# All notes

## Sec 3 - Working with Docker

Setup the Docker desktop

Managing venv is very important and it can be done via conda/pip

### File contents:

1. Dockerfile: This has instructions to execute the env

L1: FROM <https://stackoverflow.com/questions/54000157/purpose-of-from-command-docker-file>

The alpine is the Linux OS version

L2: COPY <https://phoenixnap.com/kb/docker-add-vs-copy>

COPY <src> … <dest>

We basically copy all files in the folder and create an app/ folder on Docker

L3: Set this as a working directory

Rest are project dependent

1. Other files for our app. Like python and requirements file

### 2. Build a Docker app

We need to build Docker image and Container

**Docker image**

Tag and add all files for our image

docker build -t <image-tag> .

Check all the images built on Docker with

docker images --all

Now, on Docker Desktop, we can only see it in the image, not in a container. So, we now have to run it in a container and set the right port to it

docker run --name <container-name> -p 5000:5000 <image-tag>

--publish , -p | Publish a container’s port(s) to the host

### Notes -

### Common Docker Commands

Cheatsheet: <https://www.docker.com/sites/default/files/d8/2019-09/docker-cheat-sheet.pdf>

* Get info about the container

docker ps

* Get useful commands

docker exec --help

* Get Debugging useful tools

docker logs --help

* How to connect back to a container?

On Docker container, you can open on browser or CLI by using 1st two icons on Docker container.

* Delete container and images commands respectively

docker rm

docker rmi

* Docker Hub login

docker login

* Stopping a container

docker kill

## Sec 4 - DockerHub

Easy sharing of images

### Docker Push Image

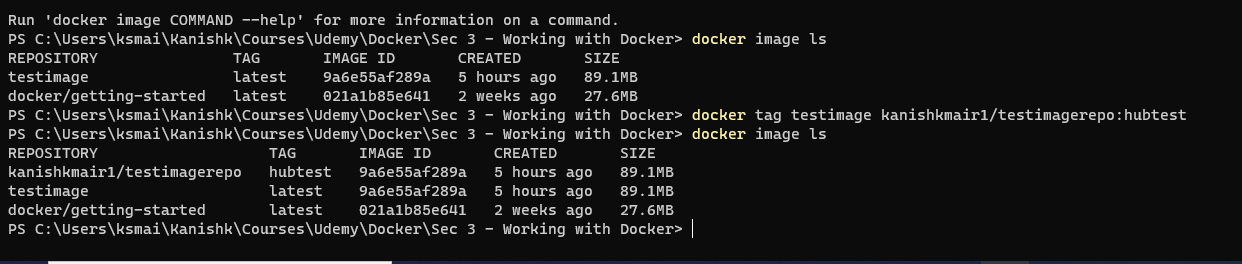
First create a repo on the UI to push our image. For example, we create ***testimagerepo***

Refer: <https://docs.docker.com/engine/reference/commandline/tag/>

docker tag <imagename> <username>/<commitreponame>:<tagname>

We can tag an image to our repo or image w/ a tag to the repo (for later, add a colon and the tag name)

On running ***docker image ls*** we get to see our commit and the tag name

We have repo with the given tag added above

Finally, push the commit to DockerHub

docker push <username>/<commitreponame>:<tagname>

### Docker Pull Image

Find trusted images from Docker Hub on Explore. For ex: We pulled Alpine in the very beginning

docker pull <reponame>

### Update Image

To update our solution, let’s say we update our code. The recommended way is we:

* **Let the Docker container stay immutable as it is**
* **Make changes only to the Docker image and commit changes to it**

### Extending an Image

After pulling, we want to extend code on the image we pull. This can be done directly in our **Dockerfile** to build our base image. Ex:

FROM <username>/<commitreponame>:<tagname>

RUN …

This basically pulls the image first and then add our own extensions to it. We can check this image in our Docker Desktop.

### Volumes



Used to share information between containers.

Docker images are read only layers. We apply changes but initial changes can be lost if original container is deleted.

We can use:

1. Volumes (Preferred)
2. Mount

Volumes are easier to manage and back up across multiple containers

**Commands:**

<https://docs.docker.com/storage/volumes/>

# Create vol

docker volume create <volname>

# Check vol

docker volume ls

# Inspect vol for debugging

docker volume inspect <volname>

# Remove

Docker volume rm <volname>

Let’s setup a file in a volume

-it interactive mode

--name for our container

--mount a more explicit version of -v (volume cmd)

docker run -it --name=newvol --mount source=testvolume,destination=/testvolume alpine

Pulls the alpine image and the interactive mode allows us to communicate directly on CLI mode.

Running `ls` in this Alpine, we can see the testvolume is present in the image.

We can add some data in this volume. After closing and running the above *docker run -it…* command, we can find all our data still present.

## Sec 5 - Build Challenge

**To build a docker image with:**

requests

Scikit-learn

pandas

Tensorflow==1.0.0

Ubuntu:18.04

## Sec 6 - Automated Builds

### Nginx

Practical workflows use NGINX which is basically a web server for all load balancing, http cache, etc.

Auto-build feature allows building automatically once we push to Git. It is like Flask in the sense that it needs it a directory (ex: of HTML files) to serve.

We use **bind volumes in Docker** to do that. In <https://docs.docker.com/storage/volumes/> it is said in the link that:

”While [bind mounts](https://docs.docker.com/storage/bind-mounts/) are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker.”



### NGINX and automation P1

**--rm** tag deletes the image after use. This is helpful since we are running in interactive mode and we can delete it once we exit the session.

docker run -it --rm --name nginxautomation -p 8080:8080 nginx

Now we can run this container on CLI.

To get into the html directory, we have to move through: usr/share/nginx

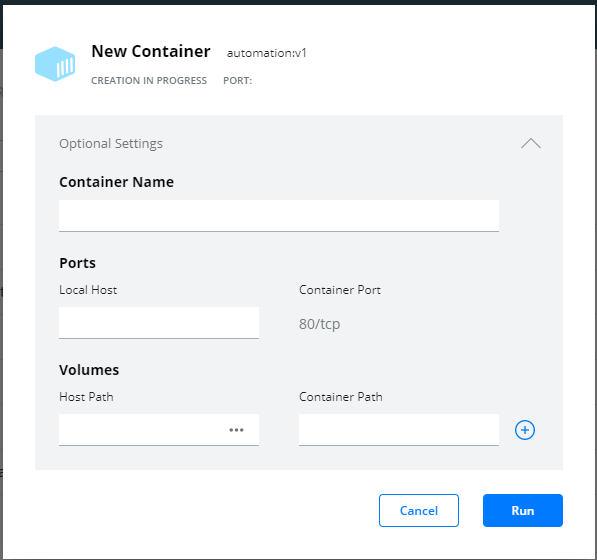
Now, to build our image, we can move build our Dockerfile here along with the html directory. And the build the image.

### NGINX and automation P2

Once a normal html is created, we can create a Dockerfile, let’s say with alpine again and finally build an image with a version

docker build -t automation:v1 .

Then, we can again do a docker run. In console too we can do docker run and provide the necessary setting as:

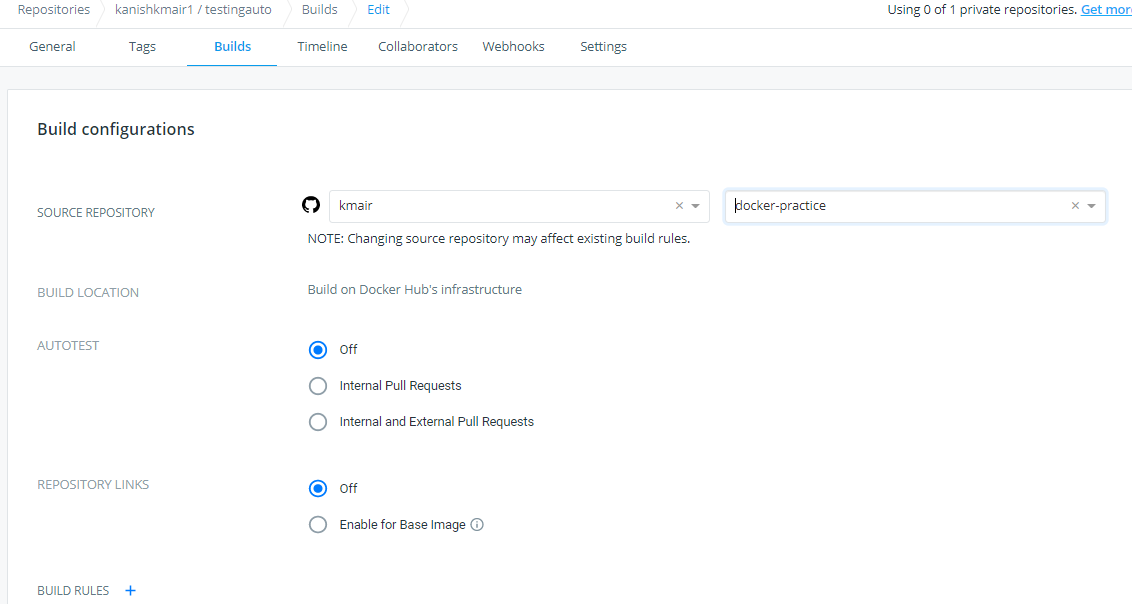


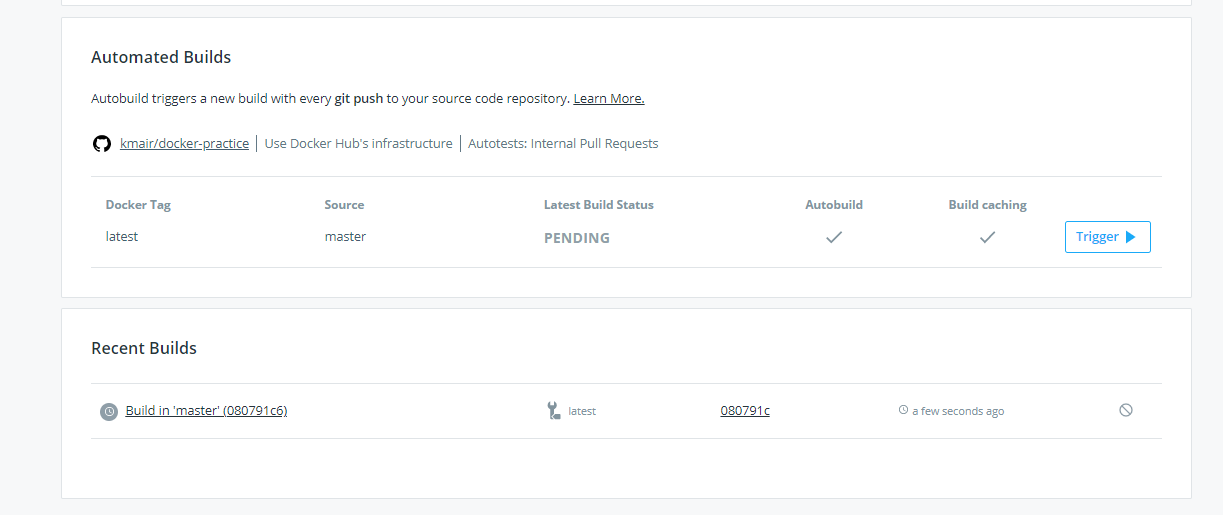
Then check the localhost mentioned above

### NGINX and automation P3

Now we want to have our Github and DockerHub repo setup.

1. Commit the previous folder to git
2. On Dockerhub, create a repo and if it’s the first time, connect your DockerHub account w/ the GitHub account.
3. Then, connect to the repo in the builds as



1. Now, let’s say we make changes to the repo. The Docker Hub would show us the automated build as (here PENDING):  
   

It will move on to IN PROGRESS and then to SUCCESS status if all goes well.

### NGINX and automation P4

To check the latest image code, we can pull from git as zip, extract, etc. or through the DockerHub:

1. Pull the image from General tab (go to public view for copying the pull)
2. Do a docker pull for this latest image
3. Go to Hub and run that image
4. Then, we can check it on our localhost.

**NOTE: The build might fail if we do not have DockerHub know the right path to the Dockerfile. By default it has the root, but we can change it in *Configure Automated Builds* if it’s w/in a specific folder.**

## Sec 7 - File challenge

Our objective is to get the output created in a container to be received locally for use.

Here, we have scrapy to run and create 2 html files that we want to download locally.

1. In the CLI of the running container, we can navigate to dockertest where we have scrapy code
2. Then run: ***scrapy crawl Books*** to create the 2 html files
3. In the container, check the working directory using: ***pwd***
4. Now there are 2 options to download these files locally:
   1. Using volumes
   2. Using docker cp command (simpler option)

docker cp <container-namer>:<pwd-directory> .

So, we basically copy the entire folder. We can do individual files w/o the **‘.’**

We can similarly send the files into the container.

## Sec 8 - Docker Compose

Overview: <https://docs.docker.com/compose/>

Basically a Dockerfile using yaml file.

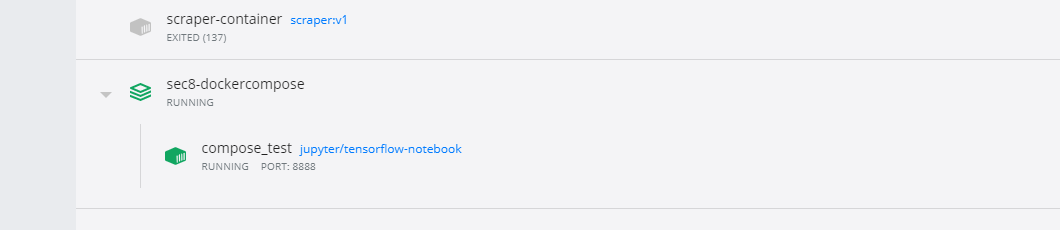
Advantage of this is in multi-container Docker project.

### Docker Compose - Jupyter

We have the yml file to create the tf environment. In order to use this instead of a Dockerfile, we use the following command

docker-compose up

Here, the up command does both ***run*** and ***build*** together. This will give a link to launch a Jupyter environment where we can see all the lusted environment set up. Also, being built through compose, the Container looks a little different as:



We can go in the compose\_test CLI and check our files as well.

Note, to Stop and remove the containers, we can run:

docker-compose down

**USEFUL:** The Docker repository has [awesome-compose](https://github.com/docker/awesome-compose) which has ways of creating projects with interlinked micro-services.

Our next project too will have nginx, SQL, etc.

### Docker Compose - Challenge

We will use the [nginx-flask-mysql](https://github.com/docker/awesome-compose/tree/master/nginx-flask-mysql) project from the above repo where we’ll need to provide info for all 3 services both on backend and proxy. Firstly, going over the repo link shows the directory structure of the 3 services and which ports are called.

“When deploying the application, docker-compose maps port 80 of the proxy service container to port 8080 of the host as specified in the file.”

This means that

Let’s go over the yaml file.

**version: "3.7"** => Python

**services:**

**db:** => Declaration of our database

**image: mysql:8.0.19**

**command: '--default-authentication-plugin=mysql\_native\_password'**

**restart: always** => Restart policy: defines how to restart the container once we exit it

**secrets:** => per service basis

**- db-password**

**volumes:** => volume to access our services

**- db-data:/var/lib/mysql**

**networks:** => network for our container we’re working with for our services to container

**- backnet**

**environment:** => Helps to access our secrets

**- MYSQL\_DATABASE=example**

**- MYSQL\_ROOT\_PASSWORD\_FILE=/run/secrets/db-password**

**backend:**

**build: backend**

**restart: always**

**secrets:**

**- db-password**

**ports:**

**- 5000:5000**

**networks:**

**- backnet**

**- frontnet**

**proxy:**

**build: proxy**

**restart: always**

**ports:**

**- 80:80**

**networks:**

**- frontnet**

**volumes:** => The volumes referenced above point here

**db-data:**

**secrets:** => The secrets referenced above point here

**db-password:**

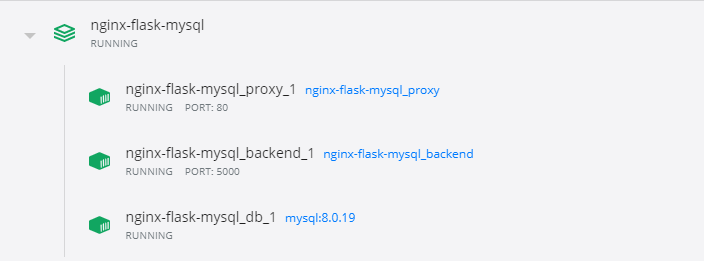
**file: db/password.txt**

**networks:**

**backnet:**

**frontnet:**

Now, in the folder, we can run the docker-compose up to build this project. We’ll have 3 containers running and open the localhost served by the proxy in our browser



## Sec 9 - Docker Swarm

### Swarm Intro

Docs: <https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/>

Might be mostly necessary for DevOps and project pipelines. We can have both physical and cloud m/c running the Docker engine. So, the advantages are running an application w/ redundancy. It creates scalable groups of Docker containers. The Manager nodes assign work to the Worker node to distribute the work with them

For building this, we use docker-stack.yml files. It looks like a compose file and can scale up or tear down an application easily.

### Swarm Deploy

We will deploy [this app](https://github.com/dockersamples/example-voting-app) with focus. After downloading the repo, we want to check the status of swarm. By running

docker info

We see that the Swarm is inactive. We initialize it with:

docker swarm init

And then we can deploy it similar to compose as:

docker stack deploy --compose-file docker-stack.yml <stack-name>

Here the stack-name is vote. We created multiple services which we can examine with:

docker stack services <stack-name>

And explore them through the ports on localhost. Finally to tear down the swarm, run:

docker stack rm <stack-name>

## Sec 10 - AWS

We’ll deploy an application on AWS with EC2. Using AWS Beanstalk can ease all of this for production environment, but presently we are using this

### Launch an instance

Launch a Linux AMI instance. We need to only configure our security instance for public DNS, add a custom TCP rule at a **Port Range of 5000.**

## Building our image on EC2

In Sec - 4, we created an image and committed it as *testimagerepo:hubtest*

Let’s add this to our instance. Now in our folder containing the instance’s pem file, we’ll launch it. But first, we need to make the pem file readable by doing:

chmod 400 <filename>.pem

Now, connect to the instance (using its IPV4 DNS) as:

ssh -i <filename>.pem ec2-user@<Public IPV4 DNS>

Next, update the env and install docker and start it as:

sudo yum update

sudo yum install docker

sudo service docker start

Now, we can pull our required image from docker hub using the pull as:

sudo docker pull <imagerepo>

Now, we can run the above container as :

sudo docker run --name <Name> -p 5000:5000 <imagerepo>

### Viewing our build

Obviously we can’t view the update on our localhost as it is running on an EC2 instance. We can access it based on our security groups. So, if we try to open the IPV4 DNS, it’ll show “This site can’t be reached” because of security groups. We can use the http instead and provide the required port to access it as:

<http://ec2-100-26-244-109.compute-1.amazonaws.com:5000/>

### AWS Deep Learning Containers

AWS has made these special containers available via ECR, EKS, etc. and can be found on the AWS Marketplace. Additionally, we can find Deep Learning images to use directly in EC2 as well.

## Sec 11 - Docker Security

### CVE and OWASP

CVE (Common Vulnerability Exposure) are bugs/exploits that can be found and resolved. We can check them on **NIST.** The NIST framework helps to let is know how to solve cybersecurity issues.

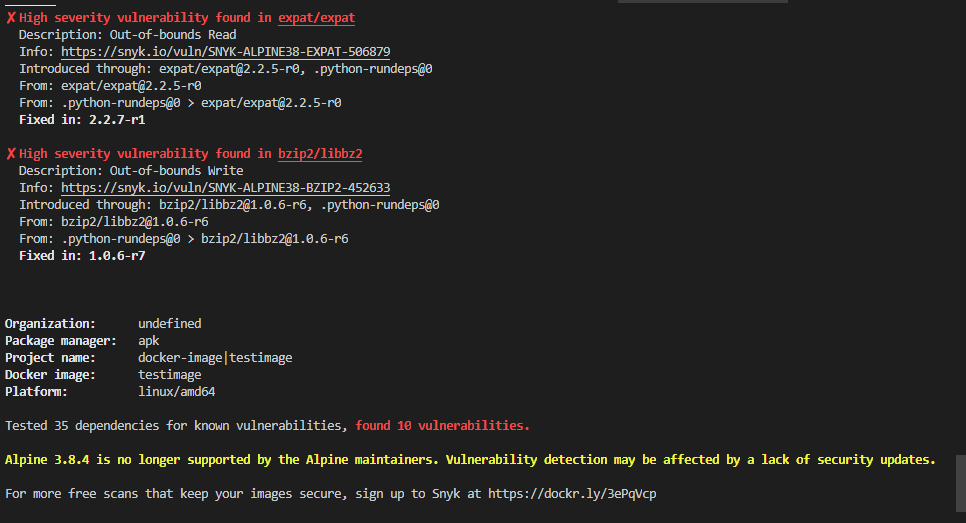
OWASP also has Docker issues.

### Docker Security Scan

We can check our image for flaws with:

docker scan <imagename>

On testing the dependencies, it can list out vulnerabilities as:

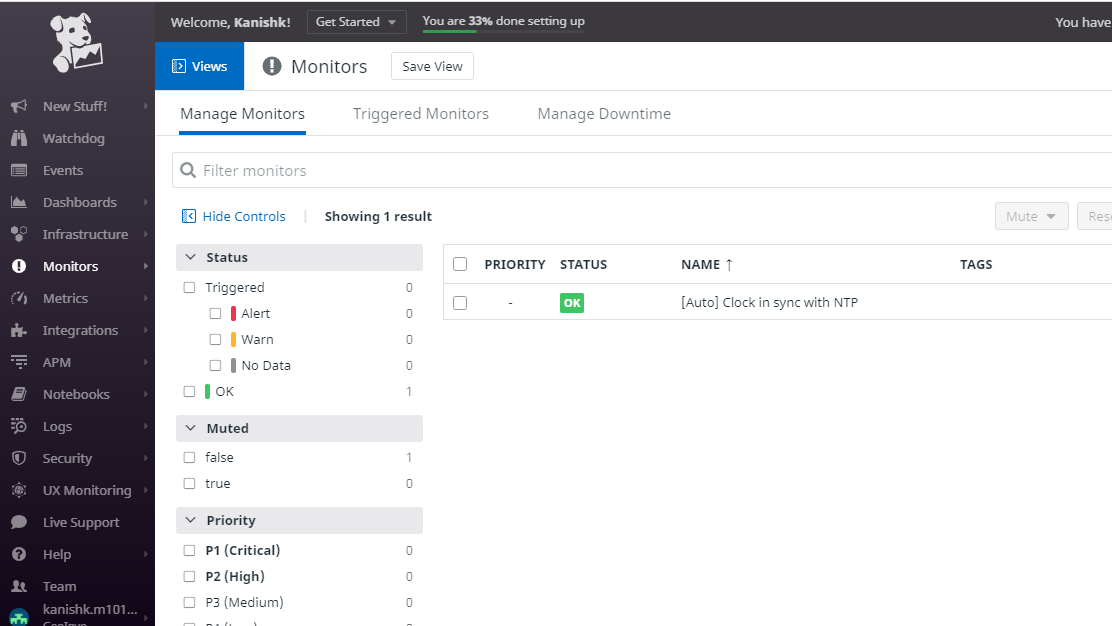


By clicking on link, we go to the details on how to solve it through the Info link.

## Sec 12 - Maintenance and Monitoring

### Monitoring with Datadog

To monitor the container performance, we can use Datadog. When we go into building pipelines for system integrations, it becomes very useful. These tools can help and monitor our Docker containers.

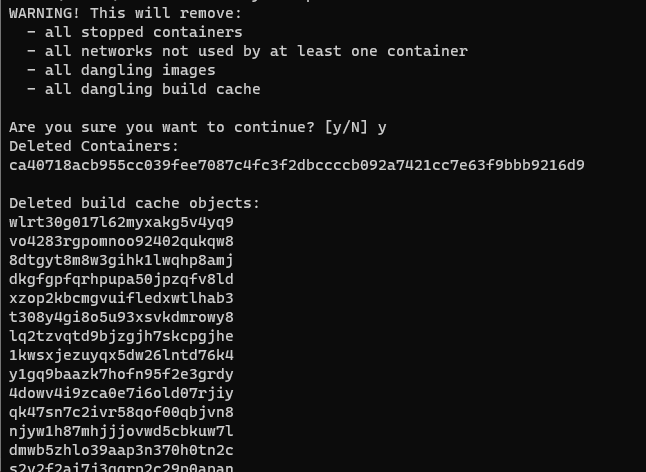


### Container Management

Cleaning of the Docker applications:

docker system prune

This is helpful for the Docker Maintenance. We remove the following (even if containers are deleted, we can spin them up again from our Docker iamges):



## Sec 13 - Kubernetes

### Intro

It’s not related to DS but is very helpful in DevOps as it is a data orchestration tool. We already have Docker swarm that is simple and easy deployments w/ CLI from docker. But Kubernetes comes up for Container Orchestration especially large scale production. It can be more complex to learn as it involves the CI/CD pipeline. Its advantages are:

1. **Flexibility**
2. **Load Balancing**
3. **Automated deployment and Rollback**
4. **Container-level resource management**

Link: <https://www.docker.com/products/kubernetes>

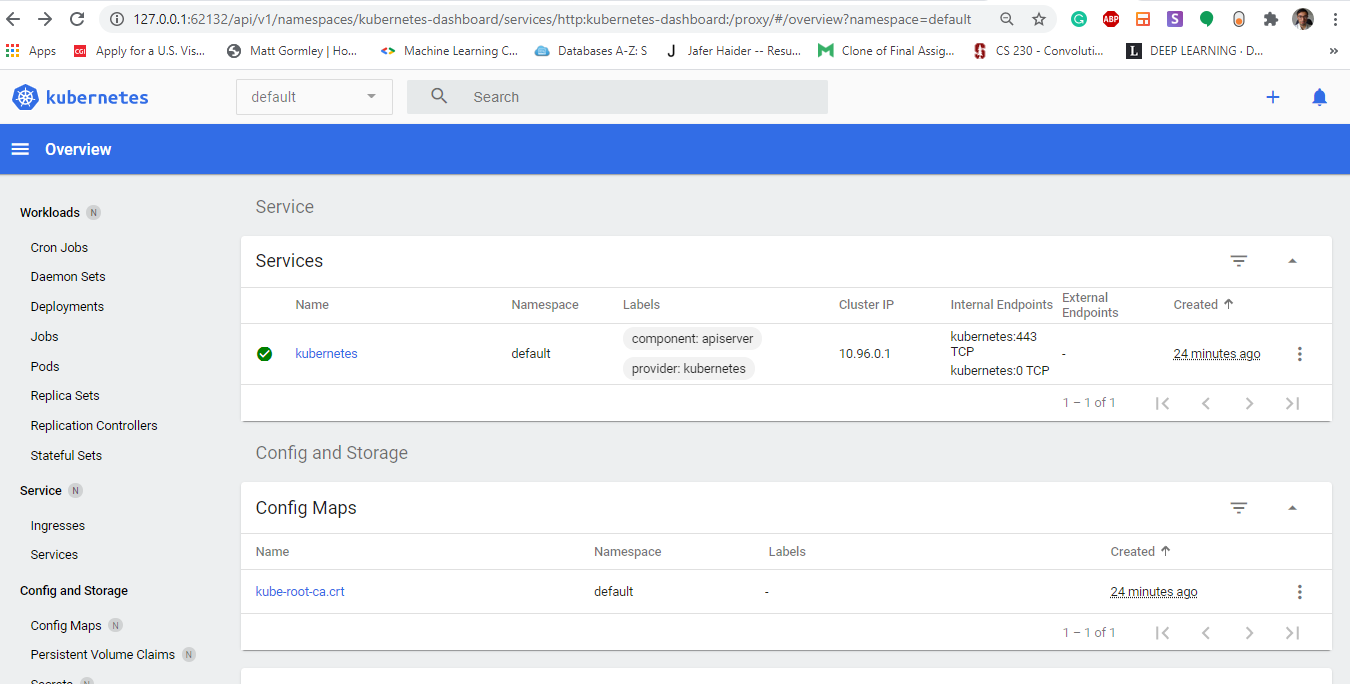
**Overview of Swarm and Kubernetes:**

1. Kubernetes uses micros-services while Swarm deploys them. We can use YAML or compose to install an application.
2. For scaling Swarm can be quicker, but Kubernetes provides much more flexibility. With deployment and scalability, we can easily customize with Kubernetes (ex when we have large demands).

### Demo

There is minikube that can be explored. Kubectl is a CLI for running commands for the clusters. Kubernetes can be run locally, so we’ll follow this minikube tutorial: <https://minikube.sigs.k8s.io/docs/start/>

After reaching Kubernetes dashboard, it opens a local link as:



This makes it easy to manage our cluster, nodes, services, etc.