Docker Deep Dive

Chapter #4

Docker Engine

Docker

UNDER THE HOOD

www.networknuts.net

DOCKER ENGINE

Under the hood

Docker Engine

DOCKER ENGINE

The core software that runs and manages containers.

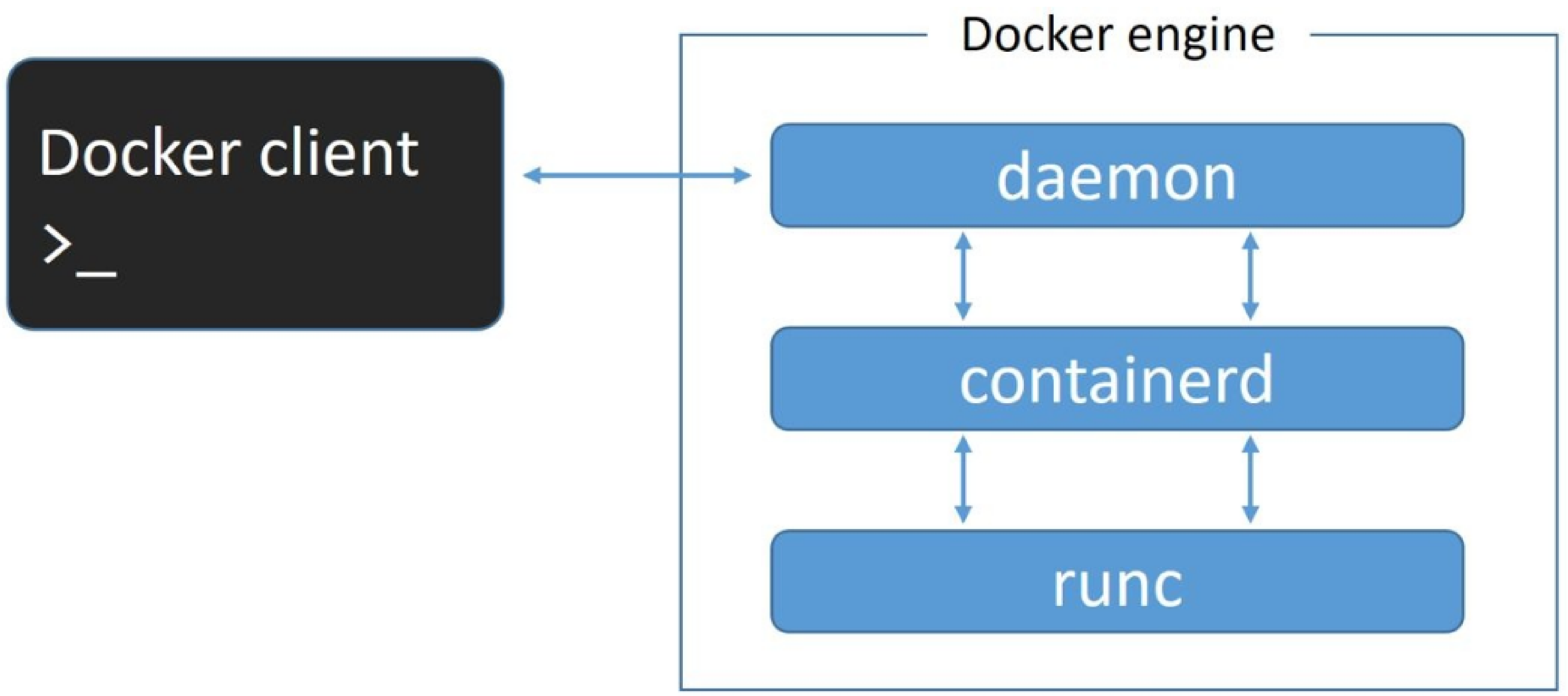
MODULAR DESIGN

Its based on open-standards outlined by the Open Container Initiative (OCI).

MAJOR COMPONENTS

Docker client, the Docker daemon, containerd, and runc.

Docker Engine: high level view



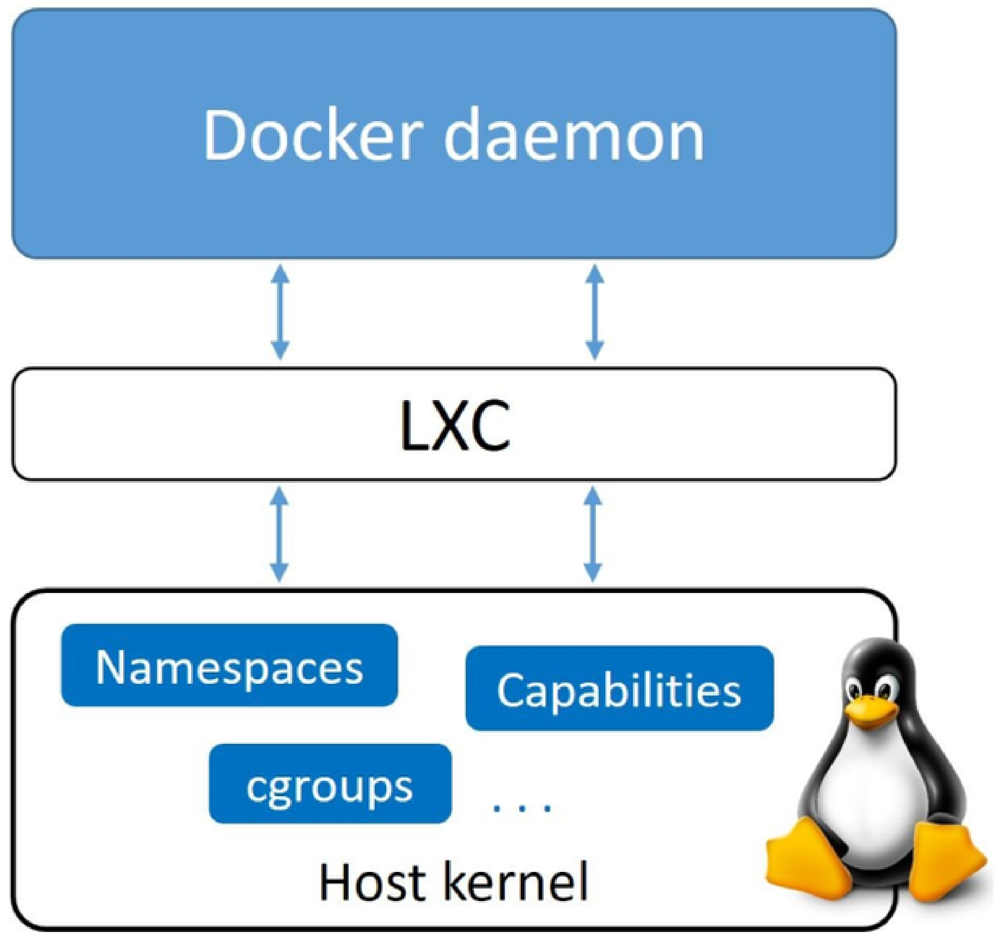
Docker Engine

When Docker was first released, the Docker engine had two major components:

1. The Docker daemon (hereafter referred to as just “the daemon”)
2. LXC

The Docker daemon was a monolithic binary. It contains all of the code for the Docker client, the Docker API, the container runtime, image builds, and much more.

The LXC component provided the daemon with access to the fundamental building-blocks of containers such as kernel namespaces and control groups (cgroups).



Getting rid of LXC

- First up, LXC is Linux-specific. This was a problem for a project that had aspirations of being multi-platform.
- Second up, being reliant on an external tool for something so core to the project was a huge risk that could hinder development.

As a result, Docker. Inc. developed their own tool called **libcontainer** as a replacement for LXC.

The goal of libcontainer was to be a platform-agnostic tool that provided Docker with access to the fundamental container building-blocks that exist inside the OS.

Libcontainer replaced LXC as the default execution driver in Docker 0.9.

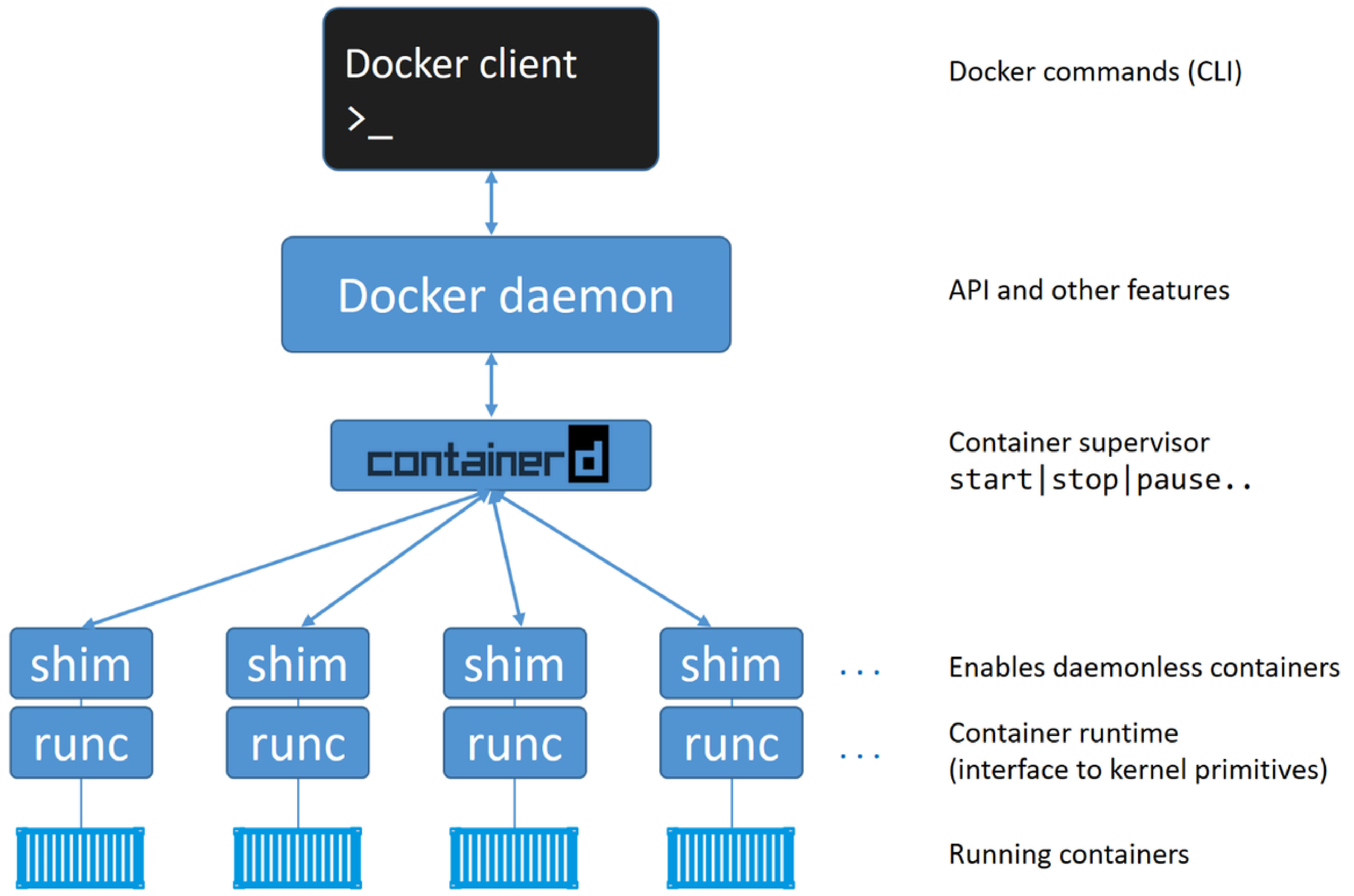
Getting rid of the monolithic Docker daemon

Over time, the monolithic nature of the Docker daemon became more & more problematic:
1. It’s hard to innovate on.
2. It got slower.
3. It wasn’t what the ecosystem (or Docker, Inc.) wanted.

Docker, Inc. was aware of these challenges, and began a huge effort to break apart the monolithic daemon and modularize it.

The aim of this work is to break out as much of the functionality as possible from the daemon, and re-implement it in smaller specialized tools. These specialized tools can be swapped out, as well as easily used by third parties to build other tools.

Now all of the container execution and container runtime code entirely removed from the daemon and refactored into small, specialized tools.



Docker daemon no longer contains any container runtime code - all container runtime code is implemented in a separate OCI-compliant layer.

By default, Docker uses a tool called **runc** for this. runc is the reference implementation of the OCI container-runtime-spec, and a goal of the runc project keep runc in lockstep with the OCI spec.

runc

runc is the reference implementation of the OCI container-runtime-spec. Docker, Inc. was heavily involved in defining the spec and developing runc.

runc is small. It’s effectively a lightweight CLI that wraps around libcontainer.

It has a single purpose in life - **to create containers**. And it’s damn good at it. And fast!

runc is also known as a container runtime.

containerd

In order to use runc, the Docker engine needed something to act as a bridge between the daemon and runc. This is where containerd comes into the picture.

It’s a container supervisor - the component that is responsible for container lifecycle operations such as; starting and stopping containers, pausing and un-pausing them, and destroying them.

Like runc, containerd is small, lightweight, and designed for a single task in life - containerd is only interested in container lifecycle operations.

containerd was developed by Docker, Inc. and donated to the Cloud Native Computing Foundation (CNCF).

How container start

The most common way of starting containers is using the Docker CLI.

```
# docker  run  --name=mywebserver  -it  centos:latest /bin/bash
```

Docker client converts them into the appropriate API payload and POSTs them to the correct API endpoint.

Once the daemon receives the command to create a new container, it makes a call to containerd.

Remember that the daemon no-longer contains any code to create containers!

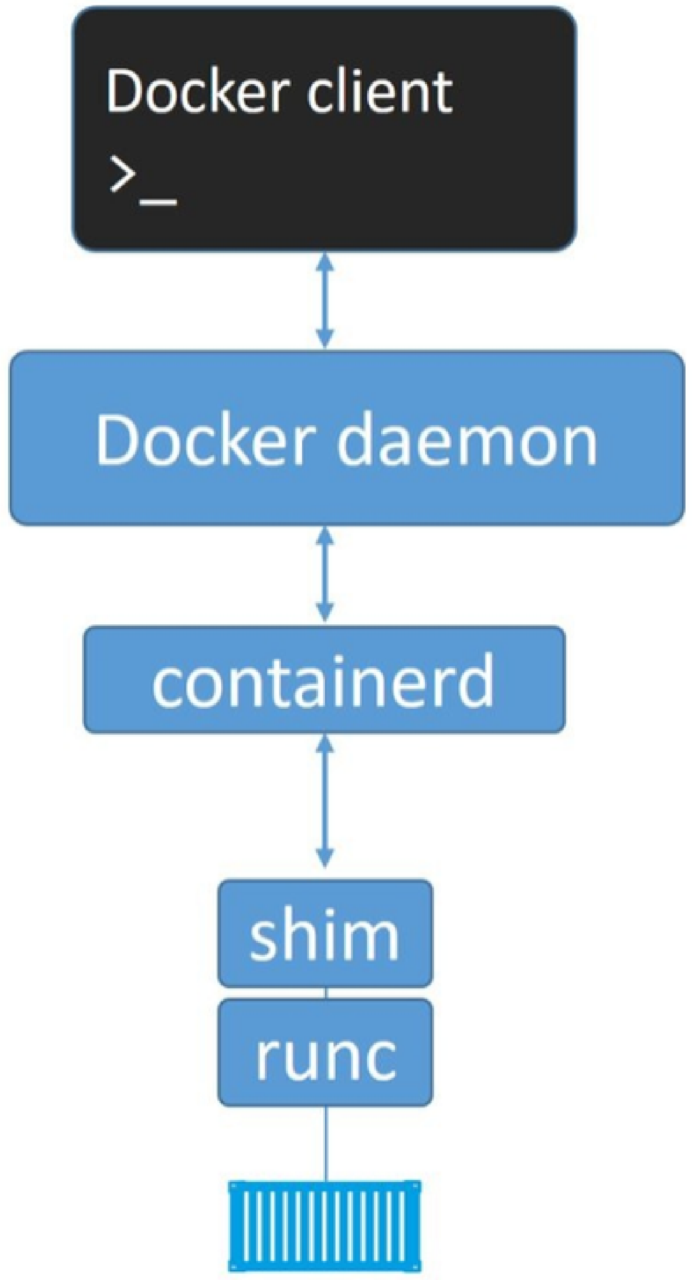
How container start

Despite its name, containerd cannot actually create containers. It uses runc to do that. It converts the required Docker image into an OCI bundle and tells runc to use this to create a new container.

runc interfaces with the OS kernel to pull together all of the constructs necessary to create a container (in Linux these include namespaces and cgroups).

The container process is started as a child-process of runc, and as soon as it is started runc will exit.

How container start



Issue `docker container run` command to **Docker API**.

Receive instruction at API endpoint.
Instruct **containerd** to start new container.

Give **runc** OCI bundle (image).
Instruct **runc** to create container.

Create container runtime

Having all of the logic and code to start and manage containers removed from the daemon means that the entire container runtime is decoupled from the Docker daemon.

We sometimes call this “daemonless containers”, and it makes it possible to perform maintenance and upgrades on the Docker daemon without impacting running containers!

In the old model, where all of container runtime logic was implemented in the daemon, starting and stopping the daemon would kill all running containers on the host. This was a huge problem in production environments - especially when you consider how frequently new versions of Docker are released! Every daemon upgrade would kill all containers on that host - not good.

Fortunately, this is no longer a problem.

What is shim?

shim is integral to the implementation of daemonless containers.

containerd uses runc to create new containers. It forks a new instance of runc for every container it creates. However, once each container is created, its parent runc process exits. This means we can run hundreds of containers without having to run hundreds of runc instances.

Once a container's parent runc process exits, the associated containerd-shim process becomes the container's parent process. Some of the responsibilities the shim performs as a container's parent include:

- Keeping any STDIN and STDOUT streams open so that when the daemon is restarted, the container doesn't terminate due to pipes being closed etc.
- Reports the container's exit status back to the daemon.

How its implemented on Linux

On a Linux system, the components we’ve discussed are implemented as separate binaries as follows: -

- dockerd (the Docker daemon)
- docker-containerd (containerd)
- docker-containerd-shim (shim)
- docker-runc (runc)

What's left in daemon

Major functionality that still exists in the daemon includes; image management, image builds, the REST API, authentication, security, core networking, and orchestration.

WHAT WE LEARNED

Docker engine is modular in design and based heavily on open-standards from the OCI.

Container execution is handled by containerd. Its a container supervisor that handles container lifecycle operations.

Docker uses runc as its default container runtime. runc is the de facto implementation of the OCI container-runtime-spec and expects to start containers from OCI-compliant bundles.

Functionality currently still inside of the Docker daemon include, but is not limited to: the API, image management, authentication, security features, core networking.