beyond that (such as username/password). *Bearer shares* operate on a similar principle, where the bearer of the shares is the one who has the right to sell them. Cash money works the same way—indeed on UK cash the notes have the phrase "I promise to pay the bearer on demand the sum of …"

The downward API items exposed are a form of Kubernetes "secret." Any secret can be created using the Kubernetes API and exposed via a file in a pod. This mechanism allows for the separation of secrets from Docker images and Kubernetes pod or deployment configuration, meaning that permissions can be handled separately from those more open items.

DISCUSSION

It's worth paying attention to this technique, as it covers a lot of ground. The key point to grasp is that Kubernetes pods have information made available to them that allows them to interact with the Kubernetes API. This allows applications to run within Kubernetes that monitor and act on activities going on around the cluster. For example, you might have an infrastructure pod that watches the API for newly sprung-up pods, investigates their activities, and records that data somewhere else.

Although role-based access control (RBAC) is outside the scope of this book, it's worth mentioning that this has implications for security, as you don't necessarily want just any user of your cluster to have this level of access. Therefore, parts of the API will require more than just a bearer token to gain access.

These security-related considerations make this technique related half to Kubernetes and half to security. Either way, this is an important technique for anyone looking to use Kubernetes "for real," to help them understand how the API works and how it potentially can be abused.

TECHNIQUE 90 Using OpenShift to run AWS APIs locally

One of the big challenges with local development is testing an application against other services. Docker can help with this if the service can be put in a container, but this leaves the large world of external third-party services unsolved.

A common solution is to have test API instances, but these often provide fake responses—a more complete test of functionality isn't possible if an application is built around a service. For example, imagine you want to use AWS S3 as an upload location for your application, where it then processes the uploads—testing this will cost money.

PROBLEM

You want to have AWS-like APIs available locally to develop against.

SOLUTION

Set up LocalStack and use the available AWS service equivalents.

In this walkthrough you're going to set up an OpenShift system using Minishift, and then run LocalStack in a pod on it. OpenShift is a RedHat-sponsored wrapper around Kubernetes that provides extra functionality more suited to enterprise production deployments of Kubernetes.

In this technique we'll cover

- The creation of routes in OpenShift
- Security context constraints
- Differences between OpenShift and Kubernetes
- Testing AWS services using public Docker images

NOTE To follow this technique you'll need to install Minishift. Minishift is similar to Minikube, which you saw in technique 89. The difference is that it contains an installation of OpenShift (covered comprehensively in technique 99).

LocalStack

LocalStack is a project that aims to give you as complete as possible a set of AWS APIs to develop against without incurring any cost. This is great for testing or trying code out before running it for real against AWS and potentially wasting time and money.

LocalStack spins up the following core Cloud APIs on your local machine:

- API Gateway at http://localhost:4567
- Kinesis at http://localhost:4568
- DynamoDB at http://localhost:4569
- DynamoDB Streams at http://localhost:4570
- Elasticsearch at http://localhost:4571

- S3 at http://localhost:4572
- Firehose at http://localhost:4573
- Lambda at http://localhost:4574
- SNS at http://localhost:4575
- SQS at http://localhost:4576
- Redshift at http://localhost:4577
- ES (Elasticsearch Service) at http://localhost:4578
- SES at http://localhost:4579
- Route53 at http://localhost:4580
- CloudFormation at http://localhost:4581
- CloudWatch at http://localhost:4582

LocalStack supports running in a Docker container, or natively on a machine. It's built on Moto, which is a mocking framework in turn built on Boto, which is a Python AWS SDK.

Running within an OpenShift cluster gives you the capability to run many of these AWS API environments. You can then create distinct endpoints for each set of services, and isolate them from one another. Also, you can worry less about resource usage, as the cluster scheduler will take care of that. But LocalStack doesn't run out of the box, so we'll guide you through what needs to be done to get it to work.

ENSURING MINISHIFT IS SET UP

At this point we assume you have Minishift set up—you should look at the official documentation on getting started at https://docs.openshift.org/latest/minishift/getting-started/index.html.

Listing 12.4 Check Minishift is set up OK

```
$ eval $(minishift oc-env)
$ oc get all
No resources found.
```

CHANGING THE DEFAULT SECURITY CONTEXT CONSTRAINTS

Security context constraints (SCCs) are an OpenShift concept that allows more granular control over Docker containers' powers. They control SELinux contexts (see technique 100), can drop capabilities from running containers (see technique 93), can determine which user the pod can run as, and so on.

To get this running, you're going to change the default `restricted` SCC. You could also create a separate SCC and apply it to a particular project, but you can try that on your own.

To change the 'restricted` SCC, you'll need to become a cluster administrator:

```
$ oc login -u system:admin
```

Then you need to edit the restricted SCC with the following command:

```
$ oc edit scc restricted
```

You'll see the definition of the `restricted` SCC.

At this point you're going to have to do two things:

- Allow containers to run as any user (in this case `root`)
- Prevent the SCC from restricting your capabilities to setuid and setgid

ALLOWING RUNASANY

The LocalStack container runs as root by default, but for security reasons, OpenShift doesn't allow containers to run as root by default. Instead it picks a UID within a very high range, and runs as that UID. Note that UIDs are numbers, as opposed to usernames, which are strings mapped to a UID.

To simplify matters, and to allow the LocalStack container to run as root, change these lines,

```
runAsUser:
 type: MustRunAsRange
```

to read as follows:

```
runAsUser:
 type: RunAsAny
```

This allows containers to run as *any* user, and not within a range of UIDs.

ALLOWING SETUID AND SETGID CAPABILITIES

When LocalStack starts up, it needs to become another user to start ElastiCache. The ElastiCache service doesn't start up as the root user.

To get around this, LocalStack `su`'s the startup command to the LocalStack user in the container. Because the `restricted` SCC explicitly disallows actions that change your user or group ID, you need to remove these restrictions. Do this by deleting these lines:

```
- SETUID
- SETGID
```

SAVING THE FILE

Once you've completed those two steps, save the file.

Make a note of the host. If you run this command,

```
$ minishift console --machine-readable | grep HOST | sed 's/^HOST=\(.*\)/\1/'
```

you'll get the host that the Minishift instance is accessible as from your machine. Note this host, as you'll need to substitute it in later.

DEPLOYING THE POD

Deploying the LocalStack is as easy as running this command:

```
$ oc new-app localstack/localstack --name="localstack"
```

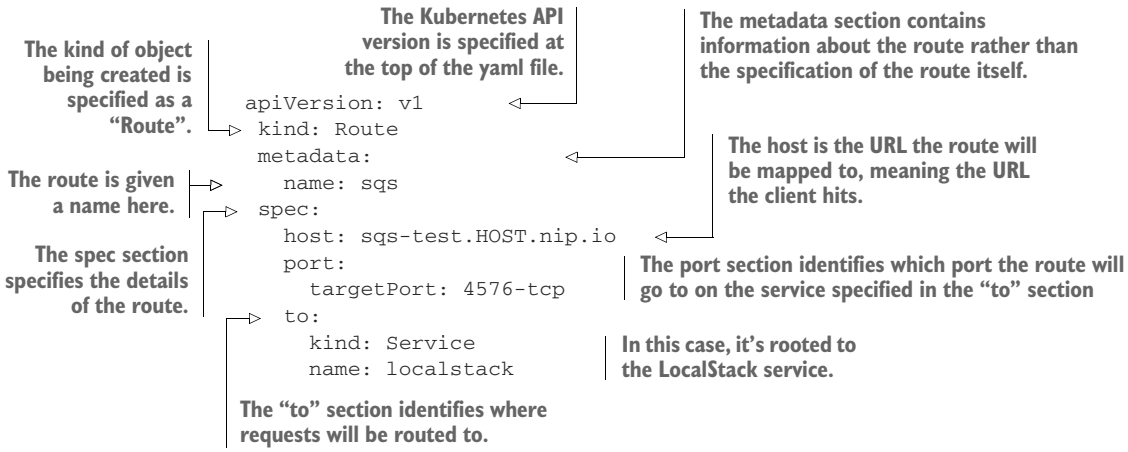
> **NOTE** If you want to take a deeper look at the localstack image, it's available at https://github.com/localstack/localstack.

This takes the localstack/localstack image and creates an OpenShift application around it for you, setting up internal services (based on the exposed ports in the LocalStack Docker image's Dockerfile), running the container in a pod, and performing various other management tasks.

CREATING THE ROUTES

If you want to access the services from outside, you need to create OpenShift routes, which create an external address for accessing services within the OpenShift network. For example, to create a route for the SQS service, create a file like the following, called route.yaml:

Listing 12.5 route.yaml

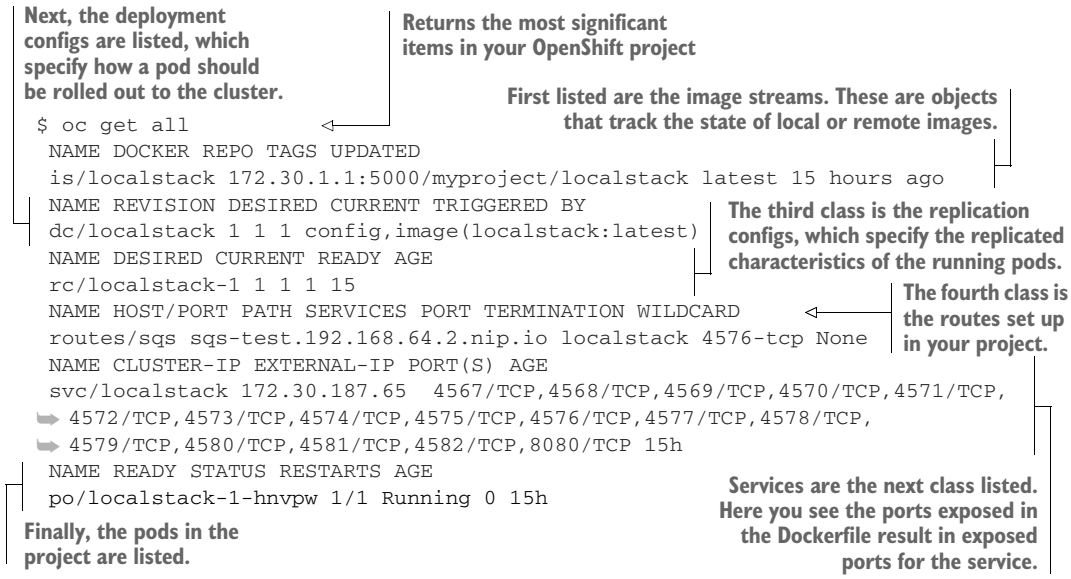


Create the route by running this command,

```
$ oc create -f route.yaml
```

which creates the route from the yaml file you just created. This process is then repeated for each service you want to set up.

Then run `oc get all` to see what you've created within your OpenShift project:

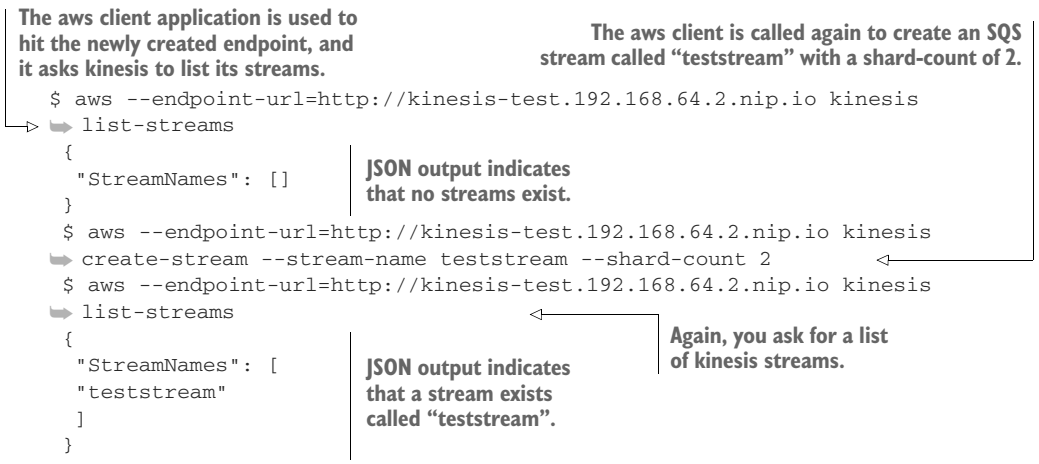


Although technically not *all* the objects available within your project, the `oc get all` command shows the ones most significant to running applications.

The SQS-like AWS service is now accessible as a URL endpoint to test your code against.

ACCESSING THE SERVICES

You can now hit the services from your host. Here's an example of creating an SQS stream:



NOTE The `aws` client is an install you'll need to make this work. Alternatively, you can curl the API endpoint directly, but we don't advise this. It's also assumed you have run `aws configure` and specified your AWS keys and default region. The actual values specified don't matter to LocalStack, as it doesn't do authentication.

Here we've covered only one type of service, but this technique is easily extended to the others listed at the beginning of this technique.

DISCUSSION

This technique has given you a sense of the power of OpenShift (and Kubernetes, on which OpenShift is based). To get a useful application spun up with a usable endpoint and all the internal wiring taken care of is in many ways the realization of the promise of portability that Docker offers, scaled up to the data centre.

For example, this could be taken further, and multiple instances of LocalStack could be spun up on the same OpenShift cluster. Tests against AWS APIs can be done in parallel without necessarily costing more resources (depending on the size of your OpenShift cluster and the demands of your tests, of course). Because this is all code, continuous integration could be set up to dynamically spin up and spin down LocalStack instances to talk to on each commit of your AWS codebase.

As well as pointing out various aspects of Kubernetes, this particular technique also demonstrates that products such as OpenShift are building on top of Kubernetes to extend its functionality. For example, security context constraints are an OpenShift concept (although security contexts are also in Kubernetes) and "routes" was a concept OpenShift created on top of Kubernetes that was eventually adapted for implementation in Kubernetes directly. Over time, features that have been developed for OpenShift have been upstreamed to Kubernetes and have become part of its offering.

You'll see OpenShift again in technique 99 where we'll look at how it can serve as a platform to securely let users run containers.

TECHNIQUE 91 Building a framework on Mesos

When discussing the multitude of orchestration possibilities, you'll probably find one, in particular, mentioned as an alternative to Kubernetes: Mesos. Typically this is followed by opaque statements like "Mesos is a framework for a framework" and "Kubernetes can be run on top of Mesos."

The most apt analogy we've come across is to think of Mesos as providing the kernel for your data center. You can't do anything useful with it alone—the value comes when you combine it with an init system and applications.

For a low-tech explanation, imagine you have a monkey sitting in front of a panel that controls of all of your machines and has the power to start and stop applications at will. Naturally, you'll need to give the monkey a *very* clear list of instructions about what to do in particular situations, when to start an application up, and so on. You could do it all yourself, but that's time-consuming and monkeys are cheap.

Mesos is the monkey!

Mesos is ideal for a company with a highly dynamic and complex infrastructure, likely with experience at rolling their own production orchestration solutions. If you don't meet these conditions, you may be better served by an off-the-shelf solution rather than spending time tailoring Mesos.

PROBLEM

You have a number of rules for controlling the startup of applications and jobs, and you want to enforce them without manually starting them on remote machines and keeping track of their status.

SOLUTION

Use Mesos, a flexible and powerful tool that provides an abstraction of resource management.

Mesos is a mature piece of software for providing an abstraction of resource management on multiple machines. It's been battle-tested in production by companies you've heard of, and, as a result, it's stable and reliable.

> **NOTE** You need Docker 1.6.2 or later for Mesos to be able to use the correct Docker API version.

Figure 12.3 shows what a generic production Mesos setup looks like.

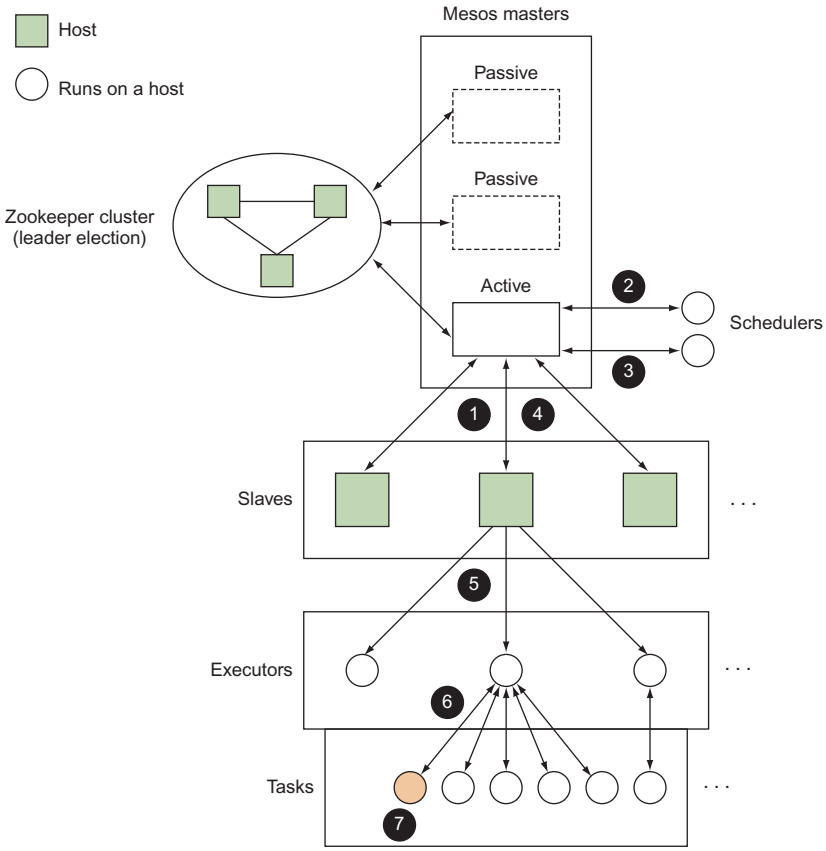


Figure 12.3 A generic production Mesos setup

Referring to this figure, you can see what the basic Mesos lifecycle for starting a task looks like:

1. A slave runs on a node, tracking resource availability and keeping the master informed.
2. The master receives information from one or more slaves about available resources and makes resource offers to schedulers.
3. A scheduler receives resource offers from the master, decides where it wants to run tasks, and communicates this back to the master.
4. The master passes on the task information to the appropriate slaves.
5. Each slave passes the task information to an existing executor on the node or starts a new one.
6. The executor reads the task information and starts the task on the node.
7. The task runs.

The Mesos project provides the master and slave, as well as a built-in shell executor. It's your job to provide a *framework* (or *application*), which consists of a scheduler (the "list of instructions" in our monkey analogy) and optionally a custom executor.

Many third-party projects provide frameworks you can drop into Mesos (and we'll look at one in more detail in the next technique), but to get a better understanding of how you can fully harness the power of Mesos with Docker, we're going to build our own framework consisting only of a scheduler. If you have highly complex logic for starting applications, this may be your final chosen route.

> **NOTE** You don't have to use Docker with Mesos, but since that's what the book is about, we will. There's a lot of detail we won't go into because Mesos is so flexible. We're also going to be running Mesos on a single computer, but we'll try to keep it as realistic as possible and point out what you need to do to go live.

We've not yet explained where Docker fits into the Mesos lifecycle—the final piece to this puzzle is that Mesos provides support for *containerizers*, allowing you to isolate your executors or tasks (or both). Docker isn't the only tool that can be used here, but it's so popular that Mesos has some Docker-specific features to get you started.

Our example will only containerize the tasks we run, because we're using the default executor. If you had a custom executor only running a language environment, where each task involves dynamically loading and executing some code, you might want to consider containerizing the executor instead. As an example use case, you might have a JVM running as an executor that loads and executes pieces of code on the fly, avoiding JVM startup overhead for potentially very small tasks.

Figure 12.4 shows what will be going on behind the scenes in our example when a new dockerized task is created.

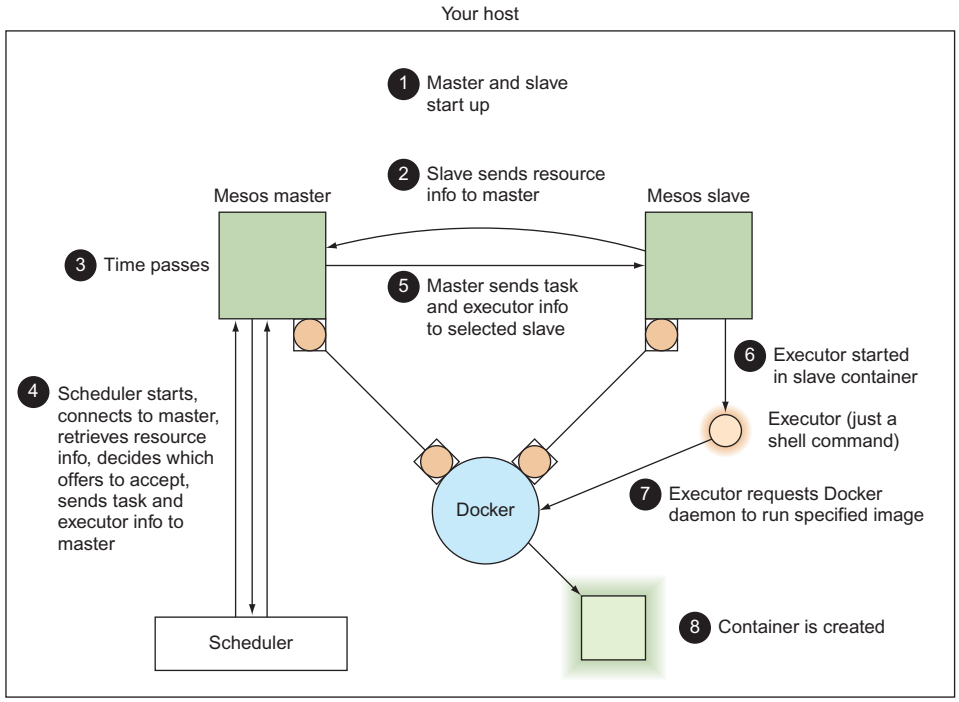


Figure 12.4 A single-host Mesos setup starting a container

Without any further ado, let's get started. First you need to start up a master:

Listing 12.6 Starting a master

```
$ docker run -d --name mesmaster redjack/mesos:0.21.0 mesos-master \
--work_dir=/opt
24e277601260dcc6df35dc20a32a81f0336ae49531c46c2c8db84fe99ac1da35
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' mesmaster
172.17.0.2
$ docker logs -f mesmaster
I0312 01:43:59.182916    1 main.cpp:167] Build: 2014-11-22 05:29:57 by root
I0312 01:43:59.183073    1 main.cpp:169] Version: 0.21.0
I0312 01:43:59.183084    1 main.cpp:172] Git tag: 0.21.0
[...]
```

The master startup is a little verbose, but you should find it stops logging quickly. Keep this terminal open so you can see what happens when you start the other containers.

NOTE Usually a Mesos setup will have multiple Mesos masters (one active and several backups), along with a Zookeeper cluster. Setting this up is documented on the "Mesos High-Availability Mode" page on the Mesos site (http://mesos.apache.org/documentation/latest/high-availability). You'd also need to expose port 5050 for external communications and use the work_dir folder as a volume to save persistent information.

You also need a slave. Unfortunately this is a little fiddly. One of the defining characteristics of Mesos is the ability to enforce resource limits on tasks, which requires the slave to have the ability to freely inspect and manage processes. As a result, the command to run the slave needs a number of outer system details to be exposed inside the container.

Listing 12.7 Starting a slave

```
$ docker run -d --name messlave --pid=host \
  -v /var/run/docker.sock:/var/run/docker.sock -v /sys:/sys \
  redjack/mesos:0.21.0 mesos-slave \
  --master=172.17.0.2:5050 --executor_registration_timeout=5mins \
  --isolation=cgroups/cpu,cgroups/mem --containerizers=docker,mesos \
  --resources="ports(*):[8000-8100]"
1b88c414527f63e24241691a96e3e3251fbb24996f3bfba3ebba91d7a541a9f5
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' messlave
172.17.0.3
$ docker logs -f messlave
I0312 01:46:43.341621 32398 main.cpp:142] Build: 2014-11-22 05:29:57 by root
I0312 01:46:43.341789 32398 main.cpp:144] Version: 0.21.0
I0312 01:46:43.341795 32398 main.cpp:147] Git tag: 0.21.0
[...]
I0312 01:46:43.554498 32429 slave.cpp:627] No credentials provided. >
Attempting to register without authentication
I0312 01:46:43.554633 32429 slave.cpp:638] Detecting new master
I0312 01:46:44.419646 32424 slave.cpp:756] Registered with master >
master@172.17.0.2:5050; given slave ID 20150312-014359-33558956-5050-1-S0
[...]
```

At this point you should also have seen some activity in the Mesos master terminal, starting with a couple of lines like these:

```
I0312 01:46:44.332494     9 master.cpp:3068] Registering slave at >
slave(1)@172.17.0.3:5051 (8c6c63023050) with id >
20150312-014359-33558956-5050-1-S0
I0312 01:46:44.333772     8 registrar.cpp:445] Applied 1 operations in >
134310ns; attempting to update the 'registry'
```

The output of these two logs shows that your slave has started and is connected to the master. If you don't see these, stop and double-check your master IP address. It can be frustrating later on to try and debug why a framework isn't starting any tasks, when there are no connected slaves to start them on.

Anyway, there's a lot going on in the command in listing 12.7. The arguments after `run` and before `redjack/mesos:0.21.0` are all Docker arguments, and they mainly consist of giving the slave container lots of information about the outside world. The arguments after `mesos-slave` are more interesting. First, `master` tells your slave where to find your master (or your Zookeeper cluster). The next three arguments, `executor_registration_timeout`, `isolation`, and `containerizers`, are all tweaks to Mesos settings that should always be applied when working with Docker. Last, but certainly

not least, you need to let the Mesos slave know what ports are acceptable to hand out as resources. By default, Mesos offers 31000–32000, but we want something a bit lower and more memorable.

Now the easy steps are out of the way and we come to the final stage of setting up Mesos—creating a scheduler.

Happily, we have an example framework ready for you to use. Let's try it out, see what it does, and then explore how it works. Keep your two `docker logs -f` commands open on your master and slave containers so you can see the communication as it happens.

The following commands will get the source repository for the example framework from GitHub and start it up.

Listing 12.8 Downloading and starting the example framework

```
$ git clone https://github.com/docker-in-practice/mesos-nc.git
$ docker run -it --rm -v $(pwd)/mesos-nc:/opt redjack/mesos:0.21.0 bash
# apt-get update && apt-get install -y python
# cd /opt
# export PYTHONUSERBASE=/usr/local
# python myframework.py 172.17.0.2:5050
I0312 02:11:07.642227   182 sched.cpp:137] Version: 0.21.0
I0312 02:11:07.645598   176 sched.cpp:234] New master detected at >
master@172.17.0.2:5050
I0312 02:11:07.645800   176 sched.cpp:242] No credentials provided. >
Attempting to register without authentication
I0312 02:11:07.648449   176 sched.cpp:408] Framework registered with >
20150312-014359-33558956-5050-1-0000
Registered with framework ID 20150312-014359-33558956-5050-1-0000
Received offer 20150312-014359-33558956-5050-1-O0. cpus: 4.0, mem: 6686.0, >
ports: 8000-8100
Creating task 0
Task 0 is in state TASK_RUNNING
[...]
Received offer 20150312-014359-33558956-5050-1-O5. cpus: 3.5, mem: 6586.0, >
ports: 8005-8100
Creating task 5
Task 5 is in state TASK_RUNNING
Received offer 20150312-014359-33558956-5050-1-O6. cpus: 3.4, mem: 6566.0, >
ports: 8006-8100
Declining offer
```

You'll note that we've mounted the Git repository inside the Mesos image. This is because it contains all the Mesos libraries we need. Unfortunately, it can be a little painful to install them otherwise.

Our `mesos-nc` framework is designed to run `echo 'hello <task id>' | nc -l <port>` on all available hosts, on all available ports from 8000 to 8005. Because of how netcat works, these "servers" will terminate as soon as you access them, be it by curl, Telnet, nc, or your browser. You can verify this by running `curl localhost:8003` in a new terminal. It will return the expected response, and your Mesos logs will show the

spawning of a task to replace the terminated one. You can also keep track of which tasks are running with `docker ps`.

It's worth pointing out here the evidence of Mesos keeping track of allocated resources and marking them as available when a task terminates. In particular, when you accessed `localhost:8003` (feel free to try it again), take a close look at the `Received offer` line—it shows two port ranges (as they're not connected), including the freshly freed one:

```
Received offer 20150312-014359-33558956-5050-1-O45. cpus: 3.5, mem: 6586.0, >
ports: 8006-8100,8003-8003
```

> **WARNING** The Mesos slave names all the containers it starts with the prefix "mesos-", and it assumes anything like that can be freely managed by the slave. Be careful with your container naming, or you might end up with the Mesos slave killing itself.

The framework code (myframework.py) is well commented, in case you're feeling adventurous. We'll go through some of the high-level design.

```
class TestScheduler
(mesos.interface.Scheduler):
[...]
    def registered(self, driver, frameworkId, masterInfo):
[...]
    def statusUpdate(self, driver, update):
[...]
    def resourceOffers(self, driver, offers):
[...]
```

All Mesos schedulers subclass the base Mesos scheduler class, and they implement a number of methods that Mesos will call at appropriate points to let your framework react to events. Although we've implemented three in the preceding snippet, two of those are optional and have been implemented to add extra logging for demonstration purposes. The only method you *must* implement is `resourceOffers`—there's not much point in a framework that doesn't know when it can launch tasks. You're free to add any additional methods for your own purposes, such as *init* and `_makeTask`, as long as they don't conflict with any of the methods Mesos expects to use, so make sure you read the documentation (http://mesos.apache.org/documentation/latest/app-framework-development-guide/).

> **TIP** If you end up writing your own framework, you'll want to look at some documentation of methods and structures. Unfortunately, at time of writing, the only generated documentation is for Java methods. Readers looking for a starting point for digging into the structures may wish to begin with the include/mesos/mesos.proto file in the Mesos source code. Good luck!

Let's look in a bit more detail at the main method of interest: `resourceOffers`. This is where the decision happens to launch tasks or decline an offer. Figure 12.5 shows the

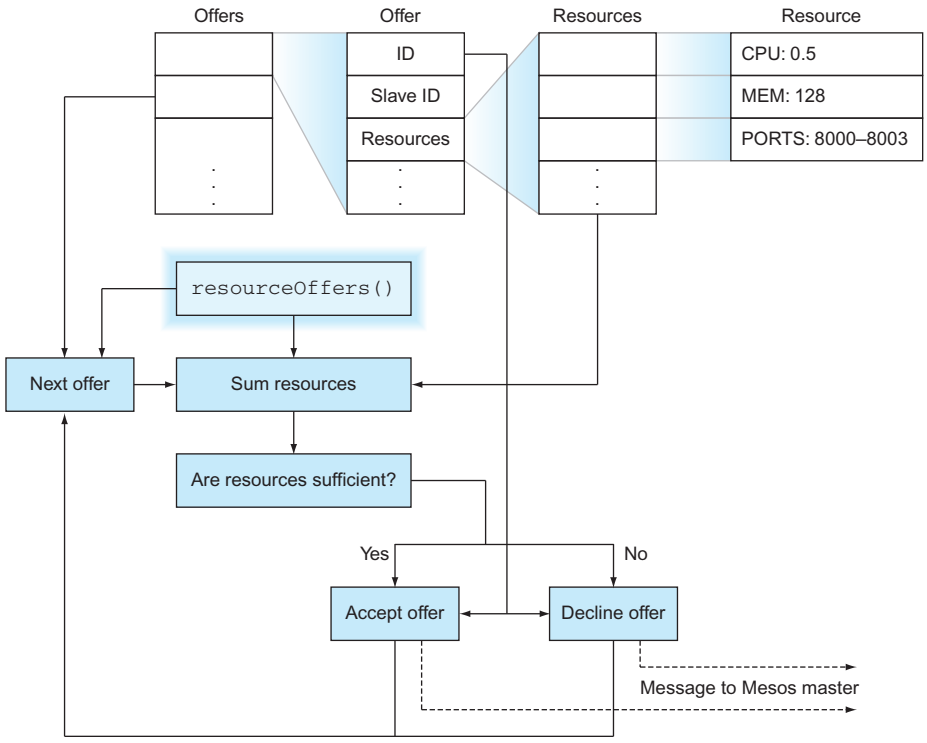


Figure 12.5 Framework `resourceOffers` execution flow

execution flow after `resourceOffers` in our framework is called by Mesos (usually because some resources have become available for use by the framework).

`resourceOffers` is given a list of offers, where each offer corresponds to a single Mesos slave. The offer contains details about the resources available to a task launched on the slave, and a typical implementation will use this information to identify the most appropriate places to launch the tasks it wants to run. Launching a task sends a message to the Mesos master, which then continues with the lifecycle outlined in figure 12.3.

DISCUSSION

It's important to note the flexibility of `resourceOffers`—your task-launching decisions can depend on any criteria you choose, from health checks of external services to the phase of the moon. This flexibility can be a burden, so premade frameworks exist to take some of this low-level detail away and simplify Mesos usage. One of these frameworks is covered in the next technique.

You may want to consult Roger Ignazio's *Mesos in Action* (Manning, 2016) for more details on what you can do with Mesos—we've only scratched the surface here, and you've seen how easily Docker slots in.

TECHNIQUE 92 Micromanaging Mesos with Marathon

By now you'll have realized that there's a lot you need to think about with Mesos, even for an extremely simple framework. Being able to rely on applications being deployed correctly is extremely important—the impact of a bug in a framework could range from the inability to deploy new applications to a full service outage.

The stakes get higher as you scale up, and unless your team is used to writing reliable dynamic deployment code, you might want to consider a more battle-tested approach—Mesos itself is very stable, but an in-house bespoke framework may not be as reliable as you'd want.

Marathon is suitable for a company without in-house deployment tooling experience but that needs a well-supported and easy-to-use solution for deploying containers in a somewhat dynamic environment.

PROBLEM

You need a reliable way to harness the power of Mesos without getting bogged down in writing your own framework.

SOLUTION

Use Marathon, a layer on top of Mesos that provides a simpler interface to get you productive faster.

Marathon is an Apache Mesos framework built by Mesosphere for managing long-running applications. The marketing materials describe it as the `init` or `upstart` daemon for your datacenter (where Mesos is the kernel). This is not an unreasonable analogy.

Marathon makes it easy to get started by allowing you to start a single container with a Mesos master, Mesos slave, and Marathon itself inside. This is useful for demos, but it isn't suitable for production Marathon deployments. To get a realistic Marathon setup, you'll need a Mesos master and slave (from the previous technique) as well as a Zookeeper instance (from technique 84). Make sure you have all this running, and we'll get started by running the Marathon container.

```
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' mesmaster
172.17.0.2
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' messlave
172.17.0.3
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' zookeeper
172.17.0.4
$ docker pull mesosphere/marathon:v0.8.2
[...]
$ docker run -d -h $(hostname) --name marathon -p 8080:8080 \
mesosphere/marathon:v0.8.2 --master 172.17.0.2:5050 --local_port_min 8000 \
--local_port_max 8100 --zk zk://172.17.0.4:2181/marathon
accd6de46cfab65572539ccffa5c2303009be7ec7dbfb49e3ab8f447453f2b93
$ docker logs -f marathon
MESOS_NATIVE_JAVA_LIBRARY is not set. Searching in /usr/lib /usr/local/lib.
MESOS_NATIVE_LIBRARY, MESOS_NATIVE_JAVA_LIBRARY set to >
'/usr/lib/libmesos.so'
```

```
[2015-06-23 19:42:14,836] INFO Starting Marathon 0.8.2 >
(mesosphere.marathon.Main$:87)
[2015-06-23 19:42:16,270] INFO Connecting to Zookeeper... >
(mesosphere.marathon.Main$:37)
[...]
[2015-06-30 18:20:07,971] INFO started processing 1 offers, >
launching at most 1 tasks per offer and 1000 tasks in total
➥ (mesosphere.marathon.tasks.IterativeOfferMatcher$:124)
[2015-06-30 18:20:07,972] INFO Launched 0 tasks on 0 offers, >
declining 1 (mesosphere.marathon.tasks.IterativeOfferMatcher$:216)
```

Like Mesos itself, Marathon is fairly chatty, but (also like Mesos) it stops fairly quickly. At this point, it will enter the loop you're familiar with from writing your own framework—considering resource offers and deciding what to do with them. Because we haven't launched anything yet, you should see no activity; hence the `declining 1` in the preceding log.

Marathon comes with a nice-looking web interface, which is why we exposed port 8080 on the host—visit http://localhost:8080 in your browser to pull it up.

We're going to dive straight into Marathon, so let's create a new application. To clarify a bit of terminology—an "app" in the Marathon world is a group of one or more tasks with exactly the same definition.

Click the New App button at the top right to bring up a dialog box you can use to define the app you want to start up. We'll continue in the vein of the framework we created ourselves by setting the ID to "marathon-nc", leaving CPU, memory, and disk space at their defaults (to match the resource limits imposed on our mesos-nc framework), and setting the command to `echo "hello $MESOS_TASK_ID" | nc -l $PORT0` (using environment variables available to the task—note, that's the number zero). Set the Ports field to 8000 as an indication of where you want to listen. For now we're going to skip over the other fields. Click Create.

Your newly defined application will now be listed on the web interface. The status will briefly show as "Deploying" before showing as "Running." Your app is now started!

If you click on the "/marathon-nc" entry in the Apps list, you'll see the unique ID of your app. You can get the full configuration from the REST API as shown in the following snippet and also verify that it's running by curling the Mesos slave container on the appropriate port. Make sure you save the full configuration returned by the REST API, as it'll come in handy later—it's been saved to app.json in the following example.

```
$ curl http://localhost:8080/v2/apps/marathon-nc/versions
{"versions":["2015-06-30T19:52:44.649Z"]}
$ curl -s \
http://localhost:8080/v2/apps/marathon-nc/versions/2015-06-30T19:52:44.649Z \
> app.json
$ cat app.json
{"id":"/marathon-nc", >
"cmd":"echo \"hello $MESOS_TASK_ID\" | nc -l $PORT0",[...]
$ curl http://172.17.0.3:8000
hello marathon-nc.f56f140e-19e9-11e5-a44d-0242ac110012
```

Note the text following "hello" in the output from `curl`ing the app—it should match the unique ID in the interface. Be quick with checking, though—running that `curl` command will make the app terminate, Marathon will relaunch it, and the unique ID in the web interface will change. Once you've verified all this, go ahead and click the Destroy App button to remove marathon-nc.

This works OK, but you may have noticed that we've not achieved what we set out to do with Marathon—orchestrate Docker containers. Although our application is within a container, it's been launched in the Mesos slave container rather than in a container of its own. Reading the Marathon documentation reveals that creating tasks inside Docker containers requires a little more configuration (as it did when writing our own framework).

Happily, the Mesos slave we started previously has both the required settings, so we just need to alter some Marathon options—in particular, app options. By taking the Marathon API response from before (saved in app.json), we can focus on adding the Marathon settings that enable Docker usage. To perform the manipulation here, we'll use the handy `jq` tool, though it's equally easy to do it via a text editor.

```
$ JQ=https://github.com/stedolan/jq/releases/download/jq-1.3/jq-linux-x86_64
$ curl -Os $JQ && mv jq-linux-x86_64 jq && chmod +x jq
$ cat >container.json <<EOF
{
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "ubuntu:14.04.2",
      "network": "BRIDGE",
      "portMappings": [{"hostPort": 8000, "containerPort": 8000}]
    }
  }
}
$ # merge the app and container details
$ cat app.json container.json | ./jq -s add > newapp.json
```

We can now send the new app definition to the API and see Marathon launch it:

```
$ curl -X POST -H 'Content-Type: application/json; charset=utf-8' \
--data-binary @newapp.json http://localhost:8080/v2/apps
{"id":"/marathon-nc", >
"cmd":"echo \"hello $MESOS_TASK_ID\" | nc -l $PORT0",[...]
$ sleep 10
$ docker ps --since=marathon
CONTAINER ID  IMAGE         COMMAND              CREATED             >
STATUS             PORTS                   NAMES
284ced88246c  ubuntu:14.04  "\"/bin/sh -c 'echo  About a minute ago  >
Up About a minute  0.0.0.0:8000->8000/tcp  mesos- >
1da85151-59c0-4469-9c50-2bfc34f1a987
$ curl localhost:8000
hello mesos-nc.675b2dc9-1f88-11e5-bc4d-0242ac11000e
$ docker ps --since=marathon
CONTAINER ID  IMAGE         COMMAND              CREATED         >
```

```
STATUS                      PORTS                    NAMES
851279a9292f  ubuntu:14.04  "\"/bin/sh -c 'echo  44 seconds ago  >
Up 43 seconds               0.0.0.0:8000->8000/tcp  mesos- >
37d84e5e-3908-405b-aa04-9524b59ba4f6
284ced88246c  ubuntu:14.04  "\"/bin/sh -c 'echo  24 minutes ago  >
Exited (0) 45 seconds ago                          mesos-1da85151-59c0-
➥ 4469-9c50-2bfc34f1a987
```

As with our custom framework in the last technique, Mesos has launched a Docker container for us with the application running. Running `curl` terminates the application and container, and a new one is automatically launched.

DISCUSSION

There are some significant differences between the custom framework from the last technique and Marathon. For example, in the custom framework we had extremely fine-grained control over accepting resource offers, to the point where we could pick and choose individual ports to listen on. In order to do a similar thing in Marathon, you'd need to impose the setting on each individual slave.

By contrast, Marathon comes with a lot of built-in features that would be error-prone to build yourself, including health checking, an event notification system, and a REST API. These aren't trivial things to implement, and using Marathon lets you operate with the assurance that you aren't the first one trying it. If nothing else, it's a lot easier to get support for Marathon than for a bespoke framework, and we've found that the documentation for Marathon is more approachable than that for Mesos.

We've covered the basics of setting up and using Marathon, but there are many more things to see and do. One of the more interesting suggestions we've seen is to use Marathon to start up other Mesos frameworks, potentially including your own bespoke one! We encourage you to explore—Mesos is a high-quality tool for orchestration, and Marathon provides a usable layer on top of it.

Summary

- You can start services on a cluster of machines with Docker swarm mode.
- Writing a custom framework for Mesos can give you fine-grained control over your container scheduling.
- The Marathon framework on top of Mesos provides a simple way to harness some of the power of Mesos.
- Kubernetes is a production-quality orchestration tool and has an API you can leverage.
- OpenShift can be used to set up a local version of some AWS services.