Docker CheatSheet (Based on tutorial on <https://docs.docker.com/get-started/>)

## List Docker CLI commands

docker

docker container --help

docker container logs --help

## Display Docker version and info

docker --version

docker version

docker info

## Create contained based on Docker image and execute

docker run hello-world

## List Docker images

docker image ls #or shortly:

docker images

## List Docker containers (running, all, all in quiet mode)

docker container ls

docker container ls --all

docker container ls -aq

- Dockerfile ex:

requirements.txt and app.py in the same folder with this Dockerfile

# Use an official Python runtime as a parent image

FROM python:2.7-slim

# Set the working directory to /app

WORKDIR /app

# Copy the current directory contents into the container at /app

ADD . /app

# Install any needed packages specified in requirements.txt

RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container

EXPOSE 80

# Define environment variable

ENV NAME World

# Run app.py when the container launches

CMD ["python", "app.py"]

* Build image, run container with image, publish image

docker build -t friendlyhello . # Create image using this directory's Dockerfile

docker run -p 4000:80 friendlyhello # Run "friendlyname" mapping port 4000 to 80

docker run -d --name demo -p 4000:80 friendlyhello # Same thing, but in detached mode

docker attach demo # Attach to above detached process

docker container ls # List all running containers

docker container ls -a # List all containers, even those not running

docker container stop <hash> # Gracefully stop the specified container

docker container kill <hash> # Force shutdown of the specified container

docker container rm <hash> # Remove specified container from this machine

docker container rm $(docker container ls -a -q) # Remove all containers

docker image ls -a # List all images on this machine

docker image rm <image id> # Remove specified image from this machine

docker image rm $(docker image ls -a -q) # Remove all images from this machine

docker login # Log in this CLI session using your Docker credentials

docker tag <image> username/repository:tag # Tag <image> for upload to registry

docker push username/repository:tag # Upload tagged image to registry

docker run username/repository:tag # Run image from a registry

Scaling Docker with Docker Compose on single machine

docker stack ls # List stacks or apps

docker stack deploy -c <composefile> <appname> # Run the specified Compose file

docker service ls # List running services associated with an app

docker service ps <service> # List tasks associated with an app

docker inspect <task or container> # Inspect task or container

docker container ls -q # List container IDs

docker stack rm <appname> # Tear down an application

docker swarm leave --force # Take down a single node swarm from the manager

**Scaling to Multi-Machine using Docker Swarm**

Multi-machine applications are made possible by joining multiple machines into a “Dockerized” cluster called a **swarm**.

docker-machine create --driver virtualbox myvm1 # Create a VM (Mac, Win7, Linux)

docker-machine create -d hyperv --hyperv-virtual-switch "myswitch" myvm1 # Win10

docker-machine env myvm1 # View basic information about your node

docker-machine ssh myvm1 "docker node ls" # List the nodes in your swarm

docker-machine ssh myvm1 "docker node inspect <node ID>" # Inspect a node

docker-machine ssh myvm1 "docker swarm join-token -q worker" # View join token

docker-machine ssh myvm1 # Open an SSH session with the VM; type "exit" to end

docker node ls # View nodes in swarm (while logged on to manager)

docker-machine ssh myvm2 "docker swarm leave" # Make the worker leave the swarm

docker-machine ssh myvm1 "docker swarm leave -f" # Make master leave, kill swarm

docker-machine ls # list VMs, asterisk shows which VM this shell is talking to

docker-machine start myvm1 # Start a VM that is currently not running

docker-machine env myvm1 # show environment variables and command for myvm1

eval $(docker-machine env myvm1) # Mac command to connect shell to myvm1

& "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env myvm1 | Invoke-Expression # Windows command to connect shell to myvm1

docker stack deploy -c <file> <app> # Deploy an app; command shell must be set to talk to manager (myvm1), uses local Compose file

docker-machine scp docker-compose.yml myvm1:~ # Copy file to node's home dir (only required if you use ssh to connect to manager and deploy the app)

docker-machine ssh myvm1 "docker stack deploy -c <file> <app>" # Deploy an app using ssh (you must have first copied the Compose file to myvm1)

eval $(docker-machine env -u) # Disconnect shell from VMs, use native docker

docker-machine stop $(docker-machine ls -q) # Stop all running VMs

docker-machine rm $(docker-machine ls -q) # Delete all VMs and their disk images

Commit new state of machine to an image

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

c3f279d17e0a ubuntu:12.04 /bin/bash 7 days ago Up 25 hours desperate\_dubinsky

197387f1b436 ubuntu:12.04 /bin/bash 7 days ago Up 25 hours focused\_hamilton

$ docker commit c3f279d17e0a svendowideit/testimage:version3

f5283438590d

$ docker images

REPOSITORY TAG ID CREATED SIZE

svendowideit/testimage version3 f5283438590d 16 seconds ago 335.7 MB

Change environment variable with commit.

The --change, -c option will apply Dockerfile instructions to the image that is created. Supported Dockerfile instructions: CMD|ENTRYPOINT|ENV|EXPOSE|LABEL|ONBUILD|USER|VOLUME|WORKDIR

$ docker inspect -f "{{ .Config.Env }}" c3f279d17e0a

[HOME=/ PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin]

$ docker commit --change "ENV DEBUG true" c3f279d17e0a svendowideit/testimage:version3

f5283438590d

$ docker inspect -f "{{ .Config.Env }}" f5283438590d

[HOME=/ PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin DEBUG=true]

**Shell access to container:**

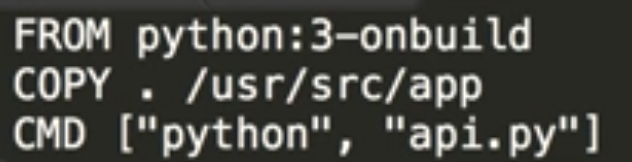
-t / --tty flag: open a pseudo terminal inside container,

-i / --interactive flag: direct STDIN (keyboard input) to machine

docker run -it myimage /bin/bash

**Special Images**

Onbuild tagged python images install the defined in requirements.txt in /usr/src/app

****

**Linking Machines Manually**

Start DB container with setting MYSQL\_ROOT\_PASSWORD environment variable:

docker run --name qas-sql -e MYSQL\_ROOT\_PASSWORD=password -d mysql:latest

Start wordpress container, linking it to above container for DB backend:

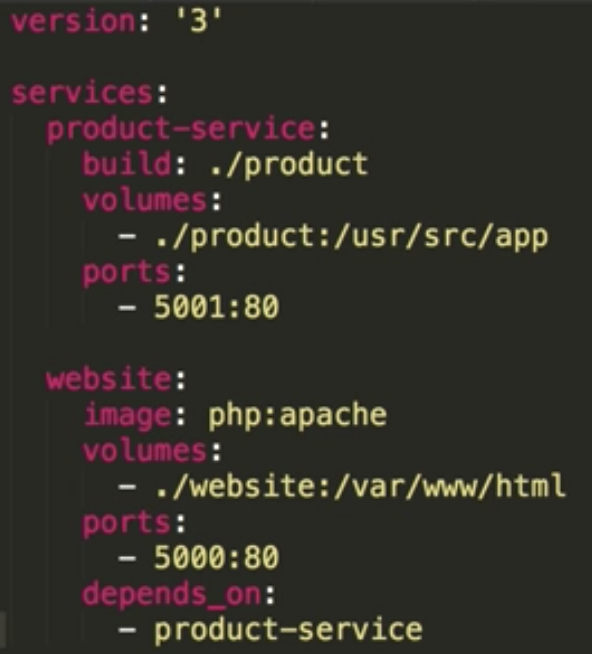
docker run --name qas-wordpress --link qas-sql:mysql -p 8080:80 -d wordpress

To get ip of machines to use in url instead of using localhost:

docker-machine ip

**docker-compose**

creates multiple machines on a network, and defines a network between them with docker-compose.yml file:

 (apache container default web location is /var/www/html)

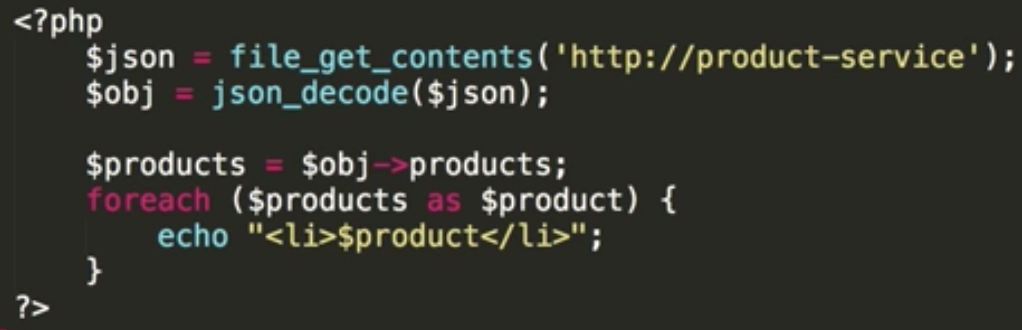
**To start all** machines, run:

docker-compose up

in docker network, machines will have their services names as domains, like:

<http://product-service:5001>

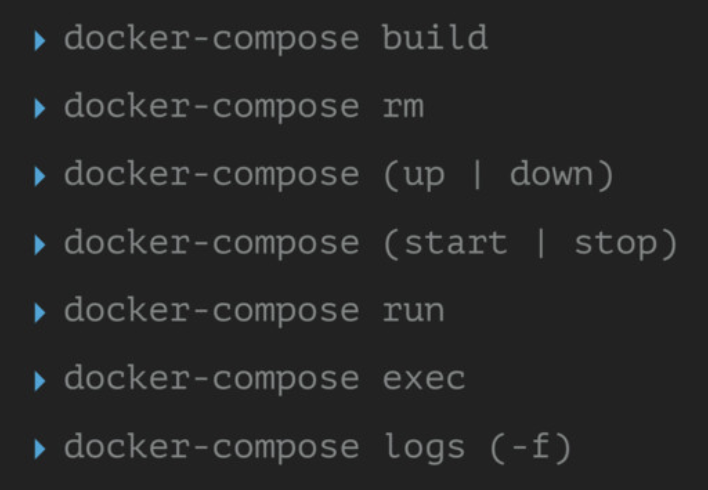
so website php code can have this call to service, with file\_get\_contents



In this case, the website is depending on product-service microservice.

**To stop all** machines, run:

docker-compose stop



In docker-compose.yml file, you can also define volumes to get data on host machine

version:”2”

services:

….

db:

image: mysql:5.7

ports:

-3306

environment:

MYSQL\_ALLOW\_EMPTY\_PASSWORD: ‘yes’

volumes:

mysql-data: /var/lib/mysql

volumes:

mysql-data:

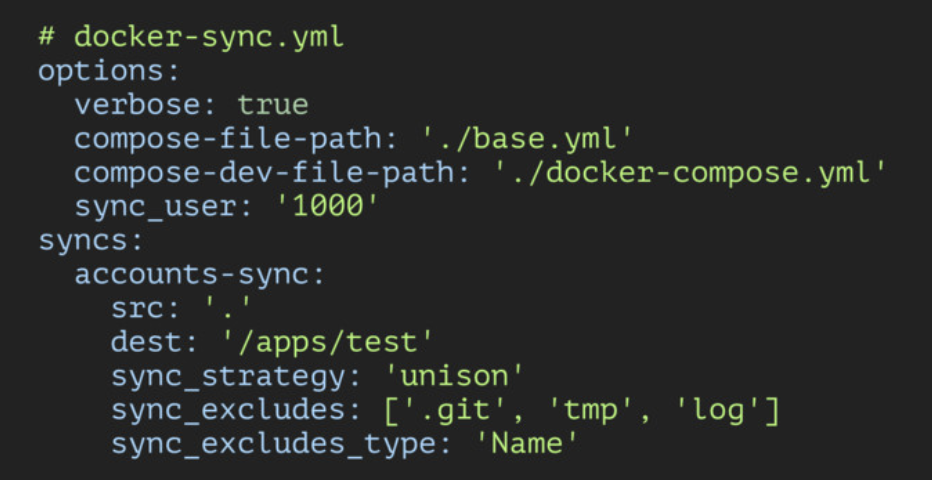
driver: local

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**docker-sync**

To sync code with containers, use docker-sync:

gem install docker-sync



To start sync services and docker-compose at the same time, run:

docker-sync-stack start

More: https://github.com/EugenMayer/docker-sync/wiki

Commandline to create volume

docker volume create –name <volume-name>

**Understanding existing images**

You may need to reverse engineeringwhat is on a server. Use

docker history <image\_id>

docker inspect <image\_id>

You can try

docker run -d <image\_id>

docker run -d example.com:5000/<image\_name>:<version>

# then to connect to running container using

docker exec -it <container name> /bin/bash

if above and directly trying terminal access:

docker run -it <image\_id> /bin/bash

gives errors, probably there is an entry point defined in image, you need to override it:

docker run -it --entrypoint /bin/bash <image\_id>