Docker and OCI Runtimes

7 min read · Nov 5, 2019

Avijit Sarkar Follow

Listen Share More

Photo by frank mckenna on Unsplash

The objective of this blog post is to deep dive into the various aspcts of OCI aka Open Container Initiative and how Docker fits in there.

What is Open Container Initiative (OCI)?

- The **Open Container Initiative** (**OCI**) is a Linux Foundation project to design

open standards for containers.

Established in June 2015 by Docker and other leaders in the container

ains two specifications: the Runtime Specification
the Image Specification (image-spec).

- OCI runtime spec defines how to run the OCI image bundle as a container.
- OCI image spec defines how to create an OCI Image, which includes an image manifest, a filesystem (layer) serialization, and an image configuration.

What are the various Container Runtimes?

Some of the the popular container runtimes are below ones:

- **containerd**: A CNCF project, it manages the complete container lifecycle of its host system that includes image management, storage and container lifecycle, supervision, execution and networking.
- **lxc**: LXC provides OS level virtualization through a virtual environment that has its own process and network space, it uses linux cgroups and namespaces to provide the isolation.
- **runc:** `runc` is a CLI tool for spawning and running containers according to the OCI specification. It was developed by Docker Inc and donated to OCI as the first OCI runtime-spec compliant reference implementation.
- **cri-o:** CRI-O is an implementation of the Kubernetes CRI (Container Runtime Interface) to enable using OCI (Open Container Initiative) compatible runtimes. It is a lightweight alternative to using Docker as the runtime for Kubernetes.
- **rkt:** rkt is an application container engine developed by

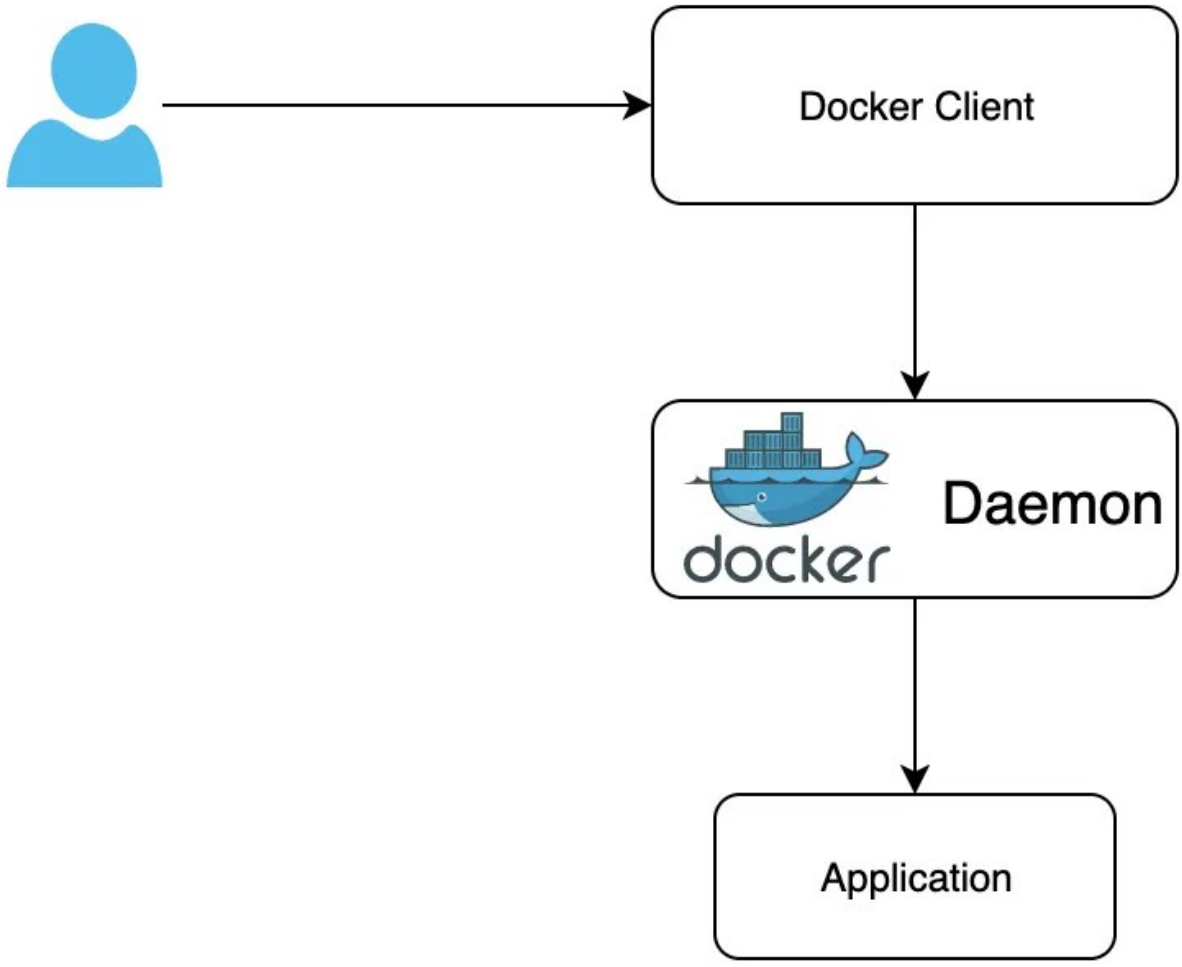
> *So in a nutshell the container runtimes does some or all of the below tasks:*

- Container image management
- Container lifecycle management

- Container creation
- Container resource management

e "rkt" does most of the tasks by itself, where as runtime like of the high level functions and uses others like "runc" for some

Okay, So what is Docker then?

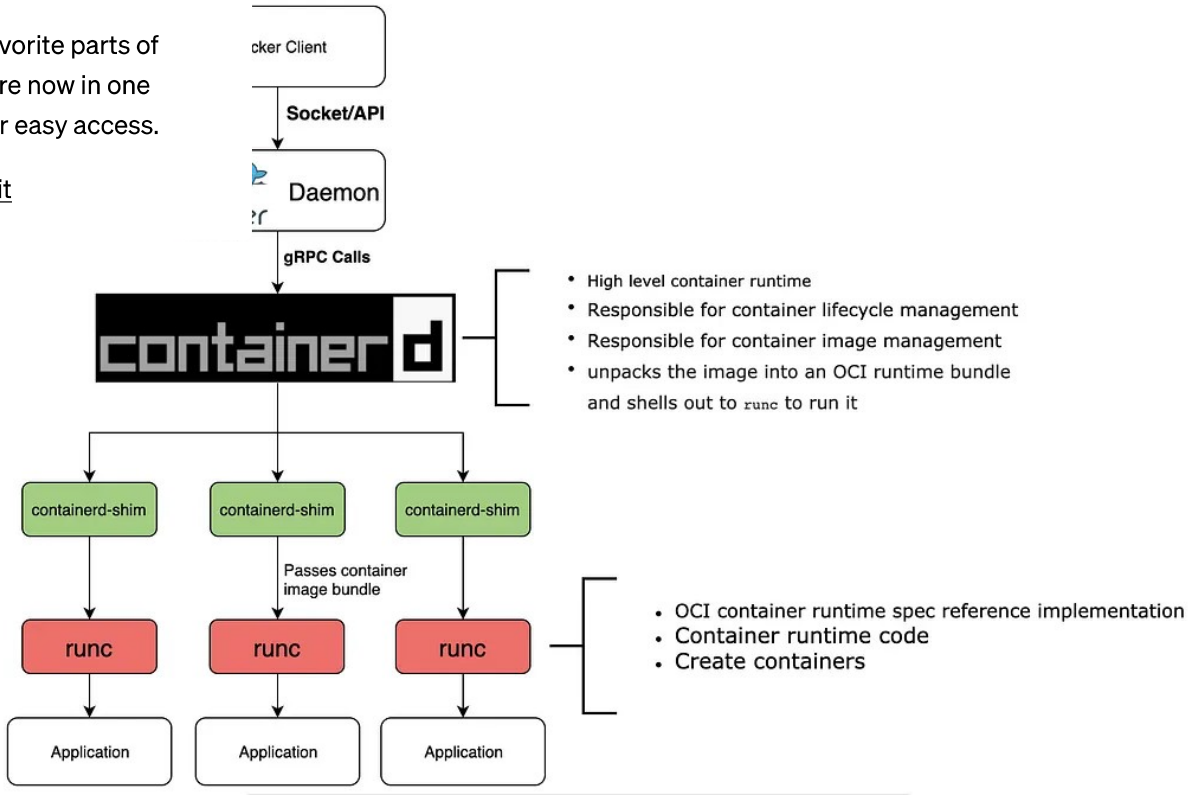


Docker Version < 1.11.0

Docker engine used to be a single monolith before version 1.11.0, the engine was responsible for all the aspects of container management like image management, lifecycle, creation, resource managment etc.

In docker version 1.11.0 major restructuring happened in the Docker engine as

below:



- The low level container runtime features were moved to a different project called runc, it was the first OCI runtime spec reference implementation. Docker donated it to OCI.
- Docker also created the “containerd” project for the supervision of the containers spawns out using “runc”
- containerd has full support for starting OCI bundles and managing their lifecycle.
- Docker published two blog posts behind the rational for createion of “runc” and “containerd”.

So, I have a vagrant CentOS box and have installed docker in it, the running docker processes looks like below:

```
[root@localhost ~]# ps aux | grep docker

root      4499  0.1  4.6 575580 23296 ?        Ssl  05:04   0:00 /
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-path=/usr/
libexec/docker/docker-proxy-current --init-path=/usr/libexec/
docker/docker-init-current --seccomp-profile=/etc/docker/
                  linux-enabled --log-driver=journald --signature-
                  e --storage-driver overlay2

                  .0  2.1 284632 10740 ?        Ssl  05:04   0:00 /
                  ntainerd-current -l unix:///var/run/docker/
                  ker-containerd.sock --metrics-interval=0 --start-
                  e-dir /var/run/docker/libcontainerd/containerd --
shim docker-containerd-shim --runtime docker-runc --runtime-args
--systemd-cgroup=true
root      4696  0.0  0.1  12520   984 pts/0    R+   05:06   0:00
grep --color=auto docker
```


Its very evident here that docker engine under the hood running containerd and runc as container runtimes.

Playing around with containerd

As mentioned above `containerd` is a high level runtime and can be installed on any linux machine following the instructions here

```
[root@localhost ~]# service containerd status
Redirecting to /bin/systemctl status containerd.service
● containerd.service - containerd container runtime
   Loaded: loaded (/etc/systemd/system/containerd.service;
disabled; vendor preset: disabled)
   Active: active (running) since Tue 2019-11-05 05:47:29 UTC;
2min 56s ago
     Docs: https://containerd.io
  Process: 24741 ExecStartPre=/sbin/modprobe overlay (code=exited,
status=0/SUCCESS)
 Main PID: 24742 (containerd)
    Tasks: 19
   Memory: 116.9M
   CGroup: /system.slice/containerd.service
           ├─24399 containerd-shim -namespace default -workdir /
var/lib/containerd/io.containerd.r...
           └─24742 /usr/local/bin/containerd
```

When we install `containerd` it installs the downstream dependencies as well like:

- runc: to run containers

inerd

ɔ support daemonless containers

What all I can do using `containerd`?

- Manage images (like downloading from docker registry)
- Manage containers (like create and running containers)
- Manage namespaces

How to interact with `containerd`?

We will be using the ctr cli for containerd.

- **Downloading images:**

```
[root@localhost ~]# ctr images pull docker.io/library/python:3
docker.io/library/python:3:
total:  333.1  (5.6 MiB/s)
unpacking linux/amd64
sha256:514a95a32b86cafafefcecc28673bb647d44c5aadf06203d39c43b9c3f6
1ed52...
done
```

- **Listing images:**

```
[root@localhost ~]# ctr images ls
REF                            TYPE
DIGEST
SIZE      PLATFORMS
LABELS
docker.io/library/python:3     application/
vnd.docker.distribution.manifest.list.v2+json
sha256:514a95a32b86cafafefcecc28673bb647d44c5aadf06203d39c43b9c3f6
1ed52 333.1 MiB linux/386,linux/amd64,linux/arm/v5,linux/arm/
v7,linux/arm64/v8,linux/ppc64le,linux/s390x,windows/amd64 -

docker.io/library/redis:latest application/
```

```
vnd.docker.distribution.manifest.list.v2+json
sha256:fe80393a67c7058590ca6b6903f64e35b50fa411b0496f604a85c526fb5
bd2d2 34.2 MiB  linux/386,linux/amd64,linux/arm/v5,linux/arm/
                ,linux/ppc64le,linux/s390x
```


g containers:

```
[root@localhost ~]# ctr run -d docker.io/library/python:3 python

[root@localhost ~]# ctr containers ls
CONTAINER    IMAGE                            RUNTIME
python       docker.io/library/python:3
io.containerd.runtime.v1.linux
```

What is this `containerd-shim` (refer the docker image above)?

So based on above command, we start a `python3` container using `containerd`, we know that `containerd` doesn't run the actual container, its the `runc` that responsible for running the container.

- The shim allows for daemonless containers. It basically sits as the parent of the container's process to facilitate a few things.
- It allows the runtimes, i.e. `runc`, to exit after it starts the container. This way we don't have to have the long running runtime processes for containers.
- It allows the container's exit status to be reported back to a higher level tool like docker without having the actual parent of the container's process running.

This is very much evident from the below command, we do see `containerd` running but no `runc`, instead we have the `containerd-shim` process running for the `python` container.

```
[root@localhost ~]# ps aux | grep containerd

root     24742  2.3  8.0 492276 40096 ?        Ssl  05:47   0:28 /
usr/local/bin/containerd

root     24829  0.0  0.7 109100  3940 ?        Sl   06:01   0:00
containerd-shim -namespace default -workdir /var/lib/containerd/
io.containerd.runtime.v1.linux/default/python -address /run/
```

```
containerd/containerd.sock -containerd-binary /usr/local/bin/
containerd
```


)out it please watch this excellent youtube video by

Playing around with runc

So `runc` is the low level container runtime (based on OCI runtime-spec) whose only job is to run containers using OCI image-spec based resource bundle.

In the above section we already saw how `containerd` can be used to start and run containers and under the good it uses `containerd-shim` to run the container using `runc`

What that means is we should be able to run a container manually using `runc` as well.

`runc` binary got installed as `containerd` dependency.

```
[root@localhost ~]# runc --help
NAME:
   runc - Open Container Initiative runtime

runc is a command line client for running applications packaged
according to
the Open Container Initiative (OCI) format and is a compliant
implementation of the
Open Container Initiative specification.

runc integrates well with existing process supervisors to provide
a production
container runtime environment for applications. It can be used
with your
existing process monitoring tools and the container will be
spawned as a
direct child of the process supervisor.

Containers are configured using bundles. A bundle for a container
is a directory
that includes a specification file named "config.json" and a root
filesystem.
The root filesystem contains the contents of the container.
```

```
To start a new instance of a container:

# runc run [ -b bundle ] <container-id>

-id>" is your name for the instance of the
 name you provide for the container instance must

ing the bundle directory using "-b" is optional.
The default
value for "bundle" is the current directory.
```


Based on above documentation all we need is an OCI image-spec based bundle (a spec file named `config.json` and the container image root filesystem) to run a container.

Lets create the container image root filesystem:

```
[root@localhost rootfs]# mkdir -p alpine/rootfs
[root@localhost rootfs]# cd alpine
[root@localhost alpine]# docker export $(docker run -d alpine) |
tar -C rootfs -xv

[root@localhost alpine]# pwd
/root/alpine/rootfs/alpine
[root@localhost alpine]# ls rootfs/
bin  dev  etc  home  lib  media  mnt  opt  proc  root  run  sbin
srv  sys  tmp  usr  var
```

We just used `docker` here to run a `alpine` image and then exported the container as tarball. Extracting the tarball gave us the root filesystem for the alpine image.

Now lets create the runtime-spec config file:

```
[root@localhost alpine]# runc spec
[root@localhost alpine]# cat config.json
{
 "ociVersion": "1.0.1-dev",
 "process": {
  "terminal": true,
  "user": {
   "uid": 0,
   "gid": 0
  },
```

```
    "args": [
     "sh"
    ],
```



```
    "env": [
     "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
    ...

    ...
    "readonlyPaths": [
     "/proc/asound",
     "/proc/bus",
     "/proc/fs",
     "/proc/irq",
     "/proc/sys",
     "/proc/sysrq-trigger"
    ]
   }
  }
```

Now to run a container is as simple as below command:

```
[root@localhost alpine]# runc run alpine-container
/ #
/ #
/ # pwd
/
/ # echo "runc started this container using the rootfs of alpine"
runc started this container using the rootfs of alpine
/ #
```

Based on the spec `config.json` file, the default command is `sh` for the image, so we entered into the shell of the container.

What we can summarize here is that the whole container runtimes under OCI is very flexible and pluggable, as long we are meeting the image-spec and runtime-spec requirements we can have custom runtimes for containers.

Docker is no longer a monolith and under the hood its using the `containerd` and