Docker Overview

Florent Glück - florent.gluck@hesge.ch

April 07, 2025

ISC - HEPIA

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 $\longrightarrow$ phenomenal developers and users attraction!
- Consequently dotCloud focused its core business on docker and changed its name to Docker, Inc.

Why Docker?

- Despite their history, containers (before Docker) didn't achieve large-scale adoption
- Main reason is their complexity: containers can be complex, hard to set up, and difficult to manage and automate
- Docker aimed to change that!

Docker's goals

- **Simplifies** the creation and management of containers

Docker's goals

- **Simplifies** the creation and management of containers
- **Ease** the process of packaging/shipping applications **independently of the underlying OS**

Docker's goals

- **Simplifies** the creation and management of containers
- **Ease** the process of packaging/shipping applications **independently of the underlying OS**
- Make applications deployment **reproducible**

Docker's goals

- **Simplifies** the creation and management of containers
- **Ease** the process of packaging/shipping applications **independently of the underlying OS**
- Make applications deployment **reproducible**
- Increase applications' **security**

What is Docker?

- Open-source engine[1] that **automates the deployment of applications into containers**
- **Platform** for developers/sysadmins to develop, ship, and run applications, based on containers

[1] Written in Go

Docker components

(a) **Docker Engine**
- docker client
- docker daemon

(b) **Images**

(c) **Containers**

(d) **Registries**

(a) Docker Engine

- Docker has a client-server architecture
- Docker clients communicate with the docker server (`dockerd` daemon) which does all the work
- Docker ships with a CLI[1] client (`docker`) and a RESTful API[2] to interact with `dockerd`
- Client and daemon can run on the same host (local) or be on different hosts (remote)

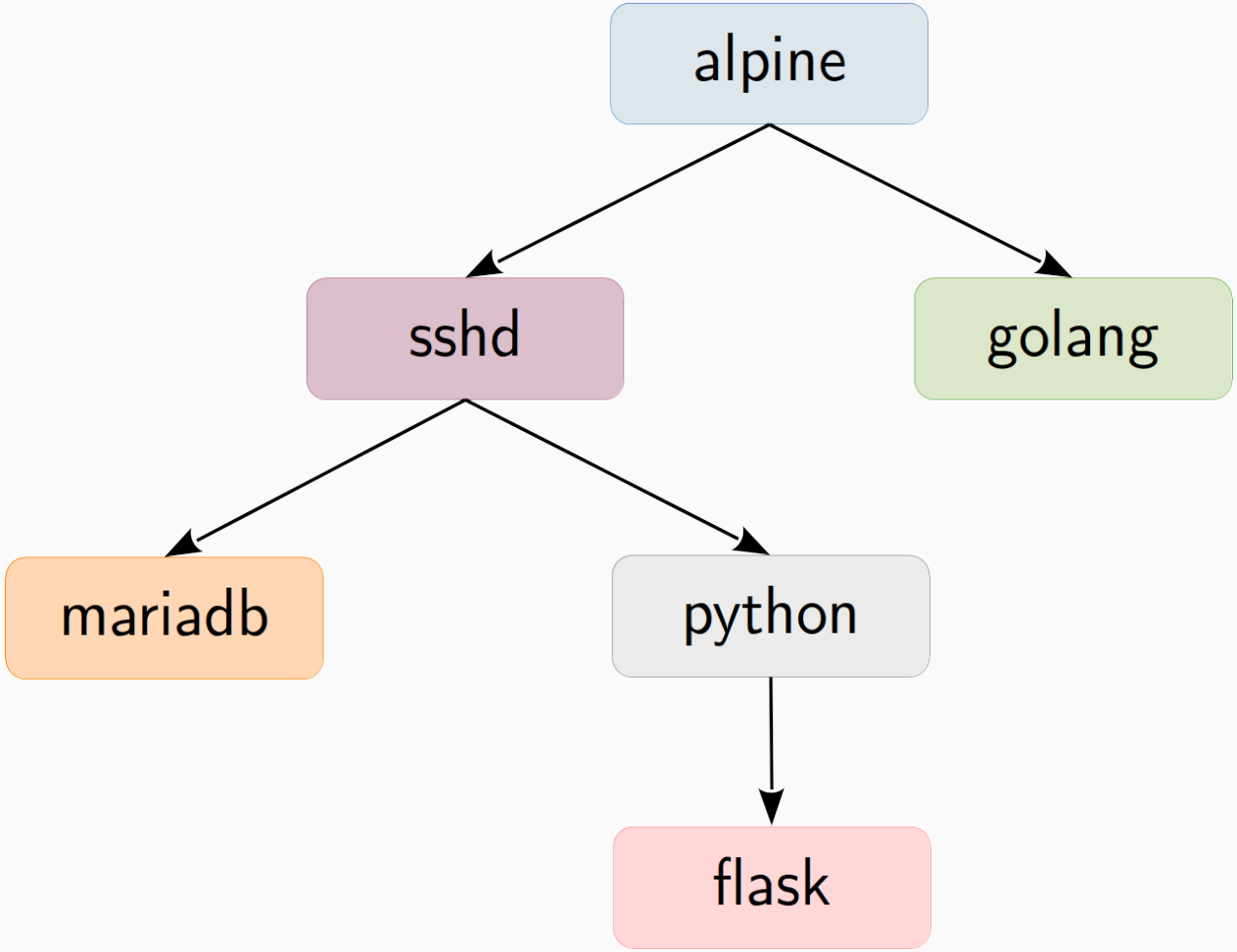
[1] Command-Line Interface

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

(b) Images

- **Every container is instantiated (created) from an image**
  - an image is a file tree containing programs and their dependencies
- **Hierarchy** of images
  - images have a parent $\leftrightarrow$ children relationship

(b) Images



(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 a GNU/Linux distribution)
  - the **default program to execute** when a container is created from the image
    - this default program is also called the **entry-point program**
  - network information (e.g. which ports should be exposed)

(c) Containers

- **Containers are instanciated from images** and contain one or more running processes
- **A container terminates when its entry-point process terminates**, **regardless** of the number of other processes **still running** in the container
- Docker helps build and deploy containers, inside which applications and services can be packaged

(c) Containers

- Analogy between container ↔ image and process ↔ program:
  - a container is an instance of an image
  - a process is an instance of a program
- **Images** = **building** aspect of docker
  - **immutable** (static)
- **Containers** = **running** aspect of docker
  - **mutable** (dynamic)

(d) 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.

(d) 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 were `dockerd` runs) to avoid downloading them again

(d) 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:24.04 and ubuntu:noble
- Regardless of the tags, an image is always identified by a unique id (hex)
- Tags for a given image can be retrieved using the registry API

(d) Registries: internal image registry

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

```
docker images
```

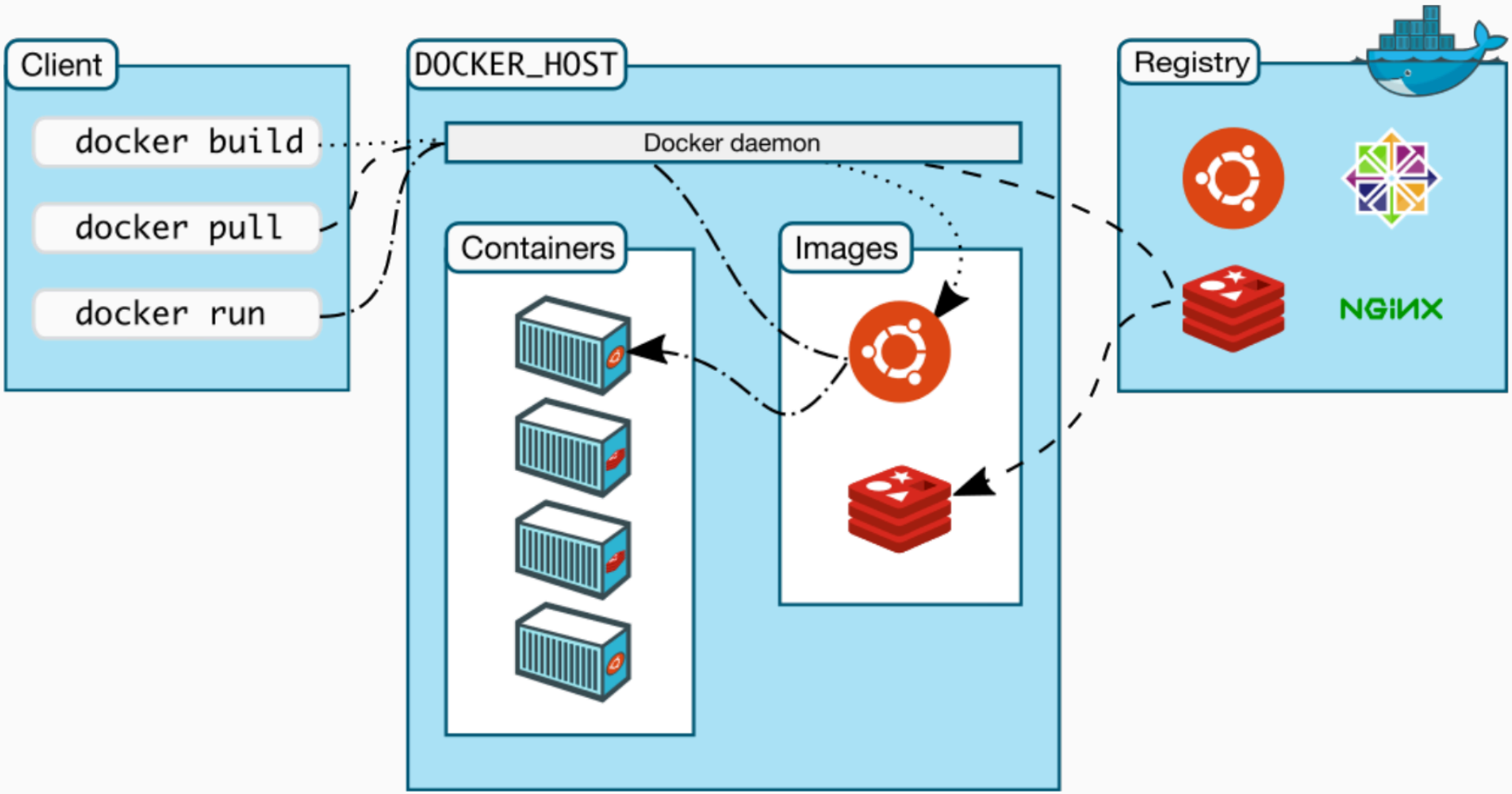
- Images are automatically downloaded by docker when needed
- Image can be manually downloaded via:

```
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)

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 $\longrightarrow$ local access
  - client $\leftrightarrow$ 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 through the use of contexts

Rootless docker daemon

- By default, `dockerd` daemon runs as root
  - consequently, root (UID 0) in the container is also root on the server!
  - **security risk!**
- Docker daemon can be configured to be rootless[1], i.e. it runs as a non-root user
  - strongly advisable, but requires specific configuration
  - unfortunately, Docker doesn't integrate well with `systemd`
- **Better to use Podman if rootless access is important**

[1] 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 as `root` or with `sudo`[1]!
  - instead, add[2] the user to the docker group:

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

[1] Rule of thumb: **NEVER** use `sudo` unless you REALLY need it!

[2] https://linuxize.com/post/how-to-add-user-to-group-in-linux/

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 terminated 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

Creating and running a first interactive container

Simplest example:

```
docker run -it debian:12
```

1. Retrieve the `debian:12` image
2. Create a container from this image
3. Bind the current terminal's `stdin`, `stdout`, `stderr` to the container (`-it`)
4. Execute the image entry point program present in the image (`bash`)
5. By using bash it's possible to browse and alter the container's filesystem!
6. When the entry point program (`bash`) exits, the container is terminated, **but it remains** on the server (in an `exited` state)
   - add `--rm` to automatically remove the container once terminated

Creating and running a non-interactive container

```
docker run --name bla --rm debian echo "Hello world"
```

1. Retrieve the `debian:latest` image
2. Create a container from this image
   - the container is named `blah` (option `--name`)
   - the container will automatically be deleted (option `--rm`)
3. Override the image entry point program with the `echo` command
4. `echo` prints the message `Hello world` to the client's terminal
5. When `echo` terminates, the container terminates as well

Listing containers and states

- `docker ps` lists all running containers
- `docker ps -a` lists all containers, including non-running ones
- When the entry point program of a container terminates, the container exits: its states switches from `running` to `exited`
- This is why when one executes `exit` or presses `ctrl+d` in a running container featuring a shell as its entry program, the container exits
- `docker history image` displays the `image`'s history, **including the entry point program**

Executing a program in a container

- Use `exec` to start another process within an **already running** container (e.g. bash to explore or alter the filesystem):

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

- Example:
  - run `nob` as a Ubuntu Noble container (`bash` is the entry point prog.):

```
docker run --name nob -it ubuntu:noble
```

  - In another terminal, execute the interactive program `top` in `nob`:

```
docker exec -it nob top
```

  - Now, there are 2 processes running in the container!

Attaching to a container

- The `attach` command attaches the terminal' `stdin`, `stdout` and `stderr` to a running container's **entry point command**
- Here, attach to the `nob` container:

```
docker attach nob
```

- **Important**: `attach` does not create a new process in the container!
- It's possible to be attached simulatenously multiple times to the same container

Detaching from a container

- It is possible to detach the standard input/output from a shell in an interactive container using the following key combination:

```
ctrl+p ctrl+q
```

Starting a container

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

```
docker start <container>
```

- The container's entry point program will be started again with the same parameters specified at creation time
- For interactive programs (e.g. bash), it's necessary to rebind the container's stdin to the current terminal with `-ia`:

```
docker start -ia
```

Stopping a container

- A container terminates (exits) as soon as its entry point process stops
- A container can also be terminated from the client's command line:

```
docker stop <container>
```

- It's also possible to send a signal to a container:

```
docker kill <container>
```

  - unless specified, the default signal is `SIGKILL`

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 terminal, but will be redirected to docker's internal logs

Reading a container's output

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

```
docker logs <options> <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

Inspecting a container

- To show detailed information about a container and its process:

```
docker inspect <container>
```

- Information is output in the JSON format
- Works for both running and terminated containers

Removing a container

- To remove a terminated container:

```
docker rm <container>
```

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

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

Running a container as a specified user/group

Running a container's program with specific user and group IDs can be achieved with the argument:

```
-u uid:gid
```

- For instance:

```
$ docker run -it --rm ubuntu
root@20c1e77f584d:/# touch /tmp/pipo && ls -l /tmp/pipo
-rw-r--r-- 1 root root 0 Mar 29 14:50 /tmp/pipo
```

vs:

```
$ docker run -u 39:60 -it --rm ubuntu
irc@9a7c3adebcdb:/$ touch /tmp/pipo && ls -l /tmp/pipo
-rw-r--r-- 1 irc games 0 Mar 29 14:55 /tmp/pipo
```

Shell: retrieving the current user/group

- Two ways to retrieve the current user:

```
$(id -u)
```

or:

```
$UID
```

- Similarily, two ways to retrieve the current group:

```
$(id -g)
```

or:

```
$GROUPS
```

Limiting a container

- `docker run` accepts arguments to impose various limits on a container
  - limits implemented using **cgroups** on the container's processes
- Non-exhaustive list (`docker run --help` for the full list):

Argument	**Description**
`--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, `dockerd` uses the 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 context

- A Docker context is a configuration mechanism allowing to manage multiple Docker environments
- Allows to easily switch between different Docker backends (local, remote with ssh, remote with https, cloud, etc.)
- Remote ssh access requires ssh public key authentication (key pair)
  - remote user must exist on the remote host and be in the `docker` group
- Use `docker context` to display commands related to Docker context

Creating and using a context

- Create a context with `docker context create`, e.g.:

```
docker context create --docker host=ssh://myuser@my.host.net myremote
```

- List available contexts with:

```
docker context ls
```

- Display the current context with:

```
docker context show
```

- Use `docker context use` to switch to a specified context, e.g.:

```
docker context use myremote
```

Docker installation

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

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

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

```
sudo systemctl start docker
```

- Enable the docker service at boot time:

```
sudo systemctl enable docker
```

Docker Desktop

- Docker Desktop is a Docker bundle/environment providing:
  - Docker Engine (dockerd + docker client)
  - Docker Compose
  - A GUI and system tray integration
  - Filesystem sharing ability with the host
  - Networking configuration
- **Based on a VM running Linux!**
- Originally created to give the ability to Windows and Mac users to use Docker

Resources

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