CHEAT SHEETS
Talan Labs

Docker Command

General Usage

Start a container in Background

```
$ docker run -d jenkins
```

Start an interactive container

```
$ docker run -it ubuntu bash
```

Start a container and remove once stopped

```
$ docker run --rm jenkins
```

Expose a port from the container on the host

```
$ docker run -p 8000:4000 -d jenkins
```

Start a named container

```
$ docker run --name myDb -d postgres
```

Stop a running container

```
$ docker stop myDb
```

Start a stoped container

```
$ docker start myDb
```

Debug

Run a shell command in a running container

```
$ docker exec -it myNamedContainer sh
```

Follow logs of a running container

```
$ docker logs -f myRunningContainer
```

Show open port of container

```
$ docker port myRunningContainer
```

Building Images

Build an image from a Dockerfile in same dir

```
$ docker build -t my-image .
```

Force rebuild an image

```
$ docker build -t my-image --no-cache .
```

Create an image from a container

```
$ docker commit sha123123 my-image
```

Remove an image

```
$ docker rmi my-image
```

Container Management

List running container

```
$ docker ps
```

List all container

```
$ docker ps -a
```

Inspect container Metadata

```
$ docker inspect sha1231234
```

List local images

```
$ docker images
```

Kill all container

```
$ docker kill $(docker ps -q)
```

Remove all stopped container

```
$ docker rm $(docker ps -q -a)
```

Removing all untagged image

```
$ docker rmi $(docker images \
  | grep "^<none>" | awk '{print $3}')
```

Volumes

Mounting a local Directory on a container

```
$ docker run -v myFolder:/data myContainer
```

Create a local volume

```
$ docker volume create --name myVolume
```

Mounting a volume on a container

```
$ docker run -v myVolume:/data myContainer
```

Destroy a volume

```
$ docker volume rm myVolume
```

List volumes

```
$ docker volume ls
```

Network

Create a local Network

```
$ docker network create myNetwork
```

Attach a container to a Network on startup

```
$ docker run --net myNetwork myContainer
```

Connect a running container to a network

```
$ docker network connect myNetwork myContainer
```

Disconnect a running container to a network

```
$ docker network disconnect myNetwork myContainer
```

docker-compose.yml

```
version: '3'

services: ①
  proxy: ②
    image: nginx:1.15.2 ③
    ports:  ④
      - "8080:8080"
    networks: ⑤
      - frontend
  web: ②
    env_file: env.env ⑥
    build: ⑦
      context: ./dir
      dockerfile: Dockerfile-alternate
      args:
        - MyARG=NicoAsArg
    ports: ④
      - "5000:5000"
    volumes: ⑧
      - .:/config
    depends_on: ⑨
      - postgresql
    working_dir: /app ⑩
    networks:  ⑤
      - database
      - frontend
  postgresql: ②
    image: postgresql ③
    networks:  ⑤
      - database

networks: ⑤
  database:
  frontend:
```

① ***services*** : docker compose run services,

② ***services names*** : each services is referenced in docker-compose using its service name and not the docker sha or docker name

③ ***images*** : instruct docker-compose that the service will use a *raw* image for the service execution

④ ***ports*** : maps container port to host port

⑤ ***networks*** : segragates services between network for discovery and security. In this example, proxy will never have access to the postgres database. But can refer to web as a known hostname, and web can access postgresql with postgresql hostname.

⑥ ***env_file*** : set list of environment variable available in the container from a file on the host - only available during execution, not build.

⑦ ***build*** : instruct docker-compose to build the container from a Dockerfile. Dockerfile filename and path can be overiden as described

⑧ ***volumes*** : volumes from host can also be mounted in the container
very usefull in developpement to have your apps changes available in the service without rebuilding the container

⑨ ***depends_on*** : wait for depended services to be started - doesn't mean it's ready, just that compose has started the depended service. watch the other side of the poster for more info on service dependencies

⑩ ***working_dir*** : same a WORKDIR in dockerfile, change to the specified directory

Dockerfile

```
LABEL maintainer="nicolas.savois@talan.com" ①

FROM debian:jessie ②

ENV nginxVer="XX.Y-Z" ③

RUN apt-get install open-ssl ④

RUN curl http://xx.org/.../nginx_${nginxVer}.deb -o nginx.deb -s && \ ⑤
    dpkg -i nginx.deb && \
    rm nginx.deb && \ ⑥
    ln -s /etc/nginx/sites-available/site /etc/nginx/sites-enabled/site

COPY nginx.conf /etc/nginx/nginx.conf ⑦

ADD myapp.conf /etc/nginx/sites-available/ ⑧

USER 1000:1000 ⑨

WORKDIR /path/to/workdir ⑩

ENTRYPOINT nginx start ⑪
```

① **LABEL** : Add a label to the metadata of the docker image

② **FROM** : The base image used to build the new image

③ **ENV** : Create and environment variable reusable later, check (5) for usage

④ **RUN** : Run a command to build the image like adding a package, touching file, etc…

⑤ **&& ** : Each line in the dockerfile create a new layer in the docker image. To avoid the layer multiplication we group commands with this shell feature

⑥ ***trick*** : Remove the downloaded file from the layer - no need to keep it once installed

⑦ **COPY** : Copy inside the image a file from the host (replace if it exists)

⑧ **ADD** : Copy inside the specified folder, - just use **COPY**, **ADD** comes with Magic around, and we all hate magic! (right?)

⑨ **USER** : Change user, goes back to the kernel and run the next commands as the user with UID:GID from the docker host (1000:1000 is the first user created on nearly all linux distribution)

⑩ **WORKDIR** : Change directory, (most expensive cd in the world)

⑪ **ENTRYPOINT** : Command run when the container start (PID=1)

Docker Compose Command

General Usage

build the container from docker-compose.yml

```
docker-compose build
```

specify non default compose file

```
docker-compose -f myConfig.yml run backup
```

specify a project name

```
docker-compose -p myproject run backup
```

used by compose to define container name with `docker ps`, defaults to the folder name

create an alias for docker-compose

```
alias dc='docker-compose'
```

will save you a lot of typing :)

Managing Composed Services

run the services in foreground

```
docker-compose up
```

run the services in background

```
docker-compose up -d
```

run only one service

```
docker-compose up web
```

stop & remove all services, volmes & network

```
docker-compose down
```

stop one service

```
docker-compose stop web
```

restart a stoped service

```
docker-compose start web
```

remove a container associated with service

```
docker-compose rm web
```

stop and remove everything

```
docker-compose rm -vfs web
```

Debuging Composed Services

Running Commands in started container

```
docker-compose exec web sh
```

Running commands in container

```
docker-compose exec web sh
```

follow logs of the containers

```
docker-compose logs -f --tail=10
```

tail only display 10 lines of history, useful when compose runs for a long time…

display running services

```
docker-compose ps
```

validate compose config and show compose file

```
docker-compose config
```

Best Practices

The Twelve Factors

α. ***Codebase***
One codebase tracked in revision control, many deploys

β. ***Dependencies***
Explicitly declare and isolate dependencies

γ. ***Config***
Store config in the environment

δ. ***Backing services***
Treat backing services as attached resources

ε. ***Build, release, run***
Strictly separate build and run stages

ζ. ***Processes***
Execute the app as one or more stateless processes

η. ***Port binding***
Export services via port binding

θ. ***Concurrency***
Scale out via the process model

ι. ***Disposability***
Maximize robustness with fast startup and graceful shutdown

κ. ***Dev/prod parity***
Keep development, staging, and production as similar as possible

λ. ***Logs***
Treat logs as event streams

μ. ***Admin processes***
Run admin/management tasks as one-off processes

Package a single application per container

Never ever use anything like a supervisor or a bash that spawn process inside a container. 1 container 1 process that's it. It would cripple the scalability offered by containers.

Properly handle PID 1, signal handling, and zombie processes

If your process dies you don't want it to leave child / zombie processes behind and plague the host resources

3 possibilities :

- run your binary from the CMD / ENTRYPOINT command in the Dockerfile - safest and easiest
- if you really need to launch a shell script that will itself launch your binary after setting things up, do it with the exec command. It will transfer the PID 1 to the binary and you will be safe
- use a custom init system inside the container like tini and use it as the entrypoint of your container. You have got to know what you are doing here.

Use vulnerability scanning in Container Registry

Security wise, it's a good practice to scan the available image in your image repository and verify that there's nothing too old.

for exemple:

- CoreOS Clair https://github.com/coreos/clair/
- Notary https://github.com/docker/notary

Docker-Compose Best Practices

Managing startup with compose

Compose will start all the service nearly at the same time, you can specify a depends_on attribute to "orchastrate the startup order" but it will not wait for the service to be up and running.

Suppose your API layer is a java spring app which takes a little while to boot, you don't want your frontend service to be up too soon without the API ready.

There's a few tricks to manage the startup order
They all rely on way to start the process in the container

wait-for-it

https://github.com/vishnubob/wait-for-it

Wrap your calls with the binary in your service command

```
version: "2"
services:
  web:
    build: .
    ports:
      - "80:8000"
    depends_on:
      - "db"
    command: ["./wait-for-it.sh", "db:5432", "--", "python", "app.py"]
  db:
    image: postgres
```

dockerize

https://github.com/jwilder/dockerize

Dockerize is a utility to simplify running applications in docker containers. It allows you to:

in Dockerfile :

```
CMD dockerize
     -template /etc/nginx/nginx.tmpl:/etc/nginx/nginx.conf
     -stdout /var/log/nginx/access.log -stderr /var/log/nginx/error.log
     -wait tcp://web:8000 nginx
```

in docker-compose :

```
web:
  command: [
    "dockerize",
     "-template /etc/nginx/nginx.tmpl:/etc/nginx/nginx.conf",
     "-stdout /var/log/nginx/access.log",
     "-stderr /var/log/nginx/error.log",
     "-wait tcp://web:8000 nginx"
   ]
```

Wait for other services to be available using TCP, HTTP(S)

```
$ dockerize -wait tcp://db:5432 -wait http://web:80
            -wait file:///tmp/generated-file
            -timeout 10s
```

Tail multiple log files to stdout and/or stderr

```
$ dockerize -stdout info.log -stdout perf.log -stderr error.log
```

Generate configuration files at container startup time

```
$ dockerize -template template1.tmpl:file1.cfg
            -template template2.tmpl:file3
```

from templates and container environment variables

Dockerfile Things

Carefully consider whether to use a public image

Only trust official image from docker hub. If you like an untrusted image, prefere copying the dockerfile and maintain it like that, it will protect from docker image change that would have suspicious activity.

Optimize for the Docker build cache

- use the same base image as much as possible
- keep the same step order accross your dockerfile
- also means your have to share the build context in your CI

Build the smallest image possible

- merge you command into a signe RUN command
- avoid switching context in the image build
    - don't change USER back and forth
    - don't set multiple ENV, use envfile
- use alpine or even distroless images
- to help you can use multi stage build when needed

Remove unnecessary tools

Don't give attacker tools and permission they could use like netcat, curl

- Compile apps in a single binary file, and add it to a scratch image.
- Run container in read-only mode with `--read-only`

Multistage Build

Multistage build is a way to decouple build container that produce a binary and the container that runs this binary.

Dockerfile

```
FROM golang:1.7.3 as builder ①
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go . ②
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app . ③

FROM alpine:latest ④
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /go/src/github.com/alexellis/href-counter/app . ⑤
CMD ["./app"] ⑥
```

① named target (as) to use in *futur* target
② copy your code in the container to build
③ build a single binary
④ 2nd target, it will be the base image for the produced image
⑤ copy the binary from the first image to the second image
⑥ run target is the copied binary

Launching the build

```
docker build -t nsa/gotest:latest .
```

Result 10MB image instead of 700MB well worth the hassle!

```
REPOSITORY          TAG                 IMAGE ID            SIZE
nsa/gotest          latest              bf19b53c1189        10.6MB
<none>              <none>              bf63bcebbf0d        695MB
```

still need to remove the *ghost* image