Service management using Docker

A brief history lesson

- Before virtualization took over, containers were the goto tool for service encapsulation
    - Solaris had Zones
    - FreeBSD and others had jails
    - Linux had LVS
- The motivation was often security, multi-tenancy and resource consolidation

Container drawbacks

- Complicated to set up
- Limited to the same OS / distribution
- No external API's or surrounding toolchains, local use was assumed
- The paradigm was still: A few big servers that do many, many things

Then virtualization took over

- With the advent of multi-core CPU's, cheap memory and hardware virtualization support, virtual environments became the norm
- More convenient than containers
  - OS - independent
  - Complete OS stacks
  - Automation features and API's that ultimately led to cloud computing
- A new paradigm: one VM per task

Cloud Computing and development projects

- Most VMs used in clouds for production are based on robust, long-term releases, such as Ubuntu 12/14.04 or RHEL

- Developers, on the other hand, run Fedora, Linux Mint etc. on their laptops

- It can be challenging to reproduce the same environment in production

    - What works for the developer may not work on a production server

The shipping problem

- Continuous integration and DevOps requires code to move quickly and easily between environments

- The code itself is in Git, but what about everything else that is needed?
    - Libraries at specific versions
    - Config files
    - Other packages

- The traditional approach would be configuration management, but that is not installed on a developers laptop...

The developers approach:

Based on a simple idea:

- The shipping industry had a similar challenge: people tried to ship all kinds of items of various sizes

- Harbor equipment and road/rail transportation was inadequate to adapt to the massive increase in shipping demand

- Solution: Standarize!

Docker

- A framework for running, managing and sharing of Linux containers
- Built on modern Linux kernel technologies such as cgroups
- A public service Docker Hub provides similar sharing as github
    - Private hub servers are also possible

The management interface containers were missing?

- Docker has solved the lack of high-level management that was needed to make containers usable for non-experts

- Combined with more secure technologies, it is now a viable alternative to spawning individual VMs for each task

Docker concepts

- A docker instance is booted from an image
    - The intention is to run only a single command inside the instance
- All changes are stored in an ephemeral filesystem
- Instances are on a local network with port forwarding from the host
- Instances can have access to local folders on the host (called volumes)
- You can commit, push and pull images just like with git

Docker use-cases

- Rapid testing and sandboxing
    - New instances spawn almost instantaneously
- A way to "ship" code all the way from developers until production
- Lean service encapsulation
- Providing efficient multi-tenancy

Docker basics

Docker support

- Docker is available for the many platforms
  - Ubuntu, Debian
  - RedHat, CentOS, Fedora
  - With boot2docker:
    - Mac OS X
    - Windows

Installing Docker on Ubuntu 14.04

- Docker advocates using either RHEL or Ubuntu for their packages
    - Docker is called docker.io on Ubuntu, but lxc-docker is the name of the docker-maintained version

- The simplest way to get the latest

```
curl -sSL https://get.docker.com/ubuntu/ | sudo sh
```

Your first instance

- Check that docker is installed and running:
  `docker ps`

- Launch your first instance:
  `docker run -i -t ubuntu /bin/bash`

- Play around

- In a separate shell, check status once again while the instance is still up:
  `docker ps`

Things to check out inside the instance

- Do you see an interface?
  try `ifconfig -a` or `ip a`

- Any running processes?
  try `ps aux`

- Is the instance online?
  try `apt-get update && apt-get install vim`

What just happened?

- The command downloaded the latest ubuntu image from the Docker Hub
- A new instance was started with an ephemeral filesystem
- The options "-i" and "-t" created an interactive session as well as a pseudo-terminal respectively
- When the shell was closed, the instance finished and 'disappeared'

Giving the instance a name

- Docker will assign every instance an ID and a random name
- Using the --name option, you can provide a name for the instance

```
docker run --name mycontainer -i -t ubuntu /bin/bash
```

- Instance names need to be unique

Starting a stopped docker instance

- The command docker ps only shows running instances
- Use docker ps -a to see all instances
- In order to start a stopped instance, run:

  `docker start <name|id>`

- A started instance will re-run the same command as specified previously
  - You may have to press enter once or twice to get the prompt back
  - Note: not all docker images start with a shell

Daemonizing a container

- If you wish to run a container in the background, add the -d option to the run command

- You can check the output inside a container using the command
`docker logs <id|name>`

- Adding -f will give continuous output, just like tail -f

- Adding -t will add timestamps to the output for better debugging

Inspecting an instance

- The command docker top <id|name> will provide a list of processes running inside the instance

- You can also run a task inside a container:
  `docker exec <id|name> command`

- You can either get the output directly by adding -t -i or run the command in the background with -d

- This is a common way to do maintenance and monitoring on a instance

- You can also run:
  `docker inspect <name|id>`
  `docker inspect -f '{{.NetworkSettings.IPAddress}}' <name|id>`

Stopping/deleting an instance

- The graceful way to shut down an instance is `docker stop <id|name>`

- In order to take it down more dramatically, run: `docker kill <id|name>`

- To delete an instance: `docker rm <id|name>`

Automatically restarting instances

- You can specify that an instance should automatically restart if it fails

- Add the --restart=always option to the run command

- Alternatively, --restart=on-failure:3, which will restart it 5 times

Docker basic networking

Networking concepts

- All docker instances are attached to a local, private 172 network
- Default behavior is that no port forwarding is enabled for an instance
- You can specify a port to be exposed when running the instance:
  - -p 80 (expose port 80 on the instance. Will use a host port between 49000 and 49900)
  - -p 80:80 (expose port 80 on the instance. Will use local port 80)

Example

- Start a docker and expose port 80

```
docker run --name=webserver -t -i -p 80 ubuntu:14.04 /bin/bash
```

- Next we inspect with docker ps

```
root@dockerhost:~# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                   NAMES

3ce81f679c54        ubuntu:14.04        "/bin/bash"         6 seconds ago       Up 4 seconds        0.0.0.0:49154->80/tcp   webserver
```

Docker images

Images

- Images are constructed from filesystem layers on top of a base filesystem (like ubuntu:14.04)
- You can create your own images using two mechanisms
  - commits
  - Docker files (the recommended method)

Image naming conventsions

- Images are usually associated with a docker hub user, a name and a tag
  - Default tag is 'latest'
- For instance:
  - shykes/couchdb
  - ubuntu:12.04
  - fedora:20

Creating an image with commits

- Start an instance from an existing image:
  docker run -t -i ubuntu:14.04 /bin/bash

- Install what you like:
  apt-get update; apt-get install apache2

- You can now commit the image:
  docker commit -m="A webserver image" <id|name>
  webserver

- Check the existence:
  docker images

Using a docker file

- Docker files are descriptive files that work as recipes for building images

- The convention is that the file is called 'Dockerfile'

- Typically, one creates a folder for the build containing a docker file and all other files you wish to copy into the image

- You then launch the docker build command on the folder

Build process

- Create a folder:
  mkdir static_webserver; cd static_webserver

- Create the docker file Dockerfile:
  # version 0.1
  FROM ubuntu:14.04
  MAINTAINER Kyrre Begnum "kyrre.begnum@hioa.no"
  RUN apt-get update
  RUN apt-get -y install apache2
  RUN echo "Hello world" > /var/www/html/index.html
  EXPOSE 80

- Build the image
  docker build -t "staticwebserver:v1" .

- Launch the instance
  docker run -P -d staticwebserver /usr/sbin/apache2ctl -D FOREGROUND

- Test: run docker ps and find the local port, then do wget -O - -q http://ip:port

The build process in detail

- The build process works in a more layered way than with docker commit
- For every "RUN" command, a new layer is created
- The finished image, therefore, is much smaller than the base image
- Docker will even keep the intermediate images in a cache
- In case one command fails, the process stops and you can run an instance from the intermediate build point and investigate
- If you build the image one more time, the build process is instant

Docker knows about git

- One exciting feature is that docker can build images based on folders in a git repository

- For example:
  docker build -t "apachephp:v1" https://git.cs.hioa.no/kyrre/apache_php_docker.git

Other build features

- CMD - Specify the command to run (can be overridden on the command line)
- WORKDIR - Set the working directory for RUN commands
- ENV - Set environment variables during build and also when creating the instance
- USER - The user to execute the CMD
- VOLUME - Attach a folder to the instance
- ADD - Copy a file into the image (can also be an URL, AND can automatically unpack archives (tar, zip, gzip))
- COPY - Like ADD but with fewer features
- ONBUILD - Triggers that are to be executed during build
- ENTRYPOINT - Overrides the default shell /bin/sh with another binary

A better webserver example

FROM ubuntu:14.04
MAINTAINER Kyrre Begnum "kyrre.begnum@hioa.no"
RUN apt-get update
RUN apt-get install -y apache2
RUN apt-get -y install libapache2-mod-php5 php5-mysql
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
ONBUILD ADD . /var/www/html/
EXPOSE 80
ENTRYPOINT ["/usr/sbin/apache2"]
CMD ["-D","FOREGROUND"]

What if you need to run several processes inside a container?

- Docker does mainly intend to run only a single process per instance
- Services are disabled from starting
- Third-party tools can be used to rather start a suite of processes, like supervisor: http://supervisord.org

Using supervisor to run apache2 and SSH (The Dockerfile)

- These lines install apache2, ssh server and supervisor

```
RUN apt-get update && apt-get install -y openssh-server apache2 supervisor
RUN mkdir -p /var/lock/apache2 /var/run/apache2 /var/run/sshd /var/log/
supervisor
```

- Also, copy a supervisor.conf file into the image:

```
COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf
```

- Expose both ports:

```
EXPOSE 22 80
```

- Run supervisor as the default command

```
CMD ["/usr/bin/supervisord"]
```

supervisord.conf

```
[supervisord]
nodaemon=true

[program:sshd]
command=/usr/sbin/sshd -D

[program:apache2]
command=/bin/bash -c "source /etc/apache2/envvars && exec /usr/sbin/apache2 -DFOREGROUND"
```

Data Volumes

Volume features

- Volumes are a shared folder between the host and one or more container instances

- You can add volumes with the -v option to the run command and the VOLUME directive in the Dockerfile

- The volumes can be shared between instances and be reused across images

Example

- Adding a simple volume
  docker run -i -t -v /opt/data ubuntu /bin/bash

- Notice, that you now have an extra mountpoint inside the instance with df -h

- In order to find the actual location of the folder, use the docker inspect id command

- They are usually located under:
  /var/lib/docker/vfs/dir/

Sharing a specific folder

- You can also specify what folder to share to a container
  docker run -t -i -v /opt/localtata:/opt/data ....

- This can also be done as read-only
  -v /opt/localdata:/opt/data:ro

- It also works for files:
  -v /opt/mydatafile:/opt/data

- This feature is not available for Dockerfiles, as they are supposed to be more portable

Volume containers

- One recommended design is to have an instance with one or more volume and then share that volume between other instances

- Create a volume container ('create' will not run it)
  `docker create -v /dbdata --name dbdata training/postgres`

- Create another container with the same volume:
  `docker run -d --volumes-from dbdata --name db1 training/postgres`

- You can have multiple instances share the same volume that way

Discussion

What have we learned?

- Docker is a simple framework for running, managing and building Linux containers
- Makes some tasks really easy
- Can significantly reduce your cloud VM footprint
- Docker is meant to be a piece of a workflow

Docker rocks the boat

- It's no secret, Docker challenges our existing paradigm of "One VM per task"
- Where do all our operations support tools go?
  - Configuration management
  - Monitoring
  - Backup
  - Logging

Is docker a new config system?

- A Dockerfile specifies many of the same things a puppet manifest would, is it a replacement?
- Can one be managed by the other?
  - Puppet apply a viable option again?
- The discussion is still ongoing...

Docker is still very single-host

- Hard to envision a distributed docker yet
- Docker Hub and private registries make distributing images easy, but what about coordination?
- Docker seems to be working on that: Docker swarms and Docker compose

Docker user interfaces

- Shipyard: https://github.com/ehazlett/shipyard
- DockerUI
- maDocker