Docker Notes

> Originally, these notes are based on the excellent Udemy course "Docker Mastery: The Complete Toolset From a Docker Captain" by Bret Fisher, Course resources at: https://github.com/tkarakai-gto/udemy-docker-mastery

Table of Contents

Basicswww

> `http://docs.docker.com`

Docker version format changed early 2017. Versions are now YY.MM based (like Ubuntu) and you can choose *Stable* (slower) or *Edge* (faster) releases. This means that newer features drop faster in Edge releases, and fixes are backported for a longer timeline to Stable releases. Everyone wins.

- `docker <command> <subcommand> <options>`
- `docker` or `docker help` - Lists docker commands.
- `docker version`

Client - The command line client talking to the server using an API

Server - Docker Engine a.k.a. the Docker Server answering to the API requests and running Docker functions

- `docker info` - More detailed info, including stats, drivers, etc.

Images - File(s) containing the application we want to run.

Container - An instance of the image running as a process.

You can have many containers running of the same image.

- `docker container run --publish 80:80 nginx` Pulls latest image (is not in local image cache), starts a container based on the image, opens container port 80 forwarding to server port 80 and runs the default command.
- `docker container ls` - Lists running containers
- `docker container ls -a` - Lists all containers
- `docker image ls` - Lists all images
- `docker container run --publish 80:80 --name web-server --detach nginx` - Pulls image (is not in local cache), starts a container based on the image using the specified name, opens container port 80 forwarding to server port 80 and runs the default command, in the background (`--detach` or `-d`)
- `docker container logs nginx -f` - Tail of container stdout with "follow".
- `docker container stop nginx` - Stops running container
- `docker container start nginx` - Starts container
- `docker container rm -f nginx` - Removes container (with "force")

Docker Containers are *NOT* lightweight VMs

- They are just processes running on the host OS.
- Limited to what resources they can access.
- Exit when the process stops.

Compare PIDs reported by `docker top nginx` with PIDs reported by the regular Linux `ps aux | grep nginx` . They both show the same!

If you use Docker Toolbox:

- `docker-machine ssh` - Teleport into the console of the Docker Engine (vs. running commands on the Client)

`docker-machine` is a Docker host mamagement CLI. By default it talks to the VM that is set up with Docker Toolbox (if on an older Windows or Mac), but it can create new Docker hosts, including on cloud platforms. It can also provision `Swarm` clusters.

> https://docs.docker.com/machine/overview/

> https://github.com/mikegcoleman/docker101/blob/master/Docker_eBook_Jan_2017.pdf

Getting inside of an existing/running container

For stopped containers:

- `docker container start -ai ubuntu` - Starts a container with an interactive session (Ubuntu and alike)

For running containers:

- `docker container exec -it redis bash` - Starts interactive session (new process) of a running container (Ubuntu and other major distros)
- `docker container exec -it redis sh` - Starts interactive session (new process) of a running container (Alpine Linux)

> https://www.digitalocean.com/community/tutorials/package-management-basics-apt-yum-dnf-pkg

Basic Monitoring

- `docker container top nginx` - Lists processes of the given running container.
- `docker container stats` - Similar to Linux `top`, for all containers.
- `docker container stats nginx` - Similar to Linux `top`, for the specified container.
- `docker container inspect nginx` - Container metadata on container, including attached networks, MAC address, etc.

Quick temporary containers

`--rm` flag is so we don't need to clean up

- `docker container run --rm -d --name redis -p 6379:6379 redis:alpine` - Quick temporary Redis:alpine instance
- `docker container run --rm -d --name mysql --publish 3306:3306 -e MYSQL_RANDOM_ROOT_PASSWORD=true mysql` - Quick temporary MySQL. Password is in the logs (`docker container logs mysql`).
- `docker container run --rm -it ubuntu` - Runs container in interactive mode and when done, it deleted the container.

Networking Basics

By default containers use the "bridge" virtual network.

Containers on the same virtual network are free to talk to each other.

Best practice is to create a new virtual network for each app if they are not related

A container can be attached to multiple virtual networks (much like PCs can have multiple NICs). Or none.

A container can skip NAT and use the host networking `--net=host` (not recommended).

A container can be on no network with the `--net=none` setting.

Then there are "network drivers" too...

- `docker container port nginx` - Shows port configuration of the container
- `docker container inspect --format '{{ .NetworkSettings.IPAddress }}' nginx` - Extracts the IP address of the container on the virtual network

> https://docs.docker.com/engine/admin/formatting/

- `docker network --help`
- `docker network ls` - Lists virtual networks
- `docker network create my_network [--driver ...]` - Create new virtual network (optionally with a driver)

> https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/

- `docker network inspect my_network` - JSON metadata about the virtual network, for example Subnet, default gateway, etc. Lists containers on it and their MAC addresses too.
- `docker container run --rm -d --network my_network nginx` - Run a container on a specific network (`--net` is a shorthand for `--network`)
- `docker network connect --help`
- `docker network connect my_network redis` - Connects container to the network (insert virtual NIC plugged into the given network)
- `docker network disconnect my_network redis` -Disconnects container from the network (remove virtual NIC plugged into the given network)
- `docker network rm my_network` - Delete virtual network

Example `docker-compose.yml` *demostrating network separation*

```
services:
    proxy:
      image: nginx-custom
      networks:
        - front
    app:
      image: myApp
      networks:
        - front
        - back
    db:
      image: postgres
      networks:
        - back

networks:
  front:
    driver: custom-driver-1
  back:
    driver: custom-driver-2
```

Name Resolution

DNS is needed to reliably identify containers that are coming and going.

Container names are used as DNS names for host resolution on the same virtual network.

- `docker container exec -it nginx ping nginx2`

The default "bridge" network does NOT have the DNS server built into it. For that you will need use the `--link` (add a link to another container). Easier to create a new network :).

Using `docker compose` makes this much easier by automatically creating virtual networks.

To create simple *round robin DNS name resolution* (a poor man's load balancer) use the `--net-alias` option. Container names must be unique, but aliases do not. Aliases will respond to DNS queries in a round-robin fashion.

- `docker container run --rm -d --net my_network --net-alias search elasticsearch:2` - (running the first container)
- `docker container run --rm -d --net my_network --net-alias search elasticsearch:2` - (running a second container, exact same command!)
- `docker container ls` - Notice the ports they use (9200, 9300), we are NOT exposing the ports on the host!
- `docker container run --rm --net my_network alpine nslookup search` - Should list both container IP addresses.
- `docker container run --rm --net my_network centos curl -s search:9200` - Calling elasticsearch by the alias. Repeatedly calling should alternate between the two containers. Check the server's `name` attribute to verify. *THIS DID NOT WORK FOR ME. IT ALWAYS CALLED THE SAME CONTAINER. ONLY SWITCHED TO THE OTHER ONE WHEN THE FIRST CONTAINER WAS STOPPED.*

Images defined

Image is the application binaries and dependencies plus the metadata on how to run it. *Officially:* "An image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime".

> https://github.com/moby/moby/blob/master/image/spec/v1.md

There are no OS inside an image. No kernel, no kernel modules (e.g. drivers).

Images use the union filesystem which are layers on top of layers of filesystem changes. Each executed OS command adds a new layer.

- `docker history nginx` - Shows layers of changes (commands)

When starting a container Docker opens a read-write layer on top of the image. When a file is changed, that file is copied into the new layer. This is called "copy-on-write" or COW.

- `docker inspect nginx` - This is the metadata about the image.

Tags

- `docker image tag --help`
- `docker image ls` - Look at the repository and tag columns

Tag is just a label for a specific image on docker hub. There can be multiple tag for the same image.

- `docker image tag nginx tkarakai/nginx:testing` - Tagging an existing image with another tag "testing". If no tag specified after the repository name, it defaults it to "latest".

"latest" does not necessarily mean "latest", it's just another label.

Docker Hub

> https://github.com/docker-library/official-images/tree/master/library

- `docker login tkarakai` - Log into docker hub by default. Stores auth key in `.docker/config.json` in Windows.
- `docker image push tkarakai/nginx:testing` - Push to docker hub.
- `docker image rm tkarakai/nginx:testing` - Remove tag (it figured based on the name that it's a tag, not an actual image) from the local repository.
- `docker logout`

Docker Hub supports private repositories. For that create the repo in docker hub first on the web interface with visibility "private", then push images.

Building Images

There are basic ways to build an image:

1. With `docker commit`
2. With a `Dockerfile` and `docker image build`

Building with `docker commit`

With `docker commit`, you first start a container, log in and make changes inside the container, stop the container and finally commit the changes as a new layer into a new image.

- `docker commit <image> <repo-name>:<tag-name>`

Building with Dockerfile, Example 1

The `Dockerfile` (the file named as such, with no extension) is a recipe for creating a Docker image. It contains "instructions". The instructions names a uppercased by convention (not required).

> https://docs.docker.com/engine/reference/builder/

The build starts on the Docker Client, so the build context (all the files needed, including the `Dockerfile` itself and maybe some files to be copied into the image) are first packed into a tarball and then sent to the Docker Engine.

Once the Docker Engine receives the build context, it starts a new container, executes the instructions, commits the new layer for each one, then at the end it stops and deletes the container.

Let's start with some basic `Dockerfile` instructions:

- `FROM` - Base image. must be the first instruction in `Dockerfile`
- `ENV` - Set environment variables. This is the way set keys and values to inject.
- `RUN` - Execute command(s)
  - Use `&&` to concatenate commands to preserve FS layers.
  - Can run shell scripts, copy files, etc.
  - Do *NOT* use commands that result in nondeterministic results, Docker will not know that it should build a new image layer. For example do not use `RUN apt-get update` by itself, append the specific apps to install/update to make it more specific, or use the `--no-cahce=true` flag in the `docker build` command.

The proper way to log is *NOT* to log to a log file, and there is no syslogd aor other syslog server either, Docker actually handles all the logging for us. All we have to do is all our logs are spit out into STDIN and STDERR. See second `RUN` command below.

- `EXPOSE` - Expose ports within the virtual network (not yet to the outside!)
- `CMD` - The default command to run every time the container is started or restarted (at runtime, not build time). Can be owerwritten with the command specified on the `docker container run` command.

Dockerfile

```
# NOTE: this example is taken from the default Dockerfile for the official nginx Docker Hub Repo
# https://hub.docker.com/_/nginx/
FROM debian:stretch-slim
# all images must have a FROM
# usually from a minimal Linux distribution like debain or (even better) alpine
# if you truly want to start with an empty container, use FROM scratch

ENV NGINX_VERSION 1.13.0-1~stretch
ENV NJS_VERSION   1.13.0.0.1.10-1~stretch
# optional environment variable that's used in later lines and set as envvar when container is running

RUN apt-get update \
        && apt-get install --no-install-recommends --no-install-suggests -y gnupg1 \
        && \
        NGINX_GPGKEY=573BFD6B3D8FBC641079A6ABABF5BD827BD9BF62; \
        found=''; \
        for server in \
                ha.pool.sks-keyservers.net \
                hkp://keyserver.ubuntu.com:80 \
                hkp://p80.pool.sks-keyservers.net:80 \
                pgp.mit.edu \
        ; do \
                echo "Fetching GPG key $NGINX_GPGKEY from $server"; \
                apt-key adv --keyserver "$server" --keyserver-options timeout=10 --recv-keys "$NGINX_GPGKEY" && found=yes && break; \
        done; \
        test -z "$found" && echo >&2 "error: failed to fetch GPG key $NGINX_GPGKEY" && exit 1; \
        apt-get remove --purge -y gnupg1 && apt-get -y --purge autoremove && rm -rf /var/lib/apt/lists/* \
        && echo "deb http://nginx.org/packages/mainline/debian/ stretch nginx" >> /etc/apt/sources.list \
        && apt-get update \
        && apt-get install --no-install-recommends --no-install-suggests -y \
                                            nginx=${NGINX_VERSION} \
                                            nginx-module-xslt=${NGINX_VERSION} \
                                            nginx-module-geoip=${NGINX_VERSION} \
                                            nginx-module-image-filter=${NGINX_VERSION} \
                                            nginx-module-njs=${NJS_VERSION} \
                                            gettext-base \
        && rm -rf /var/lib/apt/lists/*
# optional commands to run at shell inside container at build time
```

```
# this one adds package repo for nginx from nginx.org and installs it

RUN ln -sf /dev/stdout /var/log/nginx/access.log \
       && ln -sf /dev/stderr /var/log/nginx/error.log
# forward request and error logs to docker log collector

EXPOSE 80 443 8080
# expose these ports on the docker virtual network
# you still need to use -p or -P to open/forward these ports on host

CMD ["nginx", "-g", "daemon off;"]
# required: run this command when container is launched
# only one CMD allowed, so if there are mulitple, last one wins
```

`docker image build`

- `docker image build -t custom-nginx .` - Builds image (as "latest") in the current directory
- `docker image ls` - The new image should show up here.

Exercise:

1. Add a new exposed port to Dockerfile
2. Rebuild
3. Observe that it only rebuilds the last step (layer)
4. We have a new image id (overwriting the old one)
5. If you undo the `Dockerfile` changes and rebuild again, *ALL* steps are use from the cache!

The order of the instructions in `Dockerfile` is significant: `docker image build` will rebuilds every step after a change is detected in `Dockerfile`. Put most stable commands on the top, and most frequently changing commands towards the end.

Dockerfile Example 2

- `WORKDIR` - Staring root dir. Preferred to using `RUN cd /path`
- `COPY` - Copies file from the builder current directory (client side) to the image current dir.
- `ADD` - Similar to `COPY` but can add files from a URL, it can decompress source files. Generally less transparent than `COPY`, so it is less preferred.
- Use the `.dockerignore` file to exclude files to be copied with `COPY` and `ADD`
- `USER` - Sets the user name or UID to use when running the image and for any RUN, CMD and ENTRYPOINT instructions that follow it in the Dockerfile. *Highly recommended to set to a nonpriviledged user.* For example use something like `RUN useradd -ms /bin/bash mynewuser` before the `USER` instruction.

Dockerfile

```
# this same shows how we can extend/change an existing official image from Docker Hub

FROM nginx:latest
# highly recommend you always pin versions for anything beyond dev/learn

WORKDIR /usr/share/nginx/html
# change working directory to root of nginx webhost
# using WORKDIR is preferred to using 'RUN cd /some/path'

COPY index.html index.html

# I don't have to specify EXPOSE or CMD because they're in my FROM
```

Inherits all instructions from the `nginx` `Dockerfile`.

- `docker image build -t nginx-with-html .`
- `docker container run --rm -d -p 80:80 nginx-with-html`
- Open `http://locahost` (or http://192.168.99.100 if using Docker Toolbox) to see the custom html.
- `docker image tag nginx-with-html tkarakai/nginx-with-html:latest` - Tag for fun, ready to be pushed to docker hub.

Dockerfile Example 3

> https://github.com/tkarakai-gto/udemy-docker-mastery/blob/master/dockerfile-assignment-1

```
FROM node:6-alpine

EXPOSE 3000

RUN apk add --update tini \
  && mkdir -p /usr/src/app

WORKDIR /usr/src/app

COPY package.json package.json

RUN npm install \
  && npm cache clean

COPY . .

CMD ["tini", "--", "node", "./bin/www"]
```

Container Lifetime & Persistent Data

> - The 12-Factor App (Everyone Should Read: Key to Cloud Native App Design, Deployment, and Operation
> - 12 Fractured Apps (A follow-up to 12-Factor, a great article on how to do 12F correctly in containers)
> - Intro to Immutable Infrastructure Concepts

Containers are meant to be immutable and ephemeral (disposable) as a design goal. We do not change things once they are running, instead we re-deploy a whole new container.

The container should not contain any application data mixed in with the application binaries.

Persistent Data Volumes

"Volume" is a container independent filesystem. When containers are deleted, volumes are left alone. Volumes need to be deleted independently.

In the `Dockerfile`:

- `VOLUME <path>` - Creates a new volume and assigns it to the path inside the container.
- `docker image inspect mysql` - Look for "Volumes". MySQL is an container using volumes, obviously :) .
- `docker container run --rm -d --name mysql -e MYSQL_RANDOM_ROOT_PASSWORD=true mysql` - Quick temporary MySQL.
- `docker container inspect mysql` - Look for "Volumes" and "Mounts" from where the the Docker host serves the volume to the container.
- `docker volume ls` - List with the unique IDs, not really useful
- `docker volume inspect 23b472102724d408a8374babf096d251a32fab5446db942d0f8dc0757e0c3b22` - JSON metadata of the volume (this is on the Docker Host!)

Named Volumes

- `docker container run --rm -d --name mysql -e MYSQL_RANDOM_ROOT_PASSWORD=true -v mysl-vol:/var/lib/mysql mysql` - Using "mysql-vol" as the name for the `/var/lib/mysql` container dir. The container volume location can be found out from the Dockerfile (`VOLUME` line).
- `docker volume create--help` - This is required to be able to specify custom drivers and labels.

Bind Mounts

It is a mapping of the host file or directory to a container file or directory.

Skips the UFS, host files do overwrite files in container. Files are not deleted when the container is deleted.

Bind Mounts are host specific, they need specific data to be on the host to work, they cannot be used in `Dockerfile`, must be at `container run`.

Looks just like the volume option, but with an extra host full path specified and separated with a colon. Bind Mount starts with a `/`, Volume does not.

- `... run -v /Users/me/stuff:/path/in/contaier` (Mac/Linux)
- `... run -v //c/Users/me/stuff:/path/in/contaier` (Windows)

Useful for development when editing files on the host is more convenient that should be automatically picked up by the container.

- `docker container run --rm -d --name nginx -p 80:80 -v $(pwd):/usr/share/nginx/html nginx` - Use `$(pwd)` to substitute the current host dir.
- `docker container exec -it nginx bash` - Jump into the container
- `cd /usr/share/nginx/html` - Should see the files from the host

Database minor upgrade (exercise)

Instead of running the OS package management `update` of the db version, use named volumes. The new version should be the new version of the image, pointing to the original volume.

- Find volume path in the Dockerfile of postgres: https://hub.docker.com/r/library/postgres/tags/9.6.1/. It is `/var/lib/postgresql/data`
- `docker container run --rm -d --name psql -v psql:/var/lib/postgresql/data postgres:9.6.1`
- `docker container logs psql` - It should have the regular startup logs with the initial entries
- `docker container stop psql` - Stop old db version
- `docker container run --rm -d --name psql2 -v psql:/var/lib/postgresql/data postgres:9.6.2` - start new db version pointing to the existing volume
- `docker container logs psql2` - It should have a very short log, skipping the initialization

Jekyll exercise

`jekyll` is a static web site generator. Jekyll, a Static Site Generator (just as background info, no need to install) It is the tool generating github pages.

The point is developers only need to run the container (no need to install Ruby and dependencies), the developer can simply use Bind Mounts to point to his workstation's directory and edit the content files there

> https://github.com/tkarakai-gto/udemy-docker-mastery/tree/master/bindmount-sample-1

- `docker container run --rm -p 80:4000 -v $(pwd):/site bretfisher/jekyll-serve` - run from the sample directory, starts the site generator
- Go to http://192.168.99.100/
- Update the post markdown file in the `_posts` directory
- Watch the server noticing the change and regenerating the web site
- Refresh the browser

Changing running container resources

- `docker container update --help` - Change container resources, like memory, CPU, and others

Docker Compose

A way to configure relationships between containers, save `docker container run` settings in easy-to-read file, start/stop the entire environment with a single command.

It has two parts:

1. `docker-compose.yml` : a `YAML` formatted file describing:
   - containers
   - networks
   - volumes
2. `docker-compose` :
   - CLI tool for dev/test automation (uses the YAML file)

> - The YAML Format: Sample Generic YAML File
> - The YAML Format: Quick Reference
> - Compose File Version Differences (Docker Docs)
> - Docker Compose Release Downloads (good for Linux users that need to download manually)

docker-compose.yml

Versions are different, be careful.

This is just the default name, you can use another YAML file name with the `-f` option

- `docker-compose --help`

Template YAML content

```
version: '3.1'  # if no version is specificed then v1 is assumed. Recommend v2 minimum

services:  # containers. same as docker run
  servicename: # a friendly name. this is also DNS name inside network
    image: # Optional if you use build:
    command: # Optional, replace the default CMD specified by the image
    environment: # Optional, same as -e in docker run
    volumes: # Optional, same as -v in docker run
  servicename2:

volumes: # Optional, same as docker volume create

networks: # Optional, same as docker network create
```

Same as the above jekyll run command

```
version: '2'

# same as
# docker run -p 80:4000 -v $(pwd):/site bretfisher/jekyll-serve

services:
  jekyll:
    image: bretfisher/jekyll-serve
    volumes:
      - .:/site
    ports:
      - '80:4000'
```

Wordpress example

```
version: '2'

services:

  wordpress:
    image: wordpress
    ports:
      - 8080:80
    environment:
      WORDPRESS_DB_PASSWORD: example
      key: value
    volumes:
      - ./wordpress-data:/var/www/html

  mysql:
    image: mariadb
    environment:
      MYSQL_ROOT_PASSWORD: example
    volumes:
      - ./mysql-data:/var/lib/mysql
```

Example of a 3 node load balanced database cluster behind a Ghost web server

```
version: '3'

services:
  ghost:
    image: ghost
    ports:
      - "80:2368"
    environment:
      - URL=http://localhost
      - NODE_ENV=production
      - MYSQL_HOST=mysql-primary
      - MYSQL_PASSWORD=mypass
      - MYSQL_DATABASE=ghost
    volumes:
      - ./config.js:/var/lib/ghost/config.js
    depends_on:
      - mysql-primary
      - mysql-secondary
  proxysql:
    image: percona/proxysql
    environment:
      - CLUSTER_NAME=mycluster
      - CLUSTER_JOIN=mysql-primary,mysql-secondary
      - MYSQL_ROOT_PASSWORD=mypass

      - MYSQL_PROXY_USER=proxyuser
      - MYSQL_PROXY_PASSWORD=s3cret
  mysql-primary:
    image: percona/percona-xtradb-cluster:5.7
    environment:
      - CLUSTER_NAME=mycluster
      - MYSQL_ROOT_PASSWORD=mypass
      - MYSQL_DATABASE=ghost
      - MYSQL_PROXY_USER=proxyuser
      - MYSQL_PROXY_PASSWORD=s3cret
  mysql-secondary:
    image: percona/percona-xtradb-cluster:5.7
    environment:
      - CLUSTER_NAME=mycluster
      - MYSQL_ROOT_PASSWORD=mypass

      - CLUSTER_JOIN=mysql-primary
      - MYSQL_PROXY_USER=proxyuser
      - MYSQL_PROXY_PASSWORD=s3cret
    depends_on:
      - mysql-primary
```

docker-compose CLI

Program using the Docker Server API on behalf of the CLI.

- `docker-compose up` - Set up volumes, networks and start all containers
- `docker-compose down` - Stop all containers and remove content, volumes, networks
- `docker-compose down -v` - Stop all containers and remove content, volumes, networks and delete volumes

With `docker-compose` new developer on-boarding could be as simple as:

1. Install Docker
2. `git clone server/repo/project`
3. `cd repo/project`
4. `docker-compose up`

Reverse proxy example

```
version: '2'
```

```
services:
  proxy:
    image: nginx:1.11 # this will use the latest version of 1.11.x
    ports:
      - '80:80' # expose 80 on host and sent to 80 in container
    volumes:
      - ./nginx.conf:/etc/nginx/conf.d/default.conf:ro

  web:
    image: httpd  # this will use httpd:latest
```

- Notice `:ro` postfix of volume means "read-only"
- Notice that there is no need to specify networking, it will create a new one automatically if not specified
- Notice that there is no `volumes` section. Not needed unless you need named volumes and other specifics.
- `docker-compose up` will start logging ALL container logs in the same console, color coded!
- Going to `http://localhost` (or the Docker host IP) should respond with "It works!", which is served up by httpd, proxied by nginx!

Drupal with Postgres example

```
version: '2'

services:

  drupal:
    image: drupal
    ports:
      - 8080:80
    volumes:
      - drupal-modules:/var/www/html/modules
      - drupal-profiles:/var/www/html/profiles
      - drupal-themes:/var/www/html/themes
      - drupal-sites:/var/www/html/sites
  db:
    image: postgres
    environment:
      - POSTGRES_USER=drupal
      - POSTGRES_PASSWORD=dbpass

volumes:
  drupal-modules:
  drupal-profiles:
  drupal-themes:
  drupal-sites:
```

- `docker-compose up -d` - detached
- `docker-compose logs -f` - to see the logs
- `docker-compose ps` - lists containers running defined in the compose file
- `docker-compose top` - processes

Building images within compose files

We can build a custom images on demand as part of the compose file. It only build once, then uses them from the cache.

Previous reverse proxy example, with custom proxy image

```
version: '2'

services:
  proxy:
    build:                      # If the image does not exist in cache, build it
      context: .                # In the current directory
      dockerfile: nginx.Dockerfile # using custom Dockerfile
    ports:
      - '80:80'
  web:
    image: httpd
    volumes:
```

```
      - ./html:/usr/local/apache2/htdocs/
```

- `docker-compose down -rmi local` - Removes images too that might have built.

Building Drupal with custom theme retrieved from github

Dockerfile

```
FROM drupal:8.2

RUN apt-get update \
  && apt-get install -y git \
  && rm -rf /var/lib/apt/lists/*

WORKDIR /var/www/html/themes

RUN git clone --branch 8.x-3.x --single-branch --depth 1 https://git.drupal.org/project/bootstrap.git \
  && chown -R www-data:www-data bootstrap

WORKDIR /var/www/html
```

docker-compose.yml

```
version: '2'

services:
  drupal:
    build: .              # Build using the default Dockerfile in the current directory
    image: drupal-custom  # If both 'image' and 'build' are specified, Compose will name the built image as per 'image' line
    ports:
      - 8080:80
    volumes:
      - drupal-modules:/var/www/html/modules
      - drupal-profiles:/var/www/html/profiles
      - drupal-themes:/var/www/html/themes
      - drupal-sites:/var/www/html/sites
  db:
    image: postgres:9.6
    environment:
      - POSTGRES_USER=drupal
      - POSTGRES_PASSWORD=dbpass
    volumes:
      - drupal-data:/var/lib/postgresql/data

volumes:
  drupal-data:
  drupal-modules:
  dupal-profiles:
  drupal-themes:
  drupal-sites:
```

- The `extends` keyword in `docker-compose.yml` enables sharing of common configurations among different files, or even different projects entirely. Extending services is useful if you have several services that reuse a common set of configuration options.

```
verion: '2.1'

web:
  extends:
    file: common-services.yml
    service: webapp
    environment:
      - DEBUG=1
    cpu_shares: 5
```

> *WARNING* `extends` currently only works up to v2.1, NOT YET in v3.x (as of June 2017)

Scaling containers with `docker-compose`

You can start `docker-compose` with multiple instances of the services defined in the `docker-compose.yml` file.

- `docker-compose up -d --scale redis-sentinel=3` - Starts the services with 3 instances of the `sentinel` service.
- `docker-compose scale sentinel=3` - scales one of the services on an already running setup.

Swarm Mode

Multi-node container life cycle management and orchestration solution, new feature as of summer of 2016.

- First there was "Swarm classic" (before v1.12) just a container repeating commands multiple times.
- v1.12 Swarm Kit was introduced (Summer 2016).
- v1.13 Stacks and Secrets were added (Jan 2017).

Swarm is *NOT* enabled by default, to make sure that existing orchestration solutions on top of Docker keep working.

- `docker info` - Look for "Swarm: inactive"

docker swarm init

- `docker swarm init` - Enable Swarm on the node you are on.

Under the hood init does these:

- PKI and security automation
  - Root Signing Certificate created for the Swarm (on the single node we are on) that will be used to establish trust between all Managers and Workers
  - Certificate is issued to first Manager node (on this node)
  - Join tokens are created (to be used on other nodes to join the swarm)
- Enables the swarm API
- Creates the "Raft consensus" database (with the RAFT protocol) to store the Root CA, configs and Secrets. Needed to secure node communication in the cloud.
  - Encrypted by default (v1.13+)
  - No need for another key/value system to hold orchestration and secrets, all built in
  - Replicates logs among Managers via mutual TLS in "control plane"

When Swarm is enabled, it adds these commands:

- `docker swarm`
- `docker node`
- `docker service`
- `docker stack`
- `docker secret`

Swarm Roles and Definitions

Manager - Nodes managing the swarm. There is one *leader* at any given time. If current leader goes down, new manager is elected. Managers all have the swarm configuration in their local database (in memory cache) replicated. Managers communicate among themselves via secured RAFT protocol. There can be maximum 7 managers. Managers send "tasks" to "worker" nodes. Managers decide about which task should run on which worker node. The same node can also act as a worker.

Worker - Node connected to a Manager accepting work to be executed and reporting back to the manager with the execution status.

Service - In a swarm "service" replaced the `docker run` command. It can run multiple replicas of the same container.

Task - A "job" for the worker node to start a container with a certain configuration.

> - Docker 1.12 Swarm Mode Deep Dive Part 1: Topology (YouTube)
> - Docker 1.12 Swarm Mode Deep Dive Part 2: Orchestration (YouTube)
> - Heart of the SwarmKit: Topology Management (slides)
> - Heart of the SwarmKit: Store, Topology & Object Model (YouTube)
> - Raft Consensus Visualization (Our Swarm DB and how it stays in sync across nodes)

Create your first service and scale it out

- `docker node ls` - Should bring up the only node in our new swarm, with a manager that is also the leader.
- `docker node --help` - commands to bring nodes in and out of the swarm and promoting to managers or demoting to workers

> Deploy services to a swarm (Docker Docs)

- `docker service --help`
- `docker service create alpine ping 8.8.8.8` - Just to create a node and give it some work while we investigate what's going on. Returns the service id (not the container id!)
- `docker service ls` - We now have one service with 1 replica running
- `docker service ps my-service-name` - Lists the container along with the NODE info. The name of the container now has a ".1" postfix
- `docker service update my-service-name --replicas 3` - We are scaling up the number of replicas to be 3.
- `docker service ls` - Should show 3 replicas running (if we are fast enough, we would see the number going up approaching 3)
- `docker service ps my-service-name` - Should show 3 containers (here on the same node)
- `docker service update --help` - A lot more options than the `docker container update` command, because it manages the entire cluster of service nodes with the goal of the service always available in the process, like rolling updates (the blue/green process)

What happens when a container crashes?

The container orchestration system will spawn a new one! Let's try it:

- `docker container ls` - Find a container id to kill
- `docker container rm -f my-container-id` - Forceibly kill a container in the swarm
- `docker service ls` -- You should see "REPLICAS" of 2/3 for a while. Then in a matter of seconds a new container will be launched to replace the failed one.
- `docker service ps my-service-name` - Should show the failed container as well as the new one.
- `docker service rm my-service-name` - Removes all containers of the service (eventually)
- `docker container ls` - To verify that the containers are removed (give it a couple of seconds)

Building a multi-node swarm

Options to experiment:

1. http://play-with-docker.com - Free, browser based, resets after 4 hours, uses docker-in-docker (!), very cool!
2. `docker-machine` + VirtualBox - Node provisioning tool, need at least 8-16GB RAM on the host. Need version 0.10+, comes with Docker Toolbox
   - `docker-machine create node1` - Created VirtualBox VM (by default) with BusyBox
   - `docker-machine create node2` - Node 2
   - `docker-machine create node3` - Node 3
   - `docker-machine ssh node1` - SSH into the node
   - `docker-machine env node1` - Lists environment variables to set if you want the `docker` CLI to send command to that node
   - `docker info` - To verify which node `docker` is talking to
3. http://digitalocean.com + Docker install - Cloud, fast (SSD), $5-$10/node/month, but there is a coupon!
4. Roll your own anywhere you can install Docker on, Amazon, Azure, DO, Google, etc.

Let's do a 3 node swarm with digitalocean!

> Digital Ocean Coupon for $10

Droplet = Server. Use Ububtu 16.04 (long term support, recent Kernel supporting Docker), $10 server should be good

Add your SSH key.

> Create and Upload a SSH Key to Digital Ocean

Create node1, node2, node3. Then use server IP addresses to SSH in, or, use configuration to enable using host names:

> Configure SSH for Saving Options for Specific Connections

Install Docker with script at http://get.docker.com on each node.

- `curl -sSL https://get.docker.com/ | sh` or
- `wget -qO- https://get.docker.com/ | sh`

> Docker Swarm Firewall Ports

- node1: `docker swarm init --advertise-addr <VISIBLE-IP>`
- node2: Copy and execute command from node1 output: `docker swarm join --token ... IP:PORT`
- node1: `docker node ls` - Should have 2 nodes listed
- node2: `docker node ls` - Should not work, only Managers can access the swarm
- node1: `docker node update --role manager node1` - promote node2 to be a Manager. Do it for node3 too.
- node1: `docker node ls` - Should list 2 nodes as Managers (node1 as "Leader", node2 as "Reachable")
- node1: `docker swarm join-token manager` - Displays the command to execut to join a node as a manager
- node3: Execute from the output of above: `docker swarm join --token ... node3` - Join directly as a Manager
- node1: `docker node ls` - Should list all 3 as Managers.

No need to remember/save tokens, you can always get them with `docker swarm join-token ...`

Join tokens can also be changed, in case they are compromised.

Now we have a 3 node redundant swarm!

- node1: `docker service create alpine --replicas 3 ping 8.8.8.8`
- node1: `docker service ls` - Should show 3/3 running
- node1: `docker service ps` - Lists containers running *ONLY* on local node
- node1: `docker service ps node2` - Lists containers running on specific node
- node1: `docker service ps my-service-name` - Lists containers running on *ALL* nodes

Scaling out with overlay networking

- `--driver overlay` - Creates a swarm wide network where containers across multiple nodes can access each other on a VLAN
- For container-to-container traffic inside a single swarm
- Optinal IPSec (AES) tunnel based encryption on network creation, off by default because of performance reasons
- Each service can be added to multiple networks (e.g. front/back)

Exercise

- `docker network create --driver overlay my-network`
- `docker network ls` - should see "ingress" (incoming), "docker_gwbridge" (outgoing) and our "my-network"
- `docker service create --name psql --network my-network -e POSTGRES_USER=myPass postgres`
- `docker service create --name drupal --network my-network -p 80:80 drupal`
- `watch docker service ls` - See service eventally start
- Go to the IP address in the browser to set up Drupal, point to "psql" as the db server name

Scaling Out with Routing Mesh

It does not matter which IP address you use for accessing Drupal, even though it is running on a single node. This is thanks to Routing Mesh.

Routing Mesh is an incoming (ingress) packet distribution among Task for a Service. It is on out-of-the-box.

- Spawns *ALL* nodes in the swarm
- Uses IPVS Kernel primitives from the Linux Kernel (core Linux Kernel feature that has been around for a long time)
- Load balances on ALL the nodes and listening on all the nodes for traffic
- Works two ways:
  - Container-to-container in an Overlay network, using VIP (private Virtual IP that is placed in front of each service) so services can talk to each other without an external load balancer (!)
  - External incoming traffic going to published ports (all nodes listen). The decision of which node will service the incoming request is handled by the Routing Mesh, it can choose any of the Workers. Note that containers can fail and recreated on different nodes, we want it to work without changing DNS and firewall settings.

VIPs are not DMS round-robin, they are better. The problem is that sometimes DNS client caches inside the applications prevent us from using the correct (ever changing) IP, so rather than fighting DNS client configurations, we just rely on the VIP, which is kind of like having a load balancer *on each node* (on the external network) that knows which node to forward traffic to.

> Use swarm mode routing mesh (Docker Docs)

Exercise with elasticsearch

- `docker service create --name search --replicas 3 -p 9200:9200 elasticserch:2` - v2 is the easiest to deploy :)
- `docker service ps search` - Each task should be created on a different node.
- `curl localhost:9200` - Repeat this multiple times to see that each time a different node responds (look for "name" in the JSON)
- It is a *stateless* load balancer (as of v17.03). No session cookies can be used.
- It is a OSI Layer 3 (TCP), not Layer 4 (DNS) load balancer
- Both above limitations can be overcome:
  - Nginx or HAProxy LB Proxy, or
  - Docker Enterprise Edition comes with L4 LB

Exercise with Docker's Distributed Voting Demo App (Dogs vs Cats)

> https://github.com/tkarakai-gto/udemy-docker-mastery/tree/master/swarm-app-1

In the exercise (created by Docker as an example app) there are 5 modules:

- "vote" - the app that accepts user input (voting if you like dogs or cats)
- "redis" - where the "vote" app pushes the votes
- "worker" - the component that monitors "redis" for new data and pushes them to the "db"
- "db" - SQL db to store the votes
- "result" - Monitoring dashboard to see current voting stats

The "vote" and "redis" modules are on a "front" network, the "db" and "result" modules are on the "back" network, and the "worker" is on both network.

These are the steps to make all these happen (after the swarm is up):

1. `docker network create front --driver overlay`
2. `docker network create back --driver overlay`
3. `docker service create --name redis --network front --replicas 2 redis:3.2`
4. `docker service create --name db --network back --mount type=volume,source=db-data,target=/var/lib/postgresql/data postgres:9.4`
5. `docker service create --name worker --network front --network back --replicas 1 dockersamples/examplevotingapp_worker`
6. `docker service create --name vote --network front --replicas 2 -p 80:80 dockersamples/examplevotingapp_vote:before`
7. `docker service create --name result --network back -p 5001:80 dockersamples/examplevotingapp_result:before`

Stacks

- Represents stacks of servies, networks and volumes, kind of compose files for Swarms. In fact Stacks accepts compose files as its input.
- Compose file has to be version 3 or higher
- Added in v1.13
- 1 Stack is for 1 Swarm
- `docker stack deploy` manages deployment (instead of using `docker service create`).
- It prefixes the stack name in front of the service/network/volume names.
- The compose file now has the `deploy:` instruction to specify swarm specific configuration (replicas, fail over, etc.). On the other hand it no longer executes `build:` instructions.
- Compose and Stack get along fine working with the same compose file: Compose ignores the `deploy:` part, Stack ignores the `build:` part.
- `docker-compose` cli does not need to be on the Swarm.

Exercise: The above voting app with Stacks

example-voting-app-stack.yml

```
version: "3"
services:

  redis:
    image: redis:alpine
    ports:
      - "6379"
    networks:
      - frontend
    deploy:
      replicas: 2
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: on-failure
  db:
    image: postgres:9.4
    volumes:
      - db-data:/var/lib/postgresql/data
    networks:
      - backend
    deploy:
      placement:
        constraints: [node.role == manager]
  vote:
    image: dockersamples/examplevotingapp_vote:before
    ports:
      - 5000:80
    networks:
      - frontend
    depends_on:
      - redis
    deploy:
      replicas: 2
      update_config:
        parallelism: 2
      restart_policy:
        condition: on-failure
  result:
    image: dockersamples/examplevotingapp_result:before
    ports:
      - 5001:80
    networks:
      - backend
    depends_on:
      - db
    deploy:
      replicas: 1
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
```

```
        condition: on-failure

  worker:
    image: dockersamples/examplevotingapp_worker
    networks:
      - frontend
      - backend
    deploy:
      mode: replicated
      replicas: 1
      labels: [APP=VOTING]
      restart_policy:
        condition: on-failure
        delay: 10s
        max_attempts: 3
        window: 120s
      placement:
        constraints: [node.role == manager]

  visualizer:
    image: dockersamples/visualizer:stable
    ports:
      - "8080:8080"
    stop_grace_period: 1m30s
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    deploy:
      placement:
        constraints: [node.role == manager]

networks:
  frontend:
  backend:

volumes:
  db-data:
```

There is a new service in there caller `visualizer` on port 8080 that shows the nodes and the running cotainers within.

Execute with:

- `docker stack deploy -c example-voting-app-stack.yml voteapp`
- `docker stack ls` - Lists number of services
- `docker stack ps voteapp` - Lists task runnig on all the nodes
- `docker stack services voteapp` - List per service
- `docker stack deploy -c example-voting-app-stack.yml voteapp` Rerun the same command after changing the compose file will update the stack as needed (for example scale as per the change). This is not a good practice becasue the file retains the changes probably meant to be executed once.

Secrets

Protects secrets within Docker like these:

- Usernames, passwords
- TLS certificaets and keys
- SSH keys
- Any data

Up to 512kb in size.

The Swarm Raft database is encrypted on disk by default (as of Docker v1.13.0).

Only stored on disk on Manager nodes.

Default is Managers and Workers use the "control plane" to communicate with TLS and Mutual Authentication.

Secrets are fist stored in Swarm, then assigned to Services that need them.

Only containers that were assigned can see the Secrets.

Secrets look like files, but they mounted from an in-memory RAM FS at these locations:

- `/run/secrets/<secret_name>` or
- `/run/secrets/<secret_alias>` The file name is the key, the value is the file content.

Local `docker-compose` based development can use real disk file-based secrets, they will work, but obviously it is **not secure**.

- `docker secret create psql_user psql_user.txt` - Adds the secret from a file
- `echo "MyPassword" | docker secret create psql_user -` - Adds the secret from STDIN (that's what the `-` at the end is for)

None of the above two methods are secure. The first one stores the secret in a file on the disk, the second one adds the secret to the basj history. Not good for production!

- `docker secret inspect psql_user` - JSON metadata (does not give away the secret :))
- `docker service create --name psql --secret psql_user --secret psql_pass -e POSTGRES_USER_FILE=/run/secrets/psql_user -e POSTGRES_PASSWORD_FILE=/run/secrets/psql_pass postgres`
  - By convention the `_FILE` postfix will replace the original env variable.
- `docker exec -it psql.1.... bash`, then `ls /run/secrets` should show the two files.
- `docker container logs psql.1....` - should show the db starting up ok
- `docker service update --secret-rm ...` - remove. It will redeploy the container without the secret!

Secrets with Swarm Stacks

Secrets with Stacks must be version 3.1 or newer.

docker-compose.yml

```
version: "3.1"

services:
  psql:
    image: postgres
    secrets:
      - psql_user
      - psql_password
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/psql_password
      POSTGRES_USER_FILE: /run/secrets/psql_user

secrets:
  psql_user:
    file: ./psql_user.txt
  psql_password:
    file: ./psql_password.txt
```

Secrets are assigned to the services that need them. The "short form" is shown above, there is a "long form" that allows specifying the user/permissions in the Linux syntax, targetting specific (non-root) users.

The `secret` section is either `file` based (as shown above) or `external` when the secrets are precreated (like with the CLI).

- `docker stack deploy -c docker-compose.yml mydb` - Creates the secrets fists, then the network, then the services.

Don't forget to clean up the secrets after the services are gone!

docker-compose.yml - modified to use external secrets

```
version: '3.1'
```

```
services:

  drupal:
    build: .           # Build using the default Dockerfile in the current directory
    image: drupal:8.2  # If both 'image' and 'build' are specified, Compose will name the built image as per 'image' line
    ports:
      - 8080:80
    volumes:
      - drupal-modules:/var/www/html/modules
      - drupal-profiles:/var/www/html/profiles
      - drupal-themes:/var/www/html/themes
      - drupal-sites:/var/www/html/sites
  db:
    image: postgres:9.6
    environment:
      - POSTGRES_USER_FILE=/run/secrets/psql-usr
      - POSTGRES_PASSWORD_FILE=/run/secrets/psql-pw
    volumes:
      - drupal-data:/var/lib/postgresql/data

volumes:
  drupal-data:
  drupal-modules:
  drupal-profiles:
  drupal-themes:
  drupal-sites:

secrets:
  psql-usr:
    external: true
  psql-pw:
    external: true
```

- `echo "myUser" | dokcer secret create psql-usr -`
- `echo "myPassword" | dokcer secret create psql-pw -`
- `docker stack deploy -c docker-compose.yml drupal`

Secrets for local development

Use the same compose file!

- `docker-compose up -d`
- `docker-compose exec psql cat /run/secrets/psql_user` - should output the content of the secret file, just like if we were in a swarm. This is becasue secrets are automatically bind mounted to the containers by Docker. This is absolutely *NOT secure*, but it works for dev. This only works for file based secrets, not for the `external` ones. This is cool, because now the secrets can be used the same way in dev as on production, with the same compose files.

Full App lifecycle with Compose

Single set of compose files as the complexity increases.

docker-compose.yml - Base compose file

```
version: '3.1'

services:

  drupal:
    image: bretfisher/custom-drupal:latest

  postgres:
    image: postgres:9.6
```

docker-compose.override.yml - if present, this will overrides `docker-compose.yml`, for development

```
version: '3.1'

services:

  drupal:
    build: .
    ports:
      - "8080:80"
    volumes:
      - drupal-modules:/var/www/html/modules
      - drupal-profiles:/var/www/html/profiles
      - drupal-sites:/var/www/html/sites
      - ./themes:/var/www/html/themes          ## DEV ONLY to bind local dev files mounted

  postgres:
    environment:
      - POSTGRES_PASSWORD_FILE=/run/secrets/psql-pw
    secrets:
      - psql-pw
    volumes:
      - drupal-data:/var/lib/postgresql/data

volumes:
  drupal-data:
  drupal-modules:
  drupal-profiles:
  drupal-sites:
  drupal-themes:

secrets:
  psql-pw:
    file: psql-fake-password.txt   ## for DEV we have to use the file based secrets
```

- `docker-compose up` for local development with the above files

docker-compose.test.yml - for CI/Jenkins solution

```
version: '3.1'

services:

  drupal:
    image: bretfisher/custom-drupal
    build: .
    ports:
      - "80:80"
    # No volumes needed for test/CI, it will get rid of them when container is done

  postgres:
    environment:
      - POSTGRES_PASSWORD_FILE=/run/secrets/psql-pw
    secrets:
      - psql-pw
    volumes:
      - ./sample-data:/var/lib/postgresql/data   # Preloaded standard data for testing

secrets:
  psql-pw:
    file: psql-fake-password.txt
```

- `docker-compose -f docker-compose.yml -f docker-compose.test.yml up -d` for CI/testing, specifying compose files explicitely, in the order of base -> overriding files.

docker-compose.prod.yml

```
version: '3.1'
```

```
services:

  drupal:
    ports:
      - "80:80"
    volumes:
      - drupal-modules:/var/www/html/modules
      - drupal-profiles:/var/www/html/profiles
      - drupal-sites:/var/www/html/sites
      - drupal-themes:/var/www/html/themes

  postgres:
    environment:
      - POSTGRES_PASSWORD_FILE=/run/secrets/psql-pw
    secrets:
      - psql-pw
    volumes:
      - drupal-data:/var/lib/postgresql/data

volumes:
  drupal-data:
  drupal-modules:
  drupal-profiles:
  drupal-sites:
  drupal-themes:

secrets:
  psql-pw:
    external: true
```

- `docker-compose -f docker-compose.yml -f docker-compose.prod.yml config > prod.yml` - generates a "combined" YAML file, should be run by the CI solution to output to a place ready for Prod.
- `docker stack deploy ...`
- `extends` doesn't work in Stacks yet (June, 2017)

> - Using Multiple Compose Files (Docker Docs)
> - Using Compose Filed in Production (Docker Docs)

Image Storage and Distribution

They are meant to be part of the Docker ecosystem, copying images in ZIP files is not really a good solution.

Docker Hub

Docker Hub is not the only public registry! It is the most popular. It has storage plus building capabilities. Docker Hub can detect repository changes in GitHub or BitBucket! It can also rebuild your images every time one of your dependencies (the `FROM` image) are updated.

Docker Store

- New as of 2016.
- Docker software. It's like Apple App Store, controlled by Docker the company
- Some Certified, some paid for

Docker Cloud

- NOT a cloud hoster
- Web based orchestration/management system for your Docker swarms and clusters. It helps create/manage stack, services in Amazon, Azure, etc.
- Automated uilds $$
- Security scan of images $$

Docker Registry (open source)

- Written in `go`
- No UI, more like an HTTPS server
- Default port: 5000
- Can act as a registry proxy to cahce images pulled through Docker Hub
- By deault it uses TLS, unless talking to localhost
- `docker container run -d -p5000:5000 --name registry registry` - That's it.
- `docker image tag hello-world 127.0.0.1:5000/hello world` - Tags it in the local registry
- `docker image push 127.0.0.1:5000/hello-world` - Push to the local registry
- `docker image pull 127.0.0.1:5000/hello-world` - Pull from the local registry

Above examples do *NOT* retian images when they stop! For that we need a volume:

- `docker container run -d -p5000:5000 --name registry -v $(pwd)/registry-data:/var/lib/registry registry`

Registry on a Swarm

In a Swarm, you cannot use an image that is only in one node! It must be able to pull from a repository availabe for all nodes.

Becasue of the `Routing Mesh` 127.0.0.1:5000 is accessible by all nodes.

Registry can run the same way in a Swarm pretty much the same way, running as a `service`.

- `docker service create --name registry --publish 5000:5000 ... registry`
- `docker pull nginx`
- `docker tag nginx 127.0.0.1:5000/nginx`
- `docker push 127.0.0.1:5000/nginx`
- `docker service create --name nginx -p 80:80 --replicas 5 --detach=false 127.0.0.1:5000/nginx` - Creates a service from the image in the local registry!

Extra random goodies

Windows Tips

How to upgrade Docker Engine in Docker Toolbox

- When you upgrade Docker Toolbox (on an old Windows 7 laptop), it only upgrades the Docker Client. Do `docker version` to verify. In order to upgrade the Docker server (Engine) you need to issue `docker-machine upgrade default` (if yours is called "default")

How to make the `docker` command work in different colnsoles

If the `docker` command doesn't work in your Windows console (like cmder or the default Windows Command Prompt) use this to set the appropriate environment. Execute the suggested command.

- `docker-machine env --help`
- `docker-machine env --shell cmd default` -- displays the commands to execute for the `cmd` (Windows) console. There are variations for fish, cmd, powershell and tcsh. `default` is the name of the Docker Machine.

Related services, alternatives

> - https://cloud.docker.com/ - Continuous integration, delivery, registry management in the cloud. Includes one free private repository.

- https://coreos.com/rkt - Pronounced as "rocket", alternative to Docker. `rkt` comparing itself to others
- Blog: Moving from Docker to rkt

- https://kubernetes.io/ - Google backed container orchestration (works with Docker and Rkt)

- https://deis.com/ - Kubernetes based tools

Dev Tools to experiment with

- https://circleci.com/ can do continuous build/test/deploy of a Docker container, triggered by Github/Bitbucket changes, 1 container FREE.
- https://codeship.com can do pretty much the same
- https://www.digitalocean.com/pricing/ is a service much like https://aws.amazon.com/ec2/pricing/on-demand/
- Digital Ocean Coupon for $10
- Another Digital Ocean Coupon for $10

Monitoring

...other than `docker stats` and `docker service logs`

- cadvisor - "Container Advisor", real time resource usage and performance data monitoring (fancy verion of `docker stats` on a web page). Backed by Google. Simple, runs as a container alongside of the other containers, no historical data, not meant to be for production.
- DataDog - Good for production. Free for 5 hosts (1 day data retention). Need to install agents. Alerts too.

- logspout - a log router for Docker containers that runs inside Docker. Runs in a container itself. It attaches to all containers on a host, then routes their logs wherever you want. It's a mostly stateless log appliance. It's not meant for managing log files or looking at history. It is just a means to get your logs out to live somewhere else, where they belong.
- papertrail - Log management in the cloud. Free for 7 days retention (48 hours search) 100MB/**month** (!)
- loggly - Log management in the cloud. Free for 7 days retention up to 200MB/**day** (!)

Public domain related services

- http://internetbs.net - Domain name regisrar charging $9/year for `.com`, *including* whois protection. Hosted outseide the U.S..
- https://letsencrypt.org/ - Free, real SSL/TLS Certificates (with 90 days validity)
- https://www.ssllabs.com/ssltest/ - SSL test for web sites. You should see A+ rating!

Some free, self-paced training

- http://training.play-with-docker.com/
- http://container.training