Lab Outline

Online

This document

You can find this document online as

- a PDF file (https://raw.githubusercontent.com/mjbright/jupyter_notebooks/master/2016-Feb_Docker_Build_Lab/2016-Feb_Docker_Build_Lab.pdf) (http://bit.ly/1QF0XaH (http://bit.ly/1QF0XaH)) or as
- a Jupyter (http://www.jupyter.org) notebook at 2016-Feb_Docker_Build_Lab (https://github.com/mjbright/jupyter_notebooks/blob/master/2016-Feb_Docker_Build_Lab/).

This notebook is runnable in a Jupyter installation with the bash_kernel installed.

Although that is not the subject of this lab, if you want to create your own environment in which to run this lab with Docker components already installed (and even Jupyter/bash_kernel), refer to the README.md here (https://github.com/mjbright/jupyter_notebooks/blob/master/2016-Feb_Docker_Build_Lab/)

On Github: mjbright (https://github.com/mjbright)

,

On Twitter: @mjbright (http://twitter.com/mjbright)

,

Blog: mjbright.github.io (http://mjbright.github.io)
mjbrightfr AT gmail

Lab-Description

TOP

Start this lab by performing the **Lab Preparation** step.

- Lab Preparation
- Lab-p1
- Lab-p2
  - Creating a small binary with C
  - Creating a small binary with Go
  - Creating a toolset Docker image containing several executeables
  - Pushing our image to Docker Hub
  - Dockerfile best practices
  - Using a Language Stack (Node.js)
  - Using a Language Stack (Python)
  - Building complex systems with Compose
  - Rails example with Compose
  - Building Docker with Docker

References

TODO

TODO: add use of simple web server, handlebar.js-template, angular-template, go-web server, ... TODO: go web server as miniscule image ????

- **configuration to use Yann's environment**
- Removal of images
  - base: just large images present (to minimize network bandwidth)
    - alpine, ubuntu, language e.g. Python/Nodejs?), Golang, Raspi ??
- Creation of lab accounts (**hpe_tss_lab**) on DockerHub, Github and GMail
  - Attendees must take care to name their images/repos according to their pod number, e.g. **userN**

Overall description of the lab steps

NOTE: All lab steps can be considered optional, attendees may perform them in order, or jump to the section of interest to them (to get to the more complicated steps)

Lab Preparation

We first need to recuperate the source code examples:

In [ ]:
```
cd $HOME

rm -rf ~/src

git clone https://github.com/mjbright/docker-examples src
```

Lab-p1

A refresh on Docker concepts

Look at what docker version you are running. Note that the 'docker version' command reports the local client version as well as the server (docker engine) version.

In [8]:
```
docker version
```

```
Client:
 Version:      1.10.0-rc2
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   c1cdc6e
 Built:        Wed Jan 27 22:31:21 2016
 OS/Arch:      linux/amd64

Server:
 Version:      1.10.0-rc2
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   c1cdc6e
 Built:        Wed Jan 27 22:31:21 2016
 OS/Arch:      linux/amd64
```

Images are image layers

Remember that when we talk of a container image it is really a collection of image layers.

The docker info command provides information about the docker engine, see below.

In [9]:
```
docker info
```

```
Containers: 8
 Running: 0
 Paused: 0
 Stopped: 8
Images: 9
Server Version: 1.10.0-rc2
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 29
 Dirperm1 Supported: false
Execution Driver: native-0.2
Logging Driver: json-file
Plugins:
 Volume: local
 Network: null host bridge
Kernel Version: 3.13.0-55-generic
Operating System: Ubuntu 14.04.2 LTS
OSType: linux
Architecture: x86_64
CPUs: 1
Total Memory: 1.955 GiB
Name: vagrant-ubuntu-trusty-64
ID: 6LRX:E4PK:3EBE:MHJE:TJHR:NVS6:5POR:YS4C:WIWG:7F5J:Y6FU:4IZE
WARNING: No swap limit support
```

But if we look at the number of containers and images, the number of images it is not the same as provided above. Why do you think that is?

First let's list the number of running and number of stopped containers

NOTE: the value on your system will be different

In [10]:
```
docker ps | tail -n +2 | wc -l # Number of running containers ('tail -n +2'
```

```
0
```

In [11]:
```
docker ps -a | tail -n +2 | wc -l # Number of stopped/running containers ('
```

```
8
```

We can see that the number of containers reported by docker info correctly reports the number of total containers, running or not

But listing images gives a different value from the 'docker info' value

In [12]:
```
docker images | tail -n +2 | wc -l
```

```
6
```

That is because there are many intermediate image layers which are not normally listed. But we can list those layers using the '-a' option and now we see a number close to the value from 'docker info'.

(We will see later how the 'docker history' command allows us to see how the layers were created).

In [13]:
```
docker images -a | tail -n +2 | wc -l  # The number of image layers+1 (inc.
```

```
9
```

Images can include 1 static binary file or more and can even include a whole distribution. Launching a container launches a single process within that container - which may in turn span other child processes.

Let us look at an extremely small image to have an idea just how small an execitable image can be. Docker provide an official 'hello-world' image which simply echoes some output to the console.

Let's run that image to see and then investigate the image. First let's search for the image; we see that the first image is 'hello-world' which is an official build

In [16]:
```
docker search hello-world
```

```
NAME                                      DESCRIPTION
STARS     OFFICIAL   AUTOMATED
hello-world                               Hello World! (an example of mini
mal Docker...   47        [OK]
tutum/hello-world                         Image to test docker deployment
s. Has Apac...   19                  [OK]
marcells/aspnet-hello-world               ASP.NET vNext - Hello World
2                   [OK]
sbasyal/java-hello-world
1                   [OK]
carinamarina/hello-world-app              This is a sample Python web appl
ication, r...   1                   [OK]
vegasbrianc/docker-hello-world
1                   [OK]
carinamarina/hello-world-web              A Python web app, running on por
t 5000, wh...   1                   [OK]
wodge/docker-hello-world                  Hello World test for auto update
to Docker...   0                    [OK]
bencampbell/hello-world                   First automated build.
0                   [OK]
crccheck/hello-world                      Hello World web server in under
2.5 MB          0                   [OK]
mikelh/hello-world                        simplified hello world as dummy
start for ...   0                   [OK]
poojathote/hello-world                    this is 3rd POC
0                   [OK]
n8io/hello-world                          A simple hello world node.js app
to test d...   0                    [OK]
asakaguchi/docker-nodejs-hello-world      Hello World for Docker
0                   [OK]
ileontyev81/docker-hello-world            hello world test build
0                   [OK]
kevto/play2-docker-hello-world            Hello World application in Play2
to test D...   0                    [OK]
alexwelch/hello-world
0                   [OK]
vasia/docker-hello-world                  rhrthrth
0                   [OK]
samxzxy/docker-hello-world                Automated build test docker-hell
o-world         0                   [OK]
asakaguchi/magellan-nodejs-hello-world    Hello World for MAGELLAN
0                   [OK]
nirmata/hello-world
0                   [OK]
cpro/http-hello-world                     Hello world
0                   [OK]
rcarun/hello-world
0                   [OK]
chalitac/hello-world                      Just Hello World
0                   [OK]
wowgroup/hello-world                      Minimal web app for testing purp
```

```
oses            0                   [OK]
```

Let's now run that image

In [17]:
```
docker run hello-world
```

```
Hello from Docker.
This message shows that your installation appears to be working correctl
y.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs
the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which se
nt it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub accoun
t:
 https://hub.docker.com (https://hub.docker.com)

For more examples and ideas, visit:
 https://docs.docker.com/userguide/ (https://docs.docker.com/userguide/)
```

If it took a while to run, this was due to the time needed to download the image before running it - see above.

Try the command a second time to see how it runs instantaneously as there is no need to download the image which already exists locally on the 'docker engine'.

In [18]:
```
docker run hello-world
```

```
Hello from Docker.
This message shows that your installation appears to be working correctl
y.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs
the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which se
nt it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub accoun
t:
 https://hub.docker.com (https://hub.docker.com)

For more examples and ideas, visit:
 https://docs.docker.com/userguide/ (https://docs.docker.com/userguide/)
```

Let us inspect the image. We see that the file is only 960 bytes large, it must be machine code to print out the text. So we see that an image can be really very small

In [19]:
```
docker images hello-world
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
hello-world         latest              690ed74de00f        3 months ago
960 B
```

We can also inspect the image with the history command to see how it was constructed.

Note that history shows the image layers in reverse order, latest first.

From the below command we can see that the image was created from only 2 image layers.

The image was built simply by copying in a binary executable and then specifying the default command to invoke when the image is run.

In [20]:
```
docker history hello-world
```

```
IMAGE               CREATED             CREATED BY
SIZE                COMMENT
690ed74de00f        3 months ago        /bin/sh -c #(nop) CMD ["/hello"]
0 B
<missing>           3 months ago        /bin/sh -c #(nop) COPY file:1ad52
e3eaf4327c8f   960 B
```

Creating a small C Docker image

TOP

In this example we show how we can create a Docker image from a statically-linked binary.

The goal of this step is to show that we do not need an Operating System image for a Docker container.

All we need is a self-contained binary - i.e. statically linked binary.

Of course a dynamically linked binary could also be used, but in this case it's more complicated as you would have to manually add all it's dependent libraries. Let's let gcc to do that work for us!

This section comprises 2 things

- A Dockerfile to build our image from a static binary Note that it starts with "FROM scratch". Scratch is a special 'empty' image
- helloFromDocker.c

In [21]:
```
cd ~/src/createTinyC/
cat Dockerfile
```

```
FROM scratch
MAINTAINER "Docker Build Lab" <dockerlab@mjbright.net>

ADD ./helloWorld.exe /helloWorld.exe
CMD ["/helloWorld.exe"]
```

So first let's build our static binary

In [25]:
```
gcc -static helloWorld.c -o helloWorld.exe

ls -alh helloWorld.exe
```

```
-rwxrwxrwx 1 vagrant vagrant 857K Feb  2 17:33 helloWorld.exe
```

So we see that this created a binary file of approximately 857kby.

Now let's build our Docker image containing this binary

In [26]:
```
docker build -t lab/c_prog .
```

```
Sending build context to Docker daemon 881.2 kB
Step 1 : FROM scratch
 --->
Step 2 : MAINTAINER "Docker Build Lab" <dockerlab@mjbright.net>
 ---> Using cache
 ---> 736216a2a3eb
Step 3 : ADD ./helloWorld.exe /helloWorld.exe
 ---> Using cache
 ---> 0a40a3444cb7
Step 4 : CMD /helloWorld.exe
 ---> Using cache
 ---> 0932ac1b6d79
Successfully built 0932ac1b6d79
```

If we now look at the generated Docker image (below) we see an image of about 877kby.

So whilst this is larger than the 1kby hello-world image (no doubt written in assembler) it is still a very small Docker image which is only 20kbytes larger than the original binary file.

In [27]:
```
docker images lab/c_prog
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
lab/c_prog          latest              0932ac1b6d79        3 hours ago
877.2 kB
```

And now let's run that image

In [28]:
```
docker run lab/c_prog
```

```
Hello World!!
```

In [32]:
```
docker history lab/c_prog
```

```
IMAGE               CREATED             CREATED BY
SIZE                COMMENT
0932ac1b6d79        3 hours ago         /bin/sh -c #(nop) CMD ["/helloWor
ld.exe"]       0 B
0a40a3444cb7        3 hours ago         /bin/sh -c #(nop) ADD file:b9e477
15d0136bd3d8   877.2 kB
736216a2a3eb        4 hours ago         /bin/sh -c #(nop) MAINTAINER "Doc
ker Build La   0 B
```

Creating a small Go Docker image

TOP

That's fine, but isn't Go taking over the world as a systems language? Docker, Kubernetes, LXD, Rocket, ... many new tools are being written in Go.

Let's see how we can do the same exercise but building a Go statically-linked binary.

The goal of this step is as the previous step (building an image from a single statically-linked binary) but using Go, but also to demonstrate how we can use a Docker image containing a Go compiler, rather than explicitly installing a compiler.

We will do this without directly 'installing a Go compiler'

In [51]:
```
cd ~/src/createTinyGo
ls -al
```

```
total 5
drwxrwxrwx 1 vagrant vagrant    0 Feb  2 21:38 .
drwxrwxrwx 1 vagrant vagrant 4096 Feb  2 21:25 ..
-rwxrwxrwx 1 vagrant vagrant  124 Feb  2 21:38 build.sh
drwxrwxrwx 1 vagrant vagrant    0 Feb  2 21:35 src
```

In [56]:
```
docker run -it -v $PWD:/go golang go build -v hello
ls -altrh
```

```
hello
total 8.6M
drwxrwxrwx 1 vagrant vagrant 4.0K Feb  2 21:25 ..
drwxrwxrwx 1 vagrant vagrant    0 Feb  2 21:35 src
-rwxrwxrwx 1 vagrant vagrant  105 Feb  2 21:42 Dockerfile
-rwxrwxrwx 1 vagrant vagrant  155 Feb  2 21:42 build.sh
-rwxrwxrwx 1 vagrant vagrant 6.3M Feb  2 21:42 webserver
drwxrwxrwx 1 vagrant vagrant    0 Feb  2 21:43 .
-rwxrwxrwx 1 vagrant vagrant 2.3M Feb  2 21:43 hello
```

In [59]:
```
cat Dockerfile
```

```
FROM scratch
MAINTAINER "Docker Build Lab" <dockerlab@mjbright.net>

ADD ./hello /hello
CMD ["/hello"]
```

In [60]:
```
docker build -t lab/go-hello .
```

```
Sending build context to Docker daemon 8.982 MB
Step 1 : FROM scratch
 --->
Step 2 : MAINTAINER "Docker Build Lab" <dockerlab@mjbright.net>
 ---> Using cache
 ---> 736216a2a3eb
Step 3 : ADD ./hello /hello
 ---> Using cache
 ---> 56c9e7511b2f
Step 4 : CMD /hello
 ---> Using cache
 ---> 2efe3ea09cbf
Successfully built 2efe3ea09cbf
```

In [61]:
```
docker images lab/*
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
lab/go-hello        latest              2efe3ea09cbf        2 minutes ago
2.367 MB
lab/toolset         latest              f63a40539224        About an hour
ago   878.1 kB
lab/basic           latest              f880808cf281        2 hours ago
689.1 MB
lab/c_prog          latest              0932ac1b6d79        7 hours ago
877.2 kB
```

Creating a toolset Docker image containing several executeables

TOP

Now let's see how we can combine these static binaries into one image.

Let's build a new image derived from the Docker provided 'hello-world' image

The goal of this step is to show how we can combine several executables in an image, opening up the possibility of creating a

container of tools.

We will do this without directly 'installing a Go compiler'

xxxx

In [ ]:
```
cd ~/src/toolset
gcc -static helloWorld.c -o helloWorld.exe
ls -al helloWorld.exe
```

In [ ]:
```
docker build -t lab/toolset ./
```

In [32]:
```
docker history lab/toolset
```

```
IMAGE               CREATED             CREATED BY
SIZE                COMMENT
f63a40539224        3 minutes ago       /bin/sh -c #(nop) CMD ["/helloWor
ld.exe"]       0 B
6c5a5256b276        3 minutes ago       /bin/sh -c #(nop) ADD file:b9e477
15d0136bd3d8   877.2 kB
d21fae990e43        3 minutes ago       /bin/sh -c #(nop) MAINTAINER "Doc
ker Build La   0 B
690ed74de00f        3 months ago        /bin/sh -c #(nop) CMD ["/hello"]
0 B
<missing>           3 months ago        /bin/sh -c #(nop) COPY file:1ad52
e3eaf4327c8f   960 B
```

Now we are free to specify which command is to be run

In [34]:
```
docker run lab/toolset
```

```
Hello World!!
```

In [35]:
```
docker run lab/toolset /hello
```

```
Hello from Docker.
This message shows that your installation appears to be working correctl
y.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs
the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which se
nt it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub accoun
t:
 https://hub.docker.com (https://hub.docker.com)

For more examples and ideas, visit:
 https://docs.docker.com/userguide/ (https://docs.docker.com/userguide/)
```

In [36]:
```
docker run lab/toolset /helloWorld.exe
```

```
Hello World!!
```

Pushing our image to Docker Hub

TOP

Note: If you have your own account on Docker Hub you may wish to use that for this exercise.

Otherwise we will all be using the same account **'dockerlabs'** so you will need to specify a tag which distinguishes your images from your neighbours.

The goal of this step is to demonstrate how we may push an image which we have built to the Docker Hub.

First we will retag our local image to be unique. If you are on podN, then tag with userN,

e.g. if you are pod3,

```
bash docker tag lab/toolset dockerlabs/toolset:user3
```

Notice that we then have 2 toolset images with different tags.

They are otherwise identical (but they could be different) and have the same "IMAGE ID".

In [47]:
```
docker tag lab/toolset:latest dockerlabs/toolset:userN
docker images */toolset
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
dockerlabs/toolset  userN               f63a40539224        23 minutes a
go      878.1 kB
lab/toolset         latest              f63a40539224        23 minutes a
go      878.1 kB
```

First we must login to the Docker Hub

In [41]:
```
docker login -u dockerlabs -p dockerlabs -e dockerlabs@mjbright.net
```

```
WARNING: login credentials saved in /home/vagrant/.docker/config.json
Login Succeeded
```

Now we may push our image to the public Docker Hub

In [48]:
```
docker push dockerlabs/toolset:userN
```

```
The push refers to a repository [docker.io/dockerlabs/toolset]

40bca4fe2928: Preparing
5f70bf18a086: Preparing
userN: digest: sha256:ecc22b32fded8041af2e3ae64ff129d1cbee31797572287354b
743c02cd6986b size: 3658
```

In [ ]:
```

```

Dockerfile best practices

TOP

The goal of this step is to demonstrate certain Dockerfile optimizations.

- group related commands together using '&&' to reduce image layers
- if temporary files are to be removed

In [50]:
```
cd ~/src/build-best-practices
cat Dockerfile
```

```
FROM ubuntu

MAINTAINER "Docker Labs" <dockerlabs@mjbright.net>

#
# Instead of perofmring the followinf commands individually which
# involves creating a separate image layer for each RUN command:
#   RUN apt-get update
#   RUN apt-get -y -q upgrade
#   RUN rm -rf /var/lib/apt/lists/*

# Here we combine the update, upgrade and cleanup steps into one command
# - This produces less image layers (better for disk space and performanc
e)
# - This keeps image smaller by removing temporary files in the same laye
r
#     If we performed update/upgrade and then rm as a separate step there
would
#     be an intermediate layer including those files, making the overall
image larger.
#

RUN apt-get update && apt-get -y -q upgrade && rm -rf /var/lib/apt/list
s/*
```

In [ ]:

In [ ]:

Creating a Node.js application from the Node.js 'LanguageStack' Docker image

TOP

Docker provide a set of *'Language Stacks'* which are medium sized images representing the necessary dependencies for a particular language.

The goal of this step is to demonstrate the use of Docker-provided *Language Stacks*.

On the Docker Hub (https://hub.docker.com/) we can find language stacks available for a variety of languages/environments, each with different release versions (Python 2.x and Python 3.x for example):

- Node.js (Javascript) (https://hub.docker.com/_/node/)
- Python (https://hub.docker.com/_/python/)
- Ruby (https://hub.docker.com/_/ruby/)

You can browse the complete list of '*Official Images*' on the Docker Hub here (https://hub.docker.com/explore/)

Now let's look at an example of Node.js. To run a Node.js application this time we will need

In [17]:
```
time docker pull node
```

```
Using default tag: latest
latest: Pulling from library/node

81cc5f26a6a0: Pulling fs layer
a3ed95caeb02: Pulling fs layer
5a4693d81fc5: Pulling fs layer
9c2c7d262d05: Pulling fs layer
8ba9f41bd3f2: Pulling fs layer
c2a65292a17e: Pulling fs layer
Digest: sha256:3b69202057d1f98532e2d9952a445addc26a859cb95870384db24499a8
a2331f
Status: Downloaded newer image for node:latest

real    0m43.570s
user    0m0.093s
sys     0m0.060s
```

In [64]:
```
docker images node
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
node                latest              baa18fdeb577        7 days ago
643.1 MB
```

In [65]:
```
docker history node
```

```
IMAGE               CREATED             CREATED BY
SIZE                COMMENT
baa18fdeb577        7 days ago          /bin/sh -c #(nop) CMD ["node"]
0 B
<missing>           7 days ago          /bin/sh -c curl -SLO "https://nod
ejs.org/dist   36.39 MB
<missing>           7 days ago          /bin/sh -c #(nop) ENV NODE_VERSIO
N=5.5.0        0 B
<missing>           7 days ago          /bin/sh -c #(nop) ENV NPM_CONFI
G_LOGLEVEL=inf   0 B
<missing>           7 days ago          /bin/sh -c set -ex   && for key i
n     9554F0   51.75 kB
<missing>           7 days ago          /bin/sh -c apt-get update && apt-
get install    314.7 MB
<missing>           7 days ago          /bin/sh -c apt-get update && apt-
get install    122.6 MB
<missing>           7 days ago          /bin/sh -c apt-get update && apt-
get install    44.3 MB
<missing>           7 days ago          /bin/sh -c #(nop) CMD ["/bin/bas
h"]             0 B
<missing>           7 days ago          /bin/sh -c #(nop) ADD file:e5a3d2
0748c5d3dd5f   125.1 MB
```

In [ ]:
```
cd ~/src/nodeJS/
ls -altr
```

In [68]:
```
docker build -t node-hello .
```

```
Sending build context to Docker daemon 6.656 kB
Step 1 : FROM node
 ---> baa18fdeb577
Step 2 : ADD src/ /src
 ---> 41ca6d60d89b
Removing intermediate container 30a3f9e8b300
Step 3 : WORKDIR /src
 ---> Running in 856cfc757447
 ---> 993c7199a42b
Removing intermediate container 856cfc757447
Step 4 : RUN npm install
 ---> Running in a708c52e5e07
npm info it worked if it ends with ok
npm info using npm@3.3.12
npm info using node@v5.5.0
npm info attempt registry request try #1 at 9:50:24 PM
npm http request GET https://registry.npmjs.org/express (https://regist
ry.npmjs.org/express)
npm http 200 https://registry.npmjs.org/express (https://registry.npmj
```

In [ ]:
```
docker run -p 80:80 --name web -d node-hello
```

In [70]:
```
curl http://localhost
```

```
<html><body>Hello from Node.js container 9893caeb39d3</body></html>
```

Creating a Python application from the Python 'LanguageStack' Docker image

TOP

The goal of this step is to demonstrate the use of the Python *Language Stack*.

Now let's look at a Python example. To run a Node.js application this time we will need

Let's pull and examine the official 'Docker Language Stack' image of Python

Note how the earliest image layers (at the bottom of the list) have the same image ids as the earliest image layers of the Node;js image.

So we can see that they were both created from the same base.

In [11]:
```
time docker pull python
```

```
Using default tag: latest
latest: Pulling from library/python

03e1855d4f31: Already exists
a3ed95caeb02: Already exists
9269ba3950bb: Already exists
6ecee6444751: Already exists
7a0c192d4d25: Already exists
a3ed95caeb02: Already exists
66777d6149f5: Already exists
Digest: sha256:4651b83dd903ce78b1c455794f63d4108d9469a6c7fe97cd07d08a77b7
e72435
Status: Image is up to date for python:latest

real    0m3.769s
user    0m0.049s
sys     0m0.019s
```

In [12]:
```
docker images python
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
python              latest              93049cc049a6        6 days ago
689.1 MB
```

In [13]:
```
docker history python
```

```
IMAGE               CREATED             CREATED BY
SIZE                COMMENT
93049cc049a6        6 days ago          /bin/sh -c #(nop) CMD ["python3"]
0 B
<missing>           6 days ago          /bin/sh -c cd /usr/local/bin  &&
ln -s easy_i   48 B
<missing>           6 days ago          /bin/sh -c set -ex  && gpg --keys
erver ha.poo   81.53 MB
<missing>           6 days ago          /bin/sh -c #(nop) ENV PYTHON_PI
P_VERSION=7.1.   0 B
<missing>           6 days ago          /bin/sh -c #(nop) ENV PYTHON_VERS
ION=3.5.1      0 B
<missing>           6 days ago          /bin/sh -c #(nop) ENV GPG_KEY=97F
C712E4C024BB   0 B
<missing>           6 days ago          /bin/sh -c #(nop) ENV LANG=C.UTF-
8              0 B
<missing>           6 days ago          /bin/sh -c apt-get purge -y pytho
n.*            978.7 kB
<missing>           7 days ago          /bin/sh -c apt-get update && apt-
get install    314.7 MB
<missing>           7 days ago          /bin/sh -c apt-get update && apt-
get install    122.6 MB
<missing>           7 days ago          /bin/sh -c apt-get update && apt-
get install    44.3 MB
<missing>           7 days ago          /bin/sh -c #(nop) CMD ["/bin/bas
h"]             0 B
<missing>           7 days ago          /bin/sh -c #(nop) ADD file:e5a3d2
0748c5d3dd5f   125.1 MB
```

Dockerfile

Images are built from Dockerfiles which contain a series of commands used to build up a docker image. Note that each command in the Dockerfile results in a new image layer being created, no matter how trivial the command - even ENV "commands" create a nw image layer.

In the following lab we will see how images can be built systematically from a Dockerfile using the 'docker build' command.

DockerHub

When we pull an image we pull it from a Docker Registry. The DockerHub (https://hub.docker.com/) is a free to use Docker registry allowing to store your own image files (which are publicly available unless you pay for your account) and to pull other image files of other users or officially provided images.

You can create images either by

- building them from a Dockerfile (thus in a **repeatable** manner)
- building them manually by modifying a running container and *'commit'*ing it's state

The DockerHub contains images which may be

- **Automated builds** (built from a git repository)
  - Such builds are usually built from an open-source git repo and so are called **Trusted huilds** because the source code is available. *Note:* The github repo may contain binary files though
- **Official builds** are builds which are builds provided by partners or by Docker themselves

Other images may exist in the hub but their origin is unknown and so represent a security risk.

It is possible to search the DockerHub, or another Docker Registry, using the 'docker search' command with appropriate options. Other companies offer their own Docker Registry which may be freely accessible e.g. RedHat, internal to a company e.g. HPE IT, or available as part of a paid for service e.g. IBM or Amawon Web Services ECR.

Lab-p2

Docker Builds

Simple Docker Build

Let's start by creating a minimal Dockerfile which builds on the Python image we downloaded earlier

In [14]:
```
mkdir ~/test
cd    ~/test

cat > Dockerfile.basic <<EOF
#
# Dockerfile to demonstrate the simplest build
#
FROM python

MAINTAINER "xxx"

RUN python --version

CMD bash

EOF
```

```
mkdir: cannot create directory ‘/home/vagrant/test’: File exists
```

- First see what images are present
- Build a simple image based on ... medium size
- Observe build process
- Modify Dockerfile
- Rebuild - observe caching
- Move order in Dockerfile
- Rebuild
- Rebuild with no caching
- Perform Docker image
- Perform docker image -a
- Perform docker history

In [15]:
```
ls -altr Dockerfile.basic
```

```
-rw-rw-r-- 1 vagrant vagrant 114 Feb  2 19:11 Dockerfile.basic
```

We can now build a new image using this dockerfile using the below command where

- we explicitly select the Dockerfile.basic which we just created
- we specify the current directory as the context for the build (any ADD/COPY or Dockerfile files will be sourced from here)
- we specify the specific tag to se for the generated image as "*lab/basic*"

In [16]:
```
docker build -f Dockerfile.basic -t lab/basic .
```

```
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM python
 ---> 93049cc049a6
Step 2 : MAINTAINER "xxx"
 ---> Running in 713c65041098
 ---> 0c94e8192e1c
Removing intermediate container 713c65041098
Step 3 : RUN python --version
 ---> Running in da8d757d8876
Python 3.5.1
 ---> 84afb463af76
Removing intermediate container da8d757d8876
Step 4 : CMD bash
 ---> Running in 9795ae7e86c5
 ---> cff63d53a214
Removing intermediate container 9795ae7e86c5
Successfully built cff63d53a214
```

Note that during the build, the RUN commands are actually run.

They are used to build up this new image.

In this case we echo the 'Python' version string during the build process.

You can see the available options to the build command by issuing 'docker build --help'

In [17]:
```
docker build --help
```

```
Usage:  docker build [OPTIONS] PATH | URL | -

Build an image from a Dockerfile

  --build-arg=[]                    Set build-time variables
  --cpu-shares                      CPU shares (relative weight)
  --cgroup-parent                   Optional parent cgroup for the containe
r
  --cpu-period                      Limit the CPU CFS (Completely Fair Sche
duler) period
  --cpu-quota                       Limit the CPU CFS (Completely Fair Sche
duler) quota
  --cpuset-cpus                     CPUs in which to allow execution (0-3,
0,1)
  --cpuset-mems                     MEMs in which to allow execution (0-3,
0,1)
  --disable-content-trust=true      Skip image verification
  -f, --file                        Name of the Dockerfile (Default is 'PAT
H/Dockerfile')
  --force-rm                        Always remove intermediate containers
  --help                            Print usage
  --isolation                       Container isolation level
  -m, --memory                      Memory limit
  --memory-swap                     Swap limit equal to memory plus swap:
'-1' to enable unlimited swap
  --no-cache                        Do not use cache when building the imag
e
  --pull                            Always attempt to pull a newer version
of the image
  -q, --quiet                       Suppress the build output and print ima
ge ID on success
  --rm=true                         Remove intermediate containers after a
successful build
  --shm-size                        Size of /dev/shm, default value is 64MB
  -t, --tag=[]                      Name and optionally a tag in the 'nam
e:tag' format
  --ulimit=[]                       Ulimit options
```

We can see all the images available using the 'docker images' command

but if there are many, how do we see just our newly-created image?

You can see the available options to the images command by issuing 'docker images --help'

In [18]:
```
docker images --help
```

```
Usage:  docker images [OPTIONS] [REPOSITORY[:TAG]]

List images

  -a, --all          Show all images (default hides intermediate images)
  --digests          Show digests
  -f, --filter=[]    Filter output based on conditions provided
  --format           Pretty-print images using a Go template
  --help             Print usage
  --no-trunc         Don't truncate output
  -q, --quiet        Only show numeric IDs
```

So you can see your newly built 'lab/basic' with the following command:

In [19]:
```
docker images lab/basic
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
lab/basic           latest              cff63d53a214        48 seconds ag
o      689.1 MB
```

Note that if you rerun the build command, the build should run faster, you will notice how build steps recognize that this step has already been performed and so will use the image layer already available in the local cache.

Now let us see what happens if we modify our Dockerfile, by inserting a line, such as defining an environment variable.

We will use the same Dockerfile, but this time we will insert an "ENV" line

In [20]:
```
cd /root/test/

cat > Dockerfile.basic <<EOF
#
# Dockerfile to demonstrate the simplest build (with ENV line added)
#
FROM python

MAINTAINER "xxx"

RUN python --version

ENV myvar "this will force rebuilding from here on"

CMD bash

EOF
```

```
bash: cd: /root/test/: Permission denied
```

This time when we build the image we will see that the addition of a line between the "RUN" line and the "CMD" line forces rebuild of subsequent image layers.

We see 'Using cache' for Step 2 and 3 only

In [21]:
```
docker build -f Dockerfile.basic -t lab/basic .
```

```
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM python
 ---> 93049cc049a6
Step 2 : MAINTAINER "xxx"
 ---> Using cache
 ---> 0c94e8192e1c
Step 3 : RUN python --version
 ---> Using cache
 ---> 84afb463af76
Step 4 : ENV myvar "this will force rebuilding from here on"
 ---> Running in 7fee2d11d4e7
 ---> f5c03fbd8055
Removing intermediate container 7fee2d11d4e7
Step 5 : CMD bash
 ---> Running in 01b860dcc655
 ---> 9963f26be3d0
Removing intermediate container 01b860dcc655
Successfully built 9963f26be3d0
```

Similarly we can force to not use the cache with the --no-cache option.

This could be useful if we suspect the caching is not working properly due to some external change.

In [22]:
```
docker build --no-cache=true -f Dockerfile.basic -t lab/basic .
```

```
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM python
 ---> 93049cc049a6
Step 2 : MAINTAINER "xxx"
 ---> Running in a25a7b0c6b96
 ---> 3b59c02497eb
Removing intermediate container a25a7b0c6b96
Step 3 : RUN python --version
 ---> Running in 2dbe512d90c1
Python 3.5.1
 ---> 5f43942fd14e
Removing intermediate container 2dbe512d90c1
Step 4 : ENV myvar "this will force rebuilding from here on"
 ---> Running in 12141f1771e7
 ---> b3e694f22a12
Removing intermediate container 12141f1771e7
Step 5 : CMD bash
 ---> Running in 9d824dd3fe22
 ---> f880808cf281
Removing intermediate container 9d824dd3fe22
Successfully built f880808cf281
```

In [23]:
```
docker images lab/basic
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
lab/basic           latest              f880808cf281        4 seconds ago
689.1 MB
```

More complex Docker Builds

```
- extending a Dockerfile ("inheritance")
```

- Building from different base images
  - e.g. from Alpine
- Build from Language Stack
  - e.g. build a node application from Node language stack
  - e.g. build a Go application from Go language stack
- Availability of diverse (official/trusted) images
  - Do some MongoDB or something ...
  - simple web server, multi-tier??

- **Docker Small Builds**
  - Build a small image Miniature web server?
  - Build a small Go image (using a Go container to build)
    - Show how image is small
    - underline how we didn't need to install a compiler natively

        - Build an incremental image based on this
    - Perform a cross-compilation for Raspberry Pi (need a large image pre-loaded)
        - Perform file on a.out to show architecture
        - not yet a way of seeing architecture of image (either locally or on DockerHub)
        - pull/run an ARM docker image (smallest Hypriot)

In [ ]:

Docker Automated Builds

- Create an account on DockerHub if necessary
    - **OR** reuse
        - Your own account
        - or reuse lab account **hpe_tss_lab**
    - Push a **small** image to DockerHub
        - if using lab account - name your image as **userN** where N is your pod number
    - Pull the image
    - Look at GMail account ?? (optional - confusing, conflicting ... in separate browser)
- Create an automated build on DockerHub
    - Create a github account if necessary
        - **OR** reuse
            - Your own account
            - or reuse lab account **hpe_tss_lab**
    - Create a repo
        - if using lab account - name your repo as **userN** where N is your pod number
    - Checkout repo
    - Create a Dockerfile, complete it
    - commit/push repo
    - On DockerHub create automated build
    - Perform initial build
    - See "build log"
    - Modify Dockerfile
    - commit/push repo
    - See "build log"

In [ ]:

How to push to Docker Hub from Jupyter??

Login to Docker Hub:

In [ ]:
```
docker login -u dockerlabs -p dockerlabs -e dockerlabs@mjbright.net
```

WARNING: login credentials saved in ~/.docker/config.json Login Succeeded

Hypriot example

Extend it with different backends

- Node
- Python server
- ??

Example

TOP

This example heavily inspired from this article microservice with rails and Docker (http://blog.giantswarm.io/getting-started-with-microservices-using-ruby-on-rails-and-docker/) ... but not complete, too cloudy ...

The goal of this step is ...

In [ ]:
```
mkdir -p /root/src/nginx

cd /root/src/nginx

cat > Dockerfile <<EOF
FROM nginx

RUN rm -rf /usr/share/nginx/html
COPY public /usr/share/nginx/html
COPY config/deploy/nginx.conf /etc/nginx/conf.d/default.conf
EOF
```

A Dockerfile always starts with the FROM instruction. It tells Docker, which image is the base to be used. Additionally, we will just COPY the public folder and the configuration into the image. As I said before containers should be treated as immutable and due to this, we

need to create a new image whenever we change the assets. The nginx config itself is pretty straight forward:

In [58]:
```
cat > 03_nginx_config.conf <<EOF
server {
    listen       80;
    server_name  localhost;

    location / {
        root      /usr/share/nginx/html;
        index     index.html index.htm;
        try_files $uri/index.html $uri.html $uri @upstream;
    }

    location @upstream {
        proxy_pass http://rails-app:8080;
    }
}
EOF
```

In [60]:
```
cat > Dockerfile.rails <<EOF
FROM ruby:2.1.5

# throw errors if Gemfile has been modified since Gemfile.lock
RUN bundle config --global frozen 1

RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

COPY Gemfile /usr/src/app/
COPY Gemfile.lock /usr/src/app/

RUN bundle install --deployment

COPY . /usr/src/app/

ENV RAILS_ENV production

EXPOSE 8080

CMD ["/usr/src/app/bin/rails", "server", "-p", "8080"]
EOF
```

In [62]:
```
cat > rake_tasks.rb <<EOF
namespace :docker do
  task :build => ['docker:build:web', 'docker:build:app', 'docker:build:wor

  namespace :build do
    task :web => ['assets:precompile', 'assets:clean'] do
      sh 'ln -snf Dockerfile.web Dockerfile'
      sh 'sudo docker build -t "registry.giantswarm.io/yoshida/gh-recommend
      sh 'rm -f Dockerfile'
    end

    task :app => ['assets:precompile', 'assets:clean'] do
      sh 'ln -snf Dockerfile.app Dockerfile'
      sh 'sudo docker build -t "registry.giantswarm.io/yoshida/gh-recommend
      sh 'rm -f Dockerfile'
    end

    task :worker do
      sh 'ln -snf Dockerfile.worker Dockerfile'
      sh 'sudo docker build -t "registry.giantswarm.io/yoshida/gh-recommend
      sh 'rm -f Dockerfile'
    end
  end
end
EOF
```

In [63]:
```
ls -altr /root/src/nginx
```

```
total 20
drwxr-xr-x 5 root root  46 Jan 31 20:54 ..
-rw-r--r-- 1 root root 140 Jan 31 20:55 Dockerfile
-rw-r--r-- 1 root root 287 Jan 31 20:57 01_nginx_config.conf
-rw-r--r-- 1 root root 287 Jan 31 20:57 03_nginx_config.conf
-rw-r--r-- 1 root root 374 Jan 31 20:59 Dockerfile.rails
-rw-r--r-- 1 root root 802 Jan 31 21:00 rake_tasks.rb
drwxr-xr-x 2 root root 120 Jan 31 21:00 .
```

One thing needs to be mentioned though: Where does this rails-app hostname come from? Docker will provide information about linked (we will explain later what that is) containers in two ways: A bunch of environment variables and entries in the /etc/hosts file. In this case we’re gonna use the entry from the /etc/hosts.

In [55]:
```

```

```
total 4
drwxr-xr-x 5 root root  46 Jan 31 20:54 ..
drwxr-xr-x 2 root root  23 Jan 31 20:55 .
-rw-r--r-- 1 root root 140 Jan 31 20:55 Dockerfile
```

Building complex systems with Compose

TOP

In [71]:
```
cd ~/src/compose

cat docker-compose.yml
```

```
version: 2
services:
  weba:
    build: ../nodeJS
    expose:
      - 80

  webb:
    build: ../nodeJS
    expose:
      - 80

  webc:
    build: ../nodeJS
    expose:
      - 80

  haproxy:
    image: haproxy
    volumes:
     - ./haproxy:/haproxy-override
    links:
     - weba
     - webb
     - webc
    ports:
     - "80:80"
     - "70:70"

    expose:
     - "80"
     - "70"
```

In [74]:
```
docker-compose up -d
```

```
compose_webb_1 is up-to-date
compose_webc_1 is up-to-date
compose_weba_1 is up-to-date
Recreating compose_haproxy_1
```

```
docker-compose logs
Attaching to compose_haproxy_1, compose_weba_1, compose_webc_1, com
pose_webb_1
haproxy_1 | [ALERT] 032/221525 (1) : Could not open configuration f
ile /usr/local/etc/haproxy/haproxy.cfg : No such file or directory
haproxy_1 | [ALERT] 032/221646 (1) : Could not open configuration f
ile /usr/local/etc/haproxy/haproxy.cfg : No such file or directory
weba_1    | Running on http://localhost
webc_1    | Running on http://localhost
webb_1    | Running on http://localhost
```

In [80]:
```
docker-compose ps
```

```
      Name                     Command               State    Ports
-------------------------------------------------------------------
compose_haproxy_1   haproxy -f /usr/local/etc/ ...   Exit 1
compose_weba_1      node index.js                    Up       80/tcp
compose_webb_1      node index.js                    Up       80/tcp
compose_webc_1      node index.js                    Up       80/tcp
```

In [82]:
```
docker-compose scale weba=5
```

```
Creating and starting 2 ...
Creating and starting 3 ...
Creating and starting 4 ...
Creating and starting 5 ...
```

In [83]:
```
docker-compose ps
```

```
      Name                     Command               State    Ports
-------------------------------------------------------------------
compose_haproxy_1   haproxy -f /usr/local/etc/ ...   Exit 1
compose_weba_1      node index.js                    Up       80/tcp
compose_weba_2      node index.js                    Up       80/tcp
compose_weba_3      node index.js                    Up       80/tcp
compose_weba_4      node index.js                    Up       80/tcp
compose_weba_5      node index.js                    Up       80/tcp
compose_webb_1      node index.js                    Up       80/tcp
compose_webc_1      node index.js                    Up       80/tcp
```

In [85]:
```
docker-compose up --force-recreate -d
```

```
Recreating compose_webb_1
Recreating compose_webc_1
Recreating compose_weba_5
Recreating compose_weba_3
Recreating compose_weba_4
Recreating compose_weba_2
Killed
```

In [ ]:
```

```

In [84]:
```
#docker-compose logs
```

Rails Example with Compose

TOP

This example heavily inspired from this article Building Microservices with Docker and the Rails API gem (https://medium.com/connect-the-dots/building-microservices-with-docker-and-the-rails-api-gem-2a463862f5d)

The goal of this step is to have hands-on experience with Compose ...

It is recommended to use yamllint (http://www.yamllint.com/) to validate your YAML file - because it's easy to make mistakes in YAML, and Compose is picky.

In [36]:
```
cd /root
mkdir -p src/railsapi
cd src/railsapi
pwd
touch Dockerfile docker-compose.yml Gemfile Gemfile.lock
```

```
/root/src/railsapi
```

In [37]:
```
cat > Dockerfile <<EOF

FROM ruby:2.3.0

RUN apt-get update -qq && apt-get install -y build-essential libmysqlclient

RUN mkdir /railsapi

WORKDIR /railsapi

ADD Gemfile /railsapi/Gemfile

ADD Gemfile.lock /railsapi/Gemfile.lock

RUN bundle install

ADD . /railsapi

EOF
```

See References section below for information on *Compose*

In [38]:
```
cat > docker-compose.yml <<EOF

version: 2
services:
  db:
   image: mysql:latest
   ports:
     - 3306:3306
   environment:
     MYSQL_ROOT_PASSWORD: mypassword

  web:
    build: .
    command: puma
    ports:
      - 9292:9292
    links:
      - db
    volumes:
      - .:/railsapi

EOF
```

In [39]:
```
docker-compose build
```

```
db uses an image, skipping
Building web
Step 1 : FROM ruby:2.3.0
 ---> 70578fbdd1a4
Step 2 : RUN apt-get update -qq && apt-get install -y build-essential lib
mysqlclient-dev
 ---> Using cache
 ---> 542437588210
Step 3 : RUN mkdir /railsapi
 ---> Using cache
 ---> 458b06826730
Step 4 : WORKDIR /railsapi
 ---> Using cache
 ---> 972ec34e499e
Step 5 : ADD Gemfile /railsapi/Gemfile
 ---> Using cache
 ---> 60947231435f
Step 6 : ADD Gemfile.lock /railsapi/Gemfile.lock
 ---> 53c629b24a97
Removing intermediate container 1aac5f737465
Step 7 : RUN bundle install
 ---> Running in c7e1111ea35d
Fetching gem metadata from https://rubygems.org/.......... (https://ruby
gems.org/..........)
Fetching version metadata from https://rubygems.org/... (https://rubygem
s.org/...)
Fetching dependency metadata from https://rubygems.org/.. (https://rubyge
ms.org/..)
Installing rake 10.5.0
Installing i18n 0.7.0
Using json 1.8.3
Installing minitest 5.8.4
Installing thread_safe 0.3.5
Installing builder 3.2.2
Installing erubis 2.7.0
Installing mini_portile2 2.0.0
Installing rack 1.6.4
Installing mime-types 2.99
Installing thor 0.19.1
Installing arel 6.0.3
Installing byebug 8.2.1 with native extensions
Installing concurrent-ruby 1.0.0
Installing mysql2 0.4.2 with native extensions
Installing puma 2.16.0 with native extensions
Using bundler 1.11.2
Installing spring 1.6.2
Installing tzinfo 1.2.2
Installing nokogiri 1.6.7.2 with native extensions
Installing rack-test 0.6.3
Installing mail 2.6.3
Installing sprockets 3.5.2
```

```
Installing activesupport 4.2.5
Installing loofah 2.0.3
Installing rails-deprecated_sanitizer 1.0.3
Installing globalid 0.3.6
Installing activemodel 4.2.5
Installing rails-html-sanitizer 1.0.3
Installing rails-dom-testing 1.0.7
Installing activejob 4.2.5
Installing activerecord 4.2.5
Installing actionview 4.2.5
Installing actionpack 4.2.5
Installing actionmailer 4.2.5
Installing railties 4.2.5
Installing sprockets-rails 3.0.1
Installing active_model_serializers 0.10.0.rc4
Installing rails-api 0.4.0
Installing rails 4.2.5
Bundle complete! 7 Gemfile dependencies, 40 gems now installed.
Bundled gems are installed into /usr/local/bundle.
 ---> 2611f8731623
Removing intermediate container c7e1111ea35d
Step 8 : ADD . /railsapi
 ---> 7159710bbffe
Removing intermediate container 7755d7569da1
Successfully built 7159710bbffe
```

In [40]:
```
cat > Gemfile <<EOF

source 'https://rubygems.org'

gem 'rails', '4.2.5'
gem 'rails-api', '0.4.0'
gem 'mysql2'
gem 'puma'

# Use ActiveModel has_secure_password
# gem 'bcrypt', '~> 3.1.7'

# Use Capistrano for deployment
# gem 'capistrano-rails', group: :development

# Use Rack CORS for handling Cross-Origin Resource Sharing (CORS), making c
# gem 'rack-cors'

# Use ActiveModelSerializers to serialize JSON responses
gem 'active_model_serializers', '~> 0.10.0.rc3'

group :development, :test do
  # Call 'byebug' anywhere in the code to stop execution and get a debugger
  gem 'byebug'
end

group :development do # Spring speeds up development by keeping your applic
  gem 'spring'
end

EOF
```

In [41]:
```
ls -altr
```

```
total 32
-rw-r--r-- 1 root root  478 Jan 31 19:34 README.rdoc
-rw-r--r-- 1 root root  249 Jan 31 19:34 Rakefile
-rw-r--r-- 1 root root  474 Jan 31 19:34 .gitignore
-rw-r--r-- 1 root root  153 Jan 31 19:34 config.ru
drwxr-xr-x 2 root root  147 Jan 31 19:34 .
drwxr-xr-x 4 root root   34 Jan 31 19:45 ..
-rw-r--r-- 1 root root 2928 Jan 31 20:01 Gemfile.lock
-rw-r--r-- 1 root root  248 Jan 31 20:01 Dockerfile
-rw-r--r-- 1 root root  258 Jan 31 20:02 docker-compose.yml
-rw-r--r-- 1 root root  779 Jan 31 20:10 Gemfile
```

Now let's build our image

In [42]:
```
docker-compose build
```

```
db uses an image, skipping
Building web
Step 1 : FROM ruby:2.3.0
 ---> 70578fbdd1a4
Step 2 : RUN apt-get update -qq && apt-get install -y build-essential lib
mysqlclient-dev
 ---> Using cache
 ---> 542437588210
Step 3 : RUN mkdir /railsapi
 ---> Using cache
 ---> 458b06826730
Step 4 : WORKDIR /railsapi
 ---> Using cache
 ---> 972ec34e499e
Step 5 : ADD Gemfile /railsapi/Gemfile
 ---> Using cache
 ---> 60947231435f
Step 6 : ADD Gemfile.lock /railsapi/Gemfile.lock
 ---> Using cache
 ---> 53c629b24a97
Step 7 : RUN bundle install
 ---> Using cache
 ---> 2611f8731623
Step 8 : ADD . /railsapi
 ---> Using cache
 ---> 7159710bbffe
Successfully built 7159710bbffe
```

In [43]:
```
docker images
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
railsapi_web        latest              7159710bbffe        6 minutes ago
813.8 MB
lab/basic           latest              5477a1cdda04        11 minutes ag
o      689.1 MB
<none>              <none>              a192650246e9        11 minutes ag
o      689.1 MB
<none>              <none>              a7d63cd9ba3c        12 minutes ag
o      689.1 MB
<none>              <none>              8855922a993d        46 minutes ag
o      813.8 MB
<none>              <none>              bef232158200        48 minutes ag
o      746.3 MB
rails_web           latest              5152cec9abc7        53 minutes ag
o      813.8 MB
<none>              <none>              349e9cce0641        About an hour
ago   813.8 MB
<none>              <none>              96b443625f76        About an hour
ago   813.8 MB
<none>              <none>              7d9bce362118        About an hour
ago   813.8 MB
<none>              <none>              524526aa2d95        20 hours ago
813.8 MB
docker-dev          HEAD                1b3493335e9d        34 hours ago
1.868 GB
docker-dev          master              5a5f54be5a20        44 hours ago
2.058 GB
ruby                2.3.0               70578fbdd1a4        3 days ago
721.8 MB
mysql               latest              a917fce37db8        4 days ago
360.2 MB
python              latest              93049cc049a6        4 days ago
689.1 MB
node                latest              0869e88f8617        10 days ago
643 MB
ubuntu              trusty              3876b81b5a81        11 days ago
187.9 MB
alpine              latest              3376496dc94c        13 days ago
4.79 MB
swarm               latest              32d67c5a4211        7 weeks ago
17.15 MB
hello-world         latest              690ed74de00f        3 months ago
960 B
```

In [44]:
```
docker-compose run web rails-api new .
```

```
       exist
   identical  README.rdoc
   identical  Rakefile
   identical  config.ru
   identical  .gitignore
    conflict  Gemfile
Overwrite /railsapi/Gemfile? (enter "h" for help) [Ynaqdh]
```

In [46]:
```
cat > database.yml <<EOF
development:
 adapter: mysql2
 encoding: utf8
 reconnect: false
 database: inventory_manager_dev
 pool: 5
 username: root
 password: mypassword
 host: db
test:
 adapter: mysql2
 encoding: utf8
 reconnect: false
 database: inventory_manager_test
 pool: 5
 username: root
 password: mypassword
 host: db
EOF
```

In [48]:
```
docker-compose up web
```

```
railsapi_db_1 is up-to-date
Creating railsapi_web_1
Attaching to railsapi_web_1
web_1 | Puma starting in single mode...
web_1 | * Version 2.16.0 (ruby 2.3.0-p0), codename: Midwinter Nights Tran
ce
web_1 | * Min threads: 0, max threads: 16
web_1 | * Environment: development
web_1 | config.ru:3:in `require': cannot load such file -- /railsapi/conf
ig/environment (LoadError)
web_1 |         from config.ru:3:in `block in <main>'
web_1 |         from /usr/local/bundle/gems/rack-1.6.4/lib/rack/builder.r
b:55:in `instance_eval'
web_1 |         from /usr/local/bundle/gems/rack-1.6.4/lib/rack/builder.r
b:55:in `initialize'
web_1 |         from config.ru:in `new'
web_1 |         from config.ru:in `<main>'
web_1 |         from /usr/local/bundle/gems/rack-1.6.4/lib/rack/builder.r
b:49:in `eval'
web_1 |         from /usr/local/bundle/gems/rack-1.6.4/lib/rack/builder.r
b:49:in `new_from_string'
web_1 |         from /usr/local/bundle/gems/rack-1.6.4/lib/rack/builder.r
b:40:in `parse_file'
web_1 |         from /usr/local/bundle/gems/puma-2.16.0/lib/puma/configur
ation.rb:155:in `load_rackup'
web_1 |         from /usr/local/bundle/gems/puma-2.16.0/lib/puma/configur
ation.rb:99:in `app'
web_1 |         from /usr/local/bundle/gems/puma-2.16.0/lib/puma/runner.r
b:114:in `load_and_bind'
web_1 |         from /usr/local/bundle/gems/puma-2.16.0/lib/puma/single.r
b:79:in `run'
web_1 |         from /usr/local/bundle/gems/puma-2.16.0/lib/puma/cli.rb:2
14:in `run'
web_1 |         from /usr/local/bundle/gems/puma-2.16.0/bin/puma:10:in
`<top (required)>'
web_1 |         from /usr/local/bundle/bin/puma:16:in `load'
web_1 |         from /usr/local/bundle/bin/puma:16:in `<main>'
web_1 | ! Unable to load application: LoadError: cannot load such file --
/railsapi/config/environment
railsapi_web_1 exited with code 1
```

In [49]:
```
curl http://localhost:80
```

```
curl: (7) Failed to connect to localhost port 80: Connection refused
```

Building Docker with Docker

TOP

A major advantage of Docker is to simplify build environments.

Let's look at how we can build the Docker engine client/daemon binary without having to explicitly install a development environment.

The goal of this step is simply to show the ease with which we can build Docker, thanks to Docker itself.

We do not make particular use of the built image.

The process involves the following steps, several of which have already been performed so as to prevent excessive network utilisation during the lab. Nevertheless all steps are described here so that you can see just how easy it is to build Docker from scratch:

- Install make
- Clone the Docker source code
- Checkout the same code revision as our current Docker binary (client and daemon)
- Build the code - which pulls the docker-dev image containing the required version of the Go compiler
- Run the executeable to demonstrate it is correct

Installing make

In your environment we have already installed the make package, but no compiler using yum:

```
yum install make
```

Cloning the Docker source code

We have already downloaded the Docker source code from github as follows:

```
mkdir -p /root/src/docker
cd /root/src/docker
git clone https://github.com/docker/docker .
```

To build Docker we simply have to build using the

```
make build
```

command.

Checkout the source code revision corresponding to our installed Docker Engine

If we build the latest sources this may not be compatible with our installed Docker version.

This is the case. We have 1.10.0-rc2 installed, which has API version 22, but the current github source is 1.10.0-dev which has changed to API version 23. So if we build this we find that we cannot use this client to communicate with the installed daemon.

So let's checkout the code for 1.10.0-rc2.

At the time of writing this is the latest release(candidate) of the Docker engine. We can obtain that version of the source code by referring to the releases page https://github.com/docker/docker/releases (https://github.com/docker/docker/releases) and selecting the SHA1 hash of build 1.10.0-rc2

```
git checkout c1cdc6e
```

Build the code - which pulls the docker-dev image containing the required version of the Go compiler

We can build the code as follows:

```
make build
```

We have run 'make build' already, so the docker-dev image has already been downloaded (again to prevent excessive network traffic). The docker-dev image includes the required go compiler and other build tools.

Run 'make build' again and you will see a standard build process and finally where it places the compiled binary

Run the executeable to demonstrate it is correct

In preparation for the lab we built from the latest source (not the c1cdc6e version we checked out).

Run this build as follows to see that it is not compatible with the installed binary (/usr/bin/docker). We see that this binary has version 1.10.0-dev and API version 1.23 but that this cannot communicate with our installed binary which has API version 1.22.

In [ ]:
```
cd /root/src/docker; ls -altr bundles/1.10.0-dev/binary/docker-1.10.0-dev;
```

But if we run our new build - as follows - created from revision c1cdc6e of the source code (corresponding to Docker version 1.10.0-rc2) we see that it has the correct version, with the same API version and can interrogate the server.

In [ ]:
```
cd /root/src/docker; ls -altr bundles/1.10.0-rc2/binary/docker-1.10.0-rc2;
```

Docker Builds Composition and Orchestration

- Compose
  - First guide through an n-Tier linked container example
    - build
    - run with linking
  - Now just compose
- Swarm? / Mesos? / Kubernetes?
- Weave?

Future Lab Ideas

- **Docker Container Services (FUTURE e.g. for TES?)**
  - Amazon ECS
  - Digital Ocean
  - MS Azure (**best**)
  - OpenStack Magnum
    - Helion OpenStack (**best**)
- **Docker Continuous Integration (FUTURE e.g. for TES?)**
  - Continuous Integration

Demo

TOP

Demo duration (mins):

In [ ]:

In [ ]:

In [ ]:

References

TOP

- Dockerfile Reference (https://docs.docker.com/engine/reference/builder/)
- Compose file documentation (https://docs.docker.com/compose/compose-file/)
- Compose file reference (https://github.com/docker/compose/blob/1.6.0-rc1/docs/compose-file.md)

- Visualizing Docker Containers and Images (http://merrigrove.blogspot.in/2015/10/visualizing-docker-containers-and-images.html)
- Awesome Docker (https://github.com/veggiemonk/awesome-docker)
- Docker Cheat Sheet ()
- Building Good Docker Images (http://jonathan.bergknoff.com/journal/building-good-docker-images)
- How to scale a Docker Container with Docker Compose (https://www.brianchristner.io/how-to-scale-a-docker-container-with-docker-compose/)

In [ ]: