Docker overview

Florent Gluck - Florent.Gluck@hesge.ch

April 7, 2022

Docker history

- End of 2013: dotCloud Inc. made public and open-source its tool for managing customer applications: a client/server framework called **docker**
- In a few months → phenomenal developers and users attraction!
- Consequently dotCloud focused its core business on docker and changed its name to Docker, Inc.

Docker's goal

- Despite their history, containers (before Docker) haven't achieved large-scale adoption
- A large part of this results from their complexity: containers can be complex, hard to set up, and difficult to manage and automate
- Docker aimed to change that!

Why Docker?

- Isolation
- Simplicity
- Lightweight
- Workflow
- Community

What is Docker?

- Docker = open-source engine that **automates the deployment of applications into containers**
- **Platform** for developers/sysadmins to develop, ship, and run applications, based on containers
- Based on `libcontainer` and `runC`, but originally used libvirt and LXC
- **Simplifies** and standardizes the creation and **management** of containers
- Provides a **RESTful API** to perform queries and actions
- Written in Go

Docker components

Docker composed of 4 main components:

(a) **Docker Engine** (docker client + server)

(b) **Images**

(c) **Registries**

(d) **Containers**

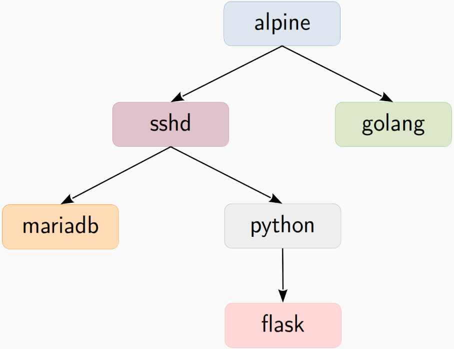
(a) Docker Engine

- Docker is a client-server application
- Docker clients talk to the docker server (`dockerd` daemon) which does all the work
- Docker ships with a command line client (`docker`) and a RESTful API[1] to interact with `dockerd`
- Client and daemon can run on the same host (local) or be on different hosts (remote)

[1] https://docs.docker.com/engine/api/latest/

(b) Images

- **Every container is instantiated from an image** → encloses a program within the image's filesystem
- **Hierarchy** of images: images have a parent ↔ children relationship.



(b) Images

- Images are to containers what classes are to instances in OOP (Object Oriented Programming)
- An image includes:
  - A full-fledged, isolated root filesystem (e.g. minimal filesystem provided by an Ubuntu distribution)
  - Process info (e.g. default process to execute when a container is created from an image or its command-line arguments)
  - Network info (e.g. which ports should be exposed)

(c) Registries

- Docker stores images that users build in registries
- Two types of registries, public and private:
  - Docker, Inc., operates the public registry for images, called the *Docker Hub*:
    - Anyone can create an account on Docker Hub and use it to share and store their own images
  - One can also run their own private registry
    - e.g. allows one to store images behind a firewall, etc.

(c) Registries

- Docker Hub hosts the main docker image registry
  - provides official images for Linux distributions and popular services (web servers, DBs, languages, etc.)
- The docker daemon has an internal registry of downloaded images
  - it caches them (on the host) to avoid downloading them again

Registries: image names

- Image names follow a precise format:

```
<repository>[:<tag>]
where <repository> ::= [<user>/]<base name>
```

- On Docker Hub, only official repositories are at root level, without a leading `<user>/`
- The same image can be associated with multiple tags, e.g. `ubuntu:20.04` and `ubuntu:focal`
- Regardless of the tags, an image is always identified by a unique hexadecimal id
- Tags for a given image can be retrieved using the registry API[2] (or via the Docker Hub website)

[2] https://docs.docker.com/registry/spec/api/

Internal image registry

- To list the images available in the internal registry, execute:

```
docker images
```

- Images are automatically downloaded by docker when needed
- Image can be manually downloaded via:

```
docker pull <image name>
docker pull <image id>
```

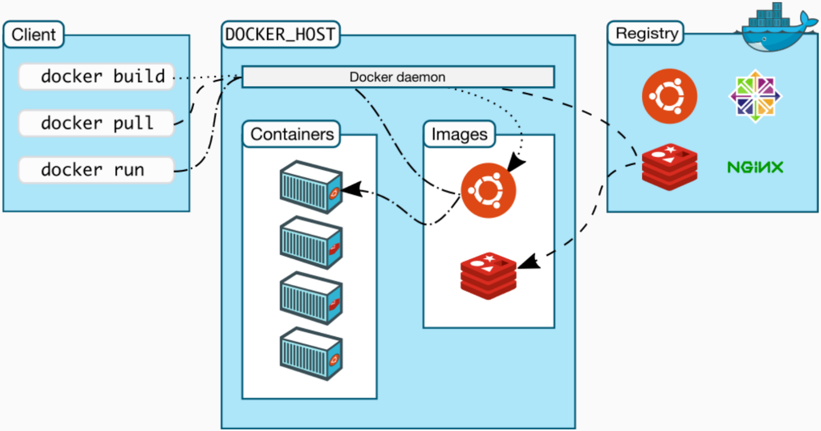
  if tag is missing in `<image name>`, `:latest` will be downloaded

- Both images and containers can be referenced via their user-friendly name or their id (full or shortened)

(d) Containers

- Docker helps to build and deploy containers inside of which one can package applications and services
- Containers are launched from images and can contain one or more running processes
- **Images = building** aspect of docker → **immutable** (static)
- **Containers = running** aspect of docker → **mutable** (dynamic)
- A docker container is:
    - An image format
    - A set of standard operations
    - An execution environment

Docker architecture and workflow



Docker daemon

- Docker daemon = `dockerd` program
- Containers are ran by the docker daemon, **not** the docker client
- By default, docker daemon and docker client are installed on the same host → local access
  - client ↔ daemon communicate through a local UNIX socket (`/var/run/docker.sock`)
- For remote access, they can be configured to communicate through SSH or HTTPS (TLS) socket

Remote docker daemon access through ssh

- Mechanism requires ssh public key authentication (key pair)
- The local user (client side) must also exist on the remote host and be in the docker group
- On the client, two ways to specify how to connect to the remote docker daemon:

```
export DOCKER_HOST=ssh://username@ip
```

or:

```
docker -H ssh://username@ip command
```

Rootless docker daemon

- By default, docker daemon runs as root
  - consequently, root (UID 0) in the container is also root on the host!
  - **security risk!**
- Docker daemon can be configured to be rootless[3], i.e. it runs as a non-root user
  - strongly advisable, but requires specific configuration
  - unfortunately, Docker doesn't integrate well with `systemd`

[3] https://docs.docker.com/engine/security/rootless/

Docker client

- Non-root users cannot execute the docker client
- To use the docker client, a user must be part of the `docker` group
  - **do not** run the docker client with `sudo`[4]!
  - instead, add the user to the docker group:

```
sudo usermod -a -G docker <user>
```

[4] Rule of thumb: **never** use sudo unless it's truly required!

Docker client commands

The set of docker (client) command line arguments is very complete:

- Running `docker` shows the available commands
- Runing `docker help <command>` shows the command-related documentation

Basic docker client commands

`docker info`	Display system-wide information
`docker search`	Search the Docker Hub for images
`docker pull`	Pull an image from a registry
`docker run`	Run a command in a new container
`docker exec`	Run a command in a running container
`docker ps`	List containers
`docker start`	Start one or more stopped containers
`docker stop`	Stop one or more running containers
`docker images`	List internal images
`docker rm`	Remove one or more containers
`docker rmi`	Remove one or more images

Example: a first container

- Simplest example:

```
docker run debian echo "Hello world"
```

- Retrieve the `debian:latest` image (from Docker Hub if not found in the internal registry)
- Execute the `echo` program present in the debian image
- Print the output to the host's terminal and terminate the execution of the container
- Once the `echo` program finishes, the container is terminated, **but it remains** on the host!
  - Add the `--rm` option to remove the container

Creating an interactive container

- To create a container and launch a bash shell inside it:

```
docker run -it debian /bin/bash
```

- Execute the program `/bin/bash` within the image's filesystem
- Connect the container's `stdin` to a pseudo-tty backed by the current terminal to support user input (→ *interactive*)
- As a result the host's prompt is replaced by a root prompt within the container
    - by using bash's commands → possible to browse and alter the container's filesystem

Listing containers and states

- `docker ps` lists all running containers
- `docker ps -a` lists all containers, including non-running ones
- When the entry point command of a container terminates, the container exits → state switches from `running` to `exited`
- This is why when one types `exit` or presses `ctrl+d` in a running Ubuntu container (which features bash as its entry command), the container exits
- `docker history image` display the history of `image`, including the entry point command

Starting a container

- To re-execute the process (*entry point*) of a stopped container, or to start a container created via `docker create`, use:

```
docker start <container>
```

- The container's process will be started again with the very same parameters specified at creation time
- For interactive processes (e.g. bash), it's necessary to rebind the container's stdin to the current terminal with:

```
docker start -ia
```

- Alternatively, use `docker attach` after starting the container

Executing a program in the container

- Might be useful to start other processes within a container (e.g. bash to explore or alter the filesystem):

```
docker exec <exec args> <container> <cmd> <cmd args>
```

- Example: execute `qalc` and connect it to the current terminal

```
docker exec -it <container> /usr/bin/qalc
```

Running background containers

- The `-d` parameter of `docker run` creates a container running in the background (i.e. *detached*)
- `stdout` and `stderr` won't be shown on the current console, but will be redirected to docker's logging infrastructure

Reading a container's output

- Docker redirects stdout and stderr of every container both to the current terminal and to docker's internal logs
- With background containers, only logs are available
- To output stdout and stderr logs for a given container

```
docker logs <container>
```

  - -f to keep the log visible, with updates printed in real-time
  - --tail=N where N is the number of most recent lines to show

Stopping a container

- A container automatically stops as soon as its entry point process stops
- A container can also be stopped from the host's command line:

```
docker stop <container>
```

- To send a signal to a container

```
docker kill <container>
```

  - unless specified, the default signal is `SIGKILL`

Inspecting a container

- To show detailed information about a container and its process, execute

```
docker inspect <container>
```

- Information is output in the JSON format
- Works for both active and stopped containers

Removing a container

- To remove a stopped container:

```
docker rm <container>
```

- Use `-f` to force deletion, typically for still running containers
- To remove all containers on the host:

```
docker rm -f $(docker ps -aq)
```

Limiting a container

- `docker run` accepts arguments to impose various limits on a container
- Non-exhaustive list (`docker run --help` for the full list):

`--cpu <d>`	limit the container to `d` “CPUs” (`d` is decimal!)
`-m <n>m`	limit the container to `n` MB of RAM
`--device-read-bps`	limit read rate from a device; format: `<device-path>:<number>[<unit>]` (unit: kb, mb, gb)
`--device-write-bps`	limit write rate to a device; format: `<device-path>:<number>[<unit>]` (unit: kb, mb, gb)
`--cap-drop list`	drop the given capabilities

Using the docker API

- Docker API described at https://docs.docker.com/engine/api/latest/
- By default, docker daemon uses UNIX socket `/var/run/docker.sock`
- Basic examples, using `curl` for HTTP requests and `jq` for JSON parsing:

```
# docker info
curl -X GET --unix-socket /var/run/docker.sock
     http://localhost/info|jq .

# docker images
curl -X GET --unix-socket /var/run/docker.sock
     http://localhost/images/json|jq '.[].RepoTags'
```

Docker installation

- To install the Docker Engine (daemon + client) on Ubuntu 20.04:

```
sudo apt-get install docker.io
```

- Make sure the docker service (deamon) is started:

```
sudo systemctl start docker.service
```

- Enable the docker service at boot time:

```
sudo systemctl enable docker.service
```

- On the client (local or remote), add your user to the `docker` group:

```
sudo usermod -a -G docker <user>
```

Resources

- Docker in Action, Jeff Nickoloff, Manning 2016
- The Docker Book, James Turnbull, December 2018
- Docker official documentation: https://docs.docker.com