**Course Intro**

Once in a decade shift all need to take part in

Mainframe to PC 90’s

Bare-metal to Virtual 00’s (VMWare)

Datacenter to Cloud 10’s – easy cheap disposable compute power

Host to Container today (Serverless)

Docker is focused on the migration experience

“By 2020, more than 50% of global organizations will be running containers in production” -Gartner

Benefit

Speed of software deployment, getting things done

Develop faster

Build faster

Test faster

Deploy faster

Recover faster

Matrix from Hell – multiple applications and dependencies that need to work together.

Containers are consistent across the board

Distribute software in the same way

Maintenance and complexity drains budgets, so innovation suffers

No code changes required

PayPal

Migrated 700+ apps

Now 150000 containers

50% dev productivity boost

Metlife

70% reduction in VM costs

66% cost reduction

67% less CPUs

Cloud Native

Windows Server 2016 supports native windows containers

Docker for Windows runs in WS 2016 but not required

Only do this for when you run locally for dev/test. NOT for prod.

**Section 3: Creating and Using Containers**

docker container stop

docker container inspect

docker container stats

**docker container run -it** // start new container interactively

**docker container exec -it** // run additional command in **existing** container, not in existing container

How to get into the container?

-it. “I” is for keeping session open to receive additional commands. “t” is to simulate terminal.

docker container run -it –name proxy nginx bash // this opens up a terminal in the container. Each container has a default behavior to run on start. By specifying “bash” we override that behavior and tell it to start a bash terminal

ls -al // list all files in the root folder of the container

type “exit” to exit bash terminal and stop container

Containers only run as long as the command in which it was started on is running!

docker container run -it –name ubuntu ubuntu // create ubuntu container. BASH is the default command in ubuntu image so it already opens the terminal.

docker container start -ai <name> // restart existing container

How to see the shell inside a running container already running mysql or nginx?

docker container exec // run additional process in running container

docker container exec -it mysql bash // starts a bash terminal on an existing mysql container

docker pull <image name> // download the latest image

Docker Networks Concepts

docker container run -p // exposes the port

docker container port <container> // port check

docker container run -p 80:80 –name webhost -d nginx // -p exposes port 80 of nginx. The left side is the HOST port and forwards traffic to port 80 of the CONTAINER.

docker container port webhost // see what ports are forwarding traffic to that container

docker container inspect –format ‘{[ .NetworkSettings.IPAddress ]}’ webhost // check IP on the container

Docker Networks CLI management

docker network ls // show networks

docker network inspect // inspect a network

docker network create –driver // create a network

docker network connect // attach a network to a container

docker network disconnect //detach a network from container

docker zero or –network bridge // default docker network that bridges through the NAT firewall through the physical network your host is connected to

docker network create my\_app\_net

docker container run -d –name new\_nginx –network my\_app\_net nginx

docker network inspect my\_app\_net

Default Security

Create your apps so front/backend sit on the same Docker network

Their inter-communication never leaves host

All externally exposed ports closed by default

You must manually expose via -p, which is better default security

Gets better with swarm and overlay networks

Docker Networks - DNS and how containers find each other

Static IPs and using IPs for talking to containers is an anti-pattern. Do your best to avoid it. IP addresses can change. Containers use the container name as a equivalent of a host name for containers talking to each other.

docker daemon has a built-in DNS server that containers use by default. Docker uses the container names as equivalent as a host name for containers talking to each other. Automatic DNS resolution using the container names.

Containers shouldn’t rely on IP addresses to communicate with each other. DNS for friendly names is built-in if you use custom networks.

Docker DNS Default Names. Docker defaults the hostname to the container’s name, but you can also set aliases.

the default bridge network driver allow containers to communicate with each other when running on the same docker host. It does not have DNS server built in by default. You can use –link to manually create links to other containers but it’s easier to just create a new network for the containers.

docker container top – list running processes

docker container stats – show I/O, CPU, performance stats

docker container logs – show logs how a container was setup

Assignment – CLI Testing

docker container run –rm -it centos:7 bash // start a centos7 container and run bash on it. Don’t need to open a port.

yum update curl // update curl in centos7 bash

curl –version // find curl version in bash

docker container run –rm -it ubuntu:14.04 bash // install ubuntu container

apt-get update && apt-get install -y curl //

--rm // removes the container as soon as you exit

Assignment: DNS Round Robin Test

Technique big companies use. There’s multiple DNS records and IP addresses behind the name you are using on the internet.

We can have multiple containers on a created network respond to the same DNS address

* Create a new virtual network (default bridge driver)
* Create two new containers from elastic search:2 image
* Research and use -network-alias search when creating them to give them additional DNS name to respond to. This works around the problem: if I can’t have multiple containers with the same name, how can I resolve DNS on my networks and maybe have the same app installed twice (DEV and Test in the same server) and in both call /search on the DNS. This is how you get around this limitation. You can keep adding aliases to your containers and they can respond just like a DNS round robin does
* Run “alpine nslookup search” with –net to see two containers list for the same DNS name
* Run “centos curl -s search:9200” with –net multiple times until you see both “name” fields show

docker network create dude

docker container run -d –network dude –network-alias search elasticsearch:2

docker container run -d –network dude –network-alias search elasticsearch:2 // create 2 elastic search images in the background. Give them both the same alias and assign to the custom network

docker container run –rm –network dude alpine nslookup search // runs nslookup on the DNS entry and cleans itself up with –rm

docker container run –rm –network dude centos curl -s search:9200 // shows the two elastic search servers

**Section 4: Container Images**

Not a complete OS. The host provides the kernel. It’s not booting up an OS, just starting an application.

Basically, your app binaries and dependencies and metadata about the image data and how to run the image

Start with images that are “official”. These have better documentation and are maintained and supported.

Images are not named. They are tagged. There can be multiple tags for an image.

docker image ls

docker pull nginx // download latest image

docker pull nginx:1 // download latest 1 version

docker pull nginx:1.11 // download latest 1.11 version

For production, always specify the exact version you want. You some other DevOps tool to manage this (Octopus)

Image ID is based upon cryptographic SHA of each image in docker hub

Image layers

Union file system concept: making layers about the changes

docker image history nginx:latest // history of the image layers

Store from blank image. Each change is a layer

Layers are cached and can be shared among images. This caching of layers helps save time and space.

Containers create a new read-write layer on top of base image

Layering “stack of pancakes”

If two containers are built from the same image, they “share” the layers from the base image. The only difference in storage is what’s running on the container top layer.

**Copy on write (COW)** If you change a file from the base image, a copy of this file is made and placed inside of the container layer

**A container is just a single read-write layer on top of an image**

So the container is just the running process and those files that are different than they were in the image

Docker image inspect // returns JSON metadata about the image

Image tagging and pushing

**docker image tag**

Official repositories. They live at the “root” namespace of the registry, so they don’t need account name in front of repo name

Tags -> pointers to specific image commits.

**Latest** tag. It’s just the default tag but image owners should assign it to the newest stable version.

**docker Image Tag SOURCE\_IMAGE[:TAG] TARGET\_IMAGE[:TAG].** If no source image is specified, then it defaults to latest tag

**docker Image Push** // uploads changed layers to a image registry

**docker login** // defaults to logging in Hub, but you can override by adding server URL

**docker logout**

Must tag images to push to docker hub

For private images, create the repository first as private, then you upload it

Dockerfile Basics

Recipe for creating docker images

<DOCKER\_FILE> docker image build -t customnginx . // build image in the current directory (.)

Each step is a line in docker file with a unique key. The key is kept so if the line doesn’t change in the docker file it’s not going to re-run it. This caching is what makes software building so fast. As you build an image the first time and you are just changing your custom source code and not the application itself, all the installation has already happened so you will have very short build times.

It’s **CRITICAL** that you keep things that change less at the top of your docker file and the things that change the most at the bottom of your docker

-f command is used to specify a dockerfile with an alias of –file

WORKDIR is the docker file stanza (command) used to change the directory in a dockerfile.

Stanzas

FROM

ENV

RUN // run commands, shell scripts inside the container

EXPOSE

CMD // final command run every time you start a container

Each stanza is a layer in the image

We can use plugins to manage all the logs in all the containers

Dockerfile Assignment I

FROM node:6-alpine

EXPOSE 3000

RUN apk add --update tini

RUN mkdir -p /usr/src/app

WORKDIR /usr/src/app

COPY package.json package.json

RUN npm install && npm cache clean –force

COPY . .

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

**docker image build -t testnode .** // build image in current directory, tag it as testnode

docker container run –rm -p 80:3000 testnode // test image locally

**Section 5: Docker Volumes**

docker volume ls

docker volume prune // cleanup unused volumes

docker container run -d mysql -e MYSQL\_ALLOW\_EMPTY\_PASSWORD=True mysql

docker volume inspect

docker container run -d mysql -e MYSQL\_ALLOW\_EMPTY\_PASSWORD=True -v mysql-db:/var/lib/mysql mysql // -v allows us to specify a new volume for the container or create a named volume

docker volume create // create volume ahead of time to specify a different driver (plugin)

When adding a bind mount to a docker run command, you can use the shortcut $(pwd), (or ${pwd} depending on your shell). What does that do? It runs the shell command to print the current working directory, to avoid having to type out the entirety of your directory location

Assignment: Named volumes

Real world scenario: database upgrades

inspect VOLUME path in tag documentation in docker hub

**VOLUME** **[/var/lib/postgresql/data]**

docker container run -d –name psql -v psql: /var/lib/postgresql/data postgres:9.6.1

docker container logs -f psql // inspect logs and -f to keep watching as it runs

docker container stop psql

docker container ps -a // list all containers and status

docker container logs <container\_name>

Bind Mounts Assignment

Bind mounts are great because you can change files in your host system and those are used inside of the containers.

Map the current directory into the site container with $(pwd)

cd <sample-directory>

docker container run -p 80:4000 -v <sample-directory>:/site bretfisher/Jekyll-serve

Edit file locally and see changes refresh in localhost:80

**Section 6: Docker-compose CLI**

Not a production grade tool but ideal for local development and test

docker-compose-up // setup volumes/networks and start all containers

docker-compose-down // stop containers and remove everything

If all your projects had a Dockerfile and docker-compose.yml then the “new developer onboarding” would be

git clone github.com/some/software

docker-compose up

docker-compose down

docker-compose top

docker-compose ps

docker-compose down -v // remove volumes

docker-compose down –rmi local // delete containers and remove local/custom images

**Recreate everything in Lesson 53**.

Docker compose file example

version: '2'

services:

drupal:

image: drupal

ports:

- "8080:80"

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:

image: postgres

environment:

- POSTGRES\_PASSWORD=mypasswd

volumes:

drupal-modules:

drupal-profiles:

drupal-sites:

drupal-themes:

Using Compose to Build

Compose can also build your custom images

Also rebuild with docker-compose build

Great for complex builds that have lots of vars or build args

Assignment: Build and Run Compose

Building custom drupal image for local testing

Use the drupal image along with the postgres image

Use ports to expose drupal on 8080

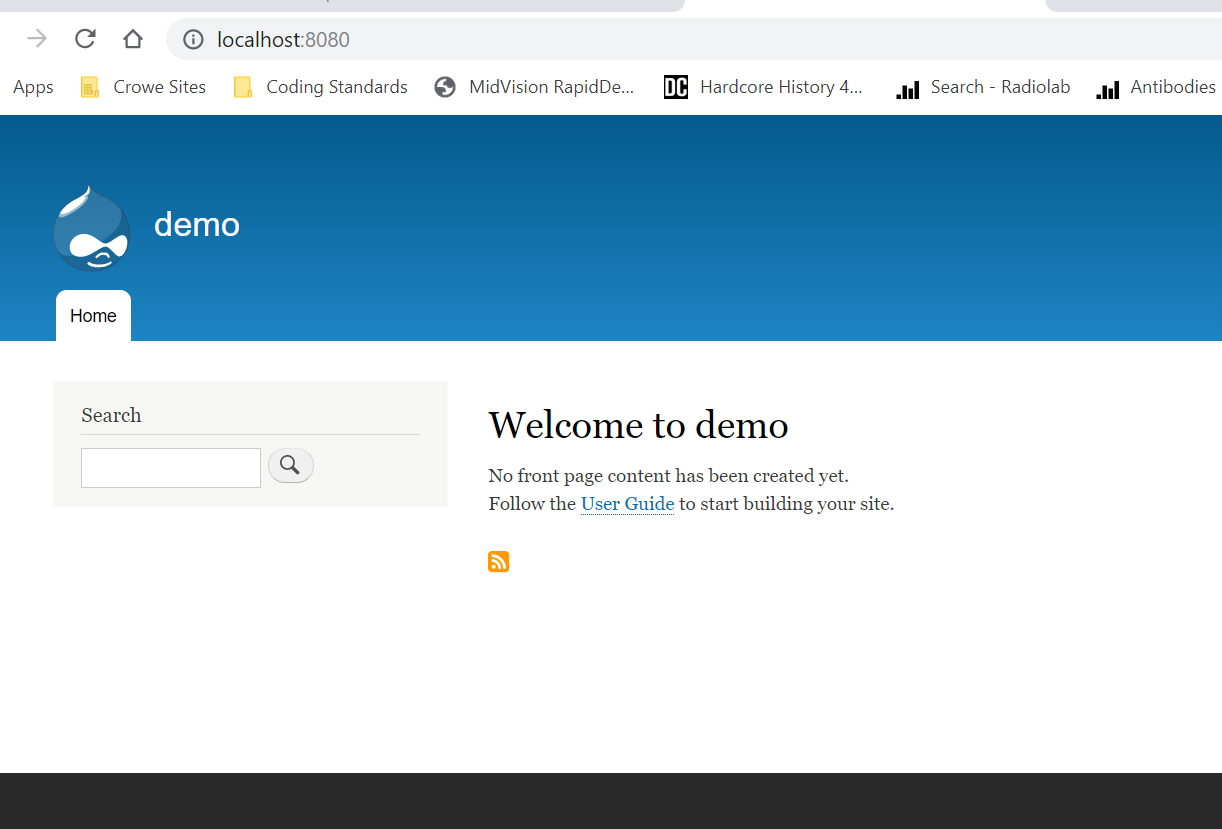
docker pull drupal

docker image inspect drupal // check out exposed port under “Exposed Ports”

// after compose is built

docker-compose up

Navigate to localhost



// cleanup

docker-compose down -v // also removes the volumes

Docker Compose File

# create your drupal and postgres config here, based off the last assignment

version: '2'

services:

    drupal:

        image: drupal

        ports:

            - "8080:80"

        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:

        image: postgres

        environment:

            - POSTGRES\_PASSWORD=mypasswd

volumes:

 drupal-modules:

 drupal-profiles:

 drupal-sites:

 drupal-themes:

Compose Adding Image Building

Compose can also build custom images

Will build them with docker-compose up if not found in cache

Also rebuild with docker-compose build

Great for complex builds that have lots of vars or build args

docker-compose down –rmi local // remove images

Assignment: Build and Run Compose

Build custom drupal image for local testing

Maybe your learning Drupal admin, or a software tester

Testing apps is easy and fun!

Start with Compose file form previous assignment

DockerFile

# create your custom drupal image here, based of official drupal

FROM drupal:8.6

# RUN apt package manager command to install git. -y is to automatically select yes. Very common. Not required but keeps it clean

RUN apt-get update && apt-get install -y git \

    && rm -rf /var/lib/apt/lists/\*

WORKDIR /var/www/html/themes

# use git to clone the theme

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

# just in case change the working directory

WORKDIR /var/www/html

Compose File

# create your drupal and postgres config here, based off the last assignment

version: '2'

services:

    drupal:

        image: custom-drupal

        build: . # build the default Dockerfile in this directory

        ports:

            - "8080:80"

        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:

        image: postgres

        environment:

            - POSTGRES\_PASSWORD=mypasswd

        volumes:

            - drupal-data:/var/lib/postgresql/data # add a volume so the database will persist across Compose restarts because docker-compose down will delete the database

volumes:

 drupal-data:

 drupal-modules:

 drupal-profiles:

 drupal-sites:

 drupal-themes:

Finally run docker-compose up

**Section 7: Swarm**

How to manage containers across many servers or nodes? How do we automate container lifecycle? How can we easily scale out/in/up/down? How can we ensure our containers are re-created if they fail?

How can we replace containers without downtime (blue/green deploy)? How can we control/track where containers get started? How can we create cross-node virtual networks? How can we ensure only trusted servers run our containers? How can we store secrets, keys, passwords and get them to the right container (and only that container)?

Swarm Mode: Built-In Orchestration. Built-in **clustering solution** to orchestrate lifecycles of containers.

A cluster of machines running docker providing a scalable platform to run many containers

Features

* Decentralized access
* High security
* Auto load balancing
* High scalability
* Roll-back a task

Containers are deployed as services. A service is a group of containers of the same image.

**docker info** // check if swarm is enabled. By default it is not

**docker swarm init** // initialize a swarm with a single node with all of the swarm features

* Created a root certificate for the swarm to establish trust and signed certificates for all nodes and managers.
* Certificate is issued for first Manager node
* Join tokens are created to join the swarm
* Raft database created to store root CA, configs and secrets. RAFT is a protocol to ensure consistency across multiple nodes. Encrypted by default on disk. No need for another key/value system for saving secrets.

docker node ls // see managers/workers

docker node COMMAND // see all commands available

docker swarm COMMAND // see all commands available for swarm

docker service --help // replaces the docker run. Pet vs. cattle analogy. Service is a cluster of containers.

docker service create alpine pine 8.8.8.8 // create a new service

docker service ls // list all services

docker service ps <name> // inspect a service

docker service update <id or name> --replicas 3 // scale up the service

docker service ps <name> // see list of replicated nodes

docker service update –help // see commands available for update. Can use this to update options in a swarm in a pattern that follows consistent availability

Service recreates the container if one of them is dropped. To remove the container you would have to remove the service

docker service rm <name> // the request is placed in a queue. Once its ready to process, swarm will remove all containers in the service

**\*\*\*Creating a 3-node Swarm\*\*\***

Make sure Swarm ports are open, see documentation

On Node 1

docker swarm init –advertise-addr <public IP address of host>

After running the command, copy the docker swarm join command that appears

On Node 2

Paste the docker swarm join command. Will be added as a worker.

On Node 1

docker node ls

You can see second node as a worker. Can’t use swarm commands on Node 2 because workers aren’t privileged. Don’t’ have access to control the swarm

docker node update –role manager node2 // promote node 2 to manager

docker swarm join-token manager // get token for node3 to join

On Node 3

Paste the docker join command created above into node 3

Now we have a 3 node swarm

On Node 1

docker service create –replicas 3 alpine ping 8.8.8.8

docker service ls // 3/3 running

docker node ps // see containers on node 1

docker node ps node2

docker node ps node3

docker service ps <name of service> // list all 3 tasks

Don’t have to operate containers in each node. You can really operate on the swarm for most things from node 1

You now have a fully operational swarm cluster!

**Section 8: Swarm Basic Features and How To Use Them**

Overlay Networking

Overlay – network driver. Create a swarm-wide bride network

Scaling out our virtual network

Overlay Multi-Host Overlay

Just choose –driver overlay when creating a network. Creates a swarm-wide bridge network so containers can access each other like a VLAN

For container-to-container traffic inside a single Swarm

Optional IPSec(AES) encryption on network creation

Each service can be connected to multiple networks

On Node 1

docker network create –driver **overlay** mydrupal // create swarm wide bridge network

docker service create –name **psql** –network mydrupal -e POSTGRES\_PASSWORD=mypwd postgres // create postgres service and connect it to the mydrupal network using the postgres image

docker service ls

docker service ps psql // see on which node is it running

docker container logs psql

docker service create –name –name **drupal** –network mydrupal -p 80:80 drupal // create another service joined to the same network, publish on 80 80 from drupal image

docker service ps // confirm services

docker service ps drupal // we see we have the db in node 1, website on node 2. How do they talk to each other? Using the service names

Overlays are great because it is as everything worked on the same subnet

// on which node is my website running in?

docker service ps drupal

docker service inspect drupal

Scaling Out With Routing Mesh

Global routing traffic

Routes incoming packets for a Service to proper task

Spans all nodes in a Swarm

Uses IPVs from Linux Kernel

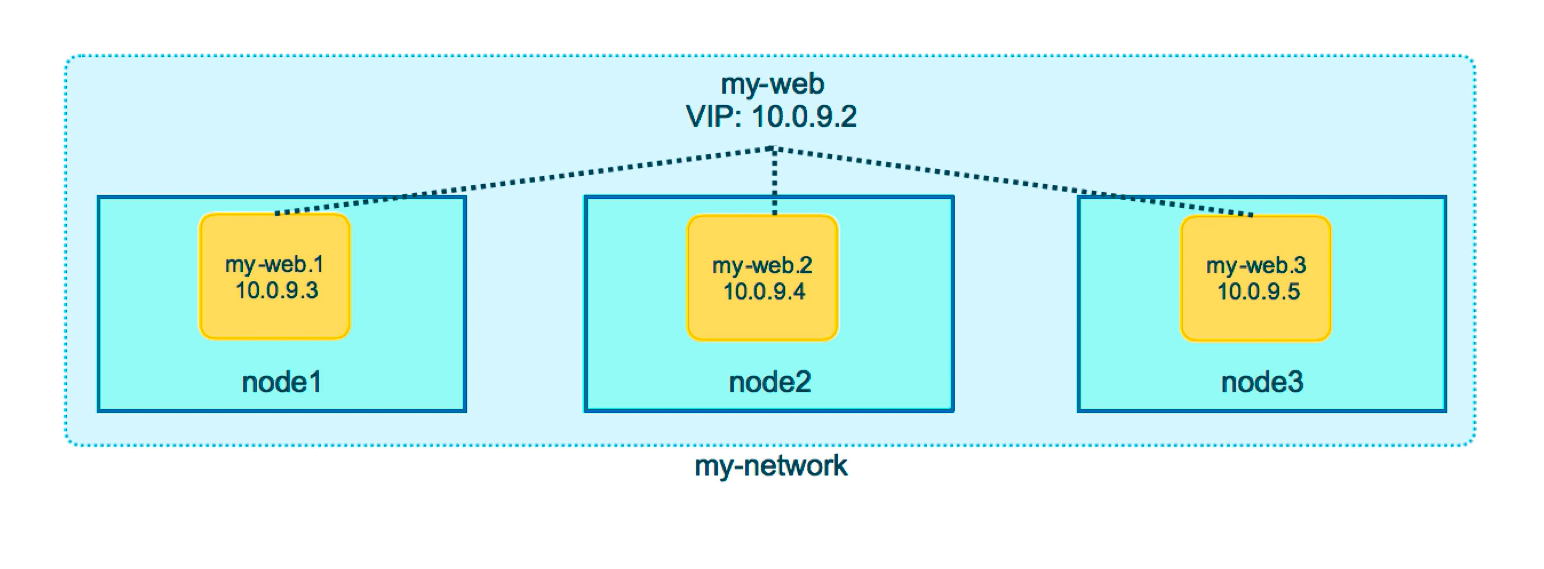
Load balances Swarm Services across their tasks, listening on all the nodes for traffic

Container-to-container in a Overlay network (uses Virtual IP or VIP that swarm puts in front of all services). Ensures the load is distributed among all the tasks for its service.

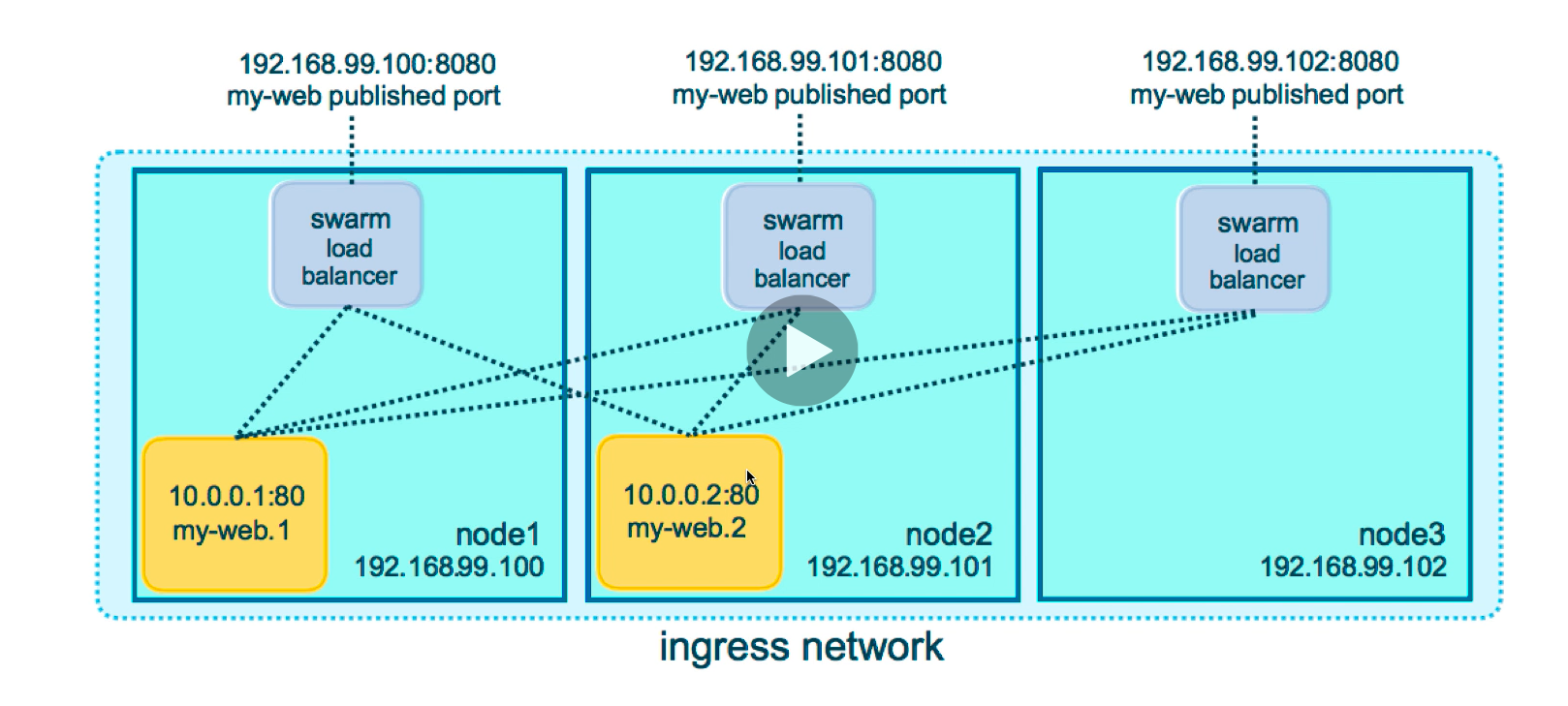
External traffic incoming to published ports (all nodes listen)

Allows app to be accessible from any node

In the network there is a private virtual network that is mapped to the DNS of the service name. By default the DNS name is the name of the service. Any containers that need to talk to the service in the swarm only need to worry about using the my-web dns. The VIP load bounces the traffic amongst all the tasks in the service. Not the same as round-robin



When receiving incoming traffic, the load balancer decides which container to get the traffic and whether the traffic goes to a local node or needs to go over the network to a different node.



docker service create –name search –replicas 3 -p 9200:9200 elasticsearch:2 // create service with 3 replicas, publish on 9200 which is default port from version 2 of elasticsearch

docker service ps search // see it created each task on a separate node by repeating it. You see the VIP acts as a load balancer and distribute the load across the three tasks.

curl localhost:9200 // get elasticsearch information

Routing mesh is a stateless load balancer. If you have to use session cookies, then add other things to solve that problem.

This LB is at OSI layer 3 (TCP), not layer 4 (DNS). Operates at the IP and Port level.

Containers and networks are a many-to-many relationship. A single container can be attached to many networks and Services can be attached to multiple Docker networks, and a network can have many containers.

Assignment: Create Multi-Service App

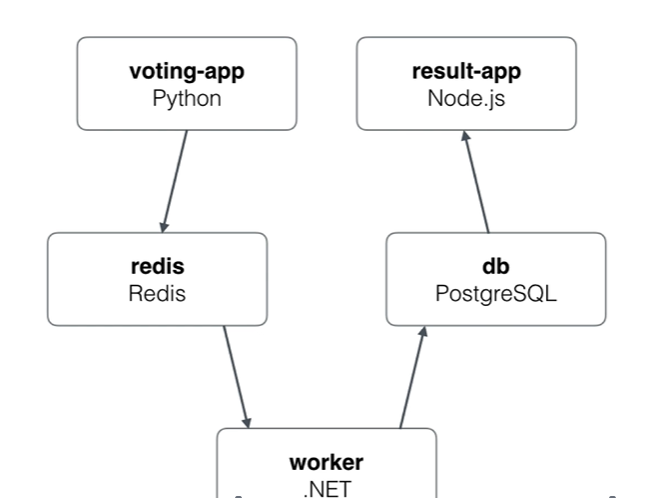
Real-world scenario of multi-tier app.

Use Docker’s Distributed Voting app.

Use swarm-app-1 directory in course repo.

1 volume, 2 networks, 5 services.

Do not build on production Swarm. Use Docker Hub images.



- vote

    - bretfisher/examplevotingapp\_vote

    - web front end for users to vote dog/cat

    - ideally published on TCP 80. Container listens on 80

    - on frontend network

    - 2+ replicas of this container

Node 1:

// create two networks to act as a firewall and give separation (protection) between the two apps

docker network create -d overlay backend

docker network create -d overlay frontend

// create the voting service

docker service create –name vote -p 80:80 –network frontend –replicas 2 dockersamples/examplevotingapp\_vote:before

// redis

docker service create –name redis –network frontend –replica 1 redis: 3.2

// worker

docker service create –name worker –network frontend –network backend dockersamples/examplevotingapp\_worker

// db

docker service create – name db –network backend –mount type=volume,source=db-data,target=/var/lib/postgresql/data postgres 9.4

// result

docker service create –name result –network backend -p 5001:80

// service logs

**docker service logs**

Stacks: Production Grade Compose

Compose files for production Swarm. A layer of abstraction. Accept compose files as their declarative definition for services, networks, and volumes. Use “docker stack deploy” rather than docker service create. Stacks manages all those objects for us, including overlay network per stack. New deploy: key in Compose file. Can’t do build:. Building shouldn’t happen on your production swarm. It should happen in your CI build. Compose ignores deploy:, Swam ignores build:. Docker-compose cli not needed on swarm server.

Multiple services, volumes, and overlay networks in your compose file. YAML file to do those things we don’t want to do in the service commands.

// open the stack file

vim example-voting-app-stack.yml

Needs version 3 or higher

docker stack deploy -c example-voting-app-stack.yml voteapp

docker stack ls

// see all of your nodes in the stack running

**docker stack ps voteapp**

// similar to service ls. Shows number of replicas

**docker stack services voteapp**

Previous two give you a complete picture of what’s going on inside of your stack

// dive into networking

docker network ls

// let’s say you update the yaml file to update the number of replicas then run command. It will recognize it is existing stack and update it

docker stack deploy -c example-voting-app-stack.yml voteapp

**Secrets Storage: Protecting Environment Variables**

Easiest secure solution for storing secrets in Swarm. Encrypted on disk, in-transit. Usernames, passwords, SSH keys, TLS certificates, API keys, etc.

Swarm Raft Db is encrypted on disk. Secrets are assigned to services. Only containers in assigned services can see them. Secrets are a swarm only thing. Without swarm you cannot have secrets. You can mount secrets into a text file in a local container but it’s not secure for production.

docker secret create psql\_user psql\_user.txt //passing a secret through a file

Echo “myDbPassword!” | docker secret create psql\_pass - // passing secrete through command line

docker secrets ls

docker secret inspect psql\_user

Only containers and services we assign them too can access unencrypted secrets

// create a service and map a secret to it so they can see the files in the container

docker service create –name psql –-secret psql\_user -–secret psql\_pass -e POSTGRESS\_PASSWORD\_FILE=/run/secrets/psql\_pass -e POSTGRESS\_USER\_FILE=/run/secrets/psql\_user postgres

docker service ps psql // find container name in service

docker exec -it <container name> bash // see what’s inside the container

ls /run/secrets // list directory files in path

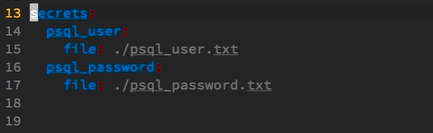
docker service update --secret-rm // remove secret. Stops and redeploys the container (because they are supposed to be immutable). Not ideal for database updates so we have to figure out a way to update database passwords.

Secrets with Stacks

docker service create - -name search - -replicas 3 -p 9200:9200 elasticsearch:2

In sample directory there is a compose file and two secrets stored in files. Version of compose file has to be 3.1 to support secrets. Define secret in the compose file. Need to tell compose where our secrets are.

Declare secrets at the bottom, then assign to services on top



docker stack deploy -c docker-compose.yml mydb

docker secret ls

docker stack rm mydb // remove stack

Don’t leave residual secrets that would be easy for people to get.

Assignment: Create Stack w/ Secrets

Add secret via “external” cli. Copy compose into a new yml file on your Swarm node1

Change version in yml file to 3.1

Change image to official drupal

Remove build

Postgres:

…

Environment:

* POSTGRES\_PASSWORD\_FILE=/run/secrets/psql-pw

Secrets:

* Psql-pw

..

Secrets:

Psql-pw:

External: true

Copy compose file to machine.

echo “abcDefFG” | docker secret create psql-pw - // create the secret

Now run docker stack deploy

docker stack deploy -c docker-compose.yml drupal

docker stack ps drupal

Open url in browser. Password from secret should be accepted when setting up the database in drupal.

Using secrets with Local Docker Compose

Can’t run docker node ls. No access to swarm database or secrets stored in it. How to deal locally? To work in test and dev?

docker-compose exec psql cat /run/secrets/psql\_user

docker bind mounts file at runtime into the container. NOT SECURE. Allows us to develop with the same variable secret information in production locally. Great to develop using same launch scripts like we would in our swarm container. We want to mash as much of production locally. Only works with file based secrets. Doesn’t work with “external”.

**Section 9: Full App Lifecycle with Compose**

Live the dream!

Single set of compose files for:

Local docker-compose up dev environment

Remove docker-compose up CI environment (integration testing)

Remote docker stack deploy production environment

\*override – when we have a standard docker compose file and it sets the defaults that are the same across all of my environments. Then override file with “docker-compose-override.yml”, it will automatically bring this is when I do docker-compose up.

Must use file-based secret for local development.

Service Updates

Up to 77 commands for the service update available. Includes rollback and healthcheck options. Also has scale & rollback subcommand for quicker access.

Swarm update examples

Just update the image used to a new version

docker service update - -image myapp:1.2.1 <servicename>

Adding an environment variable and remove a port

docker service update - -env-add NODE\_ENV=production - -publish-rm 8080

Change number of replicas of two services

docker service scale web=8 api=6

Swarm updates in stack files

Same command. Just edit the YAML file, then

Docker stack deploy -c file.yml <stackname>

More examples

docker service create -p 8088:80 - -name web nginx:1.13.7

docker service ls // confirm service is running

docker service scale web=5

docker service update - -image nginx:1.13.6 web // remove then create new service with updated image

docker service update - -publish-rm 8088 - -publish-add 9090:80 // change publish port on each of the 5 services.

docker service update – -force web // completely replaces the tasks, looks for nodes less used. Trick to uneven amount of work in nodes (rebalancing)

Healthchecks in Dockerfiles

Added in 1.12

For production, engage in test commands for Healthcheck

Supported in docker file, yaml, docker run, and swarm services

It expects exist 0 (ok) or exit 1 (error)

Three states: starting, healthy, unhealthy

Not an external monitoring replacement

Healthcheck status shows in **docker container ls**

Check last 5 healthchecks with **docker container inspect**

Docker run does nothing with healthchecks

Services will replace tasks if they fail healthcheck

Service updates wait for them before continuing

docker container run - -name p1 -d postgres

docker container ls // nothing indicating health check

docker container run - -name p2 -d - -health-cmd=”pg\_isready -U postgres || exit 1” postgres // -U is indicating user is “postgres”. No pwd is needed.

docker container inspect p2 // there is now a health option

Same but now in service

docker service create - -name p1 postgres // goes straight to running state because there is nothing else left for it to do.

docker service create - -name p2 - -health-cmd=”pg\_isready -U postgres || exit 1” postgres // stays in a starting state until first healthcheck runs. Only considers in a running state if the healthcheck passes. After 30s it will shift to the running state. Extra bonus of health concept whenever we can.

More about Swarm production, check out **Swarm Mastery** in Udemy

**Section 10: Container Registries**

An image registry needs to be part of your container plan

Docker Hub most popular public image registry

It’s really registry plus lightweight image building

Auto-build images on commit

1 free private repository on docker hub

Webhooks to trigger builds in CI (or some other process)

If you have github/bitbucket to store code, don’t create repository or push in hub. Instead, click “create automated build” allows Hub to create CI path to build image based on commit. Like a reverse webhook

“automated build” is an indication of quality of the image. Always on the latest version

Create repository links in case your image uses FROM another image so that if the source image changes, your image is automatically built

Image Registry

A private image registry for your network

De facto in private container registries

Not full featured as hub. No web ui, basic auth only

At its core, a web api and storage system written in Go

Secure your registry with TLS

Storage cleanup via garbage collection

Run a private docker registry

Register on port 5000

Re-tag existing image and push it to new registry

Prefers TLS. Secure by default. Docker won’t talk to registry without HTTPS except localhost.

For remote self-signed TLS, enable “insecure-registry” in engine (not recommended)

docker container run – d – p 5000:5000 - -name registry registry // run the registry image

Retag existing image and push it to the new registry:

**docker tag hello-world 127.0.0.1:5000/hello-world**

**docker push 127.0.0.1:5000/hello-world**

**docker container rm admiring\_stallman**

Remove that image from local cache and pull it from new registry

**docker image remove hello-world**

**docker image remove 127.0.0.1:5000/hello-world**

**docker pull 127.0.0.1:5000/hello-world**

Re-create registry using a bind mount and see how it stores data

**docker image remove hell-world**

**docker container kill registry**

**docker container rm registry**

**docker container run –d –p 5000:5000 –name registry –v %(pwd)/registry-data/… registry // registry on a volume**

**docker push // push to registry**

**ll registry-data**

**tree registry-data // show dirs. And files visually**

<https://training.play-with-docker.com/linux-registry-part2/>

Private Docker Registry with Swarm

Works the same as localhost

Because of routing mesh, all nodes can see 127.0.0.1:5000

Remember to decide how to store images

ProTip: use a hosted SaaS registry if possible

I've mentioned Docker Hub, Docker Enterprise Edition DTR (Docker Trusted Registry), and Docker Registry as three options for storing your images, but there are many 3rd party options out there.

Quay.io is a popular choice, and is very comparable to Docker Hub as a cloud-based image registry. Sysdig did a Docker Usage Report in April 2017 based off their users that shows Quay as the most popular cloud-based choice.

If you're on AWS, Azure, or Google Cloud, they all have their own registry options that are well integrated with their toolset.

If you want a self-hosted option, there's Docker EE, Quay Enterprise, and also GitLab, which comes with GitLab Container Registry, among others.

**Section 11: Docker in Production**

Don’t need in day 1:

* Fully automatic CI/CD
* Dynamic Performance Scaling
* Containerizing all or nothing
* Starting with persistent data (don’t start by including your database in the container). Start with application code
* Microservice conversion isn’t required

Focus first on Dockerfiles

More important than fancy orchestration

FROM official distros

Make it start

Log all things to stdout/stderr

Make it work for others

Make it lean

Make it scale

Don’t sweat on the image size. Only one image for all containers. Focus on quality of docker file.

DockerFile Anti-Patterns

Storing unique data in container. Define VOLUME for each location.

Using LATEST. Image builds from LATEST. Use specific FROM tags.

Image builds installs latest packages. Specify version for critical packages.

You don’t deploy random code. So don’t deploy random versions of code dependencies. You could end up with random image versions of dependencies. You start getting random quirks.

Leaving default config. Not changing app defaults or blindly copying VM config. Update configs via ENV, RUN, and ENTRYPOINT.

Environment specific. Copy in environment config at image build. Instead, use single docker file with default ENV and overwrite per environment with ENTRYPOINT script. Do not have different images per environment. Try to get as many settings in the DockerFile as defaults

The 3 Big Decisions

Containers on VM or on bare metal? Stick with what you know at first.

OS Linux distribution/kernel matters. Minimum version != best version.

Which FROM image should you use? Don’t make decision based on image size. At first match your existing deployment process

Good defaults: Swarm architecture

Baby swarm (1 node)

Docker swarm init (done!)

Gives you more features than docker run

HA swarm: 3 – node. Use odd-number swarms. Do not use even numbers. Minimum 3 for HA. All managers. One node can fail. Use when very small budget. Pet projects or Test/CI.

Anything beyond 5 nodes, stick with 5 managers and rest workers

Assume nodes will be replaced

Assume containers will be recreated

SaaS/commercial market is mature

Outsourcing opportunities: image registry, logging, monitoring, alerting

At first: one container per VM. Improve Docker management skills. Run on Docker files rather than on Puppet. Simplify your VM OS build.

Trim optional requirements at first

Focus on Dockerfile/docker compose

Watch out for anti patterns

Stick with familiar OS and FROM images

Grow swarm as you grow  
Outsource plumbing