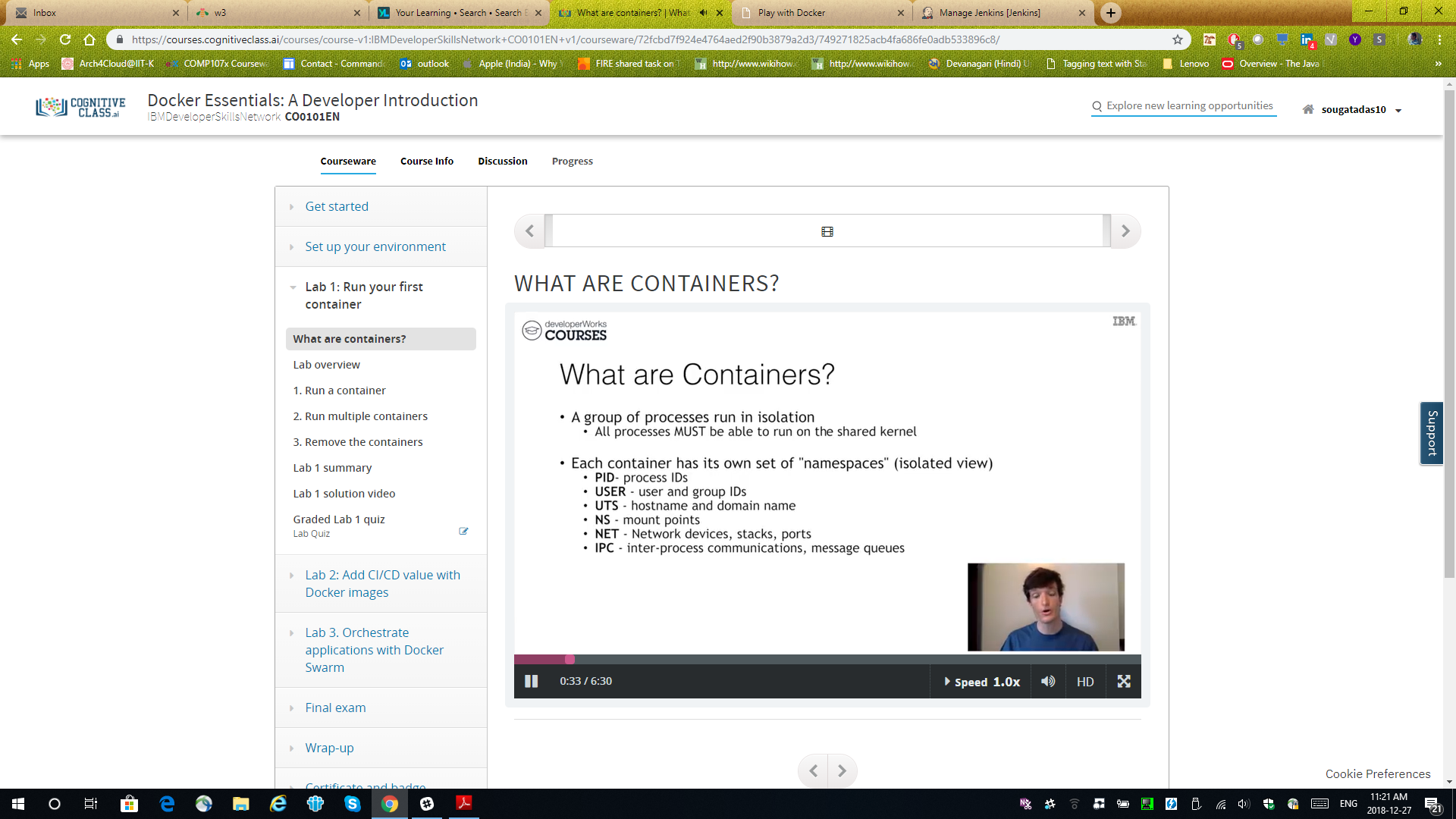
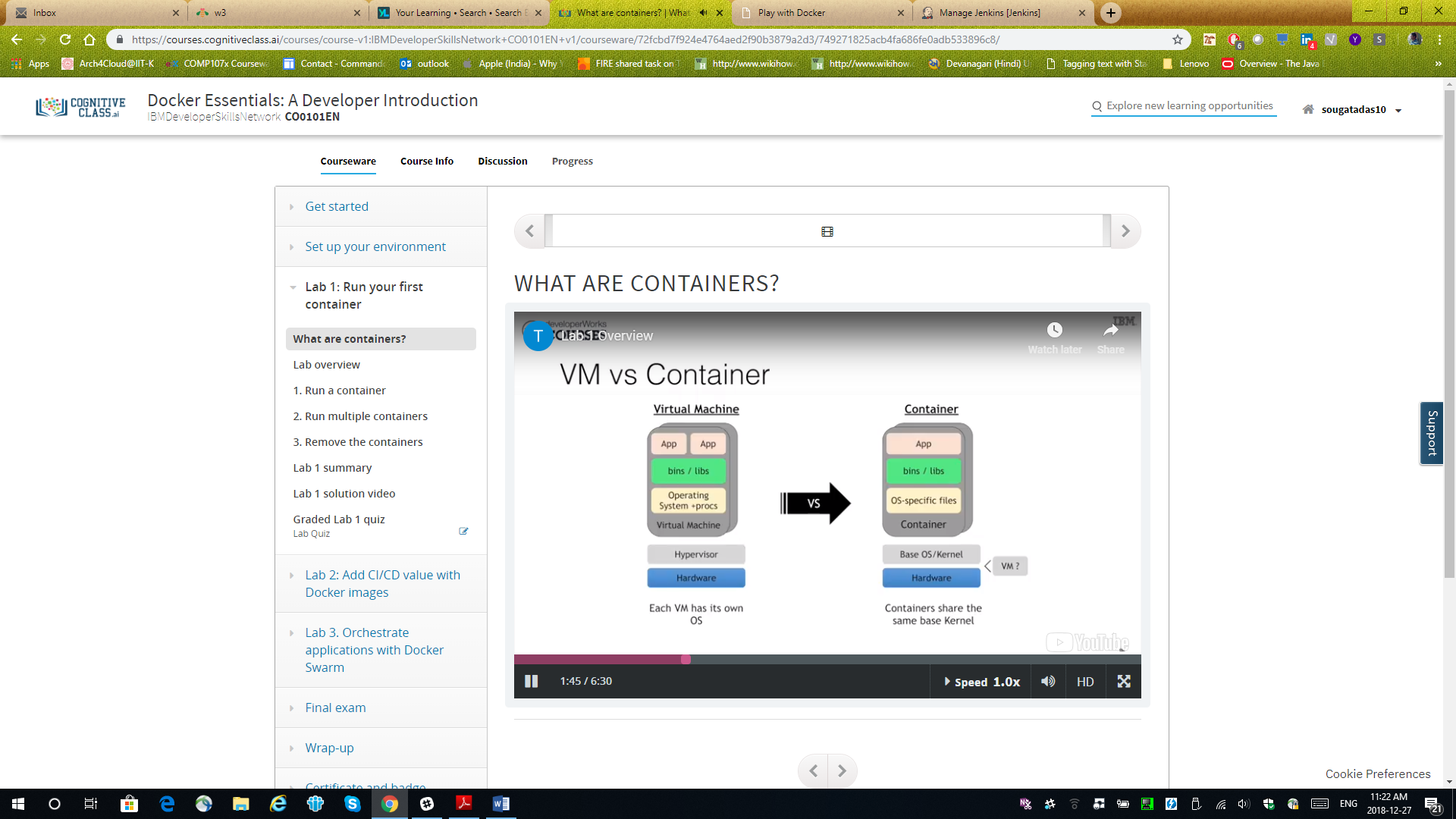
# **Dockers**

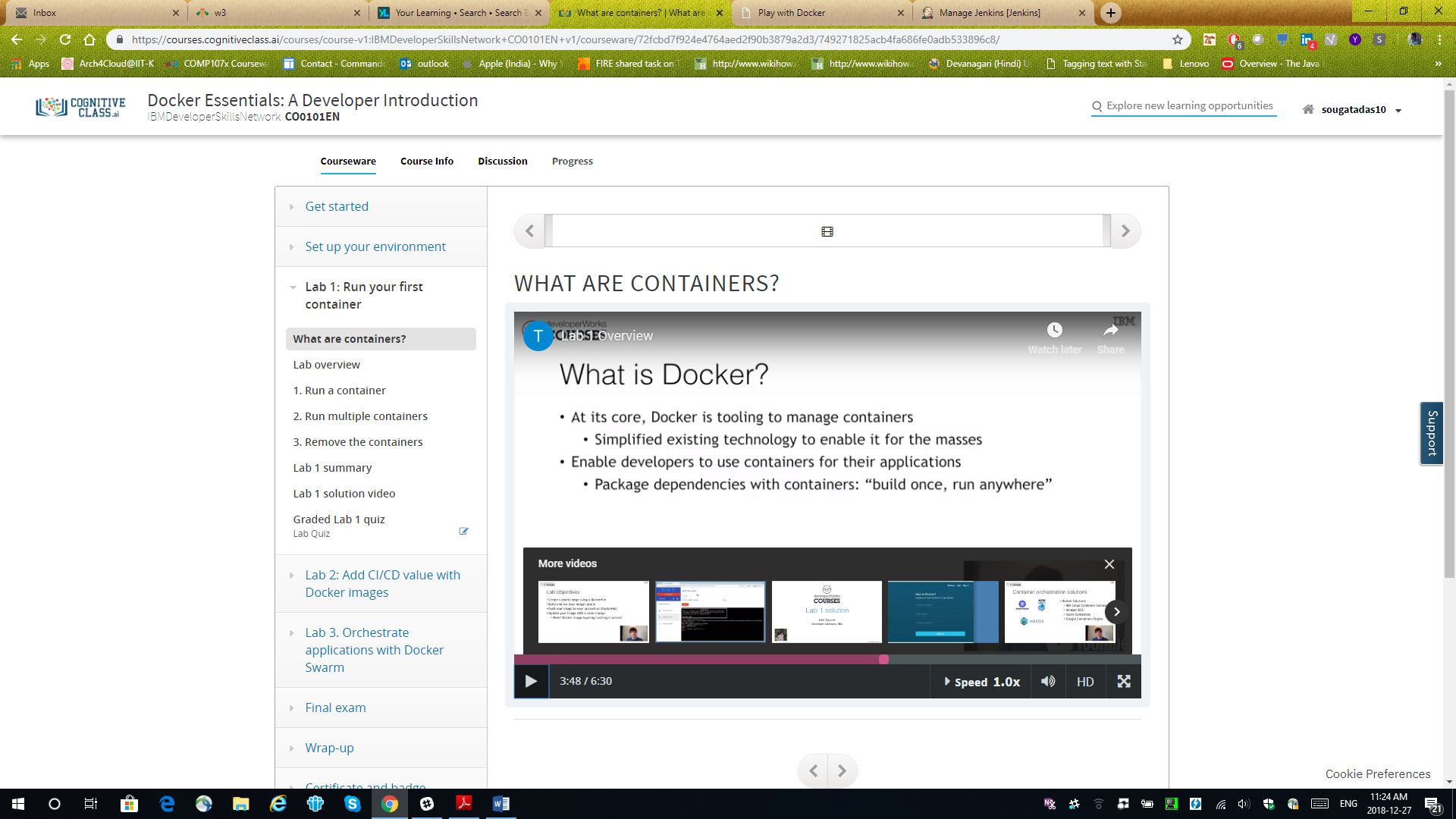
## **Containers**

Containers are just a process (or a group of processes) running in isolation, which is achieved with Linux namespaces and control groups. Linux namespaces and control groups are features that are built into the Linux kernel. Other than the Linux kernel itself, there is nothing special about containers.

What makes containers useful is the tooling that surrounds them. The labs in this course use Docker, which has been the understood standard tool for using containers to build applications. Docker provides developers and operators with a friendly interface to build, ship, and run containers on any environment.



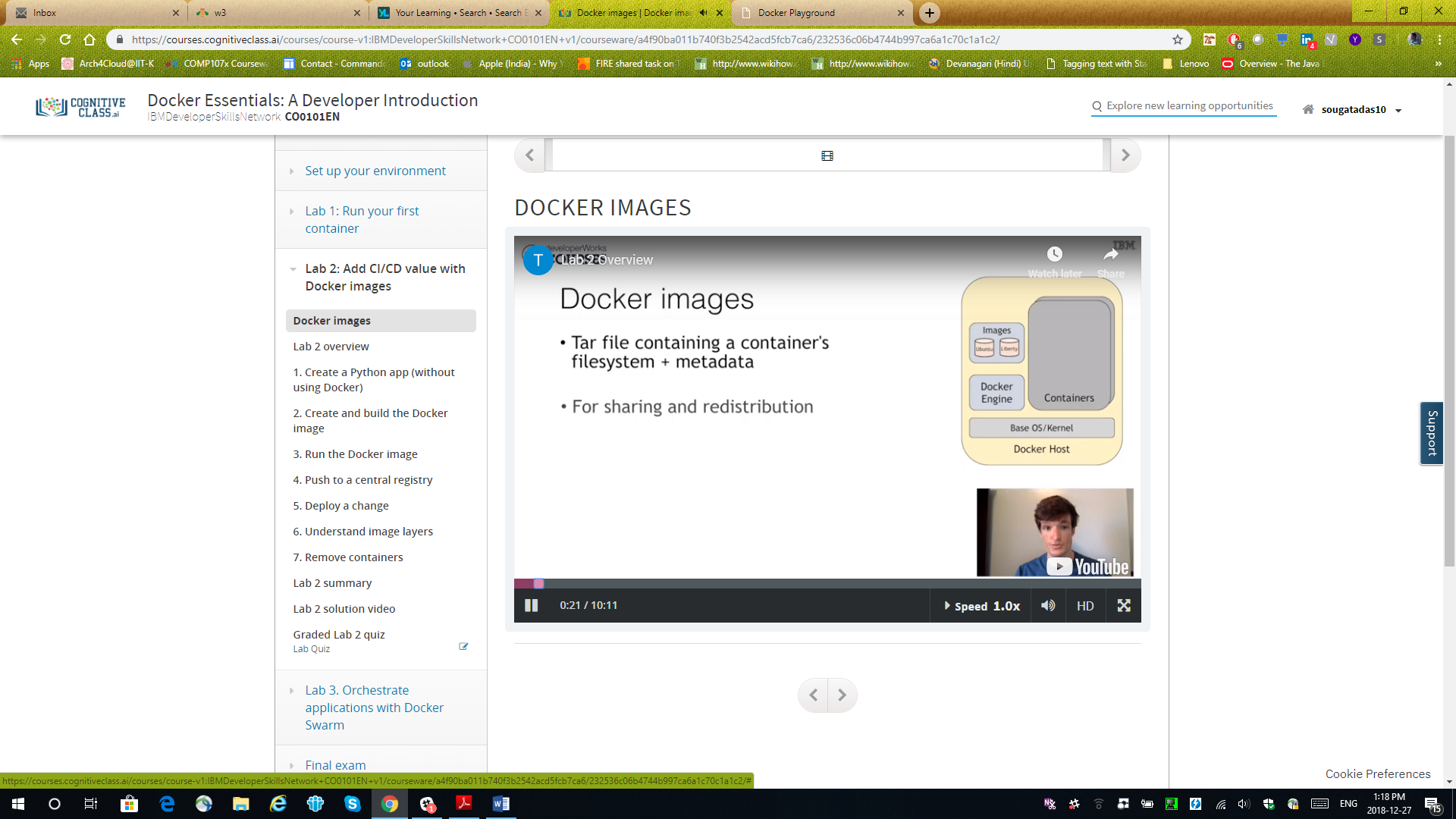


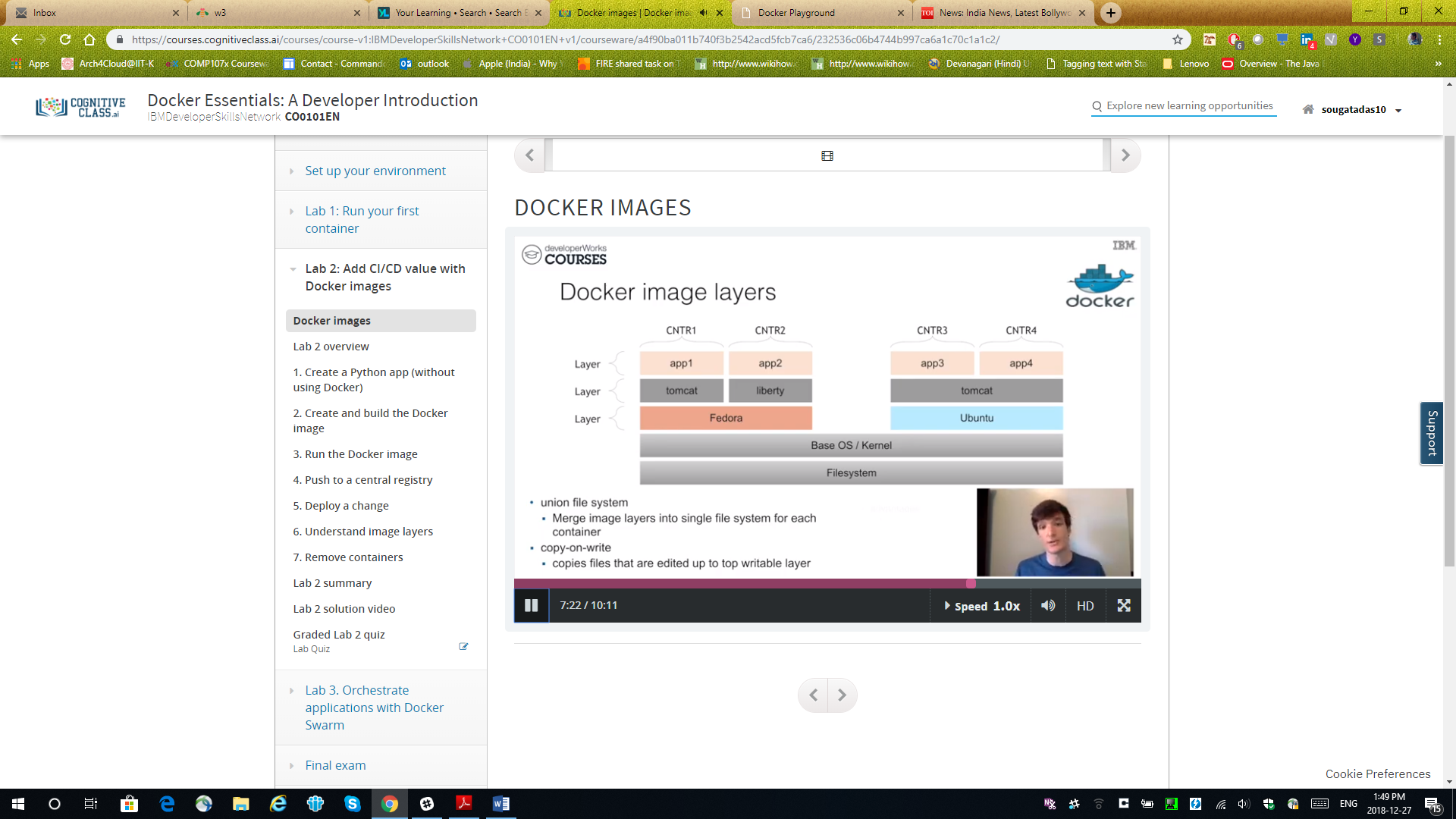


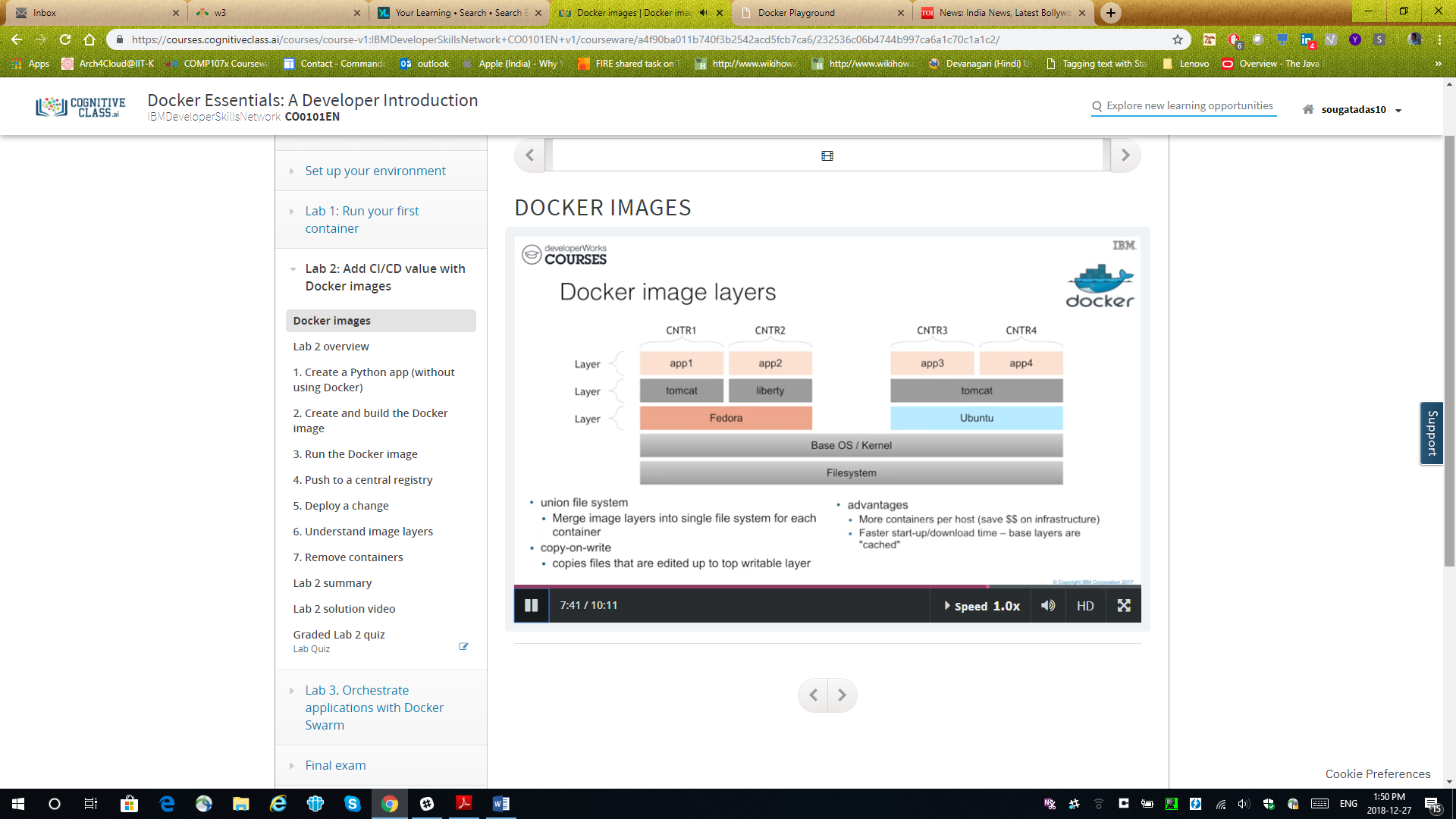
The -t flag allocates a pseudo-TTY

docker container run --detach --publish 8080:80 --name nginx nginx

You are using a couple of new flags here. The --detach flag will run this container in the background. The publish flag publishes port 80 in the container (the default port for NGINX) by using port 8080 on your host. Remember that the NET namespace gives processes of the container their own network stack. The --publishflag is a feature that can expose networking through the container onto the host.







## **Push to a central registry**

Username: sougatadas10

Password: fuse56910\*

1. Log in to the Docker registry account by entering docker login on your terminal:

$ docker login

Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to create one.

Username:

1. Tag the image with your username.

The Docker Hub naming convention is to tag your image with [dockerhub username]/[image name]. To do this, tag your previously created image python-hello-world to fit that format.

$ docker tag python-hello-world [dockerhub username]/python-hello-world

1. After you properly tag the image, use the docker push command to push your image to the Docker Hub registry:

$ docker push jzaccone/python-hello-world

The push refers to a repository [docker.io/jzaccone/python-hello-world]

2bce026769ac: Pushed

64d445ecbe93: Pushed

18b27eac38a1: Mounted from library/python

3f6f25cd8b1e: Mounted from library/python

b7af9d602a0f: Mounted from library/python

ed06208397d5: Mounted from library/python

5accac14015f: Mounted from library/python

latest: digest: sha256:508238f264616bf7bf962019d1a3826f8487ed6a48b80bf41fd3996c7175fd0f size: 1786

1. Check your image on Docker Hub in your browser.

Navigate to Docker Hub and go to your profile to see your uploaded image.

Now that your image is on Docker Hub, other developers and operators can use the docker pull command to deploy your image to other environments.

**Remember:** Docker images contain all the dependencies that they need to run an application within the image. This is useful because you no longer need to worry about environment drift (version differences) when you rely on dependencies that are installed on every environment you deploy to. You also don't need to follow more steps to provision these environments. Just one step: install docker, and that's it.

## **Understand image layers**

One of the important design properties of Docker is its use of the union file system.

Consider the Dockerfile that you created before:

FROM python:3.6.1-alpine

RUN pip install flask

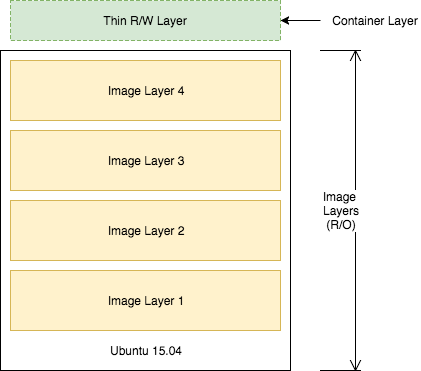
CMD ["python","app.py"]

COPY app.py /app.py

Each of these lines is a layer. Each layer contains only the delta, or changes from the layers before it. To put these layers together into a single running container, Docker uses the union file system to overlay layers transparently into a single view.

Each layer of the image is read-only except for the top layer, which is created for the container. The read/write container layer implements "copy-on-write," which means that files that are stored in lower image layers are pulled up to the read/write container layer only when edits are being made to those files. Those changes are then stored in the container layer.

The "copy-on-write" function is very fast and in almost all cases, does not have a noticeable effect on performance. You can inspect which files have been pulled up to the container level with the docker diff command. For more information, see the command-line reference on the [docker diff](https://docs.docker.com/engine/reference/commandline/diff/) command.



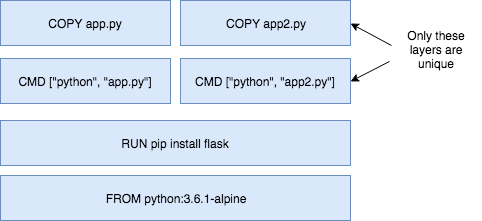
Because image layers are read-only, they can be shared by images and by running containers. For example, creating a new Python application with its own Dockerfile with similar base layers will share all the layers that it had in common with the first Python application.

FROM python:3.6.1-alpine

RUN pip install flask

CMD ["python","app2.py"]

COPY app2.py /app2.py



You can also see the sharing of layers when you start multiple containers from the same image. Because the containers use the same read-only layers, you can imagine that starting containers is very fast and has a very low footprint on the host.

You might notice that there are duplicate lines in this Dockerfile and the Dockerfile that you created earlier in this lab. Although this is a trivial example, you can pull common lines of both Dockerfiles into a base Dockerfile, which you can then point to with each of your child Dockerfiles by using the FROM command.

Image layering enables the docker caching mechanism for builds and pushes. For example, the output for your last docker push shows that some of the layers of your image already exist on the Docker Hub.

$ docker push jzaccone/python-hello-world

The push refers to a repository [docker.io/jzaccone/python-hello-world]

94525867566e: Pushed

64d445ecbe93: Layer already exists

18b27eac38a1: Layer already exists

3f6f25cd8b1e: Layer already exists

b7af9d602a0f: Layer already exists

ed06208397d5: Layer already exists

5accac14015f: Layer already exists

latest: digest: sha256:91874e88c14f217b4cab1dd5510da307bf7d9364bd39860c9cc8688573ab1a3a size: 1786

To look more closely at layers, you can use the docker image history command of the Python image you created.

$ docker image history python-hello-world

IMAGE CREATED CREATED BY SIZE COMMENT

f1b2781b3111 5 minutes ago /bin/sh -c #(nop) COPY file:0114358808a1bb... 159B

0ab91286958b 5 minutes ago /bin/sh -c #(nop) CMD ["python" "app.py"] 0B

ce41f2517c16 5 minutes ago /bin/sh -c pip install flask 10.6MB

c86415c03c37 8 days ago /bin/sh -c #(nop) CMD ["python3"] 0B

<missing> 8 days ago /bin/sh -c set -ex; apk add --no-cache -... 5.73MB

<missing> 8 days ago /bin/sh -c #(nop) ENV PYTHON\_PIP\_VERSION=... 0B

<missing> 8 days ago /bin/sh -c cd /usr/local/bin && ln -s idl... 32B

<missing> 8 days ago /bin/sh -c set -ex && apk add --no-cache ... 77.5MB

<missing> 8 days ago /bin/sh -c #(nop) ENV PYTHON\_VERSION=3.6.1 0B

<missing> 8 days ago /bin/sh -c #(nop) ENV GPG\_KEY=0D96DF4D411... 0B

<missing> 8 days ago /bin/sh -c apk add --no-cache ca-certificates 618kB

<missing> 8 days ago /bin/sh -c #(nop) ENV LANG=C.UTF-8 0B

<missing> 8 days ago /bin/sh -c #(nop) ENV PATH=/usr/local/bin... 0B

<missing> 9 days ago /bin/sh -c #(nop) CMD ["/bin/sh"] 0B

<missing> 9 days ago /bin/sh -c #(nop) ADD file:cf1b74f7af8abcf... 4.81MB

Each line represents a layer of the image. You'll notice that the top lines match to the Dockerfile that you created, and the lines below are pulled from the parent Python image. Don't worry about the <missing> tags. These are still normal layers; they have just not been given an ID by the Docker system.