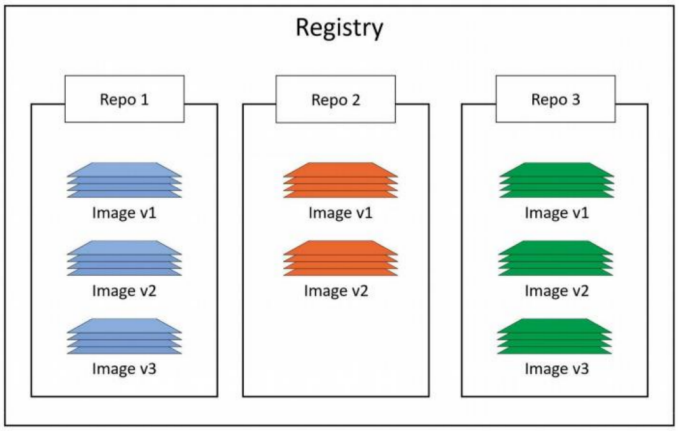
As you can see now we have two images downloaded on our docker engine.

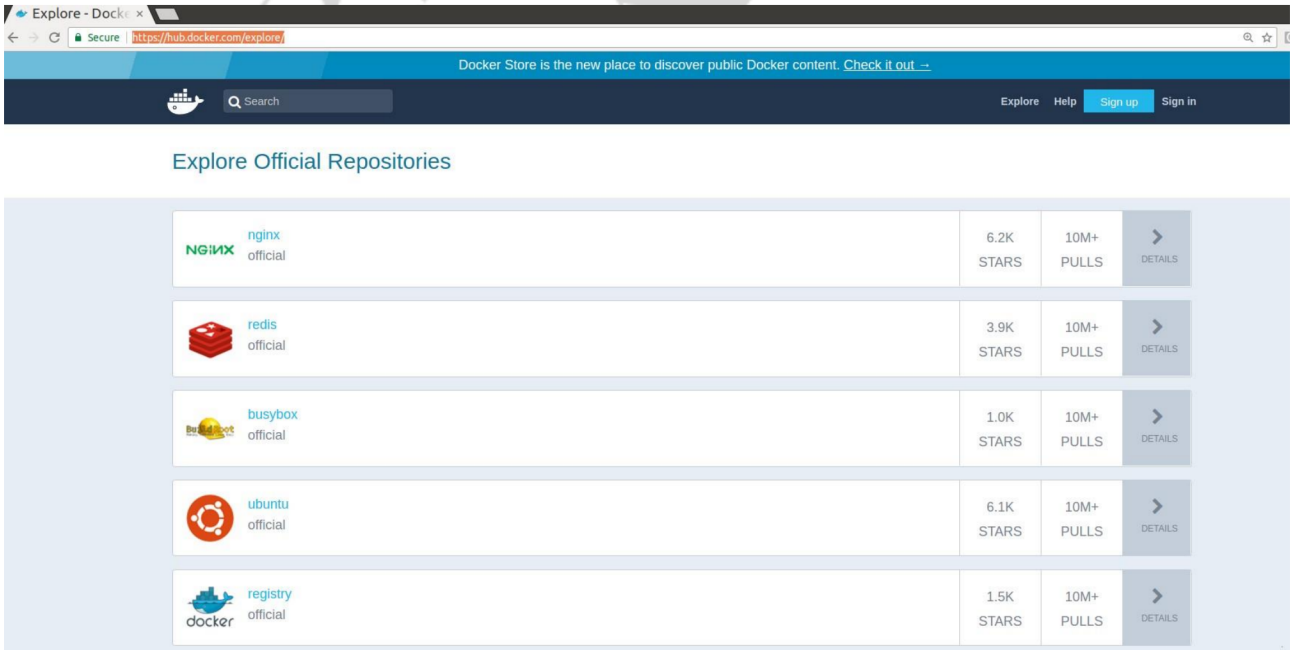
Image registries

Docker images are stored in image registries. The most common image registry is Docker Hub. Other registries exist including 3rd party registries and secure on-premises registries, but Docker Hub is the default, and it's the one we'll use in this tutorial.

https://hub.docker.com/

Image registries contain multiple image repositories. Image repositories contain images. That might be a bit confusing, so Figure 5.2 shows a picture of an image registry containing 3 repositories, and each repository contains a few images.





Docker hub contains Official and unofficial repositories.
Official repositories are from Docker, Inc. These are safe and secure images with latest up to date software.

Unofficial repositories are uploaded by anyone and are not verified by Docker, Inc.

The list below contains a few of the official repositories and shows their URLs that exist at the top level of the Docker Hub namespace:

- nginx - https://hub.docker.com/_/nginx/
- busybox - https://hub.docker.com/_/busybox/
- redis - https://hub.docker.com/_/redis/
- mongo - https://hub.docker.com/_/mongo/

Our personal images live in the unofficial repositories. Below are some examples of images in my repositories:
visualpath/myjsonsinatra - https://hub.docker.com/r/visualpath/myjsonsinatra/
visualpath/devops-docker-ci - https://hub.docker.com/r/visualpath/devops-docker-ci/

Image Tags.

While pulling a image we give the imagename:TAG and docker will reach by default to dockerhub registry and find the image with the TAG we specified.

```
docker  pull nginx:latest
```

Tag generally refers to the version of the image from the repository.
If we are looking for some other version like 1.12.0 then we can use below command.

```
docker pull nginx:1.12.0
```

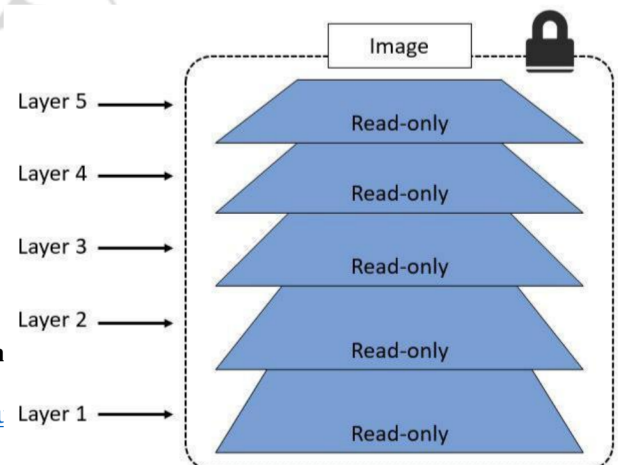
If we do not specify any tag then the default tag is latest. Latest tag does not mean that the images is latest in version, it's just the name of the tag and that's all.

```
docker pull nginx
```

Above command will download the nginx image with latest tag.

Images and layers

All Docker images are made up of one or more read-only layers as shown below.

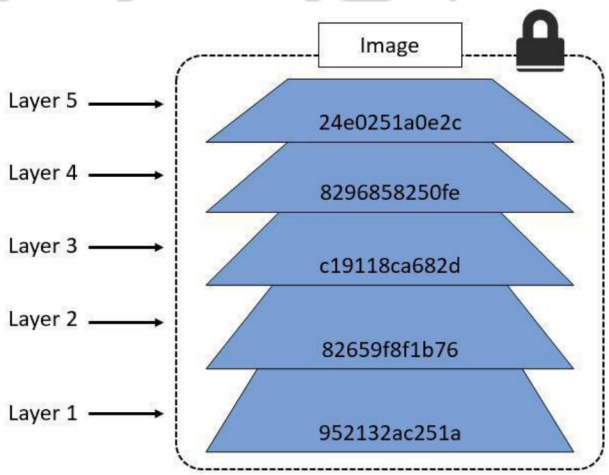


There are a few ways to see and inspect the layers that make up an image,

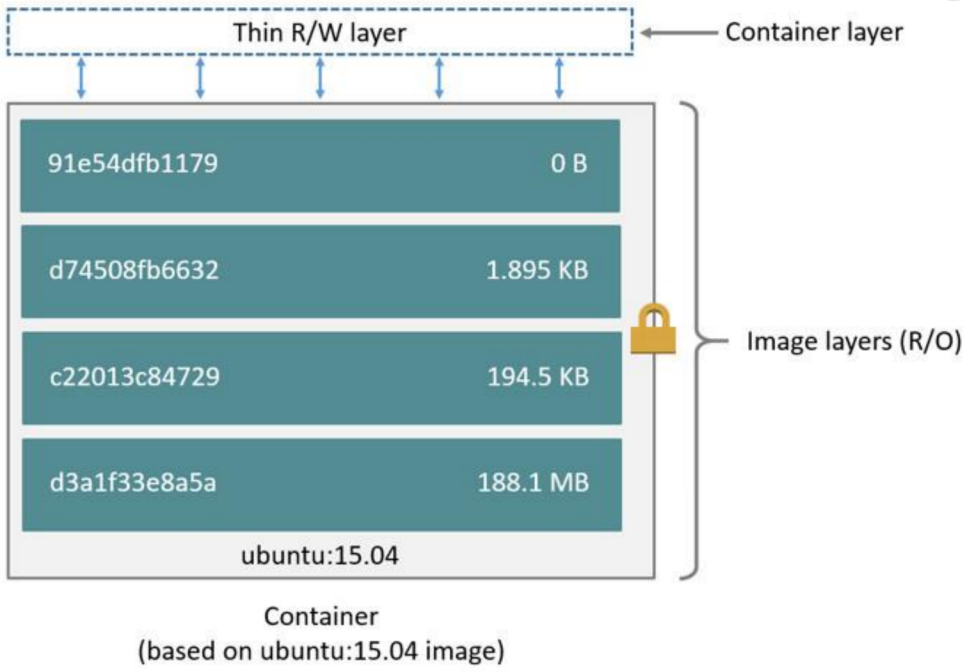
and we've already seen one of them. Let's take a second look at the output of the docker pull node:latest command from earlier:

```
$ docker pull node
Using default tag: latest
latest: Pulling from library/node
ef0380f84d05: Pull complete
24c170465c65: Pull complete
4f38f9d5c3c0: Pull complete
4125326b53d8: Pull complete
a1468c06f443: Pull complete
cb113e1791ca: Pull complete
519ce9303f95: Pull complete
f15f31d0549a: Pull complete
Digest: sha256:1d496e5c8e692dfabeb1cc8a18f01e2b501111f32c3d08e94e5402daeceb94e6
```

Each line in the output above that ends with "Pull complete" represents a layer in the image that was pulled. As we can see, this image has 5 layers.



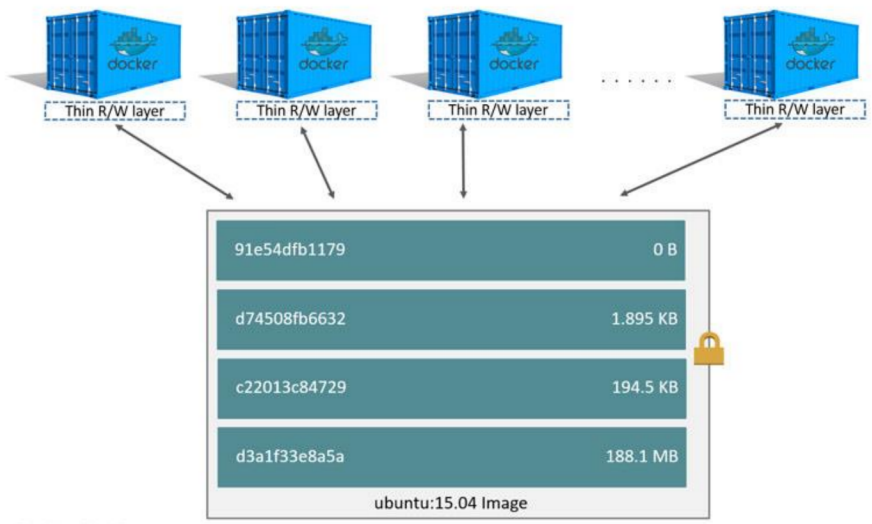
Each layer is only a set of differences from the layer before it. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the “container layer”. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on the Ubuntu 15.04 image.



Container and layers

The major difference between a container and an image is the top writable layer. All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged.

Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.



Docker uses storage drivers to manage the contents of the image layers and the writable container layer. Each storage driver handles the implementation differently, but all drivers use stackable image layers and the copy-on-write (CoW) strategy.

Another way to see the layers that make up an image is to inspect the image with the docker inspect command. The example below inspects the same ubuntu:latest image.

```
$ docker inspect node

[

    {

        "Id": "sha256:f3068bc71556e181c774ee7dadc4d3ebbf5643e95680a202779f08146332547d",

        "RepoTags": [

            "node:latest"

        ],

        "RepoDigests": [

            "node@sha256:1d496e5c8e692dfabeb1cc8a18f01e2b501111f32c3d08e94e5402daeceb94e6"

        ],

        "Parent": "",

        "Comment": "",

        "Created": "2017-06-15T17:26:33.424702587Z",

        "Container": "8c6ffc2b7a445e6f2ade22c6be3a430c772e0ab61bf0ee4fff69b36a24e9123e",

        "ContainerConfig": {

            "Hostname": "5c84359661e5",

            "Domainname": "",

            "User": "",

            "AttachStdin": false,

            "AttachStdout": false,
```

```
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
                "NPM_CONFIG_LOGLEVEL=info",
                "NODE_VERSION=8.1.2",
                "YARN_VERSION=0.24.6"
            ],
            "Cmd": [
                "/bin/sh",
                "-c",
                "#(nop) ",
                "CMD [\"node\"]"
            ],
            "ArgsEscaped": true,
            "Image": "sha256:3864d82628abf07bbdebe3d1d529aa90eef89f8dc99d06dea2a35329879c81a7",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": [],
            "Labels": {}
        },
        "DockerVersion": "17.03.1-ce",
        "Author": "",
        "Config": {
            "Hostname": "5c84359661e5",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
                "NPM_CONFIG_LOGLEVEL=info",
                "NODE_VERSION=8.1.2",
                "YARN_VERSION=0.24.6"
```

```
            ],
            "Cmd": [
                "node"
            ],
            "ArgsEscaped": true,
            "Image": "sha256:3864d82628abf07bbdebe3d1d529aa90eef89f8dc99d06dea2a35329879c81a7",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": [],
            "Labels": {}
        },
        "Architecture": "amd64",
        "Os": "linux",
        "Size": 666628404,
        "VirtualSize": 666628404,
        "GraphDriver": {
            "Data": null,
            "Name": "aufs"
        },
        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:007ab444b234be691d8dafd51839b7713324a261b16e2487bf4a3f989ded912d",
                "sha256:4902b007e6a712835de8e09c385c0f061638323c3cacc13f7190676f05dad9d7",
                "sha256:bb07d0c1008de4fd468f865764e6f1129ba53f4bfe6ab14dd5eb3ab256947ab0",
                "sha256:ecf5c2e2468e7fe6600193972ffc659214050d8829f44e5194a22997de13aab4",
                "sha256:7b3b4fef39c1df95f7a015716bc980dad38fff92ebba6de82d5add10b1258523",
                "sha256:677f02386f077da16bfbe00af5305928545c11c40dccf70f93fa332f617c1fba",
                "sha256:99c62c5bb4f21ab5b288339ce1a29e33cb3a82670ec1a9254730b6925d7da7dc",
                "sha256:0dd4309d61fe600b230931ce669a93ee0baf9b2c18f574749c3af1e3eed44b83"
            ]
        }
    }
]
```


Deleting Images

When you no longer need an image, you can delete it form your Docker host with the docker rmi command. rmi is short for remove image.

Delete the node image pulled in the previous step with the docker rmi command. The example below addresses the image by its ID.

```
$ docker rmi f3068bc71556
```

9. Containers

Container is the runtime instance of an image like we start a vm from vagrant box. We can start multiple containers from one single image.

We create container from an image by giving docker run command. Containers run until the processes running inside them exists. There should be minimum one process running inside the container with PID 1. If this process dies the containers also dies.

```
imran@DevOps:~$ docker run -it ubuntu:latest /bin/bash

root@ae141c80e436:/# ps -ef

UID        PID  PPID  C STIME TTY          TIME CMD

root         1     0  2 21:44 ?        00:00:00 /bin/bash

root        11     1  0 21:44 ?        00:00:00 ps -ef
```

In above container /bin/bash has the PID 1, this process will get killed if we hit exit command.

```
root@ae141c80e436:/# exit

exit

imran@DevOps:~$ docker ps

CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
```

As exit will logout and kill the current shell and that's our PID 1 our container also got killed with it.

docker ps -a will show all the containers running or exited.

```
imran@DevOps:~$ docker ps -a

CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES

ae141c80e436        ubuntu:latest       "/bin/bash"         12 minutes ago      Exited (0) 10
minutes ago                         zen_bell

159b1586b69b        ubuntu:latest       "/bin/bash"         12 minutes ago      Exited (0) 12
minutes ago                         hungry_noether
```

We can start a exited container by giving docker start <containerid>

```
imran@DevOps:~$ docker start ae141c80e436

ae141c80e436
```

```
imran@DevOps:~$ docker ps

CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES

ae141c80e436        ubuntu:latest       "/bin/bash"         14 minutes ago      Up 3 seconds
zen_bell
```

docker stop will stop a running container.

```
imran@DevOps:~$ docker stop ae141c80e436

ae141c80e436
```

docker rm will remove the container with its data.

```
imran@DevOps:~$ docker rm ae141c80e436

ae141c80e436
```

Running a webservice in a container.

```
imran@DevOps:~$ docker run -d --name newwebserver -p 8070:80 visualpath/devops-
docker-ci

c695f224adf6a91f971018c6934b255e5cb86d6d8f6a4445d24fa7d7f1f70967
```

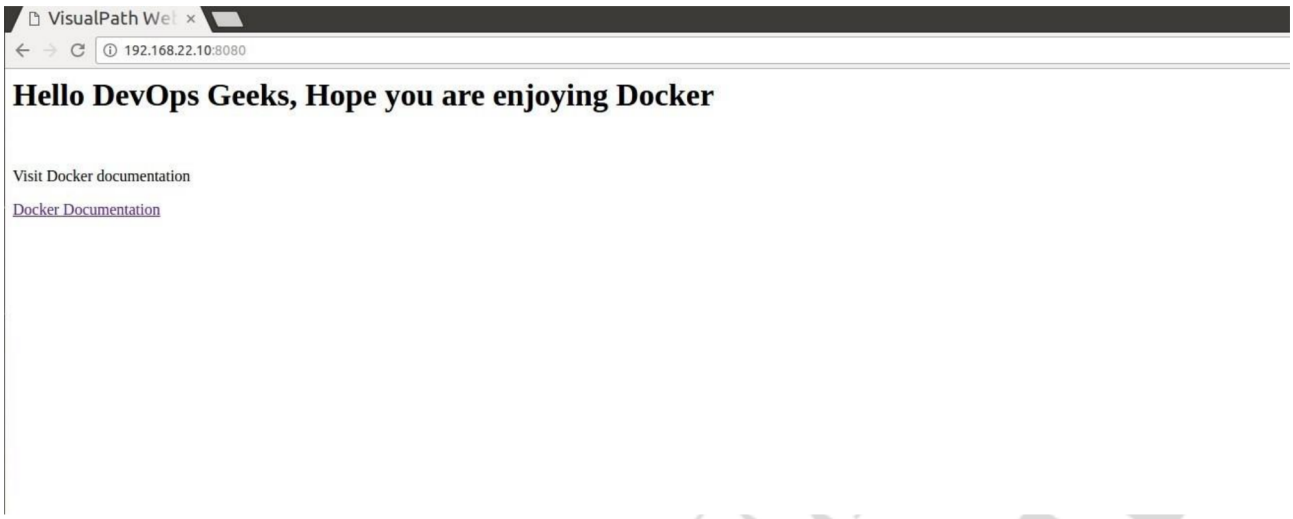
We started the above container in backgroud by option -d

we also have given name to our container by --name option

Above image is an apache webservice which is running on port 80 but can be accessed by forwarded port from host. Host Port 8070 is mapped to containers port 80. That means if we access host machines IP on port 8070 we will get service running on port 80 from the containers. Containers are not directly accessed by their IP's because container Ip's are not permanent. We will discuss that in detail in networking section. We access service running in container from host port which are redirected to containers port this is called a port forwarding.

-p 8070:80 means host port 8070 is mapped to containers port 80.

Verify the container webservice by accessing it from browser on http://hostip:8070



Check nginx, apache and Jenkins registries from dockerhub and run them to understand more about it.

```
$ docker run -p 8080:8080 -p 50000:50000 -v /your/home:/var/jenkins_home Jenkins
```

Here we are mapping two ports 8080 and 50000, host and container ports are same which is ok if your host ports are not busy.
Containers data is not persistent that means if we delete the container its data is also lost, which is obvious. But if we want to keep that data safe on host machine, we can use -v flag which is for **volumes**. Left hand side is host machine directory path and right-hand side is containers directory path which you want to save on host machine. It's similar to our vagrant sync directories.
Now even if we delete the container its data in /var/jenkins_home will be safe on the host machine in /your/home directory.

Inspecting Containers

In the previous example you might have noticed that we didn't specify a command for the container when we issued the docker run. Yet the container ran a simple web service. How did this happen? When building a Docker image, it's possible to embed a default command or process you want containers using the image to run. If we run a docker inspect command against the image we used to run our container, we'll be able to see the command/process that the container will run when it starts.

```
imran@DevOps:~$ docker inspect c695f224adf6a91f971018c6934b255e5cb86d6d8f6a4445d24fa7d7f1f70967
[
  {
    "Id": "c695f224adf6a91f971018c6934b255e5cb86d6d8f6a4445d24fa7d7f1f70967",
    "Created": "2017-06-16T22:13:41.841190294Z",
    "Path": "/bin/sh",
    "Args": [
```

```
        "-c",
        "/usr/sbin/apache2ctl -D FOREGROUND"
    ],
    "State": {
        "Status": "running",
        "Running": true,
        "Paused": false,
        "Restarting": false,
        "OOMKilled": false,
        "Dead": false,
        "Pid": 17026,
        "ExitCode": 0,
        "Error": "",
        "StartedAt": "2017-06-16T22:13:42.656496753Z",
        "FinishedAt": "0001-01-01T00:00:00Z"
```

10. Building & Shipping Images

Docker can build images automatically by reading the instructions from a Dockerfile, a text file that contains all the commands, in order, needed to build a given image. Dockerfiles adhere to a specific format and use a specific set of instructions.

Dockerfile will define what goes on in the environment inside your container. Access to resources like networking interfaces and disk drives is virtualized inside this environment, which is isolated from the rest of your system, so you have to map ports to the outside world, and be specific about what files you want to “copy in” to that environment. However, after doing that, you can expect that the build of your app defined in thisDockerfile will behave exactly the same wherever it runs.

`Dockerfile`

Create an empty directory and put this file in it, with the name Dockerfile. Take note of the comments that explain each statement.

```
# Use an official Python runtime as a base image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
```

```
ADD . /app

# Install any needed packages specified in requirements.txt
RUN pip install -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

This Dockerfile refers to a couple of things we haven't created yet, namely app.py and requirements.txt. Let's get those in place next.
The app itself

Grab these two files and place them in the same folder as Dockerfile. This completes our app, which as you can see is quite simple. When the above Dockerfile is built into an image, app.py and requirements.txtwill be present because of that Dockerfile's ADD command, and the output from app.py will be accessible over HTTP thanks to the EXPOSE command.

requirements.txt

```
Flask
Redis
```

app.py

```
from flask import Flask
from redis import Redis, RedisError
import os
import socket

# Connect to Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2, socket_timeout=2)

app = Flask(__name__)
```

```
@app.route("/")
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>cannot connect to Redis, counter disabled</i>"

    html = "<h3>Hello {name}!</h3>" \
           "<b>Hostname:</b> {hostname}<br/>" \
           "<b>Visits:</b> {visits}"
    return html.format(name=os.getenv("NAME", "world"), hostname=socket.gethostname(), visits=visits)


if __name__ == "__main__":
        app.run(host='0.0.0.0', port=80)
```

Now we see that `pip install -r requirements.txt` installs the Flask and Redis libraries for Python, and the app prints the environment variable `NAME`, as well as the output of a call to `socket.gethostname()`. Finally, because Redis isn't running (as we've only installed the Python library, and not Redis itself), we should expect that the attempt to use it here will fail and produce the error message.

> **Note**: Accessing the name of the host when inside a container retrieves the container ID, which is like the process ID for a running executable.

Build the app

That's it! You don't need Python or anything in `requirements.txt` on your system, nor will building or running this image install them on your system. It doesn't seem like you've really set up an environment with Python and Flask, but you have.

Here's what `ls` should show:

```
$ ls
Dockerfile          app.py              requirements.txt
```

Now run the build command. This creates a Docker image, which we're going to tag using `-t` so it has a friendly name.

```
docker build -t friendlyhello .
```

Where is your built image? It's in your machine's local Docker image registry:

```
$ docker images

REPOSITORY          TAG                 IMAGE ID
friendlyhello       latest              326387cea398
```

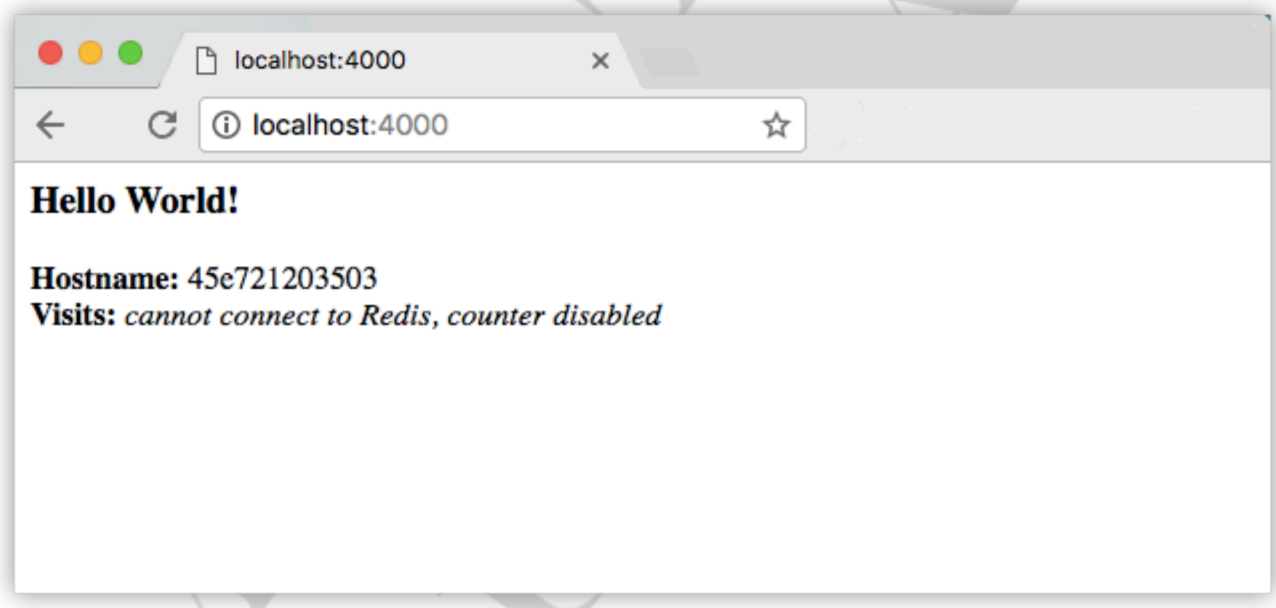
Run the app

Run the app, mapping your machine's port 4000 to the container's EXPOSEd port 80 using -p:

```
docker run -p 4000:80 friendlyhello
```

You should see a notice that Python is serving your app at http://0.0.0.0:80. But that message is coming from inside the container, which doesn't know you mapped port 80 of that container to 4000, making the correct URL http://localhost:4000.

Go to that URL in a web browser to see the display content served up on a web page, including "Hello World" text, the container ID, and the Redis error message.



You can also use the curl command in a shell to view the same content.

```
$ curl http://localhost:4000

<h3>Hello World!</h3><b>Hostname:</b> 8fc990912a14<br/><b>Visits:</b> <i>cannot connect to Redis, counter disabled</i>
```

> **Note**: This port remapping of 4000:80 is to demonstrate the difference between what you EXPOSEwithin the Dockerfile, and what you publish using docker run -p. In later steps, we'll just map port 80 on the host to port 80 in the container and use http://localhost.

Hit CTRL+C in your terminal to quit.

Now let's run the app in the background, in detached mode:

```
docker run -d -p 4000:80 friendlyhello
```

You get the long container ID for your app and then are kicked back to your terminal. Your container is running in the background. You can also see the abbreviated container ID with docker ps (and both work interchangeably when running commands):

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
1fa4ab2cf395        friendlyhello       "python app.py"     28 seconds ago
```

You'll see that CONTAINER ID matches what's on http://localhost:4000.

Now use docker stop to end the process, using the CONTAINER ID, like so:

```
docker stop 1fa4ab2cf395
```

Share your image

To demonstrate the portability of what we just created, let's upload our built image and run it somewhere else. After all, you'll need to learn how to push to registries when you want to deploy containers to production.

A registry is a collection of repositories, and a repository is a collection of images—sort of like a GitHub repository, except the code is already built. An account on a registry can create many repositories. The dockerCLI uses Docker's public registry by default.

> **Note**: We'll be using Docker's public registry here just because it's free and pre-configured, but there are many public ones to choose from, and you can even set up your own private registry using Docker Trusted Registry.

Log in with your Docker ID

If you don't have a Docker account, sign up for one at cloud.docker.com. Make note of your username.
Log in to the Docker public registry on your local machine.

```
docker login
```

Tag the image

The notation for associating a local image with a repository on a registry is username/repository:tag. The tag is optional, but recommended, since it is the mechanism that registries use to give Docker images a version. Give the repository and tag meaningful names for the context, such as get-started:part1. This will put the image in the get-started repository and tag it as part1.

Now, put it all together to tag the image. Run docker tag image with your username, repository, and tag names so that the image will upload to your desired destination. The syntax of the command is:

```
docker tag image username/repository:tag
```

For example:

```
docker tag friendlyhello john/get-started:part1
```

Run docker images to see your newly tagged image. (You can also use docker image ls.)

```
$ docker images

REPOSITORY            TAG                 IMAGE ID            CREATED             SIZE
friendlyhello         latest              d9e555c53008        3 minutes ago       195MB
john/get-started      part1               d9e555c53008        3 minutes ago       195MB
python                2.7-slim            1c7128a655f6        5 days ago          183MB
...
```

Publish the image

Upload your tagged image to the repository:

```
docker push username/repository:tag
```

Once complete, the results of this upload are publicly available. If you log in to Docker Hub, you will see the new image there, with its pull command.

Pull and run the image from the remote repository

From now on, you can use docker run and run your app on any machine with this command:

```
docker run -p 4000:80 username/repository:tag
```

If the image isn’t available locally on the machine, Docker will pull it from the repository.

```
$ docker run -p 4000:80 john/get-started:part1
```

```
Unable to find image 'john/get-started:part1' locally

part1: Pulling from orangesnap/get-started

10a267c67f42: Already exists

f68a39a6a5e4: Already exists

9beaffc0cf19: Already exists

3c1fe835fb6b: Already exists

4c9f1fa8fcb8: Already exists

ee7d8f576a14: Already exists

fbccdcced46e: Already exists

Digest: sha256:0601c866aab2adcc6498200efd0f754037e909e5fd42069adeff72d1e2439068

Status: Downloaded newer image for john/get-started:part1

 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
```

> **Note**: If you don't specify the :tag portion of these commands, the tag of :latest will be assumed, both when you build and when you run images. Docker will use the last version of the image that ran without a tag specified (not necessarily the most recent image).

No matter where docker run executes, it pulls your image, along with Python and all the dependencies from requirements.txt, and runs your code. It all travels together in a neat little package, and the host machine doesn't have to install anything but Docker to run it.

Dockerfile Instructions

We have seen in previous section that Dockerfile is used to build docker images. It contains the list of instructions that docker reads to setup an Image. There are around a dozen Instruction that we can use in our Dockerfile.

1.ADD
2.CMD
3.ENTRYPOINT
4.ENV
5.EXPOSE
6.FROM
7.MAINTAINER
8.RUN
9.USER
10.VOLUME
11.WORKDIR
12.ONBUILD

FROM

This instruction is used to set the base image for subsequent instructions. It is mandatory to set this in the first line of a Dockerfile. You can use it any number of times though.
Example:

```
FROM ubuntu:latest
```

MAINTAINER

This is a non-executable instruction used to indicate the author of the Dockerfile.
Example:

```
MAINTAINER <name>
```

RUN

This instruction lets you execute a command on top of an existing layer and create a new layer with the results of command execution.
For example, if there is a pre-condition to install PHP before running an application, you can run appropriate commands to install PHP on top of base image (say Ubuntu) like this:

```
FROM ubuntu

RUN apt-get update update apt-get install php5
```

CMD

The major difference between `CMD` and `RUN` is that `CMD` doesn't execute anything during the build time. It just specifies the intended command for the image. Whereas RUN executes the command during build time.
Note: there can be only one `CMD` instruction in a Dockerfile, if you add more, only the last one takes effect.
Example:

```
CMD "echo" "Hello World!"
```

EXPOSE

While running your service in the container you may want your container to listen on specified ports. The `EXPOSE` instruction helps you do this.
Example:

```
EXPOSE 6456
```

ENV

This instruction can be used to set the environment variables in the container.
Example:

```
ENV var_home="/var/etc"
```

COPY

This instruction is used to copy files and directories from a specified source to a destination (in the file system of the container).

Example:

```
COPY preconditions.txt /usr/temp
```

ADD

This instruction is similar to the `COPY` instruction with few added features like remote URL support in the source field and local-only tar extraction. But if you don't need an extra feature, it is suggested to use `COPY` as it is more readable.

Example:

```
ADD http://www.site.com/downloads/sample.tar.xz /usr/src
```

ENTRYPOINT

You can use this instruction to set the primary command for the image.

For example, if you have installed only one application in your image and want it to run whenever the image is executed, `ENTRYPOINT` is the instruction for you.

Note: arguments are optional, and you can pass them during the runtime with something like `docker run <image-name>`.

Also, all the elements specified using `CMD` will be overridden, except the arguments. They will be passed to the command specified in `ENTRYPOINT`.

Example:

```
CMD "Hello World!"

ENTRYPOINT echo
```

VOLUME

You can use the `VOLUME` instruction to enable access to a location on the host system from a container. Just pass the path of the location to be accessed.

Example:

```
VOLUME /data
```

USER

This is used to set the UID (or username) to use when running the image.

Example:

```
USER daemon
```

WORKDIR

This is used to set the currently active directory for other instructions such as `RUN`, `CMD`, `ENTRYPOINT`, `COPY` and `ADD`.

Note that if relative path is provided, the next `WORKDIR` instruction will take it as relative to the path of previous `WORKDIR` instruction.

Example:

```
WORKDIR /user

WORKDIR home

RUN pwd
```

This will output the path as `/user/home`.

ONBUILD

This instruction adds a trigger instruction to be executed when the image is used as the base for some other image. It behaves as if a `RUN` instruction is inserted immediately after the `FROM` instruction of the downstream Dockerfile. This is typically helpful in cases where you need a static base image with a dynamic config value that changes whenever a new image must be built (on top of the base image).

Example:

```
ONBUILD RUN rm -rf /usr/temp
```

Dockerhub has Dockerfile for every official Image that's hosted there.

Above screenshot is from nginx official repository from Dockerhub. If you see there are links to Dockerfile for every version of the image. The links points to Dockerfile hosted in github.

```
1  FROM debian:stretch-slim
2
3  MAINTAINER NGINX Docker Maintainers "docker-maint@nginx.com"
4
5  ENV NGINX_VERSION 1.13.1-1~stretch
6  ENV NJS_VERSION   1.13.1.0.1.10-1~stretch
7
8  RUN apt-get update \
9          && apt-get install --no-install-recommends --no-install-suggests -y gnupg1 \
10         && \
11         NGINX_GPGKEY=573BFD6B3D8FBC641079A6ABABF5BD827BD9BF62; \
12         found=''; \
13         for server in \
14                 ha.pool.sks-keyservers.net \
15                 hkp://keyserver.ubuntu.com:80 \
16                 hkp://p80.pool.sks-keyservers.net:80 \
17                 pgp.mit.edu \
18         ; do \
19                 echo "Fetching GPG key $NGINX_GPGKEY from $server"; \
20                 apt-key adv --keyserver "$server" --keyserver-options timeout=10 --recv-key
21         done; \
22         test -z "$found" && echo >&2 "error: failed to fetch GPG key $NGINX_GPGKEY" && exit
23         apt-get remove --purge -y gnupg1 && apt-get -y --purge autoremove && rm -rf /var/li
24         && echo "deb http://nginx.org/packages/mainline/debian/ stretch nginx" >> /etc/apt/
25         && apt-get update \
26         && apt-get install --no-install-recommends --no-install-suggests -y \
```