Docker lab

Lab-1

Install docker on your linux virtual system
https://docs.docker.com/engine/installation/
Start hello-world container to test your installation

```
[root@server ~]# docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:c3b4ada4687bbaa170745b3e4dd8ac3f194ca95b2d0518b417fb47e5879d9b5f
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

[...]
```

Inspect images and container with docker cli

Get container active:

```
[root@server ~]# docker ps
```

Get all the container

```
[root@server ~]# docker ps -a
```

Get container images downloaded locally

```
[root@server ~]# docker images
```

Search images on docker hub

```
[root@server ~]# docker search hello-world
```

Delete hello-world container

```
[root@server ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS                        PORTS
NAMES
9fdbdd35f5a5        hello-world         "/hello"              7
minutes ago       Exited (0) 7 minutes ago
relaxed_shtern
```

Delete hello-world container

```
[root@server ~]# docker rm 9fdbdd35f5a5
9fdbdd35f5a5
```

Delete hello-world image

```
[root@server ~]# docker rmi hello-world
```

Lab-2

Let's install NGINX with the following command:

```
[root@server ~]# docker run --detach --name web_test nginx:latest
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
8d691f585fa8: Pull complete
5b07f4e08ad0: Pull complete
abc291867bca: Pull complete
Digest: sha256:922c815aa4df050d4df476e92daed4231f466acc8ee90e0e774951b0fd7195a4
Status: Downloaded newer image for nginx:latest
e6cdf5e34c0900c7db3002424bba8d88dc1c7c50c79893820a7bff0ee9c3934a
[root@server ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS               NAMES
e6cdf5e34c09        nginx:latest        "nginx -g 'daemon of…"   7 seconds ago
Up 6 seconds        80/tcp              web_test
```

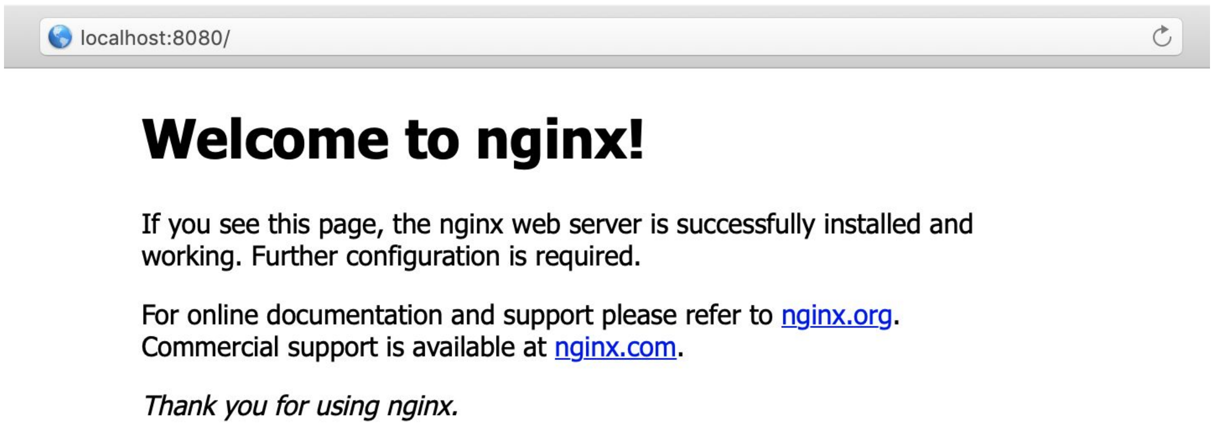
Stop docker container web_test

```
[root@server ~]# docker stop web_test
web_test
```

Start again a new nginx container binding port 80 on local system

```
[root@server ~]# docker run --detach -p 80:80 --name web nginx:latest
f8baa546333981995b13ce4d6be5478030ecbbb2fc266d4a9b12ce8199fa1077
[root@server ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                NAMES
f8baa5463339        nginx:latest        "nginx -g 'daemon of…"   44 seconds ago
Up 43 seconds       0.0.0.0:80->80/tcp   web
```

Try to open a http connection from your local browser on forwarded port 8080 (virtualbox system)



localhost:8080/

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

Inspect container log to see the connection log entry

```
[root@server ~]# docker logs web
```

```
192.168.3.2 - - [15/Nov/2019:09:41:29 +0000] "GET / HTTP/1.1" 200 612 "-"
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) AppleWebKit/605.1.15 (KHTML, like
Gecko) Version/13.0.3 Safari/605.1.15" "-"
```

Stop web container

```
[root@server ~]# docker stop web
web
[root@server ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
```

LAB-3

Start another container named web `--name web`:

```
[root@server ~]# docker run --detach --name web nginx:latest
docker: Error response from daemon: Conflict. The container name "/web" is already
in use by container
"f8baa546333981995b13ce4d6be5478030ecbbb2fc266d4a9b12ce8199fa1077". You have to
remove (or rename) that container to be able to reuse that name.
See 'docker run --help'.
```

Solution 1: rename

```
[root@server ~]# docker rename web web_old
[root@server ~]# docker run --detach --name web nginx:latest
5277054d7d7a18c37308feb6cb7e6d629cccaf55de898c4e683c91975f37b40e
```

Solution 2: stop & remove

```
[root@server ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS               NAMES
5277054d7d7a        nginx:latest        "nginx -g 'daemon of…"   53 seconds ago
Up 52 seconds       80/tcp              web
[root@server ~]# docker stop web
web
[root@server ~]# docker rm web
web
[root@server ~]# docker run --detach --name web nginx:latest
12d981cfb7288f127bc15625bfea420cac06c2f3c649e2b2c35312e29e4bae0c
[root@server ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS               NAMES
12d981cfb728        nginx:latest        "nginx -g 'daemon of…"   2 seconds ago
Up 2 seconds        80/tcp              web
```

LAB-4

Try to execute some docker command with API https://docs.docker.com/develop/sdk/examples/

LAB-5

Pull images from Docker Registry

```
[root@server ~]# docker pull httpd:latest
latest: Pulling from library/httpd
8d691f585fa8: Pull complete
8eb779d8bd44: Pull complete
574add29ec5c: Pull complete
30d7fa9ec230: Pull complete
ede292f2b031: Pull complete
Digest:
sha256:35fcab73dc9ae55db5c4ac33f5e0c7e76b7735aaddb628366bab04db6f8
ae96e
Status: Downloaded newer image for httpd:latest
docker.io/library/httpd:latest
```

Verify:

```
[root@server ~]# docker images
REPOSITORY          TAG                 IMAGE ID
CREATED             SIZE
httpd               latest              d3017f59d5e2        2
weeks ago          165MB
```

Export image to a local file

```
[root@server ~]# docker save -o httpd_latest.tar httpd:latest
[root@server ~]# ls -la httpd_latest.tar
-rw-------.  1 root root 170482176 20 nov 05.25 httpd_latest.tar
```

Remove httpd image

```
[root@server ~]# docker rmi httpd
Untagged: httpd:latest
Untagged:
httpd@sha256:35fcab73dc9ae55db5c4ac33f5e0c7e76b7735aaddb628366bab0
4db6f8ae96e
Deleted:
sha256:d3017f59d5e25daba517ac35eaf4b862dce70d2af5f27bf40bef5f936c8
b2e1f
Deleted:
sha256:c015bdd664253fc2ccdff3a425ba085e94b99ce801d6c9c5219ffaad279
362b1
Deleted:
sha256:c79505e64684e42a92353a6e3430969d8a801f327d24bdde11bd99d41cb
ef2a0
```

```
Deleted:
sha256:87158971c3545200bc870118cfa31bf5470204eea10da0b79531388b5f9
1cea2
Deleted:
sha256:0105db3a8b98aea80771956b067b64a26bd630718141bacd6feafdc5b5c
0caee
Deleted:
sha256:b67d19e65ef653823ed62a5835399c610a40e8205c16f839c5cc567954f
cf594

[root@server ~]# docker images
REPOSITORY          TAG                 IMAGE ID
CREATED             SIZE
```

Load local images from file

```
[root@server ~]# docker load -i httpd_latest.tar
b67d19e65ef6: Loading layer
[==================================================>]
72.5MB/72.5MB
71b4839cc8bf: Loading layer
[==================================================>]
2.56kB/2.56kB
e60e854dd58d: Loading layer
[==================================================>]
36.7MB/36.7MB
f69951420260: Loading layer
[==================================================>]
61.24MB/61.24MB
108b118a67d7: Loading layer
[==================================================>]
3.584kB/3.584kB
Loaded image: httpd:latest
[root@server ~]# docker images
REPOSITORY          TAG                 IMAGE ID
CREATED             SIZE
httpd               latest              d3017f59d5e2        2
weeks ago          165MB
```

Lab-6

Playing with Busybox

In this lab, we are going to run a Busybox container on our system and get a taste of the docker run command.

```
[root@server ~]# docker pull busybox
Using default tag: latest
latest: Pulling from library/busybox
0f8c40e1270f: Pull complete
Digest:
sha256:1303dbf110c57f3edf68d9f5a16c082ec06c4cf7604831669faf2c71226
0b5a0
Status: Downloaded newer image for busybox:latest
docker.io/library/busybox:latest
[root@server ~]# docker images
REPOSITORY          TAG                 IMAGE ID
CREATED             SIZE
busybox             latest              020584afccce        2
weeks ago         1.22MB
```

The pull command fetches the busybox image from the Docker registry and saves it to our system. You can use the docker images command to see a list of all images on your system.

```
[root@server ~]# docker images
REPOSITORY          TAG                 IMAGE ID
CREATED             SIZE
busybox             latest              020584afccce        2
weeks ago         1.22MB
```

Run a Docker container based on this image:

```
[root@server ~]# docker run busybox
[root@server ~]# docker ps
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS               NAMES
```

nothing happened! Is that a bug?... no!. Behind the scenes, a lot of stuff happened. When you call run, the Docker client finds the image (busybox in this case), loads up the container and then runs a command in that container. When we run docker run busybox, we didn't provide a command, so the container booted up, ran an empty command and then exited.

run this command:

```
[root@server ~]# docker run busybox echo "hello from busybox"
hello from busybox
```

In this case, the Docker client ran the **echo** command in our busybox container and then exited it. If you've noticed, all of that happened pretty quickly. Imagine booting up a virtual machine, running a command and then killing it.

```
[root@server ~]# docker ps
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS
NAMES

[root@server ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS                          PORTS
NAMES
7bdbcb110894        busybox             "echo 'hello from bu…"   2
minutes ago       Exited (0) 2 minutes ago
stupefied_bell
ec46a2d08d67        busybox             "sh"                     4
hours ago         Exited (0) 4 hours ago
naughty_nash
```

To run more than just one command in a container. Let's try that now:

```
[root@server ~]# docker run -it busybox sh
/ # ls
bin   dev   etc   home  proc  root  sys   tmp   usr   var
/ # uptime
 14:50:23 up  4:32,  0 users,  load average: 0.00, 0.01, 0.04
/ #
```

Running the run command with the -it flags attaches us to an interactive tty in the container. Now we can run as many commands in the container as we want. **Take some time to run some commands.**

Clean up all the container exited from start to now

```
[root@server ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS                              PORTS
NAMES
73fab3ef1564        busybox             "sh"
16 minutes ago      Exited (130) 58 seconds ago
priceless_lovelace
7bdbcb110894        busybox             "echo 'hello from bu…"
20 minutes ago      Exited (0) 20 minutes ago
stupefied_bell
```

```
0ac16a224243        hello-world         "/hello"                4
hours ago         Exited (0) 4 hours ago
priceless_cerf
ec46a2d08d67        busybox             "sh"                    4
hours ago         Exited (0) 4 hours ago
naughty_nash
ef86117d695d        d3017f59d5e2        "httpd-foreground"      4
hours ago         Exited (0) 4 hours ago
blissful_austin
3fb98025290b        d3017f59d5e2        "httpd-foreground"      4
hours ago         Exited (0) 4 hours ago
unruffled_mclean
[root@server ~]# docker ps -a -q -f status=exited
73fab3ef1564
7bdbcb110894
0ac16a224243
ec46a2d08d67
ef86117d695d
3fb98025290b
[root@server ~]# docker rm $(docker ps -a -q -f status=exited)
73fab3ef1564
7bdbcb110894
0ac16a224243
ec46a2d08d67
ef86117d695d
3fb98025290b
[root@server ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS               NAMES
```

Terminology

Some terminology used in this exercise on the Docker ecosystem.

Images - The blueprints of our application which form the basis of containers. In the demo above, we used the docker pull command to download the busybox image.
Containers - Created from Docker images and run the actual application. We create a container using docker run which we did using the busybox image that we downloaded. A list of running containers can be seen using the docker ps command.

Docker Daemon - The background service running on the host that manages building, running and distributing Docker containers. The daemon is the process that runs in the operating system to which clients talk to.
Docker Client - The command line tool that allows the user to interact with the daemon. More generally, there can be other forms of clients too - such as Kitematic which provide a GUI to the users.
Docker Hub - A registry of Docker images. You can think of the registry as a directory of all available Docker images. If required, one can host their own Docker registries and can use them for pulling images.

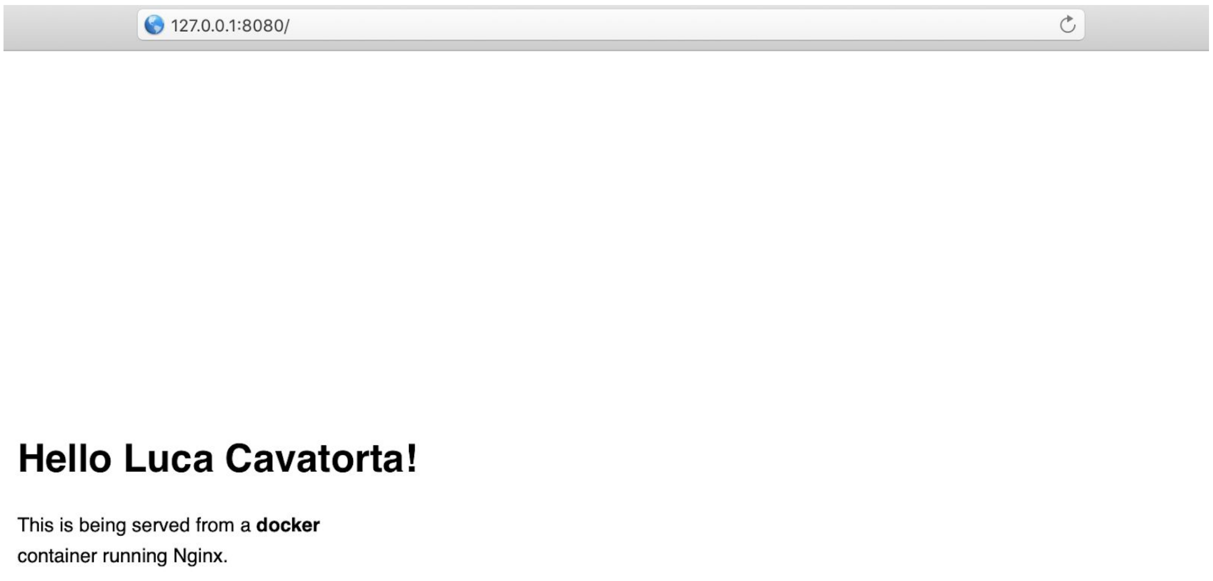
LAB-7

Run a static website in a container

The image that you are going to use is a single-page website available on the Docker Store as [dockersamples/static-site](). You can download and run the image directly in one go using docker run as follows.

```
[root@server ~]#  docker run --name static-site-2 -e AUTHOR="Your
Name" -d -p 80:80 dockersamples/static-site
fe79037a930c91b2e5b548cf369c2ee92cf6d359f0a545326cd927433464e016
[root@server ~]# docker ps
CONTAINER ID        IMAGE                       COMMAND
CREATED             STATUS              PORTS
NAMES
fe79037a930c        dockersamples/static-site   "/bin/sh -c 'cd
/usr…"   3 seconds ago       Up 2 seconds
0.0.0.0:80->80/tcp, 443/tcp   static-site-2
```

Try to open a http connection from your local browser on forwarded port 8080 (virtualbox system)



127.0.0.1:8080/

Hello Luca Cavatorta!

This is being served from a **docker** container running Nginx.

Now, let's launch a container in detached mode as shown below:

```
[root@server ~]# docker run --name static-site -e AUTHOR="Luca
Cavatorta" -d -P dockersamples/static-site
4f1d765fa9a6eec01b62f455c59199666b804cff937c553a104f180abb4a36a0
```

```
e61d12292d69556eabe2a44c16cbd54486b2527e2ce4f95438e504afb7b02810
```
In the above command:

- -d will create a container with the process detached from our terminal
- -P will publish all the exposed container ports to **random** ports on the Docker host
- -p will expose your container port selected ports on the Docker host
- **-e is how you pass environment variables to the container**
- --name allows you to specify a container name
- AUTHOR is the environment variable name and Your Name is the value that you can pass

Now you can see the ports by running the docker port command.
```
[root@server ~]# docker port static-site
443/tcp -> 0.0.0.0:32768
80/tcp -> 0.0.0.0:32769
```

If you want create a temporary bind on virtualbox to port 32769 and 32768 and try to open a local browser on forward port configured.

Clean up all
docker ps
docker stop
docker ps -a
docker rm

LAB-8

Volumes:

Bind mount volume

Create a new directory /localvolume
Add some file to /localvolume directory
for example:

```
touch /localvolume/file1
touch /localvolume/file2
```

Start a centos:8 container bind to it your `/localvolume` directory

```
[root@server ~]# docker run -it --name mycentos -v
/localvolume:/usr/localvolume:ro centos:8 bash
[root@fb6a2fd1de6d /]# cd /usr/localvolume
[root@fb6a2fd1de6d localvolume]# ls -la
file1
file2
```

Try to create a directory or a file under /usr/localvolume

```
Stop the container
```

Docker-managed volume and shared volumes

Start a centos:8 container named `mycentos2` with a new Docker volume

```
docker run -dit --name mycentos2 -v /localvolume centos:8 bash
Inspect create volume
```

Start a second centos:8 container named `mycentos3` with shared volume from `mycentos2` container

```
[root@server ~]# docker ps
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS               NAMES
30efc2065fa5        centos:8            "bash"              3
seconds ago       Up 2 seconds
mycentos3
```

```
6931f31807f4        centos:8            "bash"              2
minutes ago       Up 2 minutes
mycentos2
```

Create some file on `mycentos2` container under /localvolume

```
[root@server ~]# docker exec -it mycentos2 bash
[root@6931f31807f4 /]# cd /localvolume/
[root@6931f31807f4 localvolume]# ll
total 0
[root@6931f31807f4 localvolume]# touch file1
[root@6931f31807f4 localvolume]# ls -la
total 0
-rw-r--r--. 1 root root  0 Nov 21 10:12 file1
[root@6931f31807f4 localvolume]# exit
exit
```

Verify on mycentos3 /localvolume directory file

```
[root@server ~]# docker exec -it mycentos3 bash
[root@30efc2065fa5 /]# cd /localvolume/
[root@30efc2065fa5 localvolume]# ls -la
total 0
drwxr-xr-x. 2 root root 19 Nov 21 10:12 .
drwxr-xr-x. 1 root root 25 Nov 21 10:11 ..
-rw-r--r--. 1 root root  0 Nov 21 10:12 file1
```

Stop mycentos3 mycentos2 container and delete created docker volume

```
[root@server ~]# docker volume ls
DRIVER              VOLUME NAME
local
395b9c327d9fcad5cbea9401eb10ef28f492b3e8b7e3b79d6a7ef28f5e73494b
[root@server ~]# docker volume rm
395b9c327d9fcad5cbea9401eb10ef28f492b3e8b7e3b79d6a7ef28f5e73494b
```

If are you sure that no other volumes present are used, to delete all non used volume you can use:

```
docker volume  prune
WARNING! This will remove all local volumes not used by at least
one container.
Are you sure you want to continue? [y/N]
Deleted Volumes:
395b9c327d9fcad5cbea9401eb10ef28f492b3e8b7e3b79d6a7ef28f5e73494b
```

```
Total reclaimed space: 0B
[root@server ~]# docker volume ls
DRIVER              VOLUME NAME
```

LAB-9

Communication across links

Links allow containers to discover each other and securely transfer information about one container to another container. When you set up a link, you create a conduit between a source container and a recipient container. The recipient can then access select data about the source. To create a link, you use the `--link` flag. First, create a new container, this time one containing a database.

```
$ docker run -d --name db training/postgres
```

This creates a new container called **db** from the **training/postgres** image, which contains a **PostgreSQL** database.

Now, create a new **web** container and link it with your db container.

```
$ docker run -d -P --name web --link db:db training/webapp python
app.py
```

This links the new **web** container with the **db** container you created earlier. The --link flag takes the form:
--link <name or id>:alias

Next, inspect your linked containers with docker inspect:

```
$ docker inspect -f "{{ .HostConfig.Links }}" web

[/db:/web/db]
```

You can see that the **web** container is now linked to the **db** container **web/db**. Which allows it to access information about the db container.

```
$ docker exec -it web env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=4b3bbafd5430
TERM=xterm
DB_PORT=tcp://172.17.0.3:5432
```

```
DB_PORT_5432_TCP=tcp://172.17.0.3:5432
DB_PORT_5432_TCP_ADDR=172.17.0.3
DB_PORT_5432_TCP_PORT=5432
DB_PORT_5432_TCP_PROTO=tcp
DB_NAME=/web/db
DB_ENV_PG_VERSION=9.3
HOME=/root
```

LAB-10

docker attach

Attach local standard input, output, and error streams to a running container

Use **docker attach** to attach your terminal's standard input, output, and error (or any combination of the three) to a running container using the container's ID or name. This allows you to view its ongoing output or to control it interactively, as though the commands were running directly in your terminal.
Note: The **attach** command will display the output of the **ENTRYPOINT/CMD** process. This can appear as if the attach command is hung when in fact the process may simply not be interacting with the terminal at that time.
You can attach to the same contained process multiple times simultaneously, from different sessions on the Docker host.
To **stop** a container, use **CTRL-**c. This key sequence sends **SIGKILL** to the container. If --sig-proxy is true (the default),**CTRL-c** sends a **SIGINT** to the container. If the container was run with **-i** and **-t**, you can detach from a container and leave it running using the **CTRL-p CTRL-q** key sequence.
Note: A process running as PID 1 inside a container is treated specially by Linux: it ignores any signal with the default action. So, the process will not terminate on SIGINT or SIGTERM unless it is coded to do so.

Use user-defined bridge networks

In this example, we again start two alpine containers, but attach them to a user-defined network called alpine-net which we have already created. These containers are not connected to the default bridge network at all. We then start a third alpine container which is connected to the bridge network but not connected to alpine-net, and a fourth alpine container which is connected to both networks.

Create the alpine-net network. You do not need the --driver bridge flag since it's the default, but this example shows how to specify it.

```
$ docker network create --driver bridge alpine-net
```

List Docker's networks:

```
$ docker network ls

NETWORK ID          NAME                DRIVER              SCOPE
e9261a8c9a19        alpine-net          bridge              local
```

```
17e324f45964        bridge              bridge              local
6ed54d316334        host                host                local
7092879f2cc8        none                null                local
```

Inspect the alpine-net network. This shows you its IP address and the fact that no containers are connected to it:

```
$ docker network inspect alpine-net

[
    {
        "Name": "alpine-net",
        "Id":
"e9261a8c9a19eabf2bf1488bf5f208b99b1608f330cff585c273d39481c9b0ec"
,
        "Created": "2017-09-25T21:38:12.620046142Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

Notice that this network's gateway is **172.18.0.1**, as opposed to the default bridge network, whose gateway is **172.17.0.1 (**`docker network inspect bridge`**)**. The exact IP address may be different on your system.

Create your four containers. Notice the --network flags. You can only connect to one network during the docker run command, so you need to use docker network connect afterward to connect alpine4 to the bridge network as well.

```
$ docker run -dit --name alpine1 --network alpine-net alpine ash
$ docker run -dit --name alpine2 --network alpine-net alpine ash

$ docker run -dit --name alpine3 alpine ash

$ docker run -dit --name alpine4 --network alpine-net alpine ash

$ docker network connect bridge alpine4
```

Verify that all containers are running:

```
$ docker container ls

CONTAINER ID        IMAGE               COMMAND
CREATED              STATUS              PORTS                NAMES
156849ccd902        alpine              "ash"                41
seconds ago       Up 41 seconds                            alpine4
fa1340b8d83e        alpine              "ash"                51
seconds ago       Up 51 seconds                            alpine3
a535d969081e        alpine              "ash"                 About
a minute ago   Up About a minute                          alpine2
0a02c449a6e9        alpine              "ash"                 About
a minute ago   Up About a minute                          alpine1
```

Inspect the bridge network and the alpine-net network again:

```
$ docker network inspect bridge

[
    {
        "Name": "bridge",
        "Id":
"17e324f459648a9baaea32b248d3884da102dde19396c25b30ec800068ce6b10"
,
        "Created": "2017-06-22T20:27:43.826654485Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Containers": {

"156849ccd902b812b7d17f05d2d81532ccebe5bf788c9a79de63e12bb92fc621"
: {
                "Name": "alpine4",
                "EndpointID":
"7277c5183f0da5148b33d05f329371fce7befc5282d2619cfb23690b2adf467d"
,
                "MacAddress": "02:42:ac:11:00:03",
                "IPv4Address": "172.17.0.3/16",
                "IPv6Address": ""
            },

"fa1340b8d83eef5497166951184ad3691eb48678a3664608ec448a687b047c53"
: {
                "Name": "alpine3",
                "EndpointID":
"5ae767367dcbebc712c02d49556285e888819d4da6b69d88cd1b0d52a83af95f"
,
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade":
"true",
            "com.docker.network.bridge.host_binding_ipv4":
"0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
]
```

Containers **alpine3** and **alpine4** are connected to the bridge network.

```
$ docker network inspect alpine-net
```

```
[
    {
        "Name": "alpine-net",
        "Id":
"e9261a8c9a19eabf2bf1488bf5f208b99b1608f330cff585c273d39481c9b0ec"
,
        "Created": "2017-09-25T21:38:12.620046142Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Containers": {

"0a02c449a6e9a15113c51ab2681d72749548fb9f78fae4493e3b2e4e74199c4a"
: {
                "Name": "alpine1",
                "EndpointID":
"c83621678eff9628f4e2d52baf82c49f974c36c05cba152db4c131e8e7a64673"
,
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            },

"156849ccd902b812b7d17f05d2d81532ccebe5bf788c9a79de63e12bb92fc621"
: {
                "Name": "alpine4",
                "EndpointID":
"058bc6a5e9272b532ef9a6ea6d7f3db4c37527ae2625d1cd1421580fd0731954"
,
                "MacAddress": "02:42:ac:12:00:04",
                "IPv4Address": "172.18.0.4/16",
                "IPv6Address": ""
            },
"a535d969081e003a149be8917631215616d9401edcb4d35d53f00e75ea1db653"
: {
                "Name": "alpine2",
                "EndpointID":
"198f3141ccf2e7dba67bce358d7b71a07c5488e3867d8b7ad55a4c695ebb8740"
,
                "MacAddress": "02:42:ac:12:00:03",
                "IPv4Address": "172.18.0.3/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

Containers **alpine1**, **alpine2**, and **alpine4** are connected to the alpine-net network.

On user-defined networks like alpine-net, containers can not only communicate by IP address, but can also resolve a container name to an IP address. This capability is called automatic service discovery. Let's connect to alpine1 and test this out. alpine1 should be able to resolve alpine2 and alpine4 (and alpine1, itself) to IP addresses.

```
$ docker container attach alpine1

# ping -c 2 alpine2

PING alpine2 (172.18.0.3): 56 data bytes
64 bytes from 172.18.0.3: seq=0 ttl=64 time=0.085 ms
64 bytes from 172.18.0.3: seq=1 ttl=64 time=0.090 ms

--- alpine2 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.085/0.087/0.090 ms

# ping -c 2 alpine4

PING alpine4 (172.18.0.4): 56 data bytes
64 bytes from 172.18.0.4: seq=0 ttl=64 time=0.076 ms
64 bytes from 172.18.0.4: seq=1 ttl=64 time=0.091 ms

--- alpine4 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.076/0.083/0.091 ms
```

```
# ping -c 2 alpine1

PING alpine1 (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.026 ms
64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.054 ms

--- alpine1 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.026/0.040/0.054 ms
```

From alpine1, you should not be able to connect to alpine3 at all, since it is not on the alpine-net network.

```
# ping -c 2 alpine3

ping: bad address 'alpine3'
```

Not only that, but you can’t connect to alpine3 from alpine1 by its IP address either. Look back at the docker network inspect output for the bridge network and find alpine3’s IP address: 172.17.0.2 Try to ping it.

```
# ping -c 2 172.17.0.2

PING 172.17.0.2 (172.17.0.2): 56 data bytes

--- 172.17.0.2 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
```

Detach from alpine1 using detach sequence, **CTRL + p CTRL + q (hold down CTRL and type p followed by q)**.

Remember that alpine4 is connected to both the default bridge network and alpine-net. It should be able to reach all of the other containers. However, you will need to address alpine3 by its IP address. Attach to it and run the tests.

```
$ docker container attach alpine4

# ping -c 2 alpine1

PING alpine1 (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.074 ms
64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.082 ms

--- alpine1 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.074/0.078/0.082 ms
```

```
# ping -c 2 alpine2

PING alpine2 (172.18.0.3): 56 data bytes
64 bytes from 172.18.0.3: seq=0 ttl=64 time=0.075 ms
64 bytes from 172.18.0.3: seq=1 ttl=64 time=0.080 ms

--- alpine2 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.075/0.077/0.080 ms

# ping -c 2 alpine3
ping: bad address 'alpine3'

# ping -c 2 172.17.0.2

PING 172.17.0.2 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.089 ms
64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.075 ms

--- 172.17.0.2 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.075/0.082/0.089 ms

# ping -c 2 alpine4

PING alpine4 (172.18.0.4): 56 data bytes
64 bytes from 172.18.0.4: seq=0 ttl=64 time=0.033 ms
64 bytes from 172.18.0.4: seq=1 ttl=64 time=0.064 ms

--- alpine4 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.033/0.048/0.064 ms
```

As a final test, make sure your containers can all connect to the internet by pinging google.com. You are already attached to alpine4 so start by trying from there. Next, detach from alpine4 and connect to alpine3 (which is only attached to the bridge network) and try again. Finally, connect to alpine1 (which is only connected to the alpine-net network) and try again.

```
# ping -c 2 google.com

PING google.com (172.217.3.174): 56 data bytes
64 bytes from 172.217.3.174: seq=0 ttl=41 time=9.778 ms
64 bytes from 172.217.3.174: seq=1 ttl=41 time=9.634 ms
```

```
--- google.com ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 9.634/9.706/9.778 ms

CTRL+p CTRL+q

$ docker container attach alpine3

# ping -c 2 google.com

PING google.com (172.217.3.174): 56 data bytes
64 bytes from 172.217.3.174: seq=0 ttl=41 time=9.706 ms
64 bytes from 172.217.3.174: seq=1 ttl=41 time=9.851 ms

--- google.com ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 9.706/9.778/9.851 ms

CTRL+p CTRL+q

$ docker container attach alpine1

# ping -c 2 google.com

PING google.com (172.217.3.174): 56 data bytes
64 bytes from 172.217.3.174: seq=0 ttl=41 time=9.606 ms
64 bytes from 172.217.3.174: seq=1 ttl=41 time=9.603 ms

--- google.com ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 9.603/9.604/9.606 ms

CTRL+p CTRL+q
```
Stop and remove all containers and the alpine-net network.

```
$ docker container stop alpine1 alpine2 alpine3 alpine4

$ docker container rm alpine1 alpine2 alpine3 alpine4

$ docker network rm alpine-net
```

LAB-11

docker **commit** [OPTIONS] CONTAINER [REPOSITORY[:TAG]]

It can be useful to commit a container's file changes or settings into a new image. This allows you to debug a container by running an interactive shell, or to export a working dataset to another server. Generally, it is better to use **Dockerfiles** to manage your images in a documented and maintainable way

ENTRYPOINT:

ENTRYPOINT has two forms:

The exec form, which is the preferred form:
```
ENTRYPOINT ["executable", "param1", "param2"]
```
The shell form:
```
ENTRYPOINT command param1 param2
```

Command line arguments to docker run <image> will be appended after all elements in an exec form ENTRYPOINT, and will override all elements specified using CMD. This allows arguments to be passed to the entry point, i.e., **docker run <image> -d** will pass the **-d** argument to the **entrypoint**. You can override the ENTRYPOINT instruction using the docker run --entrypoint flag.

CMD
The CMD instruction has three forms:

- CMD ["executable","param1","param2"] (exec form, this is the preferred form)
- CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
- CMD command param1 param2 (shell form)

There can only be one CMD instruction. The main purpose of a CMD is to provide defaults for an executing container. These defaults can include an executable, or they can omit the executable, in which case you must specify an ENTRYPOINT instruction as well.
If CMD is used to provide default arguments for the ENTRYPOINT instruction, both the CMD and ENTRYPOINT instructions should be specified.

Start a new container based on centos:8 images to start a bash as entrypoint

```
[root@server ~]# docker run -dit centos:8 bash

465b93c17a36ec17cc54d053c41d27bb5822e1483d7facbb8c598f47e3385e66
```

```
[root@server ~]# docker ps

CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES

465b93c17a36        centos:8            "bash"              5 seconds ago       Up
4 seconds                             strange_khorana
```

Commit your container

```
[root@server ~]# docker commit 465b93c17a36
class/testimage:version1

sha256:61b13e14d2377d6be7d850b7af1b0b4268045680149151f25313efa076d
05050

root@server ~]# docker images

REPOSITORY                 TAG                 IMAGE ID
CREATED              SIZE

class/testimage            version1            61b13e14d237
28 seconds ago       215MB
```

Commit a container with new configuration

```
[root@server ~]# docker inspect -f "{{ .Config.Env }}" 465b93c17a36

[PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
]

[root@server ~]# docker commit --change "ENV AUTHOR=luca"
465b93c17a36  class/testimage:version2
sha256:f6e9d61b16cffaa10ece2afe1c73982edec0e5c1fcf9212014f0741ad1c6e
50c
[root@server ~]# docker images
REPOSITORY                 TAG                 IMAGE ID            CREATED
SIZE
class/testimage            version2            f6e9d61b16cf        4 seconds ago
215MB
class/testimage            version1            61b13e14d237        4 minutes ago
215MB
```

```
[root@server ~]# docker inspect -f "{{ .Config.Env }}"
f6e9d61b16cffaa10ece2afe1c73982edec0e5c1fcf9212014f0741ad1c6e50c

[PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
AUTHOR=luca]
```

Commit a container with new CMD and ENTRYPOINT instructions

```
[root@server ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
465b93c17a36        centos:8            "bash"              12 minutes ago
Up 12 minutes                           strange_khorana

[root@server ~]# docker commit --change='ENTRYPOINT ["echo"]'
--change='CMD ["hello", "world"]' 465b93c17a36
class/testimage:version3
sha256:2cc2d990114e577d92c662ed5a325a976732def93d5d31d0924cd032600
6f82e
[root@server ~]# docker images
REPOSITORY                  TAG                 IMAGE ID
CREATED             SIZE
class/testimage             version3            2cc2d990114e
7 seconds ago       215MB
class/testimage             version2            f6e9d61b16cf
8 minutes ago       215MB
class/testimage             version1            61b13e14d237
```

Run your version3 container

```
[root@server ~]# docker run class/testimage:version3
hello world
```

LAB-12

Docker Images

The goal of this lab is to create a Docker image which will run a Flask app

Create a Python Flask app that displays random cat pix

For the purposes of this exercise, we've a fun little Python Flask app that displays a random cat `.gif` every time it is loaded.

Start by creating a directory called `flask-app` where we'll create the following files:

- app.py
- requirements.txt
- templates/index.html
- Dockerfile

Make sure to cd `flask-app` before you start creating the files, because you don't want to start adding a whole bunch of other random files to your image.

app.py

Create the app.py with the following content:

```
from flask import Flask, render_template
import random

app = Flask(__name__)

# list of images
images = [

"https://www.animatedimages.org/data/media/209/animated-cat-image-0072.gif",
"https://www.animatedimages.org/data/media/209/animated-cat-image-0056.gif",
"https://www.animatedimages.org/data/media/209/animated-cat-image-0394.gif",
"https://www.animatedimages.org/data/media/209/animated-cat-image-0338.gif",
"https://www.animatedimages.org/data/media/209/animated-cat-image-0058.gif",
“https://www.animatedimages.org/data/media/209/animated-cat-image-0184.gif”,
“https://www.animatedimages.org/data/media/209/animated-cat-image-0459.gif”
]

@app.route('/')
```

```
def index():
    url = random.choice(images)
    return render_template('index.html', url=url)

if __name__ == "__main__":
    app.run(host="0.0.0.0")
```

requirements.txt

In order to install the Python modules required for our app, we need to create a file called requirements.txt and add the following line to that file:

```
Flask==0.10.1
```

templates/index.html

Create a directory called templates and create an `index.html` file in that directory with the following content in it:

```
<html>
  <head>
    <style type="text/css">
      body {
        background: black;
        color: white;
      }
      div.container {
        max-width: 500px;
        margin: 100px auto;
        border: 20px solid white;
        padding: 10px;
        text-align: center;
      }
      h4 {
        text-transform: uppercase;
      }
    </style>
  </head>
  <body>
    <div class="container">
      <h4>Cat Gif of the day</h4>
      <img src="{{url}}" />
      <br>
    </div>
  </body>
</html>
```

Write a Dockerfile

We want to create a Docker image with this web app. As mentioned above, all user images are based on a base image. Since our application is written in Python, we will build our own Python image based on Alpine. We'll do that using a Dockerfile.
A Dockerfile is a text file that contains a list of commands that the Docker daemon calls while creating an image. The Dockerfile contains all the information that Docker needs to know to run the app — a base Docker image to run from, location of your project code, any dependencies it has, and what commands to run at start-up. It is a simple way to automate the image creation process. The best part is that the commands you write in a Dockerfile are almost identical to their equivalent Linux commands. This means you don't really have to learn new syntax to create your own Dockerfiles.

1. Create a file called Dockerfile, and add content to it as described below.

We'll start by specifying our base image, using the FROM keyword:

```
FROM alpine:3.5
```

2. The next step usually is to write the commands of copying the files and installing the dependencies. But first we will install the Python pip package to the alpine linux distribution. This will not just install the pip package but any other dependencies too, which includes the python interpreter. Add the following RUN command next:

```
RUN apk add --update py2-pip
```

3. Let's add the files that make up the Flask Application.

Install all Python requirements for our app to run. This will be accomplished by adding the lines:

```
COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt
```

Copy the files you have created earlier into our image by using COPY command.

```
COPY app.py /usr/src/app/
COPY templates/index.html /usr/src/app/templates/
```

4. Specify the port number which needs to be exposed. Since our flask app is running on `5000` that's what we'll expose.

```
EXPOSE 5000
```

5. The last step is the command for running the application which is simply - `python ./app.py`. Use the CMD command to do that:

```
CMD ["python", "/usr/src/app/app.py"]
```

The primary purpose of `CMD` is to tell the container which command it should run by default when it is started.

6. Verify your Dockerfile.

Our `Dockerfile` is now ready. This is how it looks:

```
# our base image
FROM alpine:3.51

# Install python and pip
RUN apk add --update py2-pip

# install Python modules needed by the Python app
COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt

# copy files required for the app to run
COPY app.py /usr/src/app/
COPY templates/index.html /usr/src/app/templates/

# tell the port number the container should expose
EXPOSE 5000

# run the application
CMD ["python", "/usr/src/app/app.py"]
```

Build the image

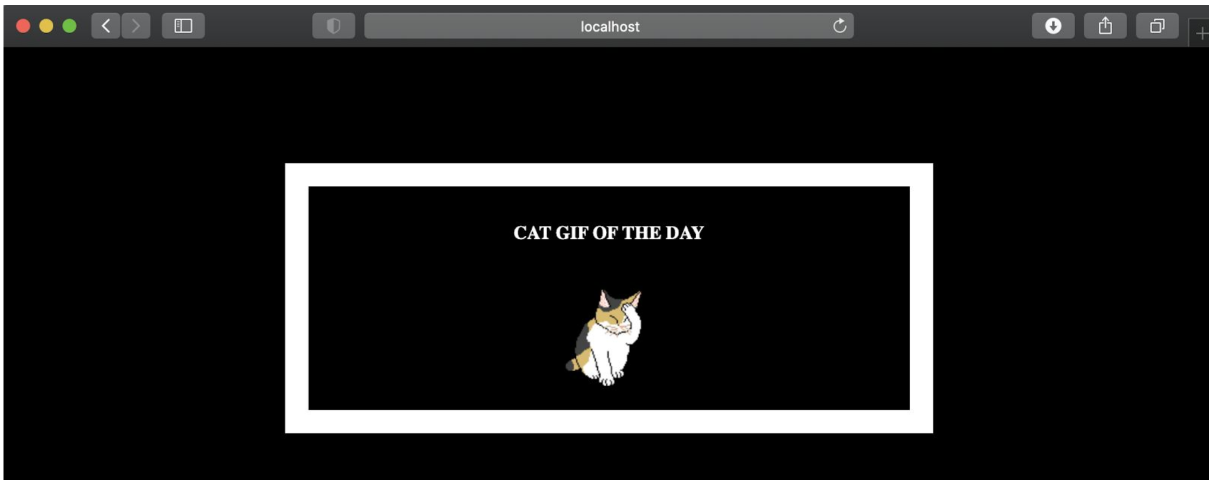
Now that you have your `Dockerfile`, you can build your image. The docker build command does the heavy-lifting of creating a docker image from a Dockerfile.
The `docker build` command is quite simple - it takes an optional tag name with the -t flag, and the location of the directory containing the Dockerfile - the **.** indicates the current directory:

```
[root@server flask-app]# docker build -t lcavatorta/myfirstapp
.
Sending build context to Docker daemon   7.68kB
Step 1/8 : FROM alpine:3.5
3.5: Pulling from library/alpine
8cae0e1ac61c: Pull complete
Digest:
sha256:66952b313e51c3bd1987d7c4ddf5dba9bc0fb6e524eed2448fa6602
46b3e76ec
Status: Downloaded newer image for alpine:3.5
 ---> f80194ae2e0c
Step 2/8 : RUN apk add --update py2-pip
 ---> Running in 1fb2f28d3369
fetch
http://dl-cdn.alpinelinux.org/alpine/v3.5/main/x86_64/APKINDEX
.tar.gz
fetch
http://dl-cdn.alpinelinux.org/alpine/v3.5/community/x86_64/APK
INDEX.tar.gz
(1/12) Installing libbz2 (1.0.6-r5)
(2/12) Installing expat (2.2.0-r1)
[..]
Successfully built e78e604f9de2
Successfully tagged lcavatorta/myfirstapp:latest

[root@server flask-app]# docker images
REPOSITORY                  TAG                 IMAGE ID
CREATED             SIZE
lcavatorta/myfirstapp       latest              e78e604f9de2
11 seconds ago      57MB
```

Run your image

```
[root@server flask-app]# docker run -p 80:5000 --name
myfirstapp lcavatorta/myfirstapp
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Open a http connection from your local browser on forwarded port 8080 (virtualbox system)



Refresh the page to get a new random gif image!!

LAB-13

Create your “hello world” image

Creating a directory called `myhelloworld` where we'll create the following files:

- start.sh
- Dockerfile

Create the **start.sh** with the following content:

```
#!/bin/sh
echo "Hello $AUTHOR"
```

N.B. Make sure to add the execution permission to the script start.sh

Create the file **Dockerfile** based on `busybox` images

```
FROM busybox:latest
```

Copy the localy start.sh into container under / directory

```
COPY start.sh /start.sh
```

Add the environment variable AUTHOR with default value Docker

```
ENV \
AUTHOR=Docker
```

The command for running the application will be now your `start.sh` script

```
CMD ["/start.sh"]
```

Your `Dockerfile` is now ready.

```
FROM busybox:latest

COPY start.sh /start.sh

ENV \
AUTHOR=Docker

CMD ["/start.sh"]
```

Build you image as follow:

```
docker build -t yourname/myhelloworld:v1 .
Sending build context to Docker daemon  3.072kB
```

```
Step 1/4 : FROM busybox:latest
 ---> 020584afccce
Step 2/4 : COPY start.sh /start.sh
 ---> Using cache
 ---> 6f9ffe05fe61
Step 3/4 : ENV AUTHOR=Docker
 ---> Using cache
 ---> 545f5ac64db5
Step 4/4 : CMD ["/start.sh"]
 ---> Using cache
 ---> 2989ecd8edd5
Successfully built 2989ecd8edd5
Successfully tagged lcavatorta/myhelloworld:latest
```

Run you image!!

```
docker run -e AUTHOR="Mario Rossi" mrossi/myhelloworld


Hello Mario Rossi
```

LAB-14

How to quickly replace environment variables in a file

The envsubst is part of the gettext internationalization and localization project for unix.

Example
Let's say, we have an existing configuration file, that want to use to configure some application at the start of container.

# my configuration file
```
server: https://someurl.com/auth
username: foo_user
password: foo_password
```

1. Create sample configuration file

Lets replace the information with the environment variables:

```
server: $SERVER_URL
username: $USER_NAME
password: $USER_PASSWORD
```

2. Create the ENV file

```
SERVER_URL=https://someurl.com/auth
USER_NAME=foo_user
USER_PASSWORD=foo_password
```

3. Substitution

To run an actual substitution, perform the following commands.

```
envsubst < config.txt > config.conf

cat config.conf
server: https://someurl.com/auth
username: foo_user
password: foo_password
```

Create your own static docker httpd site based on centos

Dockerfile:

```
FROM centos:8
ENV AUTOR=Docker
RUN yum -y install httpd gettext
WORKDIR /var/www/html
COPY Hello_docker.html /var/www/html
EXPOSE 80
CMD cd /var/www/html && envsubst < Hello_docker.html > index.html;
/usr/sbin/httpd -D FOREGROUND;
```

Hello_docker.html:

```
<!DOCTYPE html>
<html>
<head>
<style>
body {
  margin-left: 500px;
  background: #5d9ab2
url("https://www.docker.com/sites/default/files/d8/styles/role_ico
n/public/2019-07/Docker-Logo-White-RGB_Vertical-BG_0.png?itok=8Tua
c9I3") no-repeat top left;
}

.center_div {
  border: 1px solid gray;
  margin-left: auto;
  margin-right: auto;
  width: 90%;
  background-color: #d0f0f6;
  text-align: left;
  padding: 8px;
}
</style>
</head>
<body>

<div class="center_div">
  <h1>Hello $AUTHOR</h1>
</div>
```

```
</body>
</html>
```

Build you image:

```
docker build -t dockersample/lab-14:1.0 .
```

Run a new container based on your image

```
docker run -d -e AUTHOR="luca" -p 80:80 dockersample/lab-14:1.0
```

check that your container is up and running

```
docker ps
```

Check on your local browser

```
httpd://localhost:8080
```

LAB-15

no solution for this labs

1)
Create a new images based on:

- Operating System “centos:8”
- install httpd service
- expose your container to 80 port
- modify /etc/httpd/conf/httpd.conf to use /srv/www/http/ as DocumentRoot directory
- Add in Dockerfile a httpd start command at the end of file with content:
  - **CMD ["/usr/sbin/httpd","-D","FOREGROUND"]**
- Build the image
- Start your container bind a **local index.html to** /srv/www/http/index.html
- Try if httpd works from browser

2)

- Create a new centos:8 based Docker image called yourName/hello:1.0
- Once launched, the task of this image will be print the string "Hello Student" on the screen every 2 seconds, for a maximum of 5 times, and then exit by printing the string "goodbye !!"
- The word Student must be modified through the environment variable passed at the start of the container.

3)

- Modify **LAB-14**
  - remove from Dockerfile
    - `CMD cd /var/www/html && envsubst < Hello_docker.html > index.html ; /usr/sbin/httpd -D FOREGROUND;`
  - use a ENTRYPOINT to call a bash script called `entrypoint.sh`
  - the script will have to:
    - replace the environment variable inside the index.html.
    - start the httpd server

4)

- Modify point 1 to configure dynamically the httpd at container start:
- Create these two env
  - DOCUMENTROOT
  - PORT
- Add these variable into httpd.conf to configure the listen port and documentRoot entry (do not forget Directory configuration: <Directory “<DocumentRoot directory>”)

- Run example:
  ```
  docker run -d -p 80:8888 --rm -e
  "DOCUMENTROOT=/srv/lab15-3/html" -e "PORT=8888" -v
  $PWD/index:/srv/lab15-3/html yourImageName:latest
  ```

LAB-16

Deploy a registry server

Use a command like the following to start the registry container:

```
$ docker run -d -p 5000:5000 --restart=always --name registry
registry:2
```

Copy an image from Docker Hub to your registry

You can pull an image from Docker Hub and push it to your registry. The following example pulls the ubuntu:16.04 image from Docker Hub and re-tags it as my-ubuntu, then pushes it to the local registry. Finally, the ubuntu:16.04 and my-ubuntu images are deleted locally and the my-ubuntu image is pulled from the local registry.

Pull the ubuntu:16.04 image from Docker Hub.

```
$ docker pull ubuntu:16.04
```

Tag the image as localhost:5000/my-ubuntu. This creates an additional tag for the existing image. When the first part of the tag is a hostname and port, Docker interprets this as the location of a registry, when pushing.

```
$ docker tag ubuntu:16.04 localhost:5000/my-ubuntu
```

Push the image to the local registry running at localhost:5000:

```
$ docker push localhost:5000/my-ubuntu
```

Remove the locally-cached ubuntu:16.04 and localhost:5000/my-ubuntu images, so that you can test pulling the image from your registry. This does not remove the localhost:5000/my-ubuntu image from your registry.

```
$ docker image remove ubuntu:16.04
$ docker image remove localhost:5000/my-ubuntu
```

Pull the localhost:5000/my-ubuntu image from your local registry.

```
$ docker pull localhost:5000/my-ubuntu
```

Customize the published port
If you are already using port 5000, or you want to run multiple local registries to separate areas of concern, you can customize the registry's port settings. This example runs the registry on port 5001 and also names it registry-test. Remember, the first part of the -p value is the host port and the second part is the port within the container. Within the container, the registry listens on port 5000 by default.

```
$ docker run -d \
  -p 5001:5000 \
  --name registry-test \
  registry:2
```

If you want to change the port the registry listens on within the container, you can use the environment variable REGISTRY_HTTP_ADDR to change it. This command causes the registry to listen on port 5001 within the container:

```
$ docker run -d \
  -e REGISTRY_HTTP_ADDR=0.0.0.0:5001 \
  -p 5001:5001 \
  --name registry-test \
  registry:2
```

Storage customization

Customize the storage location

By default, your registry data is persisted as a docker volume on the host filesystem. If you want to store your registry contents at a specific location on your host filesystem, such as if you have an SSD or SAN mounted into a particular directory, you might decide to use a bind mount instead. A bind mount is more dependent on the filesystem layout of the Docker host, but more performant in many situations. The following example bind-mounts the host directory /mnt/registry into the registry container at /var/lib/registry/.

```
$ docker run -d \
  -p 5000:5000 \
  --restart=always \
  --name registry \
  -v /mnt/registry:/var/lib/registry \
  registry:2
```

LAB-17 - Docker Compose

YAML Syntax

YAML stands for "YAML Ain't Markup Language" and it is used in docker-compose.

YAML is to configuration what markdown is to markup. It’s basically a human-readable structured data format. It is less complex and ungainly than XML or JSON, but provides similar capabilities. It essentially allows you to provide powerful configuration settings, without having to learn a more complex code type like CSS, JavaScript, and PHP.

YAML is built from the ground up to be simple to use. At its core, a YAML file is used to describe data. One of the benefits of using YAML is that the information in a single YAML file can be easily translated to multiple language types.

YAML Basic Rules

There are some rules that YAML has in place to avoid issues related to ambiguity in relation to various languages and editing programs. These rules make it possible for a single YAML file to be interpreted consistently, regardless of which application and/or library is being used to interpret it.

- YAML files should end in .yaml or .yml
- YAML is case sensitive.
- YAML does not allow the use of tabs.

Basic Data Types

YAML excels at working with **mappings** (hashes / dictionaries), **sequences** (arrays / lists), and **scalars** (strings / numbers). While it can be used with most programming languages, it works best with languages that are built around these data structure types. This includes: PHP, Python, Perl, JavaScript, and Ruby.

Scalars

Scalars are a pretty basic concept. They are the strings and numbers that make up the data on the page. A scalar could be a boolean property, like true, integer (number) such as 5, or a string of text, like a sentence or the title of your website.

Scalars are often called variables in programming. If you were making a list of types of animals, they would be the names given to those animals.

Most scalars are unquoted, but if you are typing a string that uses punctuation and other elements that can be confused with YAML syntax (dashes, colons, etc.) you may want to quote this data using single ' or double " quotation marks. Double quotation marks allow you to use escapings to represent ASCII and Unicode characters.

```
integer: 25
string: "25"
float: 25.0
boolean: true
```

Sequences

Here is a simple sequence you might find. It is a basic list with each item in the list placed in its own line with an opening dash.

```
- Cat
- Dog
- Goldfish
```

This sequence places each item in the list at the same level. If you want to create a nested sequence with items and sub-items, you can do so by placing a single space before each dash in the sub-items. YAML uses spaces, NOT tabs, for indentation. You can see an example of this below.

```
-
 - Cat
 - Dog
 - Goldfish
-
 - Python
 - Lion
 - Tiger
```

If you wish to nest your sequences even deeper, you just need to add more levels.

```
-
  -
    - Cat
    - Dog
    - Goldfish
```

Sequences can be added to other data structure types, such as mappings or scalars.

Mappings

Mapping gives you the ability to list keys with values. This is useful in cases where you are assigning a name or a property to a specific element.

```
animal: pets
```

This example maps the value of pets to the animal key. When used in conjunction with a sequence, you can see that you are starting to build a list of pets. In the following example, the dash used to label each item counts as indentation, making the line items the child and the mapping line pets the parent.

```
pets:
 - Cat
 - Dog
 - Goldfish
```

Overview of Docker Compose

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration. To learn more about all the features of Compose, see the list of features.
Compose works in all environments: production, staging, development, testing, as well as CI workflows. You can learn more about each case in Common Use Cases.
Using Compose is basically a three-step process:

1. Define your app's environment with a Dockerfile so it can be reproduced anywhere.
2. Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
3. Run docker-compose up and Compose starts and runs your entire app.

A docker-compose.yml looks like this:

```
version: "3.8"
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Compose has commands for managing the whole lifecycle of your application:

- Start, stop, and rebuild services
- View the status of running services
- Stream the log output of running services
- Run a one-off command on a service

Service configuration reference

The Compose file is a YAML file defining services, networks and volumes. The default path for a Compose file is ./docker-compose.yml.
A service definition contains configuration that is applied to each container started for that service, much like passing command-line parameters to docker run. Likewise, network and volume definitions are analogous to docker network create and docker volume create.
As with docker run, options specified in the Dockerfile, such as CMD, EXPOSE, VOLUME, ENV, are respected by default - you don't need to specify them again in docker-compose.yml.
You can use environment variables in configuration values with a Bash-like ${VARIABLE} syntax - see variable substitution for full details.
This section contains a list of all configuration options supported by a service definition in version 3.

build

build can be specified either as a string containing a path to the build context:

```
version: "3.8"
services:
  webapp:
    build: ./dir
```

Or, as an object with the path specified under context and optionally Dockerfile and args:

```
version: "3.8"
services:
  webapp:
    build:
      context: ./dir
      dockerfile: Dockerfile-alternate
      args:
        buildno: 1
```

ARGS
Add build arguments, which are environment variables accessible only during the build process.
First, specify the arguments in your Dockerfile:

```
ARG buildno
ARG gitcommithash

RUN echo "Build number: $buildno"
RUN echo "Based on commit: $gitcommithash"
```

Then specify the arguments under the build key. You can pass a mapping or a list:

```
build:
  context: .
  args:
    buildno: 1
    gitcommithash: cdc3b19

build:
  context: .
  args:
    - buildno=1
    - gitcommithash=cdc3b19
```

If you specify image as well as build, then Compose names the built image with the webapp and optional tag specified in image:

```
build: ./dir
image: webapp:tag
```

command

Override the default command.

```
command: bundle exec thin -p 3000
```

The command can also be a list, in a manner similar to dockerfile:

```
command: ["bundle", "exec", "thin", "-p", "3000"]
```

Example

```
services:
 sshd:
   image: centos:7-sshd
   container_name: sshd
   command: ["-u", "student", "-i", "1000"]
```

container_name

Specify a custom container name, rather than a generated default name.

```
container_name: my-web-container
```

Because Docker container names must be unique, you cannot scale a service beyond 1 container if you have specified a custom name. Attempting to do so results in an error.

depends_on

Express dependency between services. Service dependencies cause the following behaviors:

- docker-compose up starts services in dependency order. In the following example, db and redis are started before web.
- docker-compose up SERVICE automatically includes SERVICE’s dependencies. In the example below, docker-compose up web also creates and starts db and redis.
- docker-compose stop stops services in dependency order. In the following example, web is stopped before db and redis.

```
version: "3.8"
services:
  web:
    build: .
```

```
      depends_on:
        - db
        - redis
    redis:
      image: redis
    db:
      image: postgres
```

env_file

Add environment variables from a file. Can be a single value or a list.
If you have specified a Compose file with docker-compose -f FILE, paths in env_file are relative to the directory that file is in.
Environment variables declared in the environment section override these values – this holds true even if those values are empty or undefined.

```
env_file: .env

env_file:
  - ./common.env
  - ./apps/web.env
  - /opt/runtime_opts.env
```

Compose expects each line in an env file to be in VAR=VAL format. Lines beginning with # are treated as comments and are ignored. Blank lines are also ignored.

```
# Set Rails/Rack environment
RACK_ENV=development
```

environment

Add environment variables. You can use either an array or a dictionary. Any boolean values (true, false, yes, no) need to be enclosed in quotes to ensure they are not converted to True or False by the YML parser.
Environment variables with only a key are resolved to their values on the machine Compose is running on, which can be helpful for secret or host-specific values.

```
environment:
  RACK_ENV: development
  SHOW: 'true'
  SESSION_SECRET:

environment:
  - RACK_ENV=development
  - SHOW=true
  - SESSION_SECRET
```

expose

Expose ports without publishing them to the host machine - they'll only be accessible to linked services. Only the internal port can be specified.

```
expose:
  - "3000"
  - "8000"
```

image

Specify the image to start the container from. Can either be a repository/tag or a partial image ID.

```
image: redis

image: ubuntu:18.04

image: tutum/influxdb

image: example-registry.com:4000/postgresql

image: a4bc65fd
```

labels

Add metadata to containers using Docker labels. You can use either an array or a dictionary. It’s recommended that you use reverse-DNS notation to prevent your labels from conflicting

with those used by other software.

```
labels:
  com.example.description: "Accounting webapp"
  com.example.department: "Finance"
  com.example.label-with-empty-value: ""

labels:
  - "com.example.description=Accounting webapp"
  - "com.example.department=Finance"
  - "com.example.label-with-empty-value"
```

network_mode

Network mode. Use the same values as the docker client --network parameter, plus the special form service:[service name].

```
network_mode: "bridge"

network_mode: "host"

network_mode: "none"

network_mode: "service:[service name]"

network_mode: "container:[container name/id]"
```

networks

Networks to join, referencing entries under the top-level networks key.

```
services:
  some-service:
    networks:
     - some-network
     - other-network
```

In the example below, two services are provided (web and worker), along with two networks (new and legacy).

```
version: "3.8"

services:
  web:
    image: "nginx:alpine"
    networks:
      - new

  worker:
    image: "my-worker-image:latest"
    networks:
      - legacy

  db:
    image: mysql
    networks:
      new:
        aliases:
          - database
      legacy:
        aliases:
          - mysql

networks:
  new:
  legacy:
```

IPV4_ADDRESS, IPV6_ADDRESS

Specify a static IP address for containers for this service when joining the network.
The corresponding network configuration in the top-level networks section must have an ipam block with subnet configurations covering each static address.
If IPv6 addressing is desired, the enable_ipv6 option must be set, and you must use a version 2.x Compose file. IPv6 options do not currently work in swarm mode.

```
An example:
version: "3.8"

services:
  app:
    image: nginx:alpine
    networks:
      app_net:
        ipv4_address: 172.16.238.10
        ipv6_address: 2001:3984:3989::10

networks:
  app_net:
    ipam:
      driver: default
      config:
        - subnet: "172.16.238.0/24"
        - subnet: "2001:3984:3989::/64"
```

ports

Expose ports.
SHORT SYNTAX
Either specify both ports (HOST:CONTAINER), or just the container port (an ephemeral host port is chosen).

```
ports:
  - "3000"
  - "3000-3005"
  - "8000:8000"
  - "9090-9091:8080-8081"
  - "49100:22"
  - "127.0.0.1:8001:8001"
  - "127.0.0.1:5000-5010:5000-5010"
  - "6060:6060/udp"
  - "12400-12500:1240"
```

LONG SYNTAX
The long form syntax allows the configuration of additional fields that can't be expressed in the short form.

- target: the port inside the container
- published: the publicly exposed port
- protocol: the port protocol (tcp or udp)
- mode: host for publishing a host port on each node, or ingress for a swarm mode port to be load balanced.

```
ports:
  - target: 80
    published: 8080
    protocol: tcp
```

```
        mode: host
```

restart

no is the default restart policy, and it does not restart a container under any circumstance. When always is specified, the container always restarts. The on-failure policy restarts a container if the exit code indicates an on-failure error. unless-stopped always restarts a container, except when the container is stopped (manually or otherwise).

```
restart: "no"
restart: always
restart: on-failure
restart: unless-stopped
```

volumes

Mount host paths or named volumes, specified as sub-options to a service.
You can mount a host path as part of a definition for a single service, and there is no need to define it in the top level volumes key.
But, if you want to reuse a volume across multiple services, then define a named volume in the top-level volumes key.
This example shows a named volume (mydata) being used by the web service, and a bind mount defined for a single service (first path under db service volumes). The db service also uses a named volume called dbdata (second path under db service volumes), but defines it using the old string format for mounting a named volume. Named volumes must be listed under the top-level volumes key, as shown.

```
version: "3.8"
services:
  web:
    image: nginx:alpine
    volumes:
      - type: volume
        source: mydata
        target: /data
        volume:
          nocopy: true
      - type: bind
        source: ./static
        target: /opt/app/static

  db:
    image: postgres:latest
    volumes:
      -
"/var/run/postgres/postgres.sock:/var/run/postgres/postgres.s
ock"
      - "dbdata:/var/lib/postgresql/data"

volumes:
  mydata:
  dbdata:
```

SHORT SYNTAX

The short syntax uses the generic [SOURCE:]TARGET[:MODE] format, where SOURCE can be either a host path or volume name. TARGET is the container path where the volume is mounted. Standard modes are ro for read-only and rw for read-write (default).
You can mount a relative path on the host, which expands relative to the directory of the Compose configuration file being used. Relative paths should always begin with . or ...

```
volumes:
  # Just specify a path and let the Engine create a volume
  - /var/lib/mysql

  # Specify an absolute path mapping
  - /opt/data:/var/lib/mysql

  # Path on the host, relative to the Compose file
  - ./cache:/tmp/cache

  # User-relative path
  - ~/configs:/etc/configs/:ro

  # Named volume
  - datavolume:/var/lib/mysql
```

domainname, hostname, ipc, mac_address, privileged, read_only, shm_size, stdin_open, tty, user, working_dir

Each of these is a single value, analogous to its docker run counterpart. Note that mac_address is a legacy option.

```
user: postgresql
working_dir: /code

domainname: foo.com
hostname: foo
```

```
    ipc: host
    mac_address: 02:42:ac:11:65:43

    privileged: true


    read_only: true
    shm_size: 64M
    stdin_open: true
    tty: true
```

Complete reference at https://docs.docker.com/compose/compose-file

Get started with Docker Compose

On this page you build a simple Python web application running on Docker Compose. The application uses the Flask framework and maintains a hit counter in Redis. While the sample uses Python, the concepts demonstrated here should be understandable even if you're not familiar with it.

Prerequisites

Make sure you have already installed both Docker Engine and Docker Compose. You don't need to install Python or Redis, as both are provided by Docker images.

Step 1: Setup

Define the application dependencies.
Create a directory for the project:

```
$ mkdir composetest
$ cd composetest
```

Create a file called **app.py** in your project directory and paste this in:

```
import time

import redis
from flask import Flask

app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)

def get_hit_count():
    retries = 5
    while True:
        try:
            return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0:
                raise exc
            retries -= 1
            time.sleep(0.5)

@app.route('/')
def hello():
    count = get_hit_count()
    return 'Hello World! I have been seen {}
times.\n'.format(count)
```

In this example, redis is the hostname of the redis container on the application's network. We use the default port for Redis, 6379.

Handling transient errors
Note the way the get_hit_count function is written. This basic retry loop lets us attempt our request multiple times if the redis service is not available. This is useful at startup while the application comes online, but also makes our application more resilient if the Redis service needs to be restarted anytime during the app's lifetime. In a cluster, this also helps handling momentary connection drops between nodes.

Create another file called **requirements.txt** in your project directory and paste this in:

```
flask
redis
```

Step 2: Create a Dockerfile

In this step, you write a Dockerfile that builds a Docker image. The image contains all the dependencies the Python application requires, including Python itself.
In your project directory, create a file named **Dockerfile** and paste the following:

```
FROM python:3.7-alpine
WORKDIR /code
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
RUN apk add --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
CMD ["flask", "run"]
```

This tells Docker to:

- Build an image starting with the Python 3.7 image.
- Set the working directory to /code.
- Set environment variables used by the flask command.
- Install gcc and other dependencies
- Copy requirements.txt and install the Python dependencies.
- Add metadata to the image to describe that the container is listening on port 5000
- Copy the current directory . in the project to the workdir . in the image.
- Set the default command for the container to flask run.

Step 3: Define services in a Compose file
Create a file called **docker-compose.yml** in your project directory and paste the following:

```
version: "3.8"
services:
  web:
    build: .
    ports:
      - "80:5000"
  redis:
    image: "redis:alpine"
```

This Compose file defines two services: web and redis.

Web service
The web service uses an image that's built from the Dockerfile in the current directory. It then binds the container and the host machine to the exposed port, 5000. This example service uses the default port for the Flask web server, 5000.

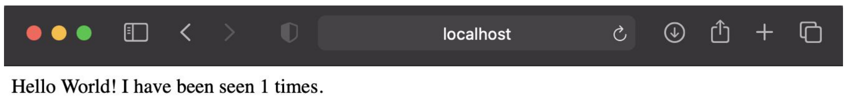
Redis service
The redis service uses a public Redis image pulled from the Docker Hub registry.
Step 4: Build and run your app with Compose

From your project directory, start up your application by running **docker-compose up**.

```
$ docker-compose up

Creating network "composetest_default" with the default driver
Creating composetest_web_1 ...
Creating composetest_redis_1 ...
Creating composetest_web_1
Creating composetest_redis_1 ... done
Attaching to composetest_web_1, composetest_redis_1
web_1    |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
redis_1  | 1:C 17 Aug 22:11:10.480 # oO0OoO0OoO0Oo Redis is starting
oO0OoO0OoO0Oo
redis_1  | 1:C 17 Aug 22:11:10.480 # Redis version=4.0.1, bits=64,
commit=00000000, modified=0, pid=1, just started
redis_1  | 1:C 17 Aug 22:11:10.480 # Warning: no config file
specified, using the default config. In order to specify a config
file use redis-server /path/to/redis.conf
web_1    |  * Restarting with stat
redis_1  | 1:M 17 Aug 22:11:10.483 * Running mode=standalone,
port=6379.
redis_1  | 1:M 17 Aug 22:11:10.483 # WARNING: The TCP backlog
setting of 511 cannot be enforced because
/proc/sys/net/core/somaxconn is set to the lower value of 128.
web_1    |  * Debugger is active!
redis_1  | 1:M 17 Aug 22:11:10.483 # Server initialized
redis_1  | 1:M 17 Aug 22:11:10.483 # WARNING you have Transparent
Huge Pages (THP) support enabled in your kernel. This will create
latency and memory usage issues with Redis. To fix this issue run
the command 'echo never >
/sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to
your /etc/rc.local in order to retain the setting after a reboot.
Redis must be restarted after THP is disabled.
```

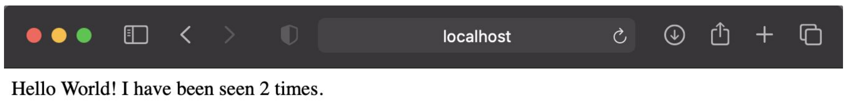
```
web_1    |  * Debugger PIN: 330-787-903
redis_1  | 1:M 17 Aug 22:11:10.483 * Ready to accept connections
```

Compose pulls a Redis image, builds an image for your code, and starts the services you defined. In this case, the code is statically copied into the image at build time.
Enter http://localhost:8080/ in a browser to see the application running.
You should see a message in your browser saying:
Hello World! I have been seen 1 times.



Hello World! I have been seen 1 times.

Refresh the page.
The number should increment.
Hello World! I have been seen 2 times.



Hello World! I have been seen 2 times.

Stop the application, either by running docker-compose down from within your project directory in the second terminal, or by hitting **CTRL+C** in the original terminal where you started the app.

Step 5: Edit the Compose file to add a bind mount
Edit docker-compose.yml in your project directory to add a bind mount for the web service:

```
version: "3.8"
services:
  web:
    build: .
    ports:
      - "80:5000"
    volumes:
      - .:/code
    environment:
      FLASK_ENV: development
  redis:
    image: "redis:alpine"
```

The new volumes key mounts the project directory (current directory) on the host to /code inside the container, allowing you to modify the code on the fly, without having to rebuild the image. The environment key sets the FLASK_ENV environment variable, which tells flask run to run in development mode and reload the code on change. This mode should only be used in development.
Step 6: Re-build and run the app with Compose
From your project directory, type docker-compose up to build the app with the updated Compose file, and run it.

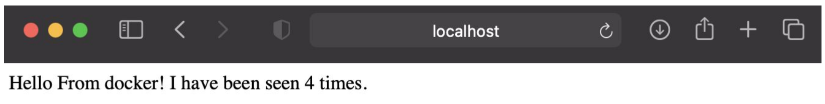
```
$ docker-compose up

Creating network "composetest_default" with the default driver
Creating composetest_web_1 ...
Creating composetest_redis_1 ...
Creating composetest_web_1
Creating composetest_redis_1 ... done
Attaching to composetest_web_1, composetest_redis_1
web_1    |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to
quit)
...
```

Check the Hello World message in a web browser again, and refresh to see the count increment.

Because the application code is now mounted into the container using a volume, you can make changes to its code and see the changes instantly, without having to rebuild the image.
Change the greeting in **app.py** and save it. For example, change the Hello World! message to Hello from Docker!:

```
return 'Hello from Docker! I have been seen {}
times.\n'.format(count)
```

Refresh the app in your browser. The greeting should be updated, and the counter should still be incrementing.



Hello From docker! I have been seen 4 times.

Stop the application hitting **CTRL+C** in the original terminal where you started the app.

Step 8: Experiment with some other commands
If you want to run your services in the **background**, you can pass the -d flag (for "**detached**" mode) to docker-compose up and use docker-compose ps to see what is currently running:

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

Starting composetest_redis_1...
Starting composetest_web_1...

```
$ docker-compose ps

Name                    Command              State        Ports
------------------------------------------------------------------
-
composetest_redis_1   /usr/local/bin/run         Up
composetest_web_1     /bin/sh -c python app.py   Up
80>5000/tcp
```

The docker-compose run command allows you to run one-off commands for your services. For example, to see what environment variables are available to the web service:

```
$ docker-compose run web env
```

If you started Compose with docker-compose up -d, stop your services once you've finished with them:

```
$ docker-compose stop
```

You can bring everything down, removing the containers entirely, with the down command. Pass --volumes to also remove the data volume used by the Redis container:

```
$ docker-compose down --volumes
```

LAB-18

try some example of docker compose
https://github.com/docker/awesome-compose