Lab Outline

Online

This document
You can find the most up-to-date version of this document online as
- an HTML file (https://cdn.rawgit.com/mjbright/jupyter_notebooks/master/2016-Feb_Docker_Build_Lab/2016-Feb_Docker_Build_Lab.html) (http://bit.ly/1T0118e (http://bit.ly/1T0118e)) or 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/)

in mjbright (https://www.linkedin.com/in/mjbright),
mjbright (https://github.com/mjbright),
@mjbright (http://twitter.com/mjbright),

Lab-Description

TOP

We first need to recuperate the source code examples:

Lab Start

Start this lab by first performing the below step:

In [52]:

```
## Lab Start:

rm -rf ~/src

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

./src/START_LAB.sh
```

```
Cloning into 'src'...
remote: Counting objects: 156, done.
remote: Compressing objects: 100% (116/116), done.
remote: Total 156 (delta 64), reused 120 (delta 31), pack-reused 0
Receiving objects: 100% (156/156), 15.60 KiB | 0 bytes/s, done.
Resolving deltas: 100% (64/64), done.
Checking connectivity... done.
```

Then procede with the following sections:

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)

Introduction

A refresh on Docker concepts

You may want to skip this section if you have already run the introductory lab.

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 [53]:

```
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 [54]:

```
docker info
```

```
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 17
Server Version: 1.10.0-rc2
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 40
 Dirperm1 Supported: false
Execution Driver: native-0.2
Logging Driver: json-file
Plugins:
 Volume: local
 Network: bridge null host
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
Username: dockerlabs
Registry: https://index.docker.io/v1/ (https://index.docker.io/v1/)
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 [55]:

```
# Show the running containers:
docker ps

# Count the number of running containers:
echo
echo "Total number of running containers:"
docker ps | tail -n +2 | wc -l
```

```
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES

Total number of running containers:
0
```

In [56]:

```
# Show all the containers (running or stopped):
docker ps -a

# Count all the containers (running or stopped):
echo
echo "Total number of containers (running or stopped):"
docker ps -a | tail -n +2 | wc -l # Number of stopped and running containers ('tail -
```

```
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES

Total number of containers (running or stopped):
0
```

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 [57]:

```
# Show the images:
docker images

# Count the images:
echo
echo "Total number of images:"
docker images | tail -n +2 | wc -l
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
<none>              <none>              7f6ea29da43b        12 minutes
ago      689.1 MB
<none>              <none>              cf3bfbdc8b8f        12 minutes
ago      689.1 MB
lab/basic           latest              586770cdbd0a        12 minutes
ago      689.1 MB
python              latest              93049cc049a6        9 days ago
689.1 MB
python              2.7                 31093b2dabe2        9 days ago
676.1 MB
node                latest              baa18fdeb577        9 days ago
643.1 MB
golang              latest              f827671e2a60        9 days ago
725.1 MB
swarm               1.1.0-rc2           81883ac55ffe        13 days ago
18.06 MB
alpine              latest              14f89d0e6257        2 weeks ago
4.794 MB

Total number of images:
9
```

That is because there are many intermediate image layers which are not normally listed. But we can

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 [58]:

```
# Show all the image layers:
docker images -a

# Count all the image layers:
echo
echo "Total number of image layers:"

docker images -a | tail -n +2 | wc -l  # The number of image layers+1 (inc. header li
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
<none>              <none>              e09e3f64b3c4        12 minutes
ago      689.1 MB
<none>              <none>              7f6ea29da43b        12 minutes
ago      689.1 MB
<none>              <none>              d1e0972af2d6        12 minutes
ago      689.1 MB
<none>              <none>              38625d4d3cf0        12 minutes
ago      689.1 MB
<none>              <none>              5a27b12d8ff4        12 minutes
ago      689.1 MB
<none>              <none>              d02362fa5ff3        12 minutes
ago      689.1 MB
<none>              <none>              cf3bfbdc8b8f        12 minutes
ago      689.1 MB
<none>              <none>              4d5347559ce7        12 minutes
ago      689.1 MB
<none>              <none>              579cc809ae9b        13 minutes
ago      689.1 MB
lab/basic           latest              586770cdbd0a        13 minutes
ago      689.1 MB
<none>              <none>              cc80fdd4e448        13 minutes
ago      689.1 MB
python              latest              93049cc049a6        9 days ago
689.1 MB
python              2.7                 31093b2dabe2        9 days ago
676.1 MB
node                latest              baa18fdeb577        9 days ago
643.1 MB
golang              latest              f827671e2a60        9 days ago
725.1 MB
swarm               1.1.0-rc2           81883ac55ffe        13 days ago
18.06 MB
alpine              latest              14f89d0e6257        2 weeks ago
4.794 MB

Total number of image layers:
17
```

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 executable 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 [59]:

```
docker search hello-world
```

```
NAME                                      DESCRIPTION
STARS     OFFICIAL   AUTOMATED
hello-world                               Hello World! (an example of mi
nimal Docker...   47        [OK]
tutum/hello-world                         Image to test docker deploymen
ts. Has Apac...   19                    [OK]
marcells/aspnet-hello-world               ASP.NET vNext - Hello World
2                    [OK]
carinamarina/hello-world-web              A Python web app, running on p
ort 5000, wh...   1                     [OK]
bonomat/nodejs-hello-world                a simple nodejs hello world co
ntainer           1                     [OK]
vegasbrianc/docker-hello-world
1                    [OK]
carinamarina/hello-world-app              This is a sample Python web ap
plication, r...   1                     [OK]
mikelh/hello-world                        simplified hello world as dumm
y start for ...   0                     [OK]
poojathote/hello-world                    this is 3rd POC
0                    [OK]
asakaguchi/docker-nodejs-hello-world      Hello World for Docker
0                    [OK]
ileontyev81/docker-hello-world            hello world test build
0                    [OK]
alexwelch/hello-world
0                    [OK]
vasia/docker-hello-world                  rhrthrth
0                    [OK]
asakaguchi/magellan-nodejs-hello-world    Hello World for MAGELLAN
0                    [OK]
samxzxy/docker-hello-world                Automated build test docker-he
llo-world         0                     [OK]
cpro/http-hello-world                     Hello world
0                    [OK]
rcarun/hello-world
0                    [OK]
kevto/play2-docker-hello-world            Hello World application in Pla
y2 to test D...   0                     [OK]
nirmata/hello-world
0                    [OK]
n8io/hello-world                          A simple hello world node.js a
pp to test d...   0                     [OK]
wodge/docker-hello-world                  Hello World test for auto upda
te to Docker...   0                     [OK]
chalitac/hello-world                      Just Hello World
0                    [OK]
wowgroup/hello-world                      Minimal web app for testing pu
rposes            0                     [OK]
bencampbell/hello-world                   First automated build.
0                    [OK]
crccheck/hello-world                      Hello World web server in unde
r 2.5 MB          0                     [OK]
```

Let's now run that image

In [60]:

```
docker run hello-world

# Note how we see the pulling of the image if not already available locally:
```

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world

03f4658f8b78: Pulling fs layer
Digest: sha256:8be990ef2aeb16dbcb9271ddfe2610fa6658d13f6dfb8bc72074cc1c
a36966a7
Status: Downloaded newer image for hello-world:latest

Hello from Docker.
This message shows that your installation appears to be working correct
ly.

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 Hu
b.
 3. The Docker daemon created a new container from that image which run
s the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which
sent 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 accou
nt:
 https://hub.docker.com (https://hub.docker.com)

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

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 [61]:

```
docker run hello-world

# The second time there is no need to repull the image:
```

```
Hello from Docker.
This message shows that your installation appears to be working correct
ly.

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 Hu
b.
 3. The Docker daemon created a new container from that image which run
s the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which
sent 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 accou
nt:
 https://hub.docker.com (https://hub.docker.com)

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

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 [62]:

```
docker images hello-world
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
hello-world         latest              690ed74de00f        3 months ag
o        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 [63]:

```
docker history hello-world
```

```
IMAGE               CREATED             CREATED BY
SIZE                COMMENT
690ed74de00f        3 months ago        /bin/sh -c #(nop) CMD ["/hell
o"]                 0 B
<missing>           3 months ago        /bin/sh -c #(nop) COPY file:1ad
52e3eaf4327c8f   960 B
```

In [64]:

```
echo
echo "Total size (in bytes) of text in 'hello-world' image:"
docker run hello-world | wc -c
```

```
Total size (in bytes) of text in 'hello-world' image:
801
```

So we see that 801 bytes of that executable is the actual text printed ! So the real program size is roughly 160 bytes (of assembler no doubt)

Basic Docker Builds

TOP

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 new 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 builds** 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 Amazon Web Services ECS.

In [66]:

```
mkdir -p ~/test
cd    ~/test
```

In the ~/test folder create a file called Dockerfile.basic with the contents shown below (the o/p of the cat command).

For this you may use vi if you are familiar, otherwise the 'nano' text editor is recommended.

Use ctrl-W to write out the file and ctrl-X to quit the editor.

In [68]:

```
cat Dockerfile.basic
```

```
#
# Dockerfile to demonstrate the simplest build
#
FROM python

MAINTAINER "Docker Build Lab" <dockerlabs@mjbright.net>

# NOTE: all RUN commands are executed at build time,
#       look at the output of the "docker build" below and you will see
the Python version.
RUN python --version

CMD bash
```

In [69]:

```
ls -altr Dockerfile.basic
```

```
-rw-rw-r-- 1 vagrant vagrant 298 Feb  4 21:56 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 with

  -f Dockerfile.basic
- we specify the current directory as the context for the build (any ADD/COPY or Dockerfile files will be sourced from here) with

  .
- we specify the specific tag to use for the generated image as "*lab/basic*" with

  -t lab/basic

In [70]:

```
docker build -f Dockerfile.basic -t lab/basic .
```

```
Sending build context to Docker daemon 64.51 kB
Step 1 : FROM python
 ---> 93049cc049a6
Step 2 : MAINTAINER "Docker Build Lab" <dockerlabs@mjbright.net>
 ---> Using cache
 ---> cc80fdd4e448
Step 3 : RUN python --version
 ---> Using cache
 ---> 579cc809ae9b
Step 4 : CMD bash
 ---> Using cache
 ---> 586770cdbd0a
Successfully built 586770cdbd0a
```

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 [71]:

```
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 contai
ner
  --cpu-period                     Limit the CPU CFS (Completely Fair Sc
heduler) period
  --cpu-quota                      Limit the CPU CFS (Completely Fair Sc
heduler) 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 'P
ATH/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 im
age
  --pull                           Always attempt to pull a newer versio
n of the image
  -q, --quiet                      Suppress the build output and print i
mage ID on success
  --rm=true                        Remove intermediate containers after
a successful build
  --shm-size                       Size of /dev/shm, default value is 64
MB
  -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 [72]:

```
docker images --help
```

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

List images

  -a, --all          Show all images (default hides intermediate image
s)
  --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 [73]:

```
docker images lab/basic
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
lab/basic           latest              586770cdbd0a        17 minutes
ago      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 [74]:

```
cd ~/test/
```

Now edit the Dockerfile.basic to have the contents as shown below (the o/p of the cat command).

In [76]:

```
cat Dockerfile.basic
```

```
#
# Dockerfile to demonstrate the simplest build
#
FROM python

MAINTAINER "Docker Build Lab" <dockerlabs@mjbright.net>

# NOTE: all RUN commands are executed at build time,
#       look at the output of the "docker build" below and you will see
the Python version.
RUN python --version

# The addition of the following line will invalidate the cache, all fol
lowing lines will
# be built, the cache will not be used:
ENV NEWVAR=somevalue

CMD bash
```

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 [77]:

```
docker build -f Dockerfile.basic -t lab/basic .
```

```
Sending build context to Docker daemon 77.31 kB
Step 1 : FROM python
 ---> 93049cc049a6
Step 2 : MAINTAINER "Docker Build Lab" <dockerlabs@mjbright.net>
 ---> Using cache
 ---> cc80fdd4e448
Step 3 : RUN python --version
 ---> Using cache
 ---> 579cc809ae9b
Step 4 : ENV NEWVAR somevalue
 ---> Running in 4757f402c5fb
 ---> 554472e06c33
Removing intermediate container 4757f402c5fb
Step 5 : CMD bash
 ---> Running in 7192072748cf
 ---> 20558c927680
Removing intermediate container 7192072748cf
Successfully built 20558c927680
```

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 [78]:

```
docker build --no-cache -f Dockerfile.basic -t lab/basic .
```

```
Sending build context to Docker daemon 77.31 kB
Step 1 : FROM python
 ---> 93049cc049a6
Step 2 : MAINTAINER "Docker Build Lab" <dockerlabs@mjbright.net>
 ---> Running in 5b0210d59236
 ---> e2395c2aa7a5
Removing intermediate container 5b0210d59236
Step 3 : RUN python --version
 ---> Running in a943c426a9ee
Python 3.5.1
 ---> a179256b8bf0
Removing intermediate container a943c426a9ee
Step 4 : ENV NEWVAR somevalue
 ---> Running in 9c899c91f55b
 ---> 12834699ace7
Removing intermediate container 9c899c91f55b
Step 5 : CMD bash
 ---> Running in 43cd245b9527
 ---> b1c72c79ea90
Removing intermediate container 43cd245b9527
Successfully built b1c72c79ea90
```

In [79]:

```
docker images lab/basic
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
lab/basic           latest              b1c72c79ea90        1 seconds a
go       689.1 MB
```

Creating small images

TOP

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

So first let's build our static binary

In [80]:

```
cd ~/src/createTinyC/

# For RHEL/Fedora/Centos only:
# First we must install *glibc-static*
#yum install -y glibc-static

gcc -static helloWorld.c -o helloWorld

ls -alh helloWorld
```

```
-rwxrwxr-x 1 vagrant vagrant 857K Feb  4 22:02 helloWorld
```

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

Now let's build our Docker image containing this binary.

You will need to recreate the Dockerfile as follows:

In [81]:

```
cat Dockerfile
```

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

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

In [82]:

```
docker build -t lab/c_prog .
```

```
Sending build context to Docker daemon 882.2 kB
Step 1 : FROM scratch
 --->
Step 2 : MAINTAINER "Docker Build Lab" <dockerlabs@mjbright.net>
 ---> Running in 1d3908c955e9
 ---> 5b37a53abbef
Removing intermediate container 1d3908c955e9
Step 3 : ADD ./helloWorld /helloWorld
 ---> f13c22c24ca5
Removing intermediate container 4d8a200f2f5f
Step 4 : CMD /helloWorld
 ---> Running in 5e5c69595417
 ---> 45069c143064
Removing intermediate container 5e5c69595417
Successfully built 45069c143064
```

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 [83]:

```
docker images lab/c_prog
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
lab/c_prog          latest              45069c143064        5 seconds a
go       877.2 kB
```

And now let's run that image

In [84]:

```
docker run lab/c_prog
```

```
Hello World!!
```

In [85]:

```
docker history lab/c_prog
```

```
IMAGE               CREATED             CREATED BY
SIZE                COMMENT
45069c143064        8 seconds ago       /bin/sh -c #(nop) CMD ["/helloW
orld"]           0 B
f13c22c24ca5        8 seconds ago       /bin/sh -c #(nop) ADD file:474e
67cc2feb0f0110   877.2 kB
5b37a53abbef        8 seconds ago       /bin/sh -c #(nop) MAINTAINER "D
ocker 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.

NOTE: We will do this **without** 'installing a Go compiler'

In [86]:

```
cd ~/src/createTinyGo
cat Dockerfile
```

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

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

NOW we invoke the golang container to build our go source code.

The following docker run

- mounts the current directory ($PWD) as /go within the container
- launches a container of the **golang** image which contains the go compiler
- invokes the command "go build -v hello" on the container to build the sources for the "hello.go" code.

The hello.go code is located under src/hello/hello.go.

This is a Go convention.

NOTE: The important thing to note here is that the compiler is within the image. We did not need to install a native Go compiler, we used an image which contains the compiler and by mounting the current directory the container can read the source code and write the executable outside the container. This is a nice pattern of providing a tool within a container.

In [87]:

```
docker run -it -v $PWD:/go golang go build hello

ls -l hello
```

```
-rwxr-xr-x 1 root root 2367272 Feb  4 22:03 hello
```

Now we can build our image including this static binary.

In [88]:

```
docker build -t lab/go-hello .
```

```
Sending build context to Docker daemon 2.377 MB
Step 1 : FROM scratch
 --->
Step 2 : MAINTAINER "Docker Build Lab" <dockerlabs@mjbright.net>
 ---> Using cache
 ---> 5b37a53abbef
Step 3 : ADD ./hello /hello
 ---> 368532705c93
Removing intermediate container 8303b13a07f5
Step 4 : CMD /hello
 ---> Running in 33e78ca46636
 ---> 75404e153500
Removing intermediate container 33e78ca46636
Successfully built 75404e153500
```

In [89]:

```
docker images lab/*
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
lab/go-hello        latest              75404e153500        2 seconds a
go       2.367 MB
lab/c_prog          latest              45069c143064        24 seconds
ago      877.2 kB
lab/basic           latest              b1c72c79ea90        40 seconds
ago      689.1 MB
```

Creating a toolset Docker image containing several executables

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' but by using the official *'golang'* image which includes the Go compiler.

In [90]:

```
cd ~/src/toolset

cp ../createTinyC/helloWorld   helloWorld
cp ../createTinyGo/hello       helloWorldGo

ls -altr
```

```
total 3192
-rw-rw-r--  1 vagrant vagrant      68 Feb  4 21:55 helloWorld.c
-rw-rw-r--  1 vagrant vagrant     181 Feb  4 21:55 Dockerfile
-rwxrwxr-x  1 vagrant vagrant     333 Feb  4 21:55 createTinyDockerImag
e.sh
drwxrwxr-x 10 vagrant vagrant    4096 Feb  4 21:55 ..
-rwxrwxr-x  1 vagrant vagrant  877152 Feb  4 22:04 helloWorld
-rwxr-xr-x  1 vagrant vagrant 2367272 Feb  4 22:04 helloWorldGo
drwxrwxr-x  2 vagrant vagrant    4096 Feb  4 22:04 .
```

Create the Dockerfile with the following contents

In [92]:

```
cat Dockerfile
```

```
FROM hello-world
MAINTAINER "Docker Build Lab" <dockerlabs@mjbright.net>

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

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

In [93]:

```
docker build -t lab/toolset ./
```

```
Sending build context to Docker daemon  3.25 MB
Step 1 : FROM hello-world
 ---> 690ed74de00f
Step 2 : MAINTAINER "Docker Build Lab" <dockerlabs@mjbright.net>
 ---> Running in af48eaf8d716
 ---> d65e580bff00
Removing intermediate container af48eaf8d716
Step 3 : ADD ./helloWorld /helloWorld
 ---> c15402facbac
Removing intermediate container 3ad406b7af86
Step 4 : CMD /helloWorld
 ---> Running in 2af85037391e
 ---> 822cfe5f6bb9
Removing intermediate container 2af85037391e
Step 5 : ADD ./helloWorldGo /helloWorldGo
 ---> 0e4cf415e1be
Removing intermediate container 8dc53041a21e
Step 6 : CMD /helloWorldGo
 ---> Running in 6d1bbf2975d0
 ---> 06585695f835
Removing intermediate container 6d1bbf2975d0
Successfully built 06585695f835
```

If we look at the history of this image we can see the different executables and CMDs which have been added including the original hello-world image.

In [94]:

```
docker history lab/toolset
```

```
IMAGE               CREATED             CREATED BY
SIZE                COMMENT
06585695f835        About a minute ago   /bin/sh -c #(nop) CMD ["/hello
WorldGo"]         0 B
0e4cf415e1be        About a minute ago   /bin/sh -c #(nop) ADD file:e82
75a9025432fdf2e   2.367 MB
822cfe5f6bb9        About a minute ago   /bin/sh -c #(nop) CMD ["/hello
World"]           0 B
c15402facbac        About a minute ago   /bin/sh -c #(nop) ADD file:474
e67cc2feb0f0110   877.2 kB
d65e580bff00        About a minute ago   /bin/sh -c #(nop) MAINTAINER
"Docker Build La   0 B
690ed74de00f        3 months ago         /bin/sh -c #(nop) CMD ["/hell
o"]                0 B
<missing>           3 months ago         /bin/sh -c #(nop) COPY file:1a
d52e3eaf4327c8f   960 B
```

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

If we don't specify the command, the last (first in the history list) will be run (so /helloWorldGo in this case)

In [95]:

```
docker run lab/toolset
```

```
Hello world from Go !!
```

Or we can explicitly choose the executable to be run.

In [96]:

```
docker run lab/toolset /hello
```

```
Hello from Docker.
This message shows that your installation appears to be working correct
ly.

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 Hu
b.
 3. The Docker daemon created a new container from that image which run
s the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which
sent 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 accou
nt:
 https://hub.docker.com (https://hub.docker.com)

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

In [97]:

```
docker run lab/toolset /helloWorld
```

```
Hello World!!
```

In [98]:

```
docker run lab/toolset /helloWorldGo
```

```
Hello world from Go !!
```

We have seen how we can combine several executables in an image, and we can imagine creating a toolset container in this way (with some more useful executable tools!)

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,

    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 [99]:

```
docker tag lab/toolset:latest dockerlabs/toolset:userN
docker images */toolset
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
dockerlabs/toolset  userN               06585695f835        2 minutes
ago       3.245 MB
lab/toolset         latest              06585695f835        2 minutes
ago       3.245 MB
```

First we must login to the Docker Hub.

Ask you instructor for the password to the dockerlabs account.

In [100]:

```
docker login -u dockerlabs -p $PASSWORD -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 [101]:

```
docker push dockerlabs/toolset:userN
```

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

51a5e96117b3: Preparing
f2625ff60edb: Preparing
5f70bf18a086: Preparing
userN: digest: sha256:b906343d1505faafdb32566ce0d6dcd8d1e57d23041af7506
40573baeb7c28d4 size: 4623
```

NOTE: The docker search command is not very useful.

and the below command doesn't show us the tags ... and so we don't know if the below image is tagged user1, user2, ...

In [102]:

```
docker search dockerlabs/
```

```
NAME                  DESCRIPTION   STARS     OFFICIAL   AUTOMATED
dockerlabs/toolset                  0
```

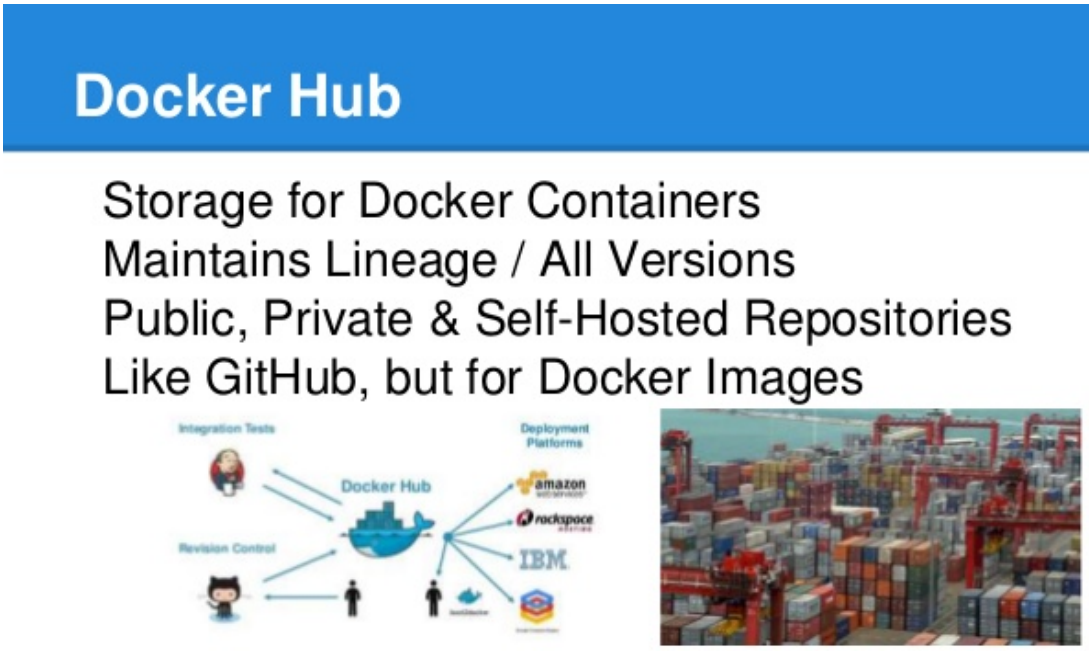
Logging on to DockerHub to see your tagged image there

So for this step, log onto DockerHub https://hub.docker.com/ (https://hub.docker.com/)

In [103]:

```
# Ignore this line: it is just to display the image below

curl -s 'http://image.slidesharecdn.com/dockerdemystifiedforsbjug-150918181554-lva1-a
```



As dockerlabs (dockerlabs AT mjbright.net) with the appropriate password (ask your instructor)

Once logged you should see the dockerlabs/toolset listed, otherwise you can search for it.

Click on the dockerlabs/toolset (https://hub.docker.com/r/dockerlabs/toolset/) link, then on the Tags (https://hub.docker.com/r/dockerlabs/toolset/tags/) link and you should now see your tagged image there.

Remove any running *'dockerlabs/toolset'* containers

We do this step to make sure we can easily delete your local dockerlabs/toolset:userN image.

These steps could be done by hand through use of 'docker ps' and 'docker ps -a' and picking containers ids corresponding to 'dockerlabs/toolset' containers to use with 'docker stop' and 'docker rm' commands.

The below expressions do this automatically for us.

In [104]:

```
IMAGE_NAME=dockerlabs/toolset

echo; echo "Currently running or stopped '$IMAGE_NAME' containers"
docker ps -a --filter=ancestor=$IMAGE_NAME

echo; echo "Stopping any running '$IMAGE_NAME' containers (so we can remove dockerlab
docker stop $(docker ps --filter=ancestor=$IMAGE_NAME) 2>/dev/null

echo; echo "Removing any stopped '$IMAGE_NAME' containers (so we can remove dockerlab
docker rm $(docker ps -a --filter=ancestor=$IMAGE_NAME) 2>/dev/null

echo; echo "There should be no more '$IMAGE_NAME' containers present:"
docker ps -a --filter=ancestor=$IMAGE_NAME
```

```
Currently running or stopped 'dockerlabs/toolset' containers
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES

Stopping any running 'dockerlabs/toolset' containers (so we can remove
dockerlabs/ image)

Removing any stopped 'dockerlabs/toolset' containers (so we can remove
dockerlabs/ image)

There should be no more 'dockerlabs/toolset' containers present:
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
```

In [105]:

```
docker images dockerlabs/*
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
dockerlabs/toolset  userN               06585695f835        11 minutes
ago      3.245 MB
```

Note that the following rmi command 'Untags' the image.

This is because it is the same - has the same image id - as our original 'lab/toolset' image.

Removing the dockerlabs/toolset image does not remove the identical 'lab/toolset' image but removes the 'dockerlabs/toolset' tag.

In [106]:

```
docker rmi dockerlabs/toolset:userN
```

```
Untagged: dockerlabs/toolset:userN
```

In [107]:

```
docker images dockerlabs/*
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
```

As we have removed ('untagged') the dockerlabs/toolset image, the following run command will download it from the Docker Hub

In [108]:

```
docker run dockerlabs/toolset:userN
```

```
Unable to find image 'dockerlabs/toolset:userN' locally
userN: Pulling from dockerlabs/toolset

03f4658f8b78: Already exists
a3ed95caeb02: Already exists
43895fb4ff42: Already exists
Digest: sha256:b906343d1505faafdb32566ce0d6dcd8d1e57d23041af750640573ba
eb7c28d4
Status: Downloaded newer image for dockerlabs/toolset:userN
Hello world from Go !!
```

In [109]:

```
docker images dockerlabs/*
```

```
REPOSITORY           TAG                 IMAGE ID            CREATED
SIZE
dockerlabs/toolset   userN               618d91642aab        13 minutes
ago      3.245 MB
```

In [110]:

```
docker run dockerlabs/toolset:userN /helloWorld
```

```
Hello World!!
```

In [111]:

```
docker run dockerlabs/toolset:userN /hello
```

```
Hello from Docker.
This message shows that your installation appears to be working correct
ly.

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 Hu
b.
 3. The Docker daemon created a new container from that image which run
s the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which
sent 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 accou
nt:
 https://hub.docker.com (https://hub.docker.com)

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

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 [112]:

```
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 comman
d
# - This produces less image layers (better for disk space and performa
nce)
# - This keeps image smaller by removing temporary files in the same la
yer
#     If we performed update/upgrade and then rm as a separate step the
re would
#     be an intermediate layer including those files, making the overal
l image larger.
#

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

TO be completed ... !!

Using the official 'Language Stack' images

TOP

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 [113]:

```
docker pull node
```

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

03e1855d4f31: Already exists
a3ed95caeb02: Already exists
9269ba3950bb: Already exists
6ecee6444751: Already exists
7a0c192d4d25: Already exists
a3ed95caeb02: Already exists
Digest: sha256:1bdda7cdd0a8f9c44ac6f51c77de9f42ed3f62efdf557dba6bcca675
084de1bd
Status: Image is up to date for node:latest
```

In [114]:

```
docker images node
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
node                latest              baa18fdeb577        9 days ago
643.1 MB
```

In [115]:

```
docker history node
```

```
IMAGE               CREATED             CREATED BY
SIZE                COMMENT
baa18fdeb577        9 days ago          /bin/sh -c #(nop) CMD ["node"]
0 B
<missing>           9 days ago          /bin/sh -c curl -SLO "https://n
odejs.org/dist   36.39 MB
<missing>           9 days ago          /bin/sh -c #(nop) ENV NODE_VERS
ION=5.5.0        0 B
<missing>           9 days ago          /bin/sh -c #(nop) ENV NPM_CONFI
G_LOGLEVEL=inf   0 B
<missing>           9 days ago          /bin/sh -c set -ex   && for key
in     9554F0   51.75 kB
<missing>           9 days ago          /bin/sh -c apt-get update && ap
t-get install    314.7 MB
<missing>           9 days ago          /bin/sh -c apt-get update && ap
t-get install    122.6 MB
<missing>           9 days ago          /bin/sh -c apt-get update && ap
t-get install    44.3 MB
<missing>           9 days ago          /bin/sh -c #(nop) CMD ["/bin/ba
sh"]             0 B
<missing>           9 days ago          /bin/sh -c #(nop) ADD file:e5a3
d20748c5d3dd5f   125.1 MB
```

In [116]:

```
cd ~/src/nodeJS/
ls -altr
```

```
total 24
drwxrwxr-x  2 vagrant vagrant 4096 Feb  4 21:55 src
-rw-rw-r--  1 vagrant vagrant  116 Feb  4 21:55 README.md
-rw-rw-r--  1 vagrant vagrant  315 Feb  4 21:55 Dockerfile
-rwxrwxr-x  1 vagrant vagrant   78 Feb  4 21:55 build_run.sh
drwxrwxr-x 10 vagrant vagrant 4096 Feb  4 21:55 ..
drwxrwxr-x  3 vagrant vagrant 4096 Feb  4 21:55 .
```

Once again edit the Dockerfile to have the contents shown below:

In [117]:

```
cat Dockerfile
```

```
FROM node

# make the src folder available in the docker image
ADD src/ /src


WORKDIR /src

# install the dependencies from the package.json file
RUN npm install

# make port 80 available outside of the image
EXPOSE 80

# start node with the index.js file of our hello-world application
CMD ["node", "index.js"]
```

Now let's build the image

In [118]:

```
docker build -t node-hello .
```

```
Sending build context to Docker daemon 6.656 kB
Step 1 : FROM node
 ---> baa18fdeb577
Step 2 : ADD src/ /src
 ---> 8c6618885ac8
Removing intermediate container f7c4e91886fd
Step 3 : WORKDIR /src
 ---> Running in 4c52826a30db
 ---> e873823852d1
Removing intermediate container 4c52826a30db
Step 4 : RUN npm install
 ---> Running in 7f7cc9bf2d2a
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 10:20:07 PM
npm http request GET https://registry.npmjs.org/express (https://r
istry.npmjs.org/express)
npm http 200 https://registry.npmjs.org/express (https://registry.
```

and run the image in the background, exposing port 80

In [119]:

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

```
7809662ff35d886dd33b8b21d9556e13606e85ec547f37266b2c6302208efa0f
```

Now let's use curl to access this container (default port for curl is 80)

In [120]:

```
curl http://localhost
```

```
<html><body>Hello from Node.js container 7809662ff35d</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 [121]:

```
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: sha256:4651b83dd903ce78b1c455794f63d4108d9469a6c7fe97cd07d08a77
b7e72435
Status: Image is up to date for python:latest
```

In [122]:

```
docker images python
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
python              latest              93049cc049a6        9 days ago
689.1 MB
python              2.7                 31093b2dabe2        9 days ago
676.1 MB
```

In [123]:

```
docker history python
```

```
IMAGE               CREATED             CREATED BY
SIZE                COMMENT
93049cc049a6        9 days ago          /bin/sh -c #(nop) CMD ["python
3"]                 0 B
<missing>           9 days ago          /bin/sh -c cd /usr/local/bin
&& ln -s easy_i   48 B
<missing>           9 days ago          /bin/sh -c set -ex  && gpg --ke
yserver ha.poo   81.53 MB
<missing>           9 days ago          /bin/sh -c #(nop) ENV PYTHON_PI
P_VERSION=7.1.   0 B
<missing>           9 days ago          /bin/sh -c #(nop) ENV PYTHON_VE
RSION=3.5.1      0 B
<missing>           9 days ago          /bin/sh -c #(nop) ENV GPG_KEY=9
7FC712E4C024BB   0 B
<missing>           9 days ago          /bin/sh -c #(nop) ENV LANG=C.UT
F-8              0 B
<missing>           9 days ago          /bin/sh -c apt-get purge -y pyt
hon.*            978.7 kB
<missing>           9 days ago          /bin/sh -c apt-get update && ap
t-get install    314.7 MB
<missing>           9 days ago          /bin/sh -c apt-get update && ap
t-get install    122.6 MB
<missing>           9 days ago          /bin/sh -c apt-get update && ap
t-get install    44.3 MB
<missing>           9 days ago          /bin/sh -c #(nop) CMD ["/bin/ba
sh"]             0 B
<missing>           9 days ago          /bin/sh -c #(nop) ADD file:e5a3
d20748c5d3dd5f   125.1 MB
```

In [124]:

```
docker run python python --version
```

```
Python 3.5.1
```

In [125]:

```
cd ~/src/python_flask
cat Dockerfile
```

```
FROM python:2.7

MAINTAINER "Docker Build Lab" <dockerlabs@mjbright.net>

WORKDIR /src

ADD requirements.txt /src/

RUN pip install -r requirements.txt

ADD . /src

CMD python flask_redis_app.py
```

In [126]:

```
docker build -t lab/python_flask .
```

```
Sending build context to Docker daemon 4.096 kB
Step 1 : FROM python:2.7
 ---> 31093b2dabe2
Step 2 : MAINTAINER "Docker Build Lab" <dockerlabs@mjbright.net>
 ---> Running in 5694bf90e230
 ---> b49d2d1c41e4
Removing intermediate container 5694bf90e230
Step 3 : WORKDIR /src
 ---> Running in 056b78f2c2ed
 ---> 87dbd1f3c183
Removing intermediate container 056b78f2c2ed
Step 4 : ADD requirements.txt /src/
 ---> 999c559fd889
Removing intermediate container ca91a1150061
Step 5 : RUN pip install -r requirements.txt
 ---> Running in dd908f954dc2
Collecting flask (from -r requirements.txt (line 1))
  Downloading Flask-0.10.1.tar.gz (544kB)
Collecting redis (from -r requirements.txt (line 2))
  Downloading redis-2.10.5-py2.py3-none-any.whl (60kB)
Collecting Werkzeug>=0.7 (from flask->-r requirements.txt (line 1))
  Downloading Werkzeug-0.11.3-py2.py3-none-any.whl (305kB)
Collecting Jinja2>=2.4 (from flask->-r requirements.txt (line 1))
  Downloading Jinja2-2.8-py2.py3-none-any.whl (263kB)
Collecting itsdangerous>=0.21 (from flask->-r requirements.txt (line
1))
  Downloading itsdangerous-0.24.tar.gz (46kB)
Collecting MarkupSafe (from Jinja2>=2.4->flask->-r requirements.txt (li
ne 1))
  Downloading MarkupSafe-0.23.tar.gz
Building wheels for collected packages: flask, itsdangerous, MarkupSafe
  Running setup.py bdist_wheel for flask
  Stored in directory: /root/.cache/pip/wheels/d2/db/61/cb9b80526b8f3ba
89248ec0a29d6da1bb6013681c930fca987
  Running setup.py bdist_wheel for itsdangerous
  Stored in directory: /root/.cache/pip/wheels/97/c0/b8/b37c320ff57e15f
993ba0ac98013eee778920b4a7b3ebae3cf
  Running setup.py bdist_wheel for MarkupSafe
  Stored in directory: /root/.cache/pip/wheels/94/a7/79/f79a998b64c1281
cb99fa9bbd33cfc9b8b5775f438218d17a7
Successfully built flask itsdangerous MarkupSafe
Installing collected packages: Werkzeug, MarkupSafe, Jinja2, itsdangero
us, flask, redis
Successfully installed Jinja2-2.8 MarkupSafe-0.23 Werkzeug-0.11.3 flas
k-0.10.1 itsdangerous-0.24 redis-2.10.5
You are using pip version 7.1.2, however version 8.0.2 is available.
You should consider upgrading via the 'pip install --upgrade pip' comma
nd.
 ---> a324a5623f13
Removing intermediate container dd908f954dc2
Step 6 : ADD . /src
 ---> 4e81515ecfe3
```

```
Removing intermediate container 5f39587c3c69
Step 7 : CMD python flask_redis_app.py
 ---> Running in 8f572157df8e
 ---> 4370d920749f
Removing intermediate container 8f572157df8e
Successfully built 4370d920749f
```

In [127]:

```
docker images lab/*
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
lab/python_flask    latest              4370d920749f        11 seconds
ago      682.9 MB
lab/toolset         latest              06585695f835        17 minutes
ago      3.245 MB
lab/go-hello        latest              75404e153500        19 minutes
ago      2.367 MB
lab/c_prog          latest              45069c143064        19 minutes
ago      877.2 kB
lab/basic           latest              b1c72c79ea90        19 minutes
ago      689.1 MB
```

Now let's run this container in the background

This example is incomplete ... to be done ...

In [*]:

```
docker run -d lab/python_flask
```

In [*]:

```
curl http://localhost:5000
```

Using Compose

TOP

Building complex systems with Compose

TOP

In [1]:

```
cd ~/src/compose
```

Create a docker-compose.yml specification file with the following contents

In [2]:

```
cat docker-compose.yml
```

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

  webb:
    build: ../createTinyGo
    command: /webserver
    # dockerfile: Dockerfile.webserver
    expose:
      - 80

  webc:
    image: python
    command: python3 -m http.server --bind 0.0.0.0 80
    expose:
      - 80

  haproxy:
    image: haproxy
    volumes:
     - ./haproxy:/usr/local/etc/haproxy/
    links:
     - weba
     - webb
     - webc
    ports:
     - "80:80"
     - "70:70"
    expose:
     - "80"
     - "70"
```

Let's look at the docker-compose options

In [3]:

```
docker-compose
```

```
Define and run multi-container applications with Docker.

Usage:
  docker-compose [-f=<arg>...] [options] [COMMAND] [ARGS...]
  docker-compose -h|--help

Options:
  -f, --file FILE           Specify an alternate compose file (default:
docker-compose.yml)
  -p, --project-name NAME   Specify an alternate project name (default:
directory name)
  --verbose                 Show more output
  -v, --version             Print version and exit

Commands:
  build              Build or rebuild services
  config             Validate and view the compose file
  create             Create services
  down               Stop and remove containers, networks, images, and
volumes
  events             Receive real time events from containers
  help               Get help on a command
  kill               Kill containers
  logs               View output from containers
  pause              Pause services
  port               Print the public port for a port binding
  ps                 List containers
  pull               Pulls service images
  restart            Restart services
  rm                 Remove stopped containers
  run                Run a one-off command
  scale              Set number of containers for a service
  start              Start services
  stop               Stop services
  unpause            Unpause services
  up                 Create and start containers
  version            Show the Docker-Compose version information
```

In [4]:

```
docker-compose stop
```

In [5]:

```
docker-compose rm -f
```

```
No stopped containers
```

This part of the lab requires some debugging also

In [6]:

```
docker-compose up -d
```

```
Building webb
Step 1 : FROM scratch
 --->
Step 2 : MAINTAINER "Docker Build Lab" <dockerlabs@mjbright.net>
 ---> Using cache
 ---> 5b37a53abbef
Step 3 : ADD ./hello /hello
 ---> 02bd6ef35abb
Removing intermediate container 758c46e42f42
Step 4 : CMD /hello
 ---> Running in 748c79ec42f9
 ---> e2256babf71f
Removing intermediate container 748c79ec42f9
Successfully built e2256babf71f
Creating compose_webb_1
ERROR: Container command not found or does not exist.
```

```
docker-compose logs
Attaching to compose_haproxy_1, compose_weba_1, compose_webc_1, compose_web
b_1
haproxy_1 | [ALERT] 032/221525 (1) : Could not open configuration file /us
r/local/etc/haproxy/haproxy.cfg : No such file or directory
haproxy_1 | [ALERT] 032/221646 (1) : Could not open configuration file /us
r/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 [7]:

```
docker-compose ps
```

```
     Name          Command      State     Ports
-----------------------------------------------
compose_webb_1   /webserver   Exit 127
```

In [8]:

```
docker-compose scale weba=5
```

```
npm http fetch GET https://registry.npmjs.org/express/-/express-4.1
2.0.tgz (https://registry.npmjs.org/express/-/express-4.12.0.tgz)
npm http fetch 200 https://registry.npmjs.org/cookie-signature/-/coo
kie-signature-1.0.6.tgz (https://registry.npmjs.org/cookie-signatur
e/-/cookie-signature-1.0.6.tgz)
npm http fetch 200 https://registry.npmjs.org/express/-/express-4.1
2.0.tgz (https://registry.npmjs.org/express/-/express-4.12.0.tgz)
npm http fetch 200 https://registry.npmjs.org/on-finished/-/on-finis
hed-2.2.1.tgz (https://registry.npmjs.org/on-finished/-/on-finished-
2.2.1.tgz)
npm http 200 https://registry.npmjs.org/qs (https://registry.npmjs.o
rg/qs)
npm info retry fetch attempt 1 at 10:27:37 PM
npm info attempt registry request try #1 at 10:27:37 PM
npm http fetch GET https://registry.npmjs.org/qs/-/qs-2.3.3.tgz (htt
ps://registry.npmjs.org/qs/-/qs-2.3.3.tgz)
npm http 200 https://registry.npmjs.org/finalhandler (https://regist
ry.npmjs.org/finalhandler)
npm info retry fetch attempt 1 at 10:27:37 PM
npm info attempt registry request try #1 at 10:27:37 PM
```

In [ ]:

```
docker-compose ps
```

In [ ]:

```
docker-compose up --force-recreate -d
```

In [ ]:

```
# TODO:
docker-compose events
```

In [ ]:

```
docker-compose ps
```

In [ ]:

```
#docker-compose logs
```

In [ ]:

```
# docker-compose up
# docker-compose down
# TODO: Add heterogeneous example ...
```

In [ ]:

```
```

TODO: extend to heterogeneous cases ...

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 [ ]:

```
cd /root
mkdir -p src/railsapi
cd src/railsapi
pwd
touch Dockerfile docker-compose.yml Gemfile Gemfile.lock
```

In [ ]:

```
cat > Dockerfile <<EOF

FROM ruby:2.3.0

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

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 [ ]:

```
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 [ ]:

```
docker-compose build
```

In [ ]:

```
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 cross-origi
# 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 console
  gem 'byebug'
end

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

EOF
```

In [ ]:

```
ls -altr
```

Now let's build our image

In [ ]:

```
docker-compose build
```

In [ ]:

```
docker images
```

In [ ]:

```
docker-compose run web rails-api new .
```

In [ ]:

```
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 [ ]:

```
docker-compose up web
```

In [ ]:

```
curl http://localhost:80
```

Building Docker with Docker

TOP

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 executable 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 executable 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; ./bundles/
```

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; ./bundles/
```

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/)
- Docker Compose Demo (https://github.com/vegasbrianc/docker-compose-demo)

In [ ]:

```
```