Deployment & Virtualization

Part 3 — Manage multiple containers

Joseph Chazalon {`firstname.lastname@epita.fr`}

March 2025

EPITA Research Laboratory (LDE)

Products as a collection of running services

Why we need more than 1 running container

- Separate software components (database, front-end, back-end...)
- Load balancing and replication
- Rolling upgrades or product variants
- ...

Challenges

- Ensure all microservices run consistently
- Upgrade them
- Deploy them on multiple hosts and configure the network accordingly
- Spawn multiple replicates to balance the load
- Limit the number of builds and distribute (private) images
- Continuous testing, integration and deployment (blue/green deploy, canary releases...)
- ...

How to manage multiple Docker containers?

Docker compose

> *The Compose specification allows one to define a platform-agnostic container based application. Such an application is designed as a set of containers which have to both run together with adequate shared resources and communication channels.*

Docker compose can manage/configure multiple containers on a single host or on a Docker swarm cluster.
You can configure:

- services (running containers)
- ports
- networks to connect services together
- volumes
- deployment (resources, cluster)

Use cases:

- Manage containers on a production server
- Avoid shell script to start your container

Docker compose example: hello world

Hello-world example:

```
version: '3'
services:
    hello:
        image: hello-world
```

Docker compose example: hello world

```
$ docker-compose up
Creating network "tmp_default" with the default driver
Pulling hello (hello-world:)...
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:b8ba256769a0ac28dd126d584e0a2011cd2877f3f76e093a7ae5
Status: Downloaded newer image for hello-world:latest
Creating tmp_hello_1 ... done
Attaching to tmp_hello_1
hello_1  |
hello_1  | Hello from Docker!
hello_1  | This message shows that your installation appears to be
hello_1  |
(…)
hello_1  |
tmp_hello_1 exited with code 0
```

Docker compose example: based on a redis image

Docker compose example from https://docs.docker.com/compose/gettingstarted/

```yaml
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
  redis:
    image: "redis:alpine"
```

Docker compose stores the run parameters for a container.

Docker compose example: front/back-end web application

```
(External user) --> 443 [frontend network]
                            |
                  +--------------------+
                  |  frontend service  |...ro...<HTTP configuration>
                  |      "webapp"      |...ro...<server certificate> #secured
                  +--------------------+
                            |
                        [backend network]
                            |
                  +--------------------+
                  |  backend service   |  r+w   ___________________
                  |     "database"     |=======( persistent volume )
                  +--------------------+        \_________________/
```

Docker compose example: front/back-end web application

The example application is composed of the following parts:

- 2 **services**, backed by Docker images: `webapp` and `database`
- 1 **secret** (HTTPS certificate), injected into the frontend
- 1 **configuration** (HTTP), injected into the frontend
- 1 **persistent volume**, attached to the backend
- 2 **networks**

Docker compose example: front/back-end web application

```yaml
services:
  frontend:
    image: awesome/webapp
    ports:
      - "443:8043"
    networks:
      - front-tier
      - back-tier
    configs:
      - httpd-config
    secrets:
      - server-certificate

  backend:
    image: awesome/database
    volumes:
      - db-data:/etc/data
    networks:
      - back-tier
```

```yaml
volumes:
  db-data:
    driver: flocker
    driver_opts:
      size: "10GiB"

configs:
  httpd-config:
    external: true

secrets:
  server-certificate:
    external: true

networks:
  # The presence of these objects
  # is sufficient to define them
  front-tier: {}
  back-tier: {}
```

Configurations, secrets, environment

Commodity tools to store and distribute configurations and secrets among a Docker swarm cluster.

- `docker config ls`
- `docker secret ls`

Configs and secrets in Docker compose files can also use local files distributed with the compose file.

Another common practice is to use **environment files** to define a set of environment variables for a service.

```
# docker-compose.yaml (1)
env_file: .env
```

```
# .env file sample content
RACK_ENV=development
VAR="quoted"
USER_INPUT=
```

```
# docker-compose.yaml (2)
env_file:
  - ./a.env
  - ./b.env
```

```
# docker-compose.yaml (3)
environment:
  RACK_ENV: development
  SHOW: "true"
  USER_INPUT:
```

Docker swarm

- Manage docker cluster
- Integrated in Docker with ‘docker swarm’ subcommand (using swarmkit)
- Internal load balancer
- Docker compose can deploy services directly to Docker swarm cluster
- Easy and lightweight setup
- No dashboard, but you can use https://www.portainer.io, https://swarmpit.io or others...

docker run, docker compose, docker swarm

run: 1 container, 1 machine

compose: n containers, 1 machine

swarm: n containers, n machines

Kubernetes (k8s)

Google version of Docker swarm with:

- Almost all docker swarm functionalities
- Auto-scaling
- Large community
- Complex installation/configuration
- Incompatible with docker CLI and compose tools
- Huge ecosystem

Alternatives to Docker

Features

1. low level container runtime (runc, crun), OCI runtime
2. high level container runtime (containerd, CRI-O)
3. build images (docker, podman, buildah), OCI image
4. registry (docker hub, quay, gitlab registry, ghcr.io)

LXC (Linux Container)

- Use case: lightweight VM
- Used to be the runtime backend of Docker before containerd/runc
- First Linux container tool
- Doesn't manage images

Example of a config file:

```
lxc.rootfs.path = /var/lib/lxc/playtime/rootfs
lxc.uts.name = playtime
lxc.arch = x86_64
lxc.include = /usr/share/lxc/config/common.conf
lxc.net.0.type = veth
lxc.net.0.link = br0
lxc.net.0.flags = up
lxc.net.0.name = eth0
lxc.net.0.hwaddr = ee:ec:fa:e9:56:7d
lxc.net.0.ipv4.address = 192.168.0.3/24
lxc.net.0.ipv4.gateway = 192.168.0.1
```

Systemd-nspawn

- Modern version of LXC based on systemd
- Low level command from systemd: systemd-nspawn, machinectl
- Doesn't manage images: you have to manually initialize your root file system with tools like debootstrap or pacstrap
- Support for OCI runtime (Open Container Initiative)

```
systemd-nspawn --machine=cbuster --boot
```

RKT (outdated)

RKT (originated from CoreOs):

- Compatibility with docker image format
- Try to be more secure by default
- Use systemd-nspawn to run container
- Actually the only alternatives to docker

Hello-world example:

```
rkt --insecure-options=image run docker://hello-world
```

OpenVZ, BSD Jails, Linux VServer, ...

Podman + Buildah

- Support multiple image formats including the OCI and Docker image formats
- High level container runtime
- Docker-compatible CLI interface with podman
- Rootless and daemonless
- Buildah to build OCI images

Serious production environments

Docker and equiv.: local, isolated view

The OCI standards barely talk about container coordination, deployment, replication, etc.

The OCR standards focus on:

- **Images:** what is an isolated service (base image, command, parameters…)
- **Runtimes:** how it should be run (isolation, exposed ports…)
- **Repositories:** how to send/receive images (API endpoints, authentication…)

Orchestration: Kubernetes (aka K8s)

https://www.redhat.com/en/topics/containers/what-is-kubernetes

- Provision images on production machines
- Sharing files and file systems between machines
- Set up (private) network interfaces among machines and containers
- Schedule the execution of containers
- Ensure availability of services, restart failing containers
- Manage reverse proxies and load balancing

There are simpler versions of Kubernetes like k3s (mini Kubernetes).

IaaS or PaaS?

Infrastructure as a Service approaches (IaaS)

Virtualization systems now support containers but target base-metal provisioning

- VMWare cluster management (vSphere, ESXi, etc.)
- Proxmox
- Apache Mesos
- OpenStack
- …

Platform as a Service (PaaS) *Usually container-centric approaches*

- Kubernetes
- OpenShift (leverages Kubernetes and Docker)
- Docker Swarm
- …

MLOps stacks

Generic application/services managers

- Rancher?

Specific to ML workloads

- Kubeflow KServe (serverless model serving)
- Ray.io (“easily” build servers for ML models)
- (many targeting LLMs specifically)
- ...