Docker Office Hours

Docker and Docker Compose for Local Development and Small Deployments

Who Am I?

- James Quacinella, Codementor for about 6 months
- Undergrad in Computer Engineering, Graduate degree in Data Analytics
- Starting programming in C and C++, now sticking with Python
- Currently using Docker when doing consulting work and on personal projects

Updated Slides can be found on github

What We Will Cover

- We'll be going over a brief overview of Docker and how to use it for development
- First we'll go over a brief overview of Docker, containers and how they work
- Then I'll introduce you to how I approach doing development work with Docker built-in from the beginning, including custom Dockerfiles and Docker Compose

Docker Intro

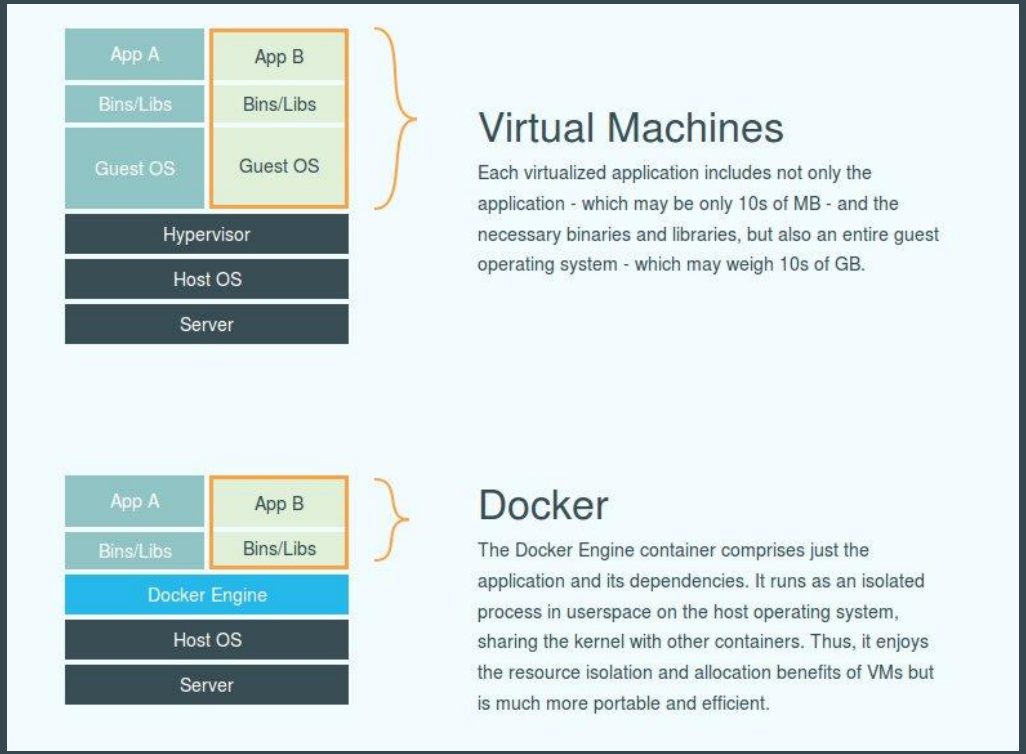
What is Docker? (1 / 2)

- From the docker website: “Docker provides a way to run applications *(think: process(es))* securely isolated in a container, packaged with all its dependencies and libraries.”

- “Because your application can always be run with the environment it expects right in the build image, testing and deployment is simpler than ever, as your build will be fully portable and ready to run as designed in any environment.”
    - This is why I force Docker into my workflow: my dev environment will be exactly like production

- Docker provides an environment for creating, running, stopping containers, where we want each container to be a “building block” of software

What is Docker? (2 / 2)



- Containers are lightweight and run without the extra load of a hypervisor
- The big difference is that virtual machines are individual Operating Systems running in parallel
- In the docker world, each container is just a **process** and its dependencies grouped together
- In some sense, this is virtualization at the **process level**, not the OS level

From: http://siliconangle.com/blog/2014/08/22/deadly-docker-why-containers-are-a-threat-to-cloud-virtualization/

What is a Container Exactly? (1 / 2)

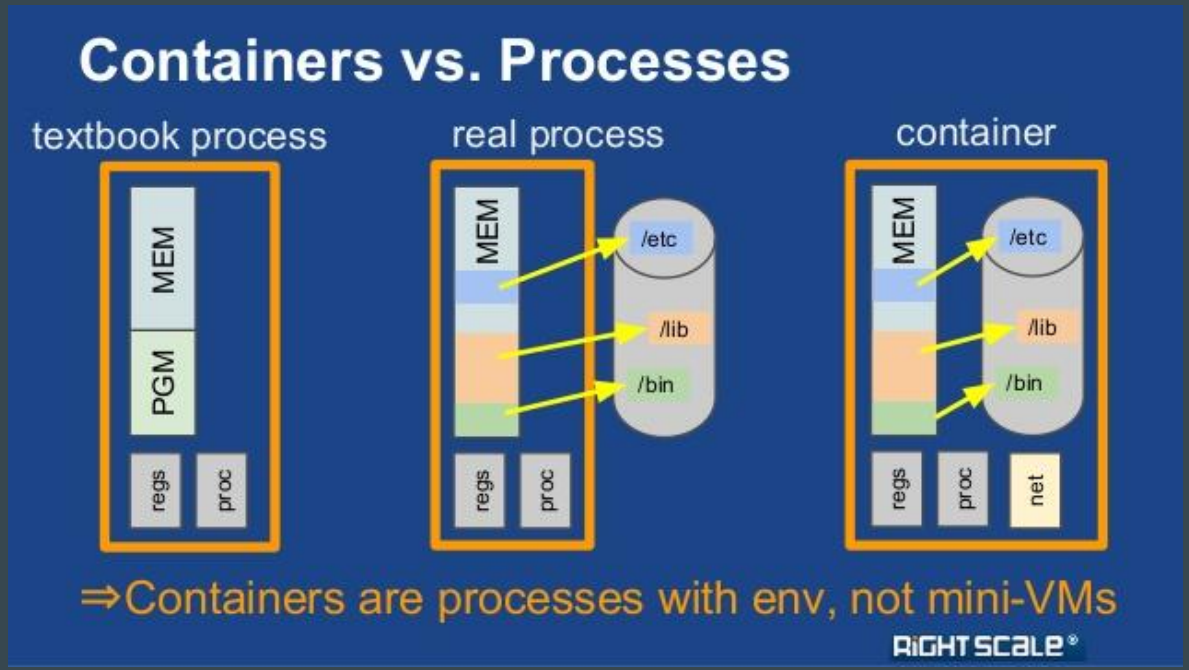
- From wikipedia:

  “Docker containers wrap up a piece of software in a complete filesystem that contains everything it needs to run: code, runtime, system tools, system libraries – anything you can install on a server. *This guarantees that it will always run the same, regardless of the environment it is running in*”

- This allows us to treat a full process and its dependencies as a single object

- We can treat containers like building blocks for software.
  - Containers get their name from shipping containers, which standardized how to ship goods long distances

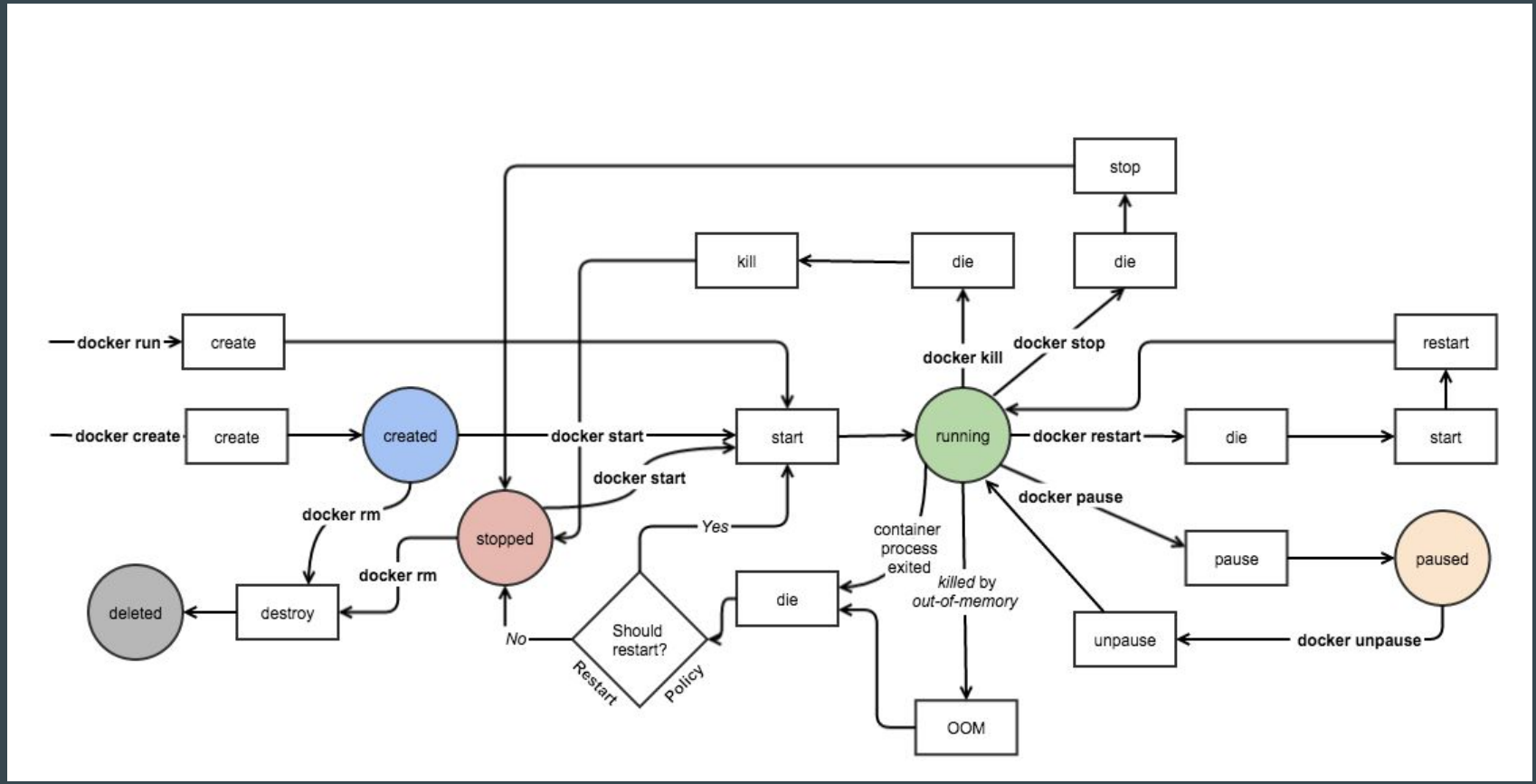
What is a Container Exactly? (2 / 2)



- A running container is simply a standard OS process plus
  - File system dependencies
  - Network connections
- **Advanced:** can place limits on resources per container

From: http://www.slideshare.net/rightscale/docker-meetup-40826948

Docker Container State Machine



From: https://docs.docker.com/engine/reference/api/docker_remote_api/

Why is this useful to developers?

- Allows you to break large applications into its constituent parts, which helps keeps cognitive load low and allows for easier debugging and more transparency

- Wrap up complex application instructions in a way that helps document how to build and configure your software

- If you use Docker for development as well as production, then you have identical stacks.
    - This reduces the cognitive load of knowing how to use multiple setups
    - Prevents the “it works for me” excuses for when things break

- Do not have to worry about crazy compilation steps to install software and maintain on your machine. This also helps keep your development work to be independent of your OS (so an ubuntu upgrade doesn’t start touching my development libraries, etc)

- And more ...

Docker Vocabulary

- **Container:** A run-able process and its resource dependencies grouped together into a single unit

- **Image:** A blueprint for creating containers. Much like class versus object, or VM image versus a VM
    - Example: the Redis Docker Image allows you to create one or more actual Redis containers that can / will run the Redis process

- **Dockerfile:** A text file containing instructions for building a docker image

- **Registry:** An external service for storing / referencing images that you can name and version. Dockerhub is a registry provided to you by Docker, but you can setup a private one if need be

- **Volume:** Like a traditional Linux ‘mount’, a volume is a folder on the docker host system that is mapped into a running container

Docker Ecosystem

- **Docker Engine:** This is the core of Docker. This daemon presents a REST API and is responsible for actually running containers

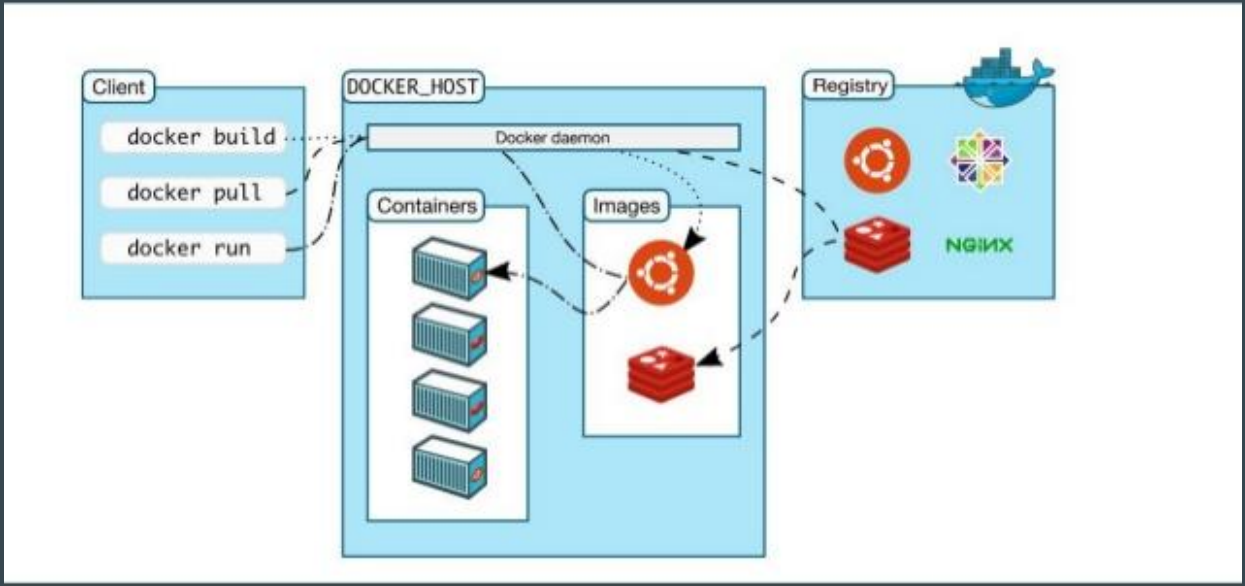
- **Docker Machine:** Provision a virtual machine, on many different cloud providers, to run a Docker Engine. Essentially provisions engines.

- **Docker Swarm:** Docker's distributed container platform to cluster many Docker Machine instances together

- **Docker Compose:** Coordinate containers by describing how containers fit together to provide a whole application

Docker Architecture

Docker Components



- The docker command line client sends request to a docker engine running on a docker machine
- This daemon (i.e. docker engine) is in control of starting and stopping containers

From: http://www.slideshare.net/then4way/introduction-to-docker-and-docker-compose

Tools We'll Be Using

- We'll be using docker engine, machine and compose
- We'll also be using VirtualBox for setting up a local docker engine, and DigitalOcean for a cloud docker engine
- We will not be using Swarm, as that will be in a future presentation

Installation

- Docker Installation Notes
- For docker machine on linux:

```
 $ curl -L https://github.com/docker/machine/releases/download/v0.8.2/docker-machine-`uname
-s`-`uname -m` > /usr/local/bin/docker-machine && chmod +x /usr/local/bin/docker-machine
```

- For docker compose on linux:

```
$ curl -L "https://github.com/docker/compose/releases/download/1.9.0/docker-compose-$(uname
-s)-$(uname -m)" -o /usr/local/bin/docker-compose && chmod +x /usr/local/bin/docker-compose
```

- Virtualbox Installation Notes

Docker For Development
Example: Python + Flask App

Local Development

- Here we will cover how to think about how to use Docker for a local development environment

- Let's say we have been hired to build a small REST server
  - I have chosen Python and Flask as my tools but you could use other stacks as well
  - This talk is opinionated in what tools I choose to use, but should be generally applicable to other languages

- We'll call the application **Flask-Tester**

- To see finished repositories, please check:
  - `git clone https://github.com/jquacinella/flask-tester-backend`
  - `git clone https://github.com/jquacinella/flask-tester-docker`

Initial Steps: Repositories

- I like to keep multiple repositories when building applications:
  - One for the application code itself, in this case Python and Flask code
  - One for the docker orchestration of the application

- Commands:

```
mkdir code; cd code
git clone https://github.com/jquacinella/flask-tester-backend
cd ../; git clone https://github.com/jquacinella/flask-tester-docker
```

- This creates two folders, one for each repo, which will start building now

Overview of Flask App

- This is not a talk about Python or Flask, but how to think about deploying applications locally and remotely using docker

- Here is my wsgi.py in github for this application. This is the initial version without redis. The version on master has the updated code with redis

- **Tip #1:** Make your application respond to env variables for configuration loading. We will not do this here but will in future

- **Tip #2:** Make sure your setup will reload the application on source changes, to allow live coding from your local machine

Overall Workflow

1. Create a docker machine to host your docker engine
    - **NOTE:** since I am on Linux I could use a docker engine installed locally on my laptop. However, AFAIK, using docker on Windows or MacOSX requires a VM, so I am doing it this way to be as general as possible
2. Create an image for a container that can host and run our python application
3. Outline volumes needed for source code and config files
4. Write a compose file to allow us to automate the creation and deletion of the application container w/ proper volumes
5. Write some scripts to wrap docker compose

Workflow Step 1: Docker Machine

- Create a virtualbox VM called flask-tester-engine:

```
$ docker-machine create --driver virtualbox flask-tester-engine
```

- See what machines you have available:

```
$ docker-machine ls
```

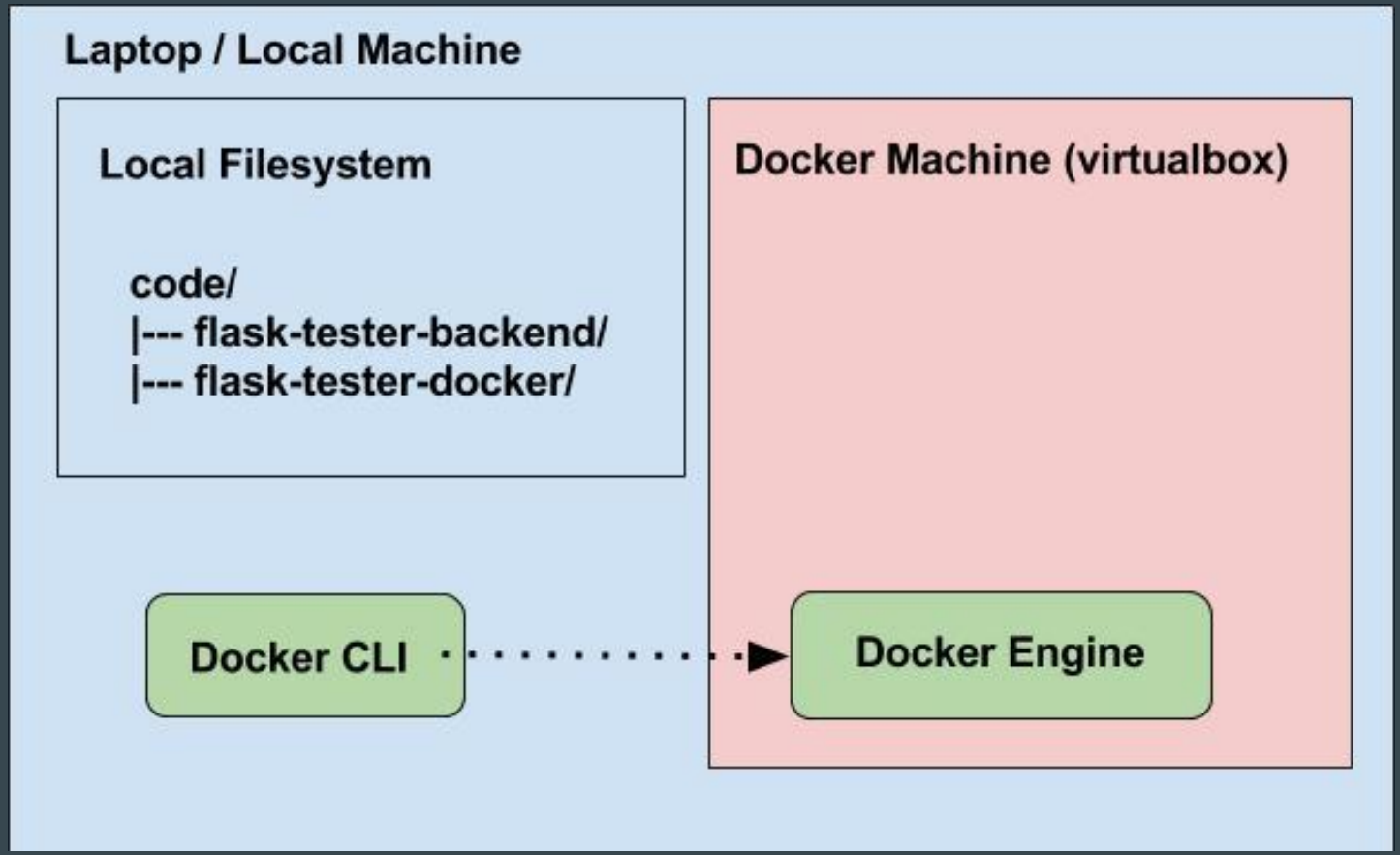
- Tell docker client to use this machine as your docker engine:

```
$ docker-machine env flask-tester-engine
$ eval $(docker-machine env flask-tester-engine)
```

- Run a docker command to see all containers, which you should see none right:

```
$ docker ps --all
```

Workflow Step 1: Docker Machine w/o Shared Volume



Workflow Step 1: Docker Machine Shared Volume

- Docker machine sets up the Virtualbox with a mount from your local /home directory to the VM
  - I do not like this automatic setup and prefer updating the mount manually
  - See for more https://github.com/docker/machine/issues/1814 for more details

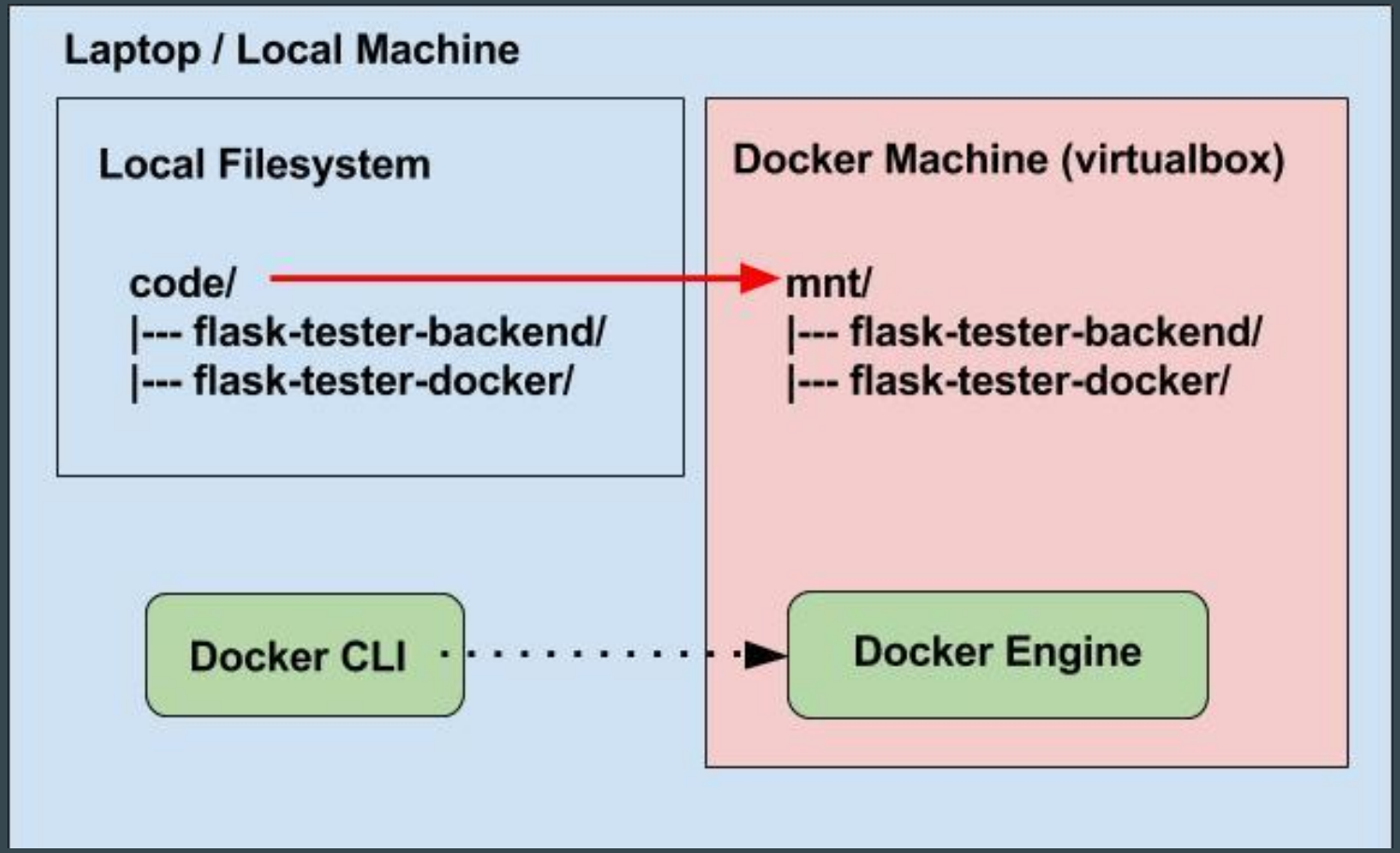
- `$ VBoxManage sharedfolder add flask-tester-engine --automount --name "code" --hostpath /home/james/Code/Tutoring/CodementorOfficeHours/code`

- `$ docker-machine ssh flask-tester-engine 'sudo mkdir --parents /mnt/code'`
- `$ docker-machine ssh flask-tester-engine 'sudo mount -t vboxsf code /mnt/code'`

- `$ docker-machine ssh flask-tester-engine 'ls -l /mnt/code'`

```
drwxrwxr-x 1 root     root       4096 Nov 29 19:53 flask-tester-backend
drwxrwxr-x 1 root     root       4096 Nov 29 19:53 flask-tester-docker
```

Workflow Step 1: Docker Machine w/ Shared Volume



Workflow Step 2: Create an Image

- “What are the dependencies?” is a good question to ask when thinking about what need to be in a container.
  - There are always two things to think about: the process the container is wrapping and what files that process needs
  - The docker philosophy is to have one process per container, but there is much discussion about this. Check https://github.com/phusion/baseimage-docker for an alternative view
- What will be our central process?
  - Our application is going to be uwsgi process
  - This process will start up threads running our Python Flask app
- What file dependencies do we have?
  - We need the Python source files
  - Files related to uwsgi, including a binary and conf files
  - Access to open port 80 for client connections

Workflow Step 2: Create an Image (1 / 3)

- I like to base my images on phusion-baseimage. It's a little controversial, but I think it works well for beginners.

- However, to minimize the things we go over, we'll use the standard Ubuntu image
  - Will introduce phusion-baseimage in the next session

- Given this base image, what do we need to do to get it running our application:
  - We need to install uwsgi in the container filesystem
  - We need to configure the container to run uwsgi as its main process
  - We need to get our source code in the container

- First, we'll write a Dockerfile to install the software we need

Workflow Step 2: Create an Image (2 / 3)

```
# Inspired by https://github.com/atupal/dockerfile.flask-uwsgi-nginx/blob/master/Dockerfile
# Base our image off the ubuntu docker container from Dockerhub
from ubuntu:14.04

# Update ubuntu filesystem with updates
run apt-get update

# Install python and easy_install from Ubuntu repos
run apt-get install -y build-essential python python-dev python-setuptools

# Install pip, and then use pip to install flask and uwsgi
run easy_install pip
run pip install uwsgi flask

# We are exposing a network port
expose 80

# uWSGI command that this container is running
cmd ["/usr/local/bin/uwsgi", "--http", ":80", "--wsgi-file", "/src/wsgi.py", "--py-autoreload", "1"]
```

Source: https://github.com/jquacinella/flask-tester-docker/blob/master/images/flask_tester/Dockerfile

Workflow Step 2: Create an Image (3 / 3)

```
# In flask-tester-docker repo:
$ mkdir -p images/flask_tester

$ vim images/flask_tester/Dockerfile   # content from before

$ docker build -t flask_tester:0.1 images/flask_tester    # Build an image from the dockerfile
... a lot of output ...

$ docker images    # We can see our image and the ubuntu image it is based on
REPOSITORY      TAG             IMAGE ID        CREATED          SIZE
flask_tester    0.1             5cb4ef863e80    2 minutes ago    439.7 MB
ubuntu          14.04           aae2b63c4946    About an hour ago   188 MB
```

- Notice each image is identified by a name and a tag; the tag is useful for keeping track of versions, so when we update the Dockerfile later in the talk, we'll update the tag from 0.1 to 0.2

Workflow Step 3: Volumes for Code and Config

- What else does our container need? Config and Data / Source Code
- Volume for source code
  - we'll use the virtual machine mount to tell the container to map the directory *in the VM* to a /src folder *in the container*
- Volume for config
  - Normally I would use a uwsgi config file and mount it via a volume, but in our case to keep things simple, I am running uwsgi only with CLI arguments
  - This allows us to update configs and not have to rebuild the image; makes debugging much easier and allows config files to be in git

Interlude: Run The Application w/o Compose

- With everything we have, we should be able to test our app using docker run, which is the command to create and start a container process:

```
$ docker run -d --name flask_tester_container -p 80:80 -v /mnt/code/flask-tester-backend:/src
flask_tester:0.1

$ docker ps --all
CONTAINER ID    IMAGE           COMMAND         CREATED         STATUS          PORTS                   NAMES
0fc5c1909584    flask_tester:0.1 "supervisord -n" 7 seconds ago   Up 7 seconds    0.0.0.0:80->80/tcp   flask_tester_container
```

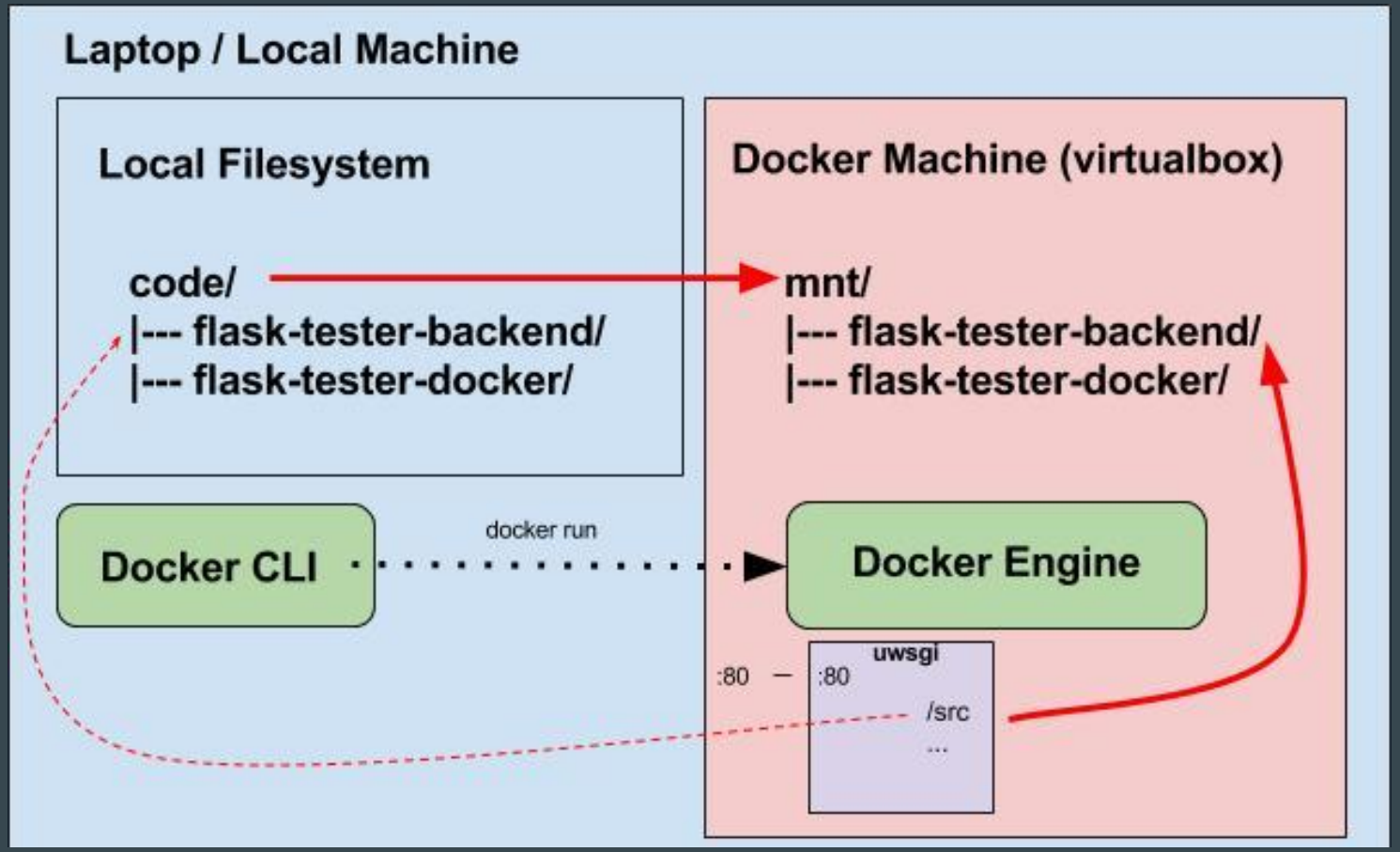
- Remember, everything is in reference to the virtual machine, so the volume mount is in reference to /mnt, which is mapped to your top-level directory on your machine

Interlude: Examine the Container

- You can always ‘jump into’ the container use the docker exec command:

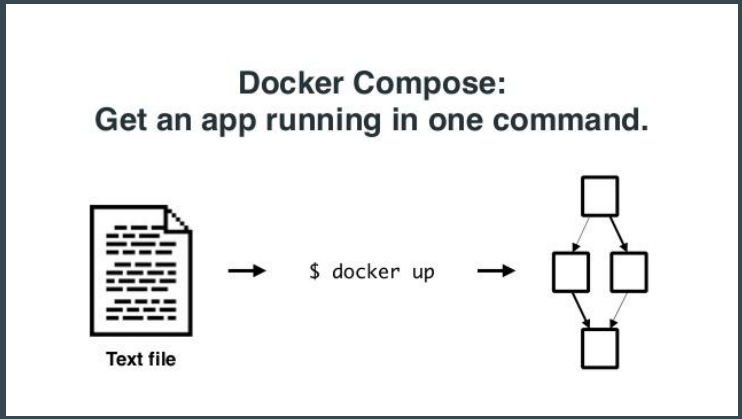
```
$ docker exec -it flask_tester /bin/bash
root@flask_tester:/# ps auxww
USER       PID %CPU %MEM    VSZ   RSS TTY    STAT START   TIME COMMAND
root       1  0.1  1.9  68364 20256 ?       Ss   19:36   0:00 /usr/local/bin/uwsgi --http :80
--wsgi-file /src/wsgi.py --py-autoreload 1
root       7  0.3  1.8 146580 18636 ?       Sl   19:36   0:00 /usr/local/bin/uwsgi --http :80
--wsgi-file /src/wsgi.py --py-autoreload 1
root       8  0.0  2.2  76560 22912 ?       S    19:36   0:00 /usr/local/bin/uwsgi --http :80
--wsgi-file /src/wsgi.py --py-autoreload 1
root       23  0.6  0.3  18140  3172 ?      Ss   19:39   0:00 /bin/bash
root       38  0.0  0.2  15568  2092 ?      R+   19:39   0:00 ps auxww
root@flask_tester:/# exit
exit
$
```

Interlude: Volume and Container Diagram



Workflow Step 4: Docker Compose (1 / 4)

- Automate the bringing up and down of the application container, so we don't have to remember the run command and all its options
- We'll use v2 of the compose syntax and we'll write a file to construct the container just like we did with docker run:
- Compose file located on github



From: http://www.slideshare.net/Docker/compose-breakout-aanand-prasad

Workflow Step 4: Docker Compose (2 / 4)

```
version: '2'
services:
    flask_tester_app:
        container_name: flask_tester
        hostname: flask_tester
        image: flask_tester:0.1
        volumes:
             - /mnt/code/flask-tester-backend:/src
        ports:
             - "80:80"
```

Workflow Step 4: Docker Compose (3 / 4)

```
$ /usr/local/bin/docker-compose -f compose/flask_tester.yml -p local up -d
Creating network "local_default" with the default driver
Creating flask_tester

$ /usr/local/bin/docker-compose -f compose/flask_tester.yml -p local ps
      Name                    Command                State       Ports
-------------------------------------------------------------------------
flask_tester   /usr/local/bin/uwsgi --htt ...   Up    0.0.0.0:80->80/tcp

$ docker ps --all
CONTAINER ID      IMAGE             COMMAND                  CREATED           STATUS            PORTS              NAM
9098751a9976      flask_tester:0.1  "/usr/local/bin/uwsgi"   7 seconds ago     Up 7 seconds      0.0.0.0:80->80/tcp
flask_tester

$ /usr/local/bin/docker-compose -f compose/flask_tester.yml -p local down
Stopping flask_tester ... done
Removing flask_tester ... done
Removing network local_default
```

Workflow Step 4: Docker Compose (4 / 4)

- One thing we are ‘skipping’ here is that compose sets up a ‘network’ for all your containers that are in the same compose file
- This means the two containers can communicate over the network using the local DNS served up by the Docker Engine (using either the container name or service name)

```
$ docker exec -it flask_tester /bin/bash
root@flask_tester:/# ping redis
PING redis (172.18.0.2) 56(84) bytes of data.
64 bytes from flask_tester_redis.local_default (172.18.0.2): icmp_seq=1 ttl=64 time=0.142 ms
64 bytes from flask_tester_redis.local_default (172.18.0.2): icmp_seq=2 ttl=64 time=0.176 ms
^C
--- redis ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.142/0.159/0.176/0.017 ms
```

Workflow Step 5: Wrapper Scripts

- I like to wrap these commands in simple scripts that I keep in the scripts/ subfolder.
- Github link

Example: Adding Redis to the Mix

Workflow Update to Add Redis (1 / 2)

- Redis is a NoSQL database that can store various data-types. We'll use it in our simple Flask application as a way to show how you can build your app with docker components
- We'll have to:
  - Add the python 'redis' library to the flask-tester container Dockerfile
  - Create a new container to house the redis process and its dependencies
    - We'll use the redis container from Dockerhub
  - Update our compose file to link them together
    - You can reference the other container using its name using DNS provided by docker

Workflow Update to Add Redis (2 / 2)

```
$ ... update dockerfile for app ...

$ docker build -t flask_tester:0.2 images/flask_tester # Build image flask_tester:0.2

$ ... update compose file ...

$ ./scripts/stop_app.sh  # Stop all containers for this app

$ mkdir data/redis       # Create folder for persistent redis data

$ ./scripts/start_app.sh # Re-create all the containers

$ curl $(docker-machine ip flask-tester-engine)  # cURL access to our app

Counter: 27
```

Example: Scale to the Cloud

Create a VM on a Cloud Provider

- You can now use docker machine to create a VM with a docker engine on a cloud provider, like DigitalOcean and AWS

- Each provider is implemented by a driver, and each driver has custom options
  - To see options for each, check the documentation
  - Examples:

```
$ docker-machine create --driver digitalocean --digitalocean-access-token xxxxx
docker-sandbox

$ docker-machine create --driver amazonec2 --amazonec2-access-key AKI*******
--amazonec2-secret-key 8T93C*******  aws-sandbox
```

Create a VM on a Cloud Provider

- Let's create an example DO machine:

```
$ ./scripts/create_do_machine.sh
Running pre-create checks...
Creating machine...
(flask-tester-do) Creating SSH key...
(flask-tester-do) Creating Digital Ocean droplet...
(flask-tester-do) Waiting for IP address to be assigned to the Droplet...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with ubuntu(systemd)...
Installing Docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this virtual machine, run:
docker-machine env flask-tester-do
```

Create a VM on a Cloud Provider

```
$ eval $(./scripts/use_do_machine.sh)
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
```

- But we have a **big issue now**: the new machine has no volumes! How will it know how to mount the source code? Solution:
  - Update the Dockerfile to first copy the source code into the container; a volume will ‘overwrite’ this allowing us flexibility when developing
  - This creates an image with our source code baked in
  - We then push this image to my **private Docker Registry on AWS**
  - Create new compose file to use this new image and to NOT mount the source code
- TL;DR: Need an image with the source inside of it and not a volume; and this image needs to be in a registry that is accessible from our docker machines

Some Caveats

- We have some issues here, mostly that when building an image, Docker will not allow you to grab files from outside the context (or directory) where the Dockerfile is. This is for security reasons.
- To get around this, we will use a Linux feature to mount our source folder into the folder with the Dockerfile:
  - `sudo mount --bind <path/to/dir>/code/flask-tester-backend images/flask_tester/src`
  - For windows, you might have to symlink or not have two repos but have your Dockerfile closer to the source
  - For MacOS X, the mount command should work

Demo AWS Registry

More information here: http://docs.aws.amazon.com/AmazonECR/latest/userguide/what-is-ecr.html

Demo On Digital Ocean Machine

Next Time ...

Future?

- Understand Swarm clustering
- Setting up a swarm cluster using docker machine
- Service Discovery via ZooKeeper
- How to setup the same Flask + Redis application across it

- ... OR Anything you want me to cover regarding Docker

Questions?