TAMBOOTCAMP-DOCKER

DOCKER LAB

Create an account on github.com

Install git on your laptop:

http://git-scm.com/downloads

Install docker on your laptop:

https://docs.docker.com/get-started/

Get an account on docker hub:

https://hub.docker.com/

PART 1

It's time to build your first container

Open a terminal on your laptop

Login to docker hub using your username and password. It should prompt you for username/password

```
% docker login
Login Succeeded
```

Let's make sure docker is working properly. Enter the following command:

```
% docker version
```

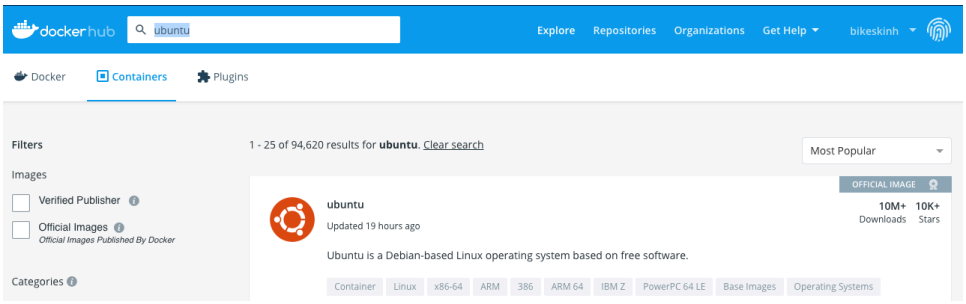
Let's get a list of available docker commands using the help utility. Enter the following command:

```
% docker help | more
```

Start with an existing "Official" image on docker hub

The pull command fetches the ubuntu linux "Official" image from docker hub and saves it to your laptop. You can then see a list of images pulled to your laptop with the "image ls" command. You can specify the version number (ubuntu:20.04) or just use (ubuntu:latest) to specify the latest version of the image

Go to the docker hub web interface (hub.docker.com), select the "Explore" tab and search for "ubuntu"



% docker pull ubuntu:latest

Using default tag: latest
latest: Pulling from library/ubuntu
ba3557a56b15: Pull complete
Digest: sha256:a75afd8b57e7f34e4dad8d65e2c7ba2e1975c795ce1ee22fa34f8cf46f96a3be
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest

% docker image ls

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	latest	28f6e2705743	5 days ago	5.61MB

Now that we have the image pulled to your laptop, let's run the image in a docker container. This command runs the ubuntu image in a container in "interactive" mode and opens a "bash" shell into the container.

% docker run -it ubuntu bash

root@364d57dac987:/#

You can now try out basic linux commands from the first tambootcamp Cloud Native session.

Open up a second terminal session and enter:
% docker container ls

You should see the running container:

```
CONTAINER ID  IMAGE     COMMAND   CREATED         STATUS         PORTS  NAMES
364d57dac987  ubuntu    "bash"    2 minutes ago   Up 2 minutes          practical_kalam
```

When you are finished, type:
exit

You now pulled your first docker image and ran it in a container!! Congrats!

Now let's delete the image from your laptop. The "rmi" command removes the image and the "-f" command forces it to be deleted.

% docker rmi -f ubuntu

```
Untagged: ubuntu:latest
Untagged: ubuntu@sha256:703218c0465075f4425e58fac086e09e1de5c340b12976ab9eb8ad26615c3715
Deleted: sha256:f63181f19b2fe819156dcb068b3b5bc036820bec7014c5f77277cfa341d4cb5e
```

==========

PART 2

1.0 Open a terminal on your laptop

2.0 Create a directory similar to:

% mkdir /Users/<your username>/github.com/<your username>

3.0 Cd into the directory:

% cd /Users/<your username>/github.com/<your username>

4.0 Clone an example project from GitHub :

git clone https://github.com/dockersamples/node-bulletin-board.git

```
Cloning into 'node-bulletin-board'...
remote: Enumerating objects: 152, done.
remote: Total 152 (delta 0), reused 0 (delta 0), pack-reused 152
Receiving objects: 100% (152/152), 190.11 KiB | 1.88 MiB/s, done.

    Resolving deltas: 100% (69/69), done.
```

This is a simple bulletin board application, written in node.js. In this example, let's imagine you wrote this app, and are now trying to containerize it.

5.0 The git clone command pulls down the source code from github and places it onto your laptop in the following directory:

/Users/dmazur/Docker/node-bulletin-board/bulletin-board-app

6.0 Change into the directory

% cd node-bulletin-board/bulletin-board-app/

```
% ls
Dockerfile      app.js      fonts       package.json      server.js
LICENSE         backend     index.html  readme.md         site.css
```

7.0 Have a look at the file called Dockerfile

Dockerfile describes how to assemble a private filesystem for a container, and can also contain some metadata describing how to run a container based on this image. The bulletin board app Dockerfile looks like this:

```
FROM node:current-slim

WORKDIR /usr/src/app
COPY package.json .
RUN npm install

EXPOSE 8080
CMD [ "npm", "start" ]

COPY . .
```

Writing a Dockerfile is the first step to containerizing an application. You can think of these Dockerfile commands as a step-by-step recipe on how to build up our image. This one takes the following steps:

Start FROM the pre-existing node:current-slim image. This is an *official image*, built by the node.js vendors and validated by Docker to be a high-quality image containing the node current-slim interpreter and basic dependencies.

Use WORKDIR to specify that all subsequent actions should be taken from the directory /usr/src/app *in your image filesystem* (never the host's filesystem).

COPY the file package.json from your host to the present location (.) in your image (so in this case, to /usr/src/app/package.json)

RUN the command npm install inside your image filesystem (which will read package.json to determine your app's node dependencies, and install them)

The next thing we need to specify is the port number that needs to be exposed. Since our app is running on port 8080, that's what we'll indicate.

COPY in the rest of your app's source code from your host to your image filesystem.

You can see that these are much the same steps you might have taken to set up and install your app on your host - but capturing these as a Dockerfile allows us to do the same thing inside a portable, isolated Docker image.

The steps above built up the filesystem of our image, but there's one more line in our Dockerfile. The CMD directive is our first example of specifying some metadata in our image that describes how to run a container based off of this image. In this case, it's saying that the containerized process that this image is meant to support is npm start.

8.0 Make sure you're in the directory node-bulletin-board/bulletin-board-app in a terminal or powershell, and build your bulletin board image:

Note: Don't forget the dot (.) after build. in the cmd below

```
% docker build -t <your dockerhub id>/bulletin-board-app .
```

9.0 You'll see Docker step through each instruction in your Dockerfile, building up your image as it goes. If successful, the build process should end with a message

```
.
.
.
Successfully built 3b9bbbfdc581
Successfully tagged bikeskinh/bulletin-board-app:latest
```

10.0 Take a look at your image

```
% docker image ls
REPOSITORY                  TAG       IMAGE ID      CREATED         SIZE
Bikeskinh/Bulletin-board-app            latest      3b9bbbfdc581   52 seconds ago   178MB
```

11.0 Run the image in a container

```
docker run -p 8000:8080 --detach --name mybb bikeskinh/bulletin-board-app:latest
```

We used a couple of common flags here:

- --publish asks Docker to forward traffic incoming on the host's port 8000, to the container's port 8080 (containers have their own private set of ports, so if we want to reach one from the network, we have to forward traffic to it in this way; otherwise, firewall rules will prevent all network traffic from reaching your container, as a default security posture).
- --detach asks Docker to run this container in the background.
- --name lets us specify a name with which we can refer to our container in subsequent commands, in this case bb.

Also notice, we didn't specify what process we wanted our container to run. We didn't have to, since we used the CMD directive when building our Dockerfile; thanks to this, Docker knows to automatically run the process npm start inside our container when it starts up.

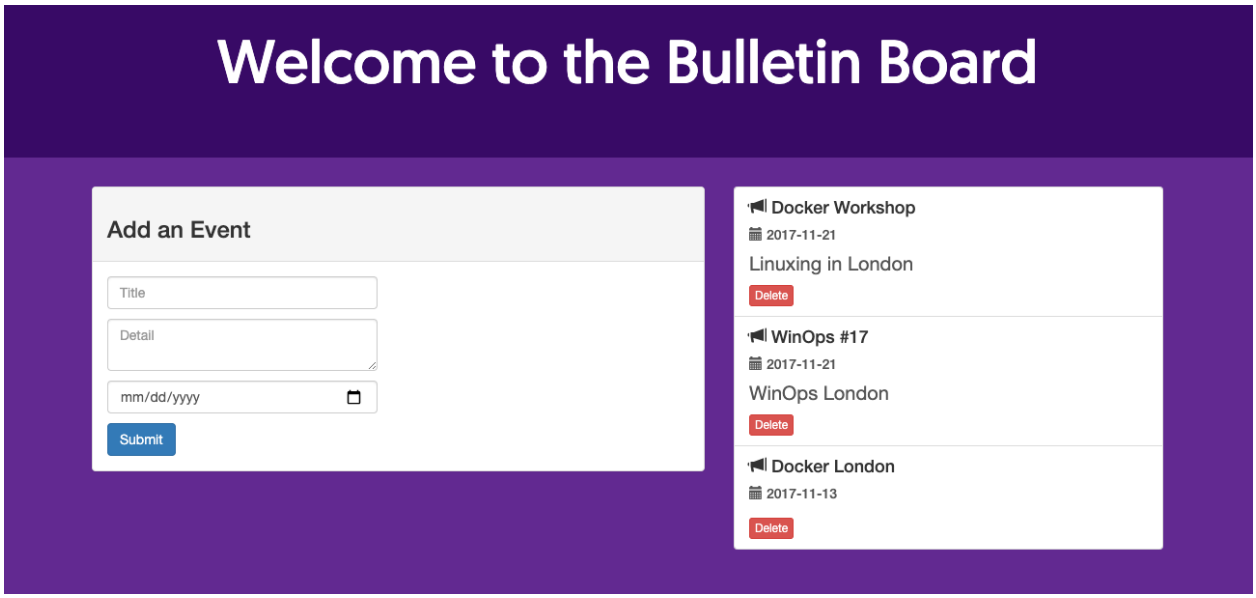
12.0 Look at the running container

Docker container ls

```
$ docker container ls
CONTAINER ID   IMAGE                                COMMAND                  CREATED         STATUS         PORTS                    NAMES
6c33a4263f57   bikeskinh/bulletin-board-app:latest   "docker-entrypoint.s…"   49 seconds ago   Up 48 seconds   0.0.0.0:8000->8080/tcp   mybb
```

13.0 Visit your application in a browser at **localhost:8000.**

You should see your bulletin board application up and running. At this step, we would normally do everything we could to ensure our container works the way we expected; now would be the time to run unit tests, for example.



14.0 Stop the running container

% Docker container stop mybb

PART 4

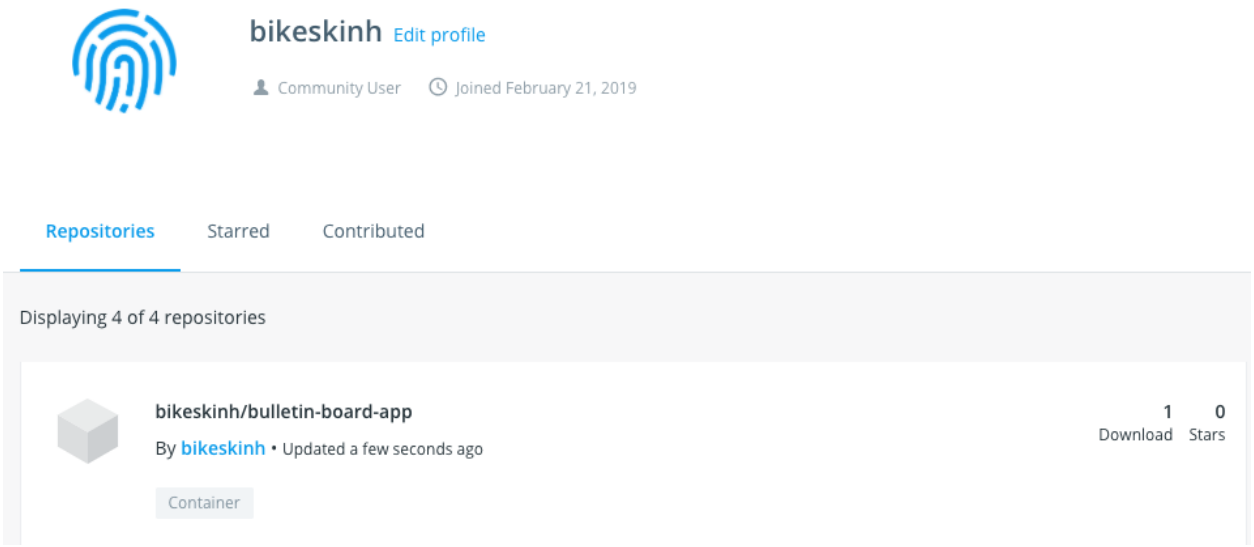
15.0 Let's push the image to docker hub

% docker login -u <username> -p <your pwd>
Login Succeeded

% docker push bikeskinh/bulletin-board-app:latest

The push refers to repository [docker.io/bikeskinh/bulletin-board-app]
fbd272a38c6f: Pushed
0e9f00803ce4: Pushed
2cfaf93fa4a9: Pushed
803fb7cc3128: Pushed
abbe5fef3a1a: Mounted from library/node
5efde78fb2a6: Mounted from library/node
f8a9a6b9dae2: Mounted from library/node
98738a12a3e5: Mounted from library/node
55d13762c439: Mounted from library/node
latest: digest: sha256:b1fb784382b05f8c9e6dcfd630e9ff41647c0b8356b842d1f5a0128a7ec94cec size: 2201

16.0 Go to hub.docker.com and see your respository



Finally, lets get rid of the container and image

% docker rm -f mybb

LAB END