# OpenShift IN ACTION

Jamie Duncan
John Osborne
Foreword by Jim Whitehurst

SAMPLE CHAPTER

MANNING

*OpenShift in Action*

by Jamie Duncan
and John Osborne

**Chapter 3**

# *brief contents*

# 3 Containers are Linux

**This chapter covers**

- How OpenShift, Kubernetes, and docker work together
- How containers isolate processes with namespaces

In the previous chapter, you deployed your first applications in OpenShift. In this chapter, we'll look deeper into your OpenShift cluster and investigate how these containers isolate their processes on the application node.

Knowledge of how containers work in a platform like OpenShift is some of the most powerful information in IT right now. This fundamental understanding of how a container actually works as part of a Linux server informs how systems are designed and how issues are analyzed when they inevitably occur.

This is a challenging chapter—not because you'll be editing a lot of configurations and making complex changes, but because we're talking about the fundamental layers of abstraction that make a container a container. Let's get started by attempting to define exactly what a container is.

## 3.1 Defining containers

You can find five different container experts and ask them to define what a container is, and you're likely to get five different answers. The following are some of our personal favorites, all of which are correct from a certain perspective:

- A transportable unit to move applications around. This is a typical developer's answer.
- A fancy Linux process (one of our personal favorites).
- A more effective way to isolate processes on a Linux system. This is a more operations-centered answer.

What we need to untangle is the fact that they're all correct, depending on your point of view.

In chapter 1, we talked about how OpenShift uses Kubernetes and docker to orchestrate and deploy applications in containers in your cluster. But we haven't talked much about which application component is created by each of these services. Before we move forward, it's important for you to understand these responsibilities as you begin interacting with application components directly.

## 3.2 How OpenShift components work together

When you deploy an application in OpenShift, the request starts in the OpenShift API. We discussed this process at a high level in chapter 2. To really understand how containers isolate the processes within them, we need take a more detailed look at how these services work together to deploy your application. The relationship between OpenShift, Kubernetes, docker, and, ultimately, the Linux kernel is a chain of dependencies.

When you deploy an application in OpenShift, the process starts with the OpenShift services.

### 3.2.1 OpenShift manages deployments

Deploying applications begins with application components that are unique to OpenShift. The process is as follows:

1. OpenShift creates a custom container image using your source code and the builder image template you specified. For example, app-cli and app-gui use the PHP builder image.
2. This image is uploaded to the OpenShift container image registry.
3. OpenShift creates a build config to document how your application is built. This includes which image was created, the builder image used, the location of the source code, and other information.
4. OpenShift creates a deployment config to control deployments and deploy and update your applications. Information in deployment configs includes the number of replicas, the upgrade method, and application-specific variables and mounted volumes.
5. OpenShift creates a deployment, which represents a single deployed version of an application. Each unique application deployment is associated with your application's deployment config component.

6 The OpenShift internal load balancer is updated with an entry for the DNS record for the application. This entry will be linked to a component that's created by Kubernetes, which we'll get to shortly.

7 OpenShift creates an image stream component. In OpenShift, an image stream monitors the builder image, deployment config, and other components for changes. If a change is detected, image streams can trigger application redeployments to reflect changes.

Figure 3.1 shows how these components are linked together. When a developer creates source code and triggers a new application deployment (in this case, using the `oc` command-line tool), OpenShift creates the deployment config, image stream, and build config components.

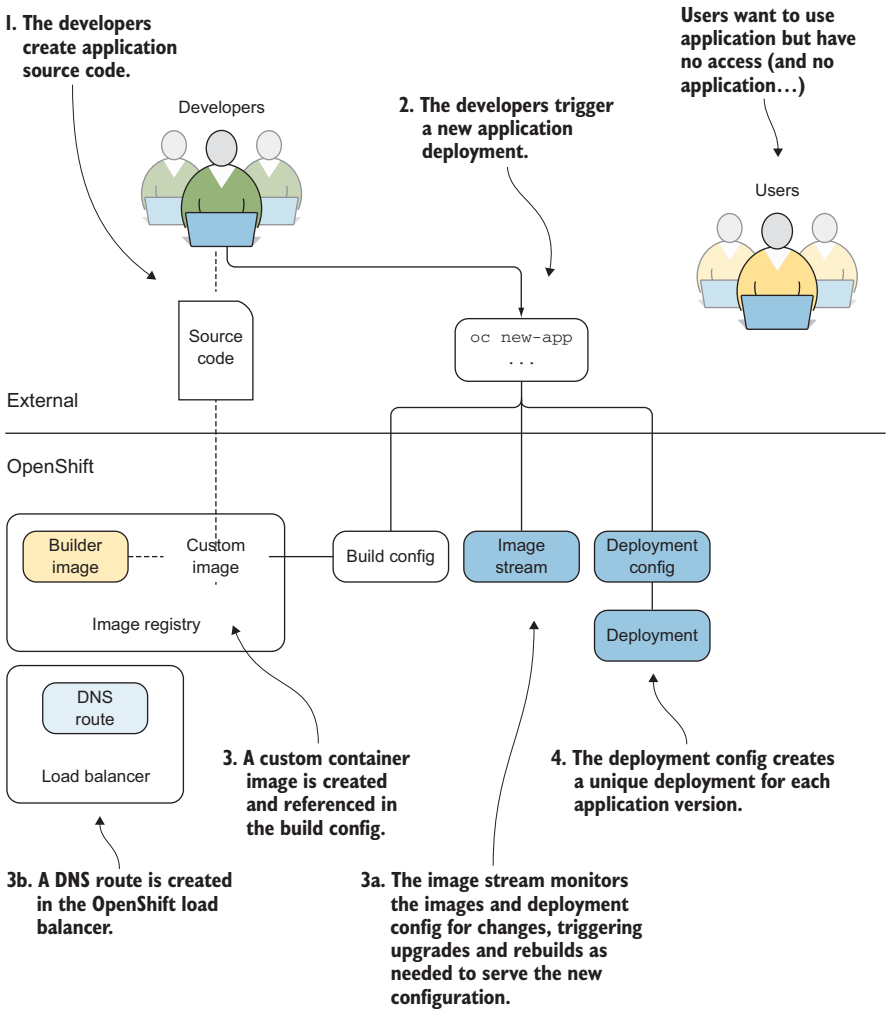



Figure 3.1 Application components created by OpenShift during application deployment

The build config creates an application-specific custom container image using the specified builder image and source code. This image is stored in the OpenShift image registry. The deployment config component creates an application deployment that's unique for each version of the application. The image stream is created and monitors for changes to the deployment config and related images in the internal registry. The DNS route is also created and will be linked to a Kubernetes object.

In figure 3.1, notice that the users are sitting by themselves with no access to the application. There *is* no application. OpenShift depends on Kubernetes, as well as docker, to get the deployed application to the user. Next, we'll look at Kubernetes' responsibilities in OpenShift.

### 3.2.2 Kubernetes schedules applications across nodes

Kubernetes is the orchestration engine at the heart of OpenShift. In many ways, an OpenShift cluster is a Kubernetes cluster. When you initially deployed app-cli, Kubernetes created several application components:

- *Replication controller*—Scales the application as needed in Kubernetes. This component also ensures that the desired number of replicas in the deployment config is maintained at all times.
- *Service*—Exposes the application. A Kubernetes service is a single IP address that's used to access all the active pods for an application deployment. When you scale an application up or down, the number of pods changes, but they're all accessed through a single service.
- *Pods*—Represent the smallest scalable unit in OpenShift.

**NOTE** Typically, a single pod is made up of a single container. But in some situations, it makes sense to have a single pod consist of multiple containers.

Figure 3.2 illustrates the relationships between the Kubernetes components that are created. The replication controller dictates how many pods are created for an initial application deployment and is linked to the OpenShift deployment component.

Also linked to the pod component is a Kubernetes service. The service represents all the pods deployed by a replication controller. It provides a single IP address in OpenShift to access your application as it's scaled up and down on different nodes in your cluster. The service is the internal IP address that's referenced in the route created in the OpenShift load balancer.

**NOTE** The relationship between deployments and replication controllers is how applications are deployed, scaled, and upgraded. When changes are made to a deployment config, a new deployment is created, which in turn creates a new replication controller. The replication controller then creates the desired number of pods within the cluster, which is where your application is actually deployed.

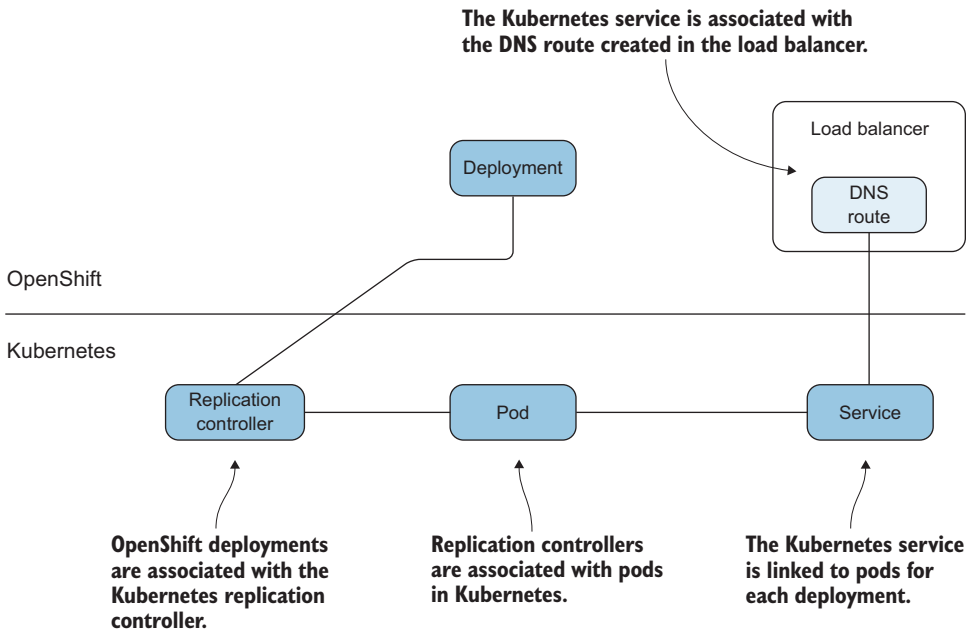



Figure 3.2 Kubernetes components that are created when applications are deployed

We're getting closer to the application itself, but we haven't gotten there yet. Kubernetes is used to orchestrate containers in an OpenShift cluster. But on each application node, Kubernetes depends on docker to create the containers for each application deployment.

### 3.2.3 Docker creates containers

Docker is a *container runtime*. A container runtime is the application on a server that creates, maintains, and removes containers. A container runtime can act as a standalone tool on a laptop or a single server, but it's at its most powerful when being orchestrated across a cluster by a tool like Kubernetes.

> **NOTE** Docker is currently the container runtime for OpenShift. But a new runtime is supported as of OpenShift 3.9. It's called cri-o, and you can find more information at http://cri-o.io.

Kubernetes controls docker to create containers that house the application. These containers use the custom base image as the starting point for the files that are visible to applications in the container. Finally, the docker container is associated with the Kubernetes pod (see figure 3.3).

To isolate the libraries and applications in the container image, along with other server resources, docker uses Linux kernel components. These kernel-level resources are the components that isolate the applications in your container from everything else on the application node. Let's look at these next.

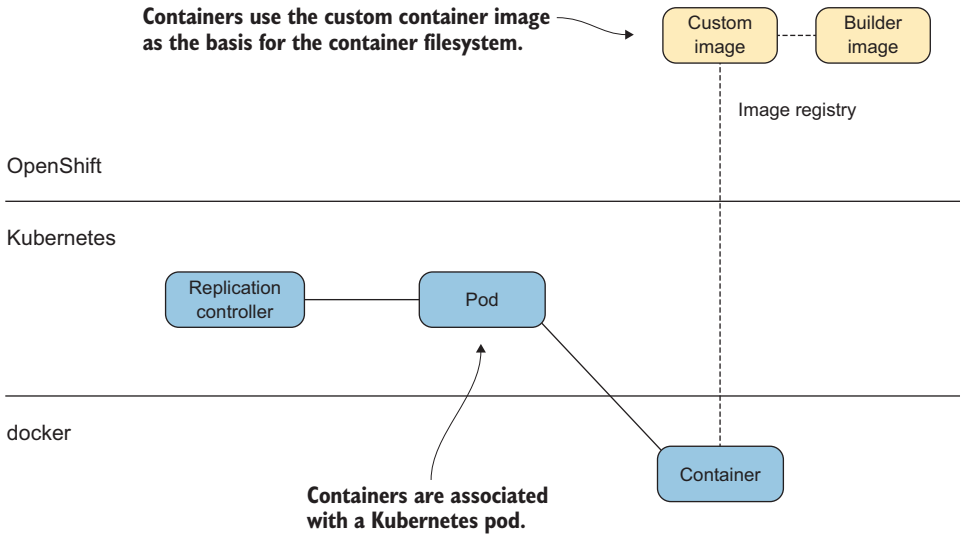



**Figure 3.3 Docker containers are associated with Kubernetes pods.**

### 3.2.4 Linux isolates and limits resources

We're down to the core of what makes a container a container in OpenShift and Linux. Docker uses three Linux kernel components to isolate the applications running in containers it creates and limit their access to resources on the host:

- *Linux namespaces*—Provide isolation for the resources running in the container. Although the term is the same, this is a different concept than Kubernetes namespaces (http://mng.bz/X8yz), which are roughly analogous to an OpenShift project. We'll discuss these in more depth in chapter 7. For the sake of brevity, in this chapter, when we reference namespaces, we're talking about Linux namespaces.
- *Control groups (cgroups)*—Provide maximum, guaranteed access limits for CPU and memory on the application node. We'll look at cgroups in depth in chapter 9.
- *SELinux contexts*—Prevent the container applications from improperly accessing resources on the host or in other containers. An SELinux context is a unique label that's applied to a container's resources on the application node. This unique label prevents the container from accessing anything that doesn't have a matching label on the host. We'll discuss SELinux contexts in more depth in chapter 11.

The docker daemon creates these kernel resources dynamically when the container is created. These resources are associated with the applications that are launched for the corresponding container; your application is now running in a container (figure 3.4).

Applications in OpenShift are run and associated with these kernel components. They provide the isolation that you see from inside a container. In upcoming sections,

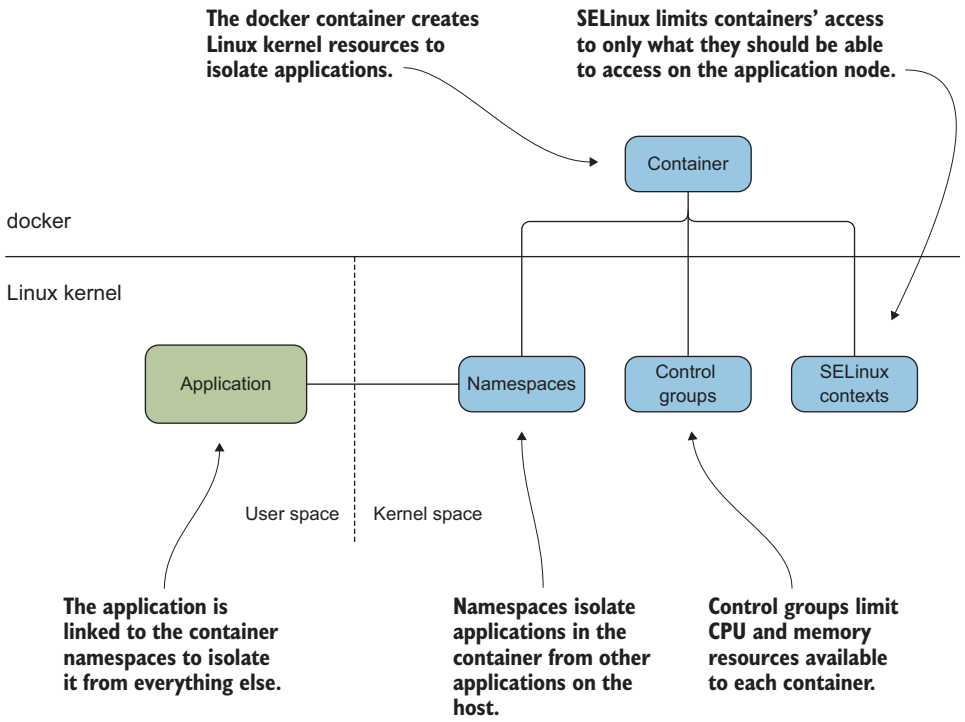



**Figure 3.4 Linux kernel components used to isolate containers**

we'll discuss how you can investigate a container from the application node. From the point of view of being inside the container, an application only has the resources allocated to it that are included in its unique namespaces. Let's confirm that next.

### Userspace and kernelspace

A Linux server is separated into two primary resource groups: the *userspace* and the *kernelspace*. The userspace is where applications run. Any process that isn't part of the kernel is considered part of the userspace on a Linux server.

The kernelspace is the kernel itself. Without special administrator privileges like those the root user has, users can't make changes to code that's running in the kernelspace.

The applications in a container run in the userspace, but the components that isolate the applications in the container run in the kernelspace. That means containers are isolated using kernel components that can't be modified from inside the container.

In the previous sections, we looked at each individual layer of OpenShift. Let's put all of these together before we dive down into the weeds of the Linux kernel.

### 3.2.5 Putting it all together

The automated workflow that's executed when you deploy an application in OpenShift includes OpenShift, Kubernetes, docker, and the Linux kernel. The interactions and dependencies stretch across multiple services, as outlined in figure 3.5.

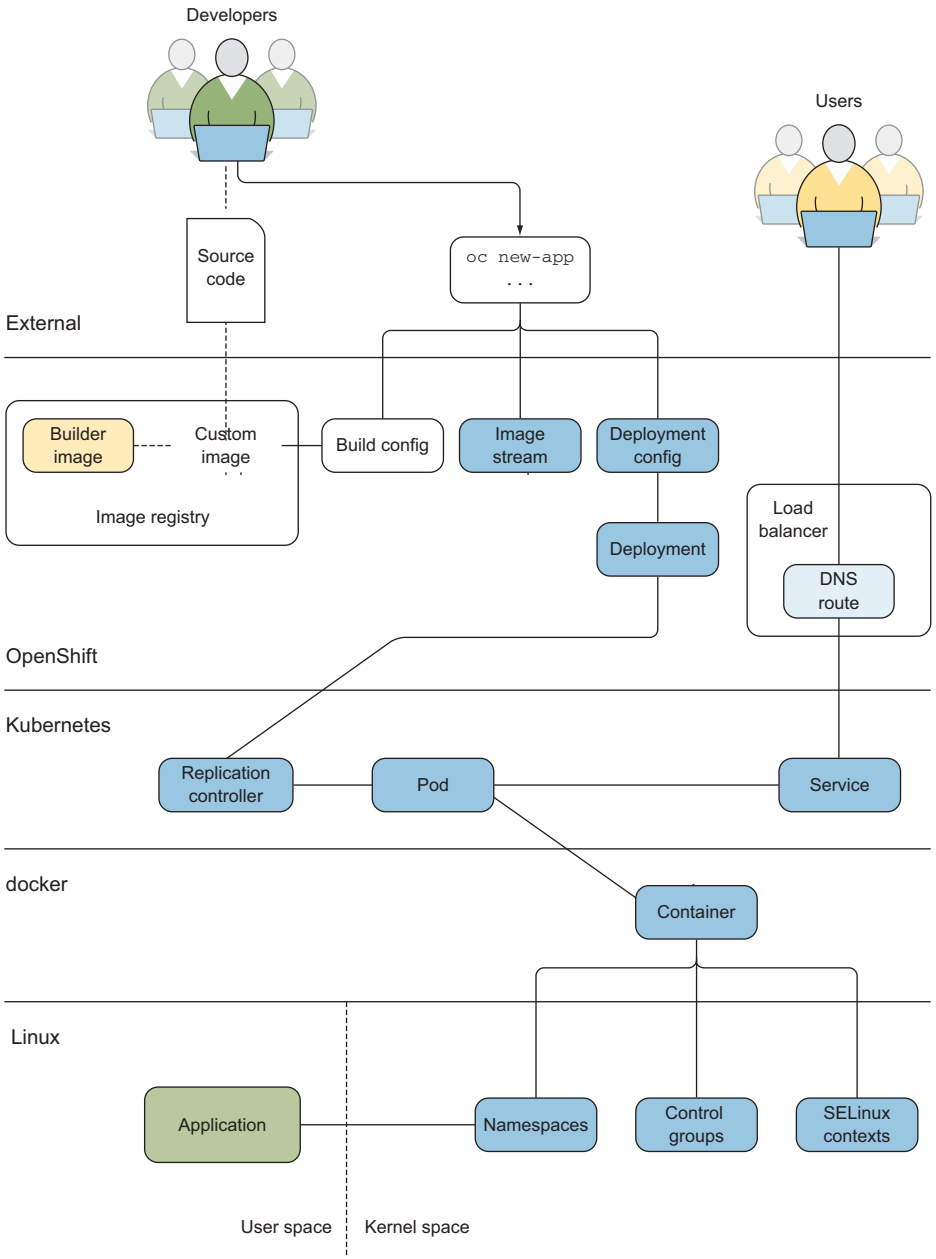



**Figure 3.5 OpenShift deployment including components that make up the container**

Developers and users interact primarily with OpenShift and its services. OpenShift works with Kubernetes to ensure that user requests are fulfilled and applications are delivered consistently according to the developer's designs.

As you'll recall, one of the acceptable definitions for a container earlier in this chapter was that they're "fancy processes." We developed this definition by explaining how a container takes an application process and uses namespaces to limit access to resources on the host. We'll continue to develop this definition by interacting with these fancy processes in more depth in chapters 9 and 10.

Like any other process running on a Linux server, each container has an assigned process ID (PID) on the application node.

## 3.3 Application isolation with kernel namespaces

Armed with the PID for the current app-cli container, you can begin to analyze how containers isolate process resources with Linux namespaces. Earlier in this chapter, we discussed how kernel namespaces are used to isolate the applications in a container from the other processes on a host. Docker creates a unique set of namespaces to isolate the resources in each container. Looking again at figure 3.4, the application is linked to the namespaces because they're unique for each container. Cgroups and SELinux are both configured to include information for a newly created container, but those kernel resources are shared among all containers running on the application node.

To get a list of the namespaces that were created for app-cli, use the `lsns` command. You need the PID for app-cli to pass as a parameter to `lsns`. Appendix C walks you through how to use the docker daemon to get the host PID for a container, along with some other helpful docker commands. Use this appendix as a reference to get the host PID for your app-cli container.

The `lsns` command accepts a PID with the `-p` option and outputs the namespaces associated with that PID. The output for `lsns` has the following six columns:

- `NS`—Inode associated with the namespace
- `TYPE`—Type of namespace created
- `NPROCS`—Number of processes associated with the namespace
- `PID`—Process used to create the namespace
- `USER`—User that owns the namespace
- `COMMAND`—Command executed to launch the process to create the namespace

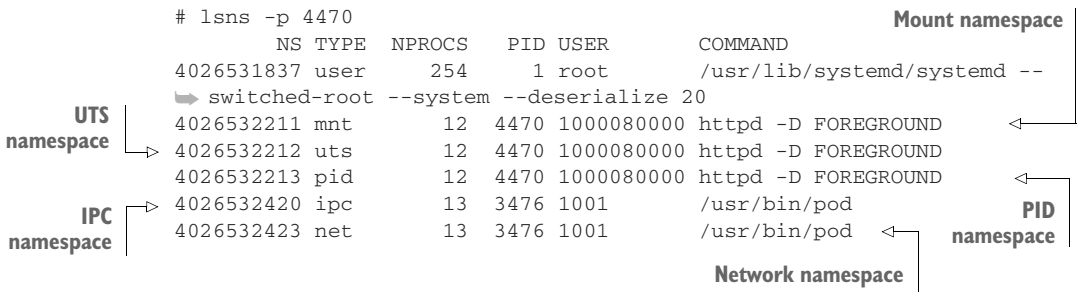
When you run the command, the output from `lsns` shows six namespaces for app-cli. Five of these namespaces are unique to app-cli and provide the container isolation that we're discussing in this chapter. There are also two additional namespaces in Linux that aren't used directly by OpenShift. The *user namespace* isn't currently used by OpenShift, and the *cgroup namespace* is shared between all containers on the system.

> **NOTE** On an OpenShift application node, the user namespace is shared across all applications on the host. The user namespace was created by PID 1 on the host, has over 200 processes associated with it, and is associated with the `systemd` command. The other namespaces associated with the app-cli PID have far fewer processes and aren't owned by PID 1 on the host.

OpenShift uses five Linux namespaces to isolate processes and resources on application nodes. Coming up with a concise definition for exactly what a namespace does is a little difficult. Two analogies best describe their most important properties, if you'll forgive a little poetic license:

- Namespaces are like paper walls in the Linux kernel. They're lightweight and easy to stand up and tear down, but they offer sufficient privacy when they're in place.
- Namespaces are similar to two-way mirrors. From within the container, only the resources in the namespace are available. But with proper tooling, you can see what's in a namespace from the host system.

The following snippet lists all namespaces for app-cli with `lsns`:




```
# lsns -p 4470
        NS TYPE  NPROCS   PID USER       COMMAND
4026531837 user     254     1 root       /usr/lib/systemd/systemd --
➥ switched-root --system --deserialize 20
4026532211 mnt       12  4470 1000080000 httpd -D FOREGROUND
4026532212 uts       12  4470 1000080000 httpd -D FOREGROUND
4026532213 pid       12  4470 1000080000 httpd -D FOREGROUND
4026532420 ipc       13  3476 1001       /usr/bin/pod
4026532423 net       13  3476 1001       /usr/bin/pod
```

As you can see, the five namespaces that OpenShift uses to isolate applications are as follows:

- *Mount*—Ensures that only the correct content is available to applications in the container
- *Network*—Gives each container its own isolated network stack
- *PID*—Provides each container with its own set of PID counters
- *UTS*—Gives each container its own hostname and domain name
- *IPC*—Provides shared memory isolation for each container

There are currently two additional namespaces in the Linux kernel that aren't used by OpenShift:

- *Cgroup*—Cgroups are used as a shared resource on an OpenShift node, so this namespace isn't required for effective isolation.

- *User*—This namespace can map a user in a container to a different user on the host. For example, a user with ID 0 in the container could have user ID 5000 when interacting with resources outside the container. This feature can be enabled in OpenShift, but there are issues with performance and node configuration that fall out of scope for our example cluster. If you'd like more information on enabling the user namespace to work with docker, and thus with OpenShift, see the article "Hardening Docker Hosts with User Namespaces" by Chris Binnie (Linux.com, http://mng.bz/Giwd).

**What is /usr/bin/pod?**

The IPC and network namespaces are associated with a different PID for an application called /usr/bin/pod. This is a pseudo-application that's used for containers created by Kubernetes.

Under most circumstances, a pod consists of one container. There are conditions, however, where a single pod may contain multiple containers. Those situations are outside the scope of this chapter; but when this happens, all the containers in the pod share these namespaces. That means they share a single IP address and can communicate with shared memory devices as though they're on the same host.

We'll discuss the five namespaces used by OpenShift with examples, including how they enhance your security posture and how they isolate their associated resources. Let's start with the mount namespace.

### 3.3.1 The mount namespace

The mount namespace isolates filesystem content, ensuring that content assigned to the container by OpenShift is the only content available to the processes running in the container. The mount namespace for the app-cli container allows the applications in the container to access only the content in the custom app-cli container image, and any information stored on the persistent volume associated with the persistent volume claim (PVC) for app-cli (see figure 3.6).

**NOTE** Applications always need persistent storage. Persistent storage allows data to persist when a pod is removed from the cluster. It also allows data to be shared between multiple pods when needed. You'll learn how to configure and use persistent storage on an NFS server with OpenShift in chapter 7.

The root filesystem, based on the app-cli container image, is a little more difficult to uncover, but we'll do that next.

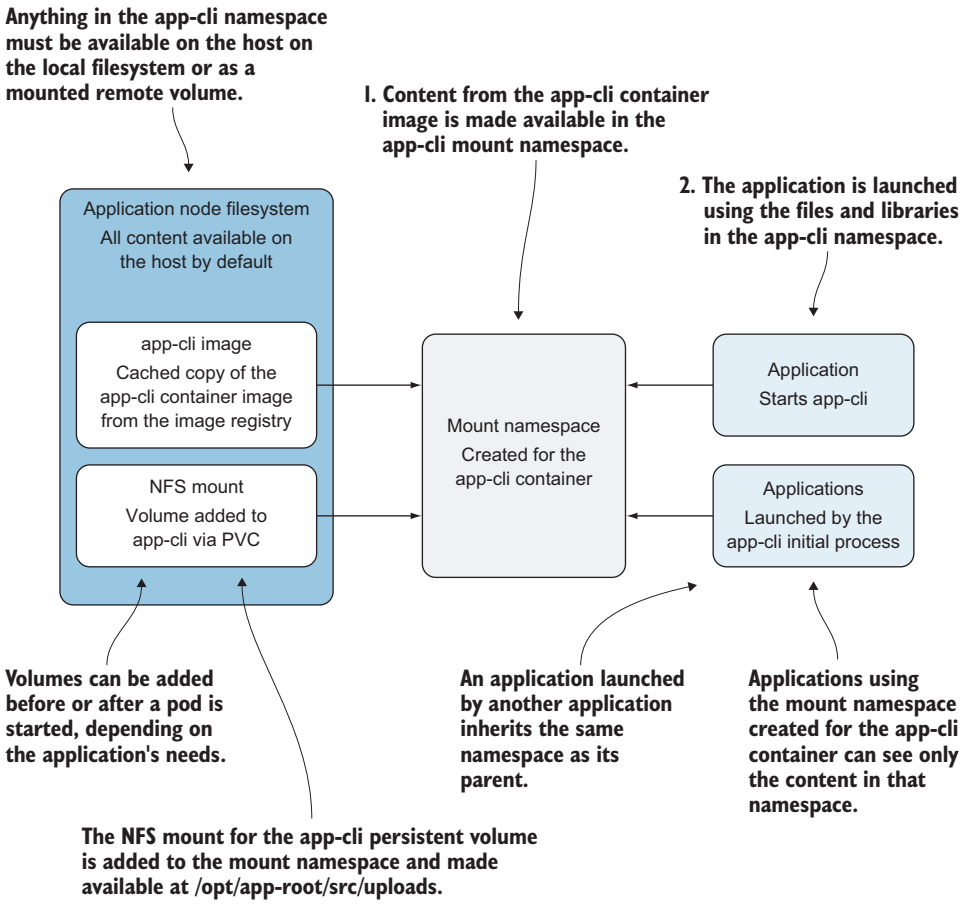



**Figure 3.6 The mount namespace takes selected content and makes it available to the app-cli applications.**

### ACCESSING CONTAINER ROOT FILESYSTEMS

When you configured OpenShift, you specified a block device for docker to use for container storage. Your OpenShift configuration uses logical volume management (LVM) on this device for container storage. Each container gets its own logical volume (LV) when it's created. This storage solution is fast and scales well for large production clusters.

To view all LVs created by docker on your host, run the `lsblk` command. This command shows all block devices on your host, as well as any LVs. It confirms that docker has been creating LVs for your containers:

```
# lsblk
NAME                                    MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
vda                                     253:0    0    8G  0 disk
└─vda1                                  253:1    0    8G  0 part /
vdb                                     253:16   0   20G  0 disk
```

```
└─vdb1                                  253:17   0   20G  0 part
  ├─docker_vg-docker--pool_tmeta        252:0    0   24M  0 lvm
  | └─docker_vg-docker--pool            252:2    0    8G  0 lvm
  |   ├─docker-253:1-10125-e27ee79f...  252:3    0   10G  0 dm
  |   ├─docker-253:1-10125-6ec90d0f...  252:4    0   10G  0 dm
...
  └─docker_vg-docker--pool_tdata        252:1    0    8G  0 lvm
    └─docker_vg-docker--pool            252:2    0    8G  0 lvm
      ├─docker-253:1-10125-e27ee79f...  252:3    0   10G  0 dm
      ├─docker-253:1-10125-6ec90d0f...  252:4    0   10G  0 dm
  ...
```

The LV device that the app-cli container uses for storage is recorded in the information from `docker inspect`. To get the LV for your app-cli container, run the following command:

```
docker inspect -f '{{ .GraphDriver.Data.DeviceName }}' fae8e211e7a7
```

You'll get a value similar to `docker-253:1-10125-8bd64caed0421039e83ee4f1cdcbcf25708e3da97081d43a99b6d20a3eb09c98`. This is the name for the LV that's being used as the root filesystem for the app-cli container.

Unfortunately, when you run the following `mount` command to see where this LV is mounted, you don't get any results:

```
mount | grep docker-253:1-10125-
➥ 8bd64caed0421039e83ee4f1cdcbcf25708e3da97081d43a99b6d20a3eb09c9
```

You can't see the LV for app-cli because it's in a different namespace. No, we're not kidding. The mount namespace for your application containers is created in a different mount namespace from your application node's operating system.

When the docker daemon starts, it creates its own mount namespace to contain filesystem content for the containers it creates. You can confirm this by running `lsns` for the docker process. To get the PID for the main docker process, run the following `pgrep` command (the process `dockerd-current` is the name for the main docker daemon process):

```
# pgrep -f dockerd-current
```

Once you have the docker daemon's PID, you can use `lsns` to view its namespaces. You can tell from the output that the docker daemon is using the system namespaces created by `systemd` when the server booted, except for the mount namespace:

```
# lsns -p 2385
        NS TYPE  NPROCS   PID USER COMMAND
4026531836 pid      221     1 root /usr/lib/systemd/systemd --switched-root
➥ --system --deserialize 20
4026531837 user     254     1 root /usr/lib/systemd/systemd --switched-root
➥ --system --deserialize 20
4026531838 uts      223     1 root /usr/lib/systemd/systemd --switched-root
➥ --system --deserialize 20
```

```
4026531839 ipc      221     1 root /usr/lib/systemd/systemd --switched-root
➥ --system --deserialize 20
4026531956 net      223     1 root /usr/lib/systemd/systemd --switched-root
➥ --system --deserialize 20
4026532298 mnt       12  2385 root /usr/bin/dockerd-current --add-runtime
➥ docker-runc=/usr/libexec/docker/docker-runc-current
--default-runtime=docker-runc --exec-opt native.cgroupdriver=systemd
➥ --userland-proxy-p
```

You can use a command-line tool named `nsenter` to enter an active namespace for another application. It's a great tool to use when you need to troubleshoot a container that isn't performing as it should. To use `nsenter`, you give it a PID for the container with the `--target` option and then instruct it regarding which namespaces you want to enter for that PID:

```
$ nsenter --target 2385
```

When you run the command, you arrive at a prompt similar to your previous prompt. The big difference is that now you're operating from inside the namespace you specified. Run `mount` from within docker's mount namespace and `grep` for your app-cli LV (the output is trimmed for clarity):

```
mount | grep docker-253:1-10125-8bd64cae...
/dev/mapper/docker-253:1-10125-8bd64cae... on ➥
/var/lib/docker/devicemapper/mnt/8bd64cae... type xfs (rw,relatime,➥
context="system_u:object_r:svirt_sandbox_file_t:s0:c4,c9",nouuid,attr2,inode64,
➥ sunit=1024,swidth=1024,noquota)
```

From inside docker's mount namespace, the `mount` command output includes the mount point for the root filesystem for app-cli. The LV that docker created for app-cli is mounted on the application node at /var/lib/docker/devicemapper/mnt/8bd64-cae… (directory name trimmed for clarity).

Go to that directory while in the docker daemon mount namespace, and you'll find a directory named rootfs. This directory is the filesystem for your app-cli container:

```
# ls -al rootfs
total 32
-rw-r--r--.  1 root root 15759 Aug  1 17:24 anaconda-post.log
lrwxrwxrwx.  1 root root     7 Aug  1 17:23 bin -> usr/bin
drwxr-xr-x.  3 root root    18 Sep 14 22:18 boot
drwxr-xr-x.  4 root root    43 Sep 21 23:19 dev
drwxr-xr-x. 53 root root  4096 Sep 21 23:19 etc
-rw-r--r--.  1 root root  7388 Sep 14 22:16 help.1
drwxr-xr-x.  2 root root     6 Nov  5  2016 home
lrwxrwxrwx.  1 root root     7 Aug  1 17:23 lib -> usr/lib
lrwxrwxrwx.  1 root root     9 Aug  1 17:23 lib64 -> usr/lib64
drwx------.  2 root root     6 Aug  1 17:23 lost+found
drwxr-xr-x.  2 root root     6 Nov  5  2016 media
drwxr-xr-x.  2 root root     6 Nov  5  2016 mnt
drwxr-xr-x.  4 root root    32 Sep 14 22:05 opt
```

```
drwxr-xr-x.  2 root root     6 Aug  1 17:23 proc
dr-xr-x---.  2 root root   137 Aug  1 17:24 root
drwxr-xr-x. 11 root root   145 Sep 13 15:35 run
lrwxrwxrwx.  1 root root     8 Aug  1 17:23 sbin -> usr/sbin
...
```

It's been quite a journey to uncover the root filesystem for app-cli. You've used information from the docker daemon to use multiple command-line tools, including `nsenter`, to change from the default mount namespace for your server to the namespace created by the docker daemon. You've done a lot of work to find an isolated filesystem. Docker does this automatically at the request of OpenShift every time a container is created. Understanding how this process works, and where the artifacts are created, is important when you're using containers every day for your application workloads.

From the point of view of the applications running in the app-cli container, all that's available to them is what's in the rootfs directory, because the mount namespace created for the container isolates its content (see figure 3.7). Understanding how mount namespaces function on an application node, and knowing how to enter a container namespace manually, are invaluable tools when you're troubleshooting a container that's not functioning as designed.

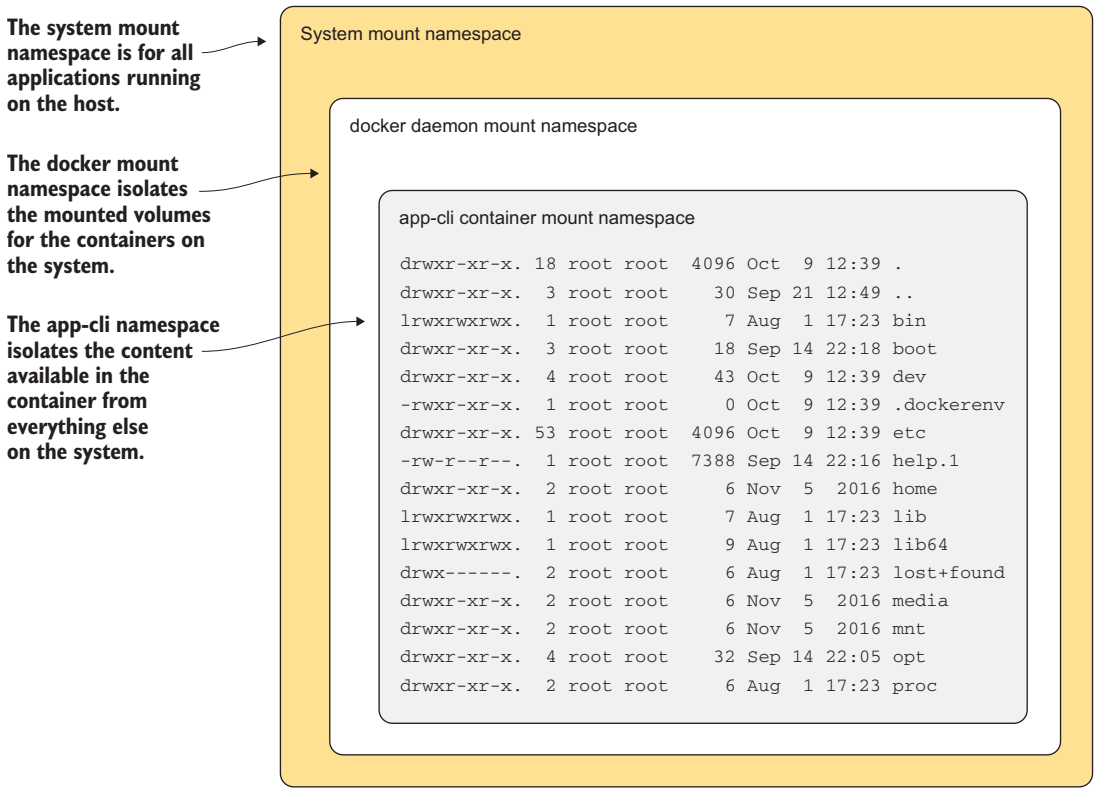



**Figure 3.7 The app-cli mount namespace isolates the contents of the rootfs directory.**

Press Ctrl-D to exit the docker daemon's mount namespace and return to the default namespace for your application node. Next, we'll discuss the UTS namespace. It won't be as involved an investigation as the mount namespace, but the UTS namespace is useful for an application platform like OpenShift that deploys horizontally scalable applications across a cluster of servers.

### 3.3.2 The UTS namespace

UTS stands for *Unix time sharing* in the Linux kernel. The UTS namespace lets each container have its own hostname and domain name.

> **Time sharing**
>
> It can be confusing to talk about time sharing when the UTS namespace has nothing to do with managing the system clock. Time sharing originally referred to multiple users sharing time on a system simultaneously. Back in the 1970s, when this concept was created, it was a novel idea.
>
> The UTS data structure in the Linux kernel had its beginnings then. This is where the hostname, domain name, and other system information are retained. If you'd like to see all the information in that structure, run `uname -a` on a Linux server. That command queries the same data structure.

The easiest way to view the hostname for a server is to run the `hostname` command, as follows:

```
# hostname
```

You could use `nsenter` to enter the UTS namespace for the app-cli container, the same way you entered the mount namespace in the previous section. But there are additional tools that will execute a command in the namespaces for a running container.

> **NOTE** On the application node, if you use the nip.io domain discussed in appendix A, your hostname should look similar to `ocp2.192.168.122.101.nip.io`.

One of those tools is the `docker exec` command. To get the hostname value for a running container, pass `docker exec` a container's short ID and the same `hostname` command you want to run in the container. Docker executes the specified command for you in the container's namespaces and returns the value. The hostname for each OpenShift container is its pod name:

```
# docker exec fae8e211e7a7 hostname
app-cli-1-18k2s
```

Each container has its own hostname because of its unique UTS namespace. If you scale up app-cli, the container in each pod will have a unique hostname as well. The

value of this is identifying data coming from each container in a scaled-up system. To confirm that each container has a unique hostname, log in to your cluster as your developer user:

```
oc login -u developer -p developer https://ocp1.192.168.122.100.nip.io:8443
```

The `oc` command-line tool has functionality that's similar to `docker exec`. Instead of passing in the short ID for the container, however, you can pass it the pod in which you want to execute the command. After logging in to your `oc` client, scale the app-cli application to two pods with the following command:

```
oc scale dc/app-cli --replicas=2
```

This will cause an update to your app-cli deployment config and trigger the creation of a new app-cli pod. You can get the new pod's name by running the command `oc get pods --show-all=false`. The `show-all=false` option prevents the output of pods in a Completed state, so you see only active pods in the output.

Because the container hostname is its corresponding pod name in OpenShift, you know which pod you were working with using docker directly:

```
$ oc get pods --show-all=false
NAME              READY     STATUS    RESTARTS   AGE
app-cli-1-18k2s   1/1       Running   1          5d
app-cli-1-9hsz1   1/1       Running   0          42m
app-gui-1-165d9   1/1       Running   1          5d
```

Original app-cli pod

New app-cli pod

To get the hostname from your new pod, use the `oc exec` command. It's similar to `docker exec`, but instead of a container's short ID, you use the pod name to specify where you want the command to run. The hostname for your new pod matches the pod name, just like your original pod:

```
$ oc exec app-cli-1-9hsz1 hostname
app-cli-1-9hsz1
```

When you're troubleshooting application-level issues on your cluster, this is an incredibly useful benefit provided by the UTS namespace. Now that you know how hostnames work in containers, we'll investigate the PID namespace.

### 3.3.3 PIDs in containers

Because PIDs are how one application sends signals and information to other applications, isolating visible PIDs in a container to only the applications in it is an important security feature. This is accomplished using the PID namespace.

On a Linux server, the `ps` command shows all running processes, along with their associated PIDs, on the host. This command typically has a lot of output on a busy system. The `--ppid` option limits the output to a single PID and any child processes it has spawned.

From your application node, run `ps` with the `--ppid` option, and include the PID you obtained for your app-cli container. Here you can see that the process for PID 4470 is `httpd` and that it has spawned several other processes:

```
# ps --ppid 4470
   PID TTY          TIME CMD
  4506 ?        00:00:00 cat
  4510 ?        00:00:01 cat
  4542 ?        00:02:55 httpd
  4544 ?        00:03:01 httpd
  4548 ?        00:03:01 httpd
  4565 ?        00:03:01 httpd
  4568 ?        00:03:01 httpd
  4571 ?        00:03:01 httpd
  4574 ?        00:03:00 httpd
  4577 ?        00:03:01 httpd
  6486 ?        00:03:01 httpd
```

Use `oc exec` to get the output of `ps` for the app-cli pod that matches the PID you collected earlier. If you've forgotten, you can compare the hostname in the docker container to the pod name. From inside the container, don't use the `--ppid` option, because you want to see all the PIDs visible from within the app-cli container.

When you run the following command, the output is similar to that from the previous command:

```
$ oc exec app-cli-1-18k2s ps
   PID TTY          TIME CMD
     1 ?        00:00:27 httpd
    18 ?        00:00:00 cat
    19 ?        00:00:01 cat
    20 ?        00:02:55 httpd
    22 ?        00:03:00 httpd
    26 ?        00:03:00 httpd
    43 ?        00:03:00 httpd
    46 ?        00:03:01 httpd
    49 ?        00:03:01 httpd
    52 ?        00:03:00 httpd
    55 ?        00:03:00 httpd
    60 ?        00:03:01 httpd
    83 ?        00:00:00 ps
```

There are three main differences in the output:

- The initial `httpd` command (PID 4470) is listed in the output.
- The `ps` command is listed in the output.
- The PIDs are completely different.

Each container has a unique PID namespace. That means from inside the container, the initial command that started the container (PID 4470) is viewed as PID 1. All the processes it spawned also have PIDs in the same container-specific namespace.

**NOTE** Applications that are created by a process already in a container automatically inherit the container's namespace. This makes it easier for applications in the container to communicate.

So far, we've discussed how filesystems, hostnames, and PIDs are isolated in a container. Next, let's take a quick look at how shared memory resources are isolated.

### 3.3.4 Shared memory resources

Applications can be designed to share memory resources. For example, application A can write a value into a special, shared section of system memory, and the value can be read and used by application B. The following shared memory resources, documented at http://mng.bz/Xjai, are isolated for each container in OpenShift:

- POSIX message queue interfaces in /proc/sys/fs/mqueue
- The following shared memory parameters:
  - msgmax
  - msgmnb
  - msgmni
  - sem
  - shmall
  - shmmax
  - shmmni
  - shm_rmid_forced
- IPC interfaces in /proc/sysvipc

If a container is destroyed, shared memory resources are destroyed as well. Because these resources are application-specific, you'll work with them more in chapter 8 when you deploy a stateful application.

The last namespace to discuss is the network namespace.

### 3.3.5 Container networking

The fifth kernel namespace that's used by docker to isolate containers in OpenShift is the network namespace. There's nothing funny about the name for this namespace. The network namespace isolates network resources and traffic in a container. The resources in this definition mean the entire TCP/IP stack is used by applications in the container.

Chapter 10 is dedicated to going deep into OpenShift's software-defined networking, but we need to illustrate in this chapter how the view from within the container is drastically different than the view from your host.

The PHP builder image you used to create app-cli and app-gui doesn't have the `ip` utility installed. You could install it into the running container using `yum`. But a faster way is to use `nsenter`. Earlier, you used `nsenter` to enter the mount namespace of the docker process so you could view the root filesystem for app-cli.

### The OSI model

It would be great if we could go through the OSI model here. Unfortunately, it's out of scope for this book. In short, it's a model to describe how data travels in a TCP/IP network. There are seven layers. You'll often hear about layer 3 devices, or a layer 2 switch; when someone says that, they're referring to the layer of the OSI model on which a particular device operates. Additionally, the OSI model is a great tool to use any time you need to understand how data moves through any system or application.

If you haven't read up on the OSI model before, it's worth your time to look at the article "The OSI Model Explained: How to Understand (and Remember) the 7 Layer Network Model" by Keith Shaw (*Network World*, http://mng.bz/CQCE).

If you run `nsenter` and include a command as the last argument, then instead of opening an interactive session in that namespace, the command is executed in the specified namespace and returns the results. Using this tool, you can run the `ip` command from your server's default namespace in the network namespace of your app-cli container.

If you compare this to the output from running the `/sbin/ip a` command on your host, the differences are obvious. Your application node will have 10 or more active network interfaces. These represent the physical and software-defined devices that make OpenShift function securely. But in the app-cli container, you have a container-specific loopback interface and a single network interface with a unique MAC and IP address:

```
# nsenter -t 5136 -n /sbin/ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
➥ state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
3: eth0@if12: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue
➥ state UP
    link/ether 0a:58:0a:81:00:2e brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.129.0.46/23 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::858:aff:fe81:2e/64 scope link
       valid_lft forever preferred_lft forever
```

Loopback device in the container

eth0 device in the container

IP address for eth0

MAC address for eth0

The network namespace is the first component in the OpenShift networking solution. We'll discuss how network traffic gets in and out of containers in chapter 10, when we cover OpenShift networking in depth.

In OpenShift, isolating processes doesn't happen in the application, or even in the userspace on the application node. This is a key difference between other types of software clusters, and even some other container-based solutions. In OpenShift, isolation

and resource limits are enforced in the Linux kernel on the application nodes. Isolation with kernel namespaces provides a much smaller attack surface. An exploit that would let someone break out from a container would have to exist in the container runtime or the kernel itself. With OpenShift, as we'll discuss in depth in chapter 11 when we examine security principles in OpenShift, configuration of the kernel and the container runtime is tightly controlled.

The last point we'd like to make in this chapter echoes how we began the discussion. Fundamental knowledge of how containers work and use the Linux kernel is invaluable. When you need to manage your cluster or troubleshoot issues when they arise, this knowledge lets you think about containers in terms of what they're doing all the way to the bottom of the Linux kernel. That makes solving issues and creating stable configurations easier to accomplish.

Before you move on, clean up by reverting back to a single replica of the app-cli application with the following command:

```
oc scale dc/app-cli --replicas=1
```

## 3.4 Summary

- OpenShift orchestrates Kubernetes and docker to deploy and manage applications in containers.
- Multiple levels of management are available in your OpenShift cluster that can be used for different levels of information.
- Containers isolate processes in containers using kernel namespaces.
- You can interact with namespaces from the host using special applications and tools.

SOFTWARE DEVELOPMENT/LINUX

# OpenShift IN ACTION

Duncan • Osborne

Containers let you package everything into one neat place, and with Red Hat OpenShift you can build, deploy, and run those packages all in one place! Combining Docker and Kubernetes, OpenShift is a powerful platform for cluster management, scaling, and upgrading your enterprise apps.

OpenShift in Action is a full reference to Red Hat OpenShift that breaks down this robust container platform so you can use it day-to-day. Starting with how to deploy and run your first application, you'll go deep into OpenShift. You'll discover crystal-clear explanations of namespaces, cgroups, and SELinux, learn to prepare a cluster, and even tackle advanced details like software-defined networks and security, with real-world examples you can take to your own work. It doesn't matter why you use OpenShift—by the end of this book you'll be able to handle every aspect of it, inside and out!

## What's Inside

- Written by lead OpenShift architects
- Rock-solid fundamentals of Docker and Kubernetes
- Keep mission-critical applications up and running
- Manage persistent storage

For DevOps engineers and administrators working in a Linux-based distributed environment.

Jamie Duncan is a cloud solutions architect for Red Hat, focusing on large-scale OpenShift deployments. John Osborne is a principal OpenShift architect for Red Hat.

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

"The first holistic view of OpenShift in print … a soup-to-nuts approach that combines both the developer and operator perspectives."

—From the Foreword by Jim Whitehurst, Red Hat

"At last, a much-needed guide to OpenShift! An excellent read crammed with practical hands-on exercises."

—Michael Bright, Containous

"The definitive guide to the base technologies of the containers era."

—Ioannis Sermetziadis Numbrs Personal Finance

"An essential resource. Gives a clear picture of a complex ecosystem."

—Bruno Vernay, Schneider Electric



ISBN-13: 978-1-61729-483-9
ISBN-10: 1-61729-483-7
54499
9 781617 294839

MANNING $44.99 / Can $59.99 [INCLUDING eBOOK]